Paradigmas y Lenguajes

Parte 3

Lic. Ricardo Monzón

PROGRAMACION FUNCIONAL INTRODUCCION.

INTRODUCCION A LA PROGRAMACION FUNCIONAL - LISP

- Lisp fué propuesto por John McCarthy a finales de los 50.
- Es una alternativa al modelo de computación tradicional, es un lenguaje dirigido a procesado simbólico en lugar de numérico, implementando el modelo de las funciones recursivas que proporcionan una definición sintáctica y semánticamente clara.
- Las **listas** son la base tanto de los programas como de los datos en 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

- El único fundamento de control era la recursión y las condicionales simples. Si se necesitaban funciones más complejas se definían en función de las primitivas.
- Muchos de los programas desarrollados dentro del ámbito de la Inteligencia Artificial se implementaron en LISP. Existen formalismos de más alto nivel como los sistemas de producción, sistemas basados en objetos, deducción de teoremas lógicos, sistemas expertos basados en reglas, etc. utilizando al Lisp como lenguaje para su implementación.
- Muchos entornos modernos para la creación de sistemas expertos están implementados en Lisp.

INTRODUCCION A LA PROGRAMACION FUNCIONAL - LISP

La popularidad de LISP como lenguaje de IA se debe, entre otros a:

- Es un lenguaje de manipulación de símbolos
- Inicialmente era un lenguaje interpretado lo que suponía gran rapidez de desarrollo.
- Es un programa uniforme. Los programas y los datos tienen una misma estructura: La lista.
- Existen excelentes compiladores que generan código muy eficiente.
- Existen entornos sofisticados de programación construídos sobre LISP.

INTRODUCCION A LA PROGRAMACION FUNCIONAL - LISP

Características de 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

SIMBOLOS, ATOMOS Y CONSES

Los objetivos fundamentales de este tema son:

- conocer los diferentes tipos de datos que puede apuntar un símbolo en memoria,
- dar ejemplos de diferentes tipos de datos atómicos,
- escribir gráficamente un símbolo con apuntadores del nombre en memoria,
- conocer si una lista es una lista de verdad o una lista punteada,
- conocer si un elemento es un átomo, una lista o ninguno de los dos.
- determinar el número de elementos del nivel superior de la lista,
- dibujar celdas y apuntadores que representen las estructuras CONS y las relaciones de sus elementos en memoria.
- convertir las celdas y apuntadores, que representan listas de verdad y listas punteadas, en listas con niveles únicos o múltiples.

SIMBOLOS LISP Y OBJETOS SIMBOLICOS

- En Lisp todo son objetos que se representan mediante símbolos. La regla de oro de Lisp (lenguaje funcional) es la evaluación de sus objetos. Existen diferentes reglas de evaluación dependiendo del tipo de objeto.
- Cada símbolo tiene asociados cinco componentes que lo caracterizan: nombre imprimible, el valor, una definición de función, una lista de propiedades y, un paquete en el que se define dicho símbolo.
- Lisp siempre trabaja con apuntadores, por ello cada uno de los atributos mencionados NO contienen el dato sino un apuntador al mismo. El *nombre imprimible* corresponderá con un nombre simple (secuencia de caracteres).
- Lisp en general, y a menos que se diga lo contrario, convertirá automáticamente las minúsculas en mayúsculas.

SIMBOLOS LISP Y OBJETOS SIMBOLICOS

Los tipos de *valores* que puede tomar un símbolo son:

- el tipo **atom** (atómicos) como un número, un carácter, otro símbolo, un tipo string.
- el tipo **cons**, como una lista.
- una **estructura** definida por el usuario.
- tipos de datos **propios** del usuario.

En cualquier caso, 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

Los **átomos** son estructuras básicas en Lisp. Todos los elementos que no son de tipo **cons** (construcciones de listas) son ATOMOS. 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

- 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"**

- Son utilizados a menudo para manipular textos y puede considerarse como series de caracteres o como vectores de caracteres. Un string puede examinarse para determinar qué caracteres lo componen. La representación imprimible de un string comienza con " y finaliza por ".

TIPO ATOM

- Los **nombres de variables**. El atributo valor del símbolo permite que dicho símbolo actúe como una variable. Todo nombre de variable debe ser un símbolo atómico. El nombre imprimible del símbolo 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 un **símbolo** es más complicada que la de los objetos (enteros, fraccionarios, coma flotante, complejos, caracteres y string) anteriormente vistos. En general, el LISP intentará aplicar la regla de devolver el valor del símbolo.

TIPO ATOM

- **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

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 listas.

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

TIPO CONS – Ejemplos de Listas

```
(1 2 3 4)                ; lista de enteros
(a b c d)                ; lista de símbolos
(#\a #\b #\c #\d)        ; lista de caracteres
(4 algo de "si")         ; lista con entero, símbolos y un string.
(sqrt 2)                 ; lista con 1er a   función
(+ 1 3);                           "
(- 4 6 1);                         "
```

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

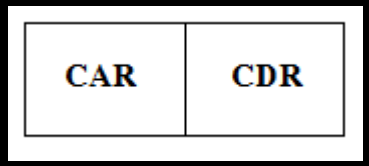
TIPO CONS

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

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**.



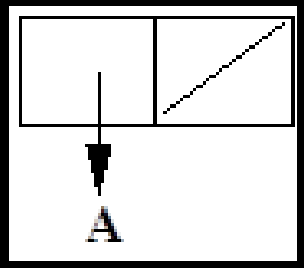
El CAR es la primera parte de la estructura y contiene un apuntador al primer elemento de la lista. El CDR es la segunda parte y contiene un apuntador a la siguiente estructura CONS que contiene los siguientes elementos de la lista o si no hay más elementos un apuntador a NIL. Una lista es unidireccional.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

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

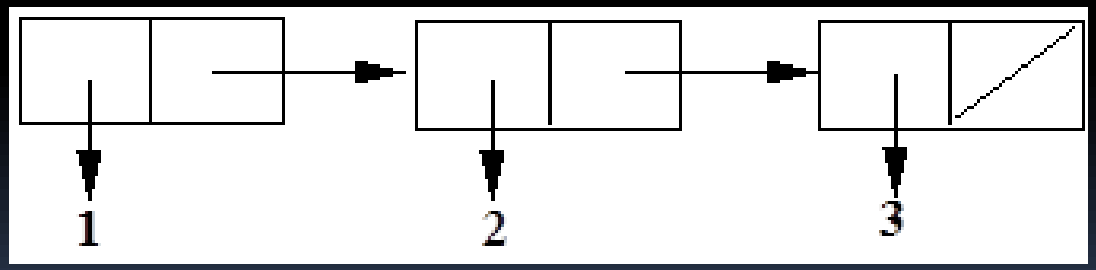
Algunas LISTAS y su representación con ESTRUCTURAS CONS

- () NO TIENE UNA ESTRUCTURA CONS

- (A)

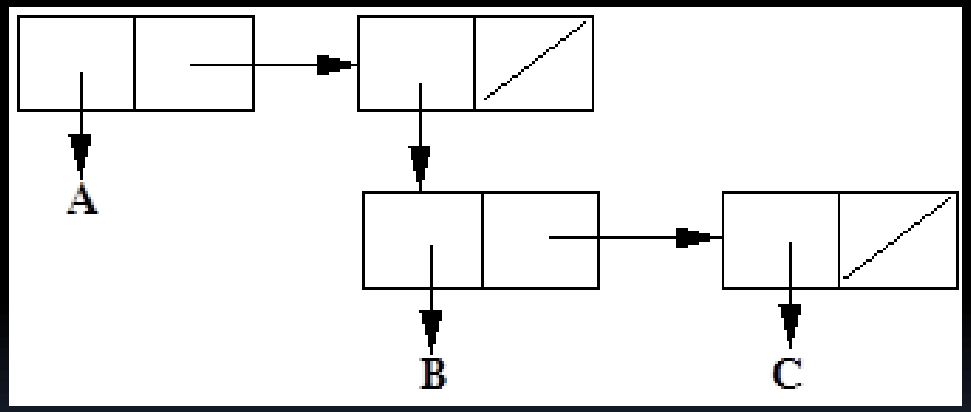


- (1 2 3)



Algunas LISTAS y su representación con ESTRUCTURAS CONS

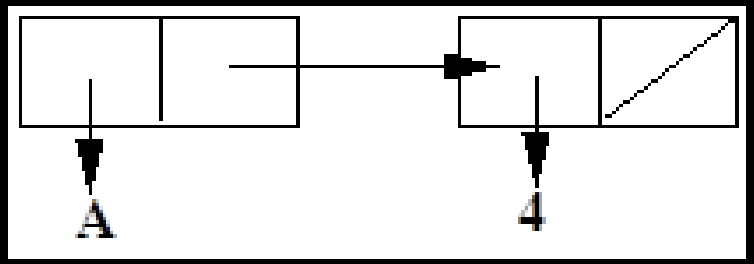
- (A (B C))



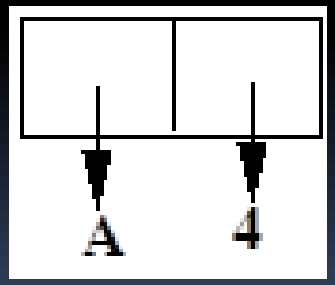
Listas Punteadas

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 punteada (A . 4)



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

Listas Punteadas

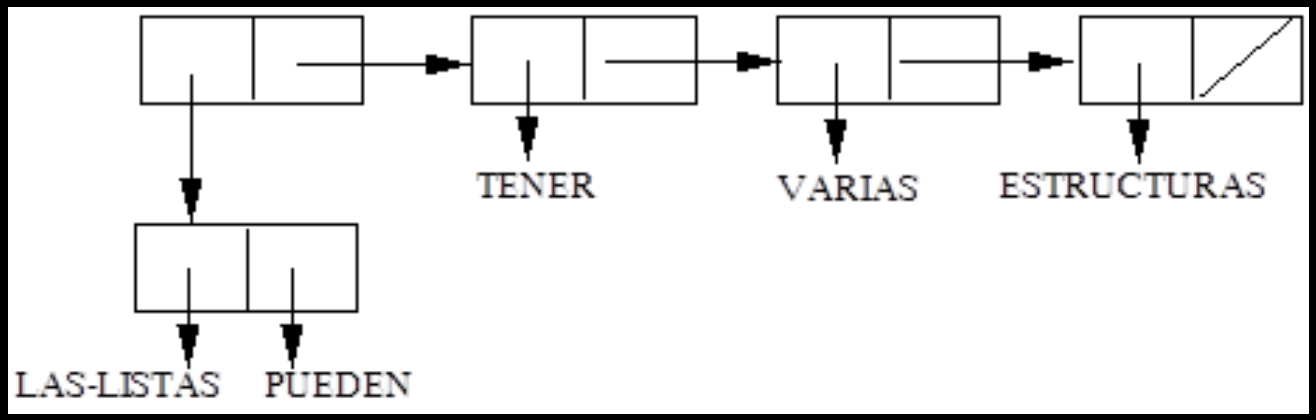
Cuando el último elemento de una lista no apunta a NIL, se genera una lista punteada. La ventaja de las listas punteadas es el ahorro de una casilla CONS. Sin embargo, se pierde en flexibilidad ya que el tratamiento de una lista no podrá depender de la marca de fin de lista. Además en una lista , que 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)

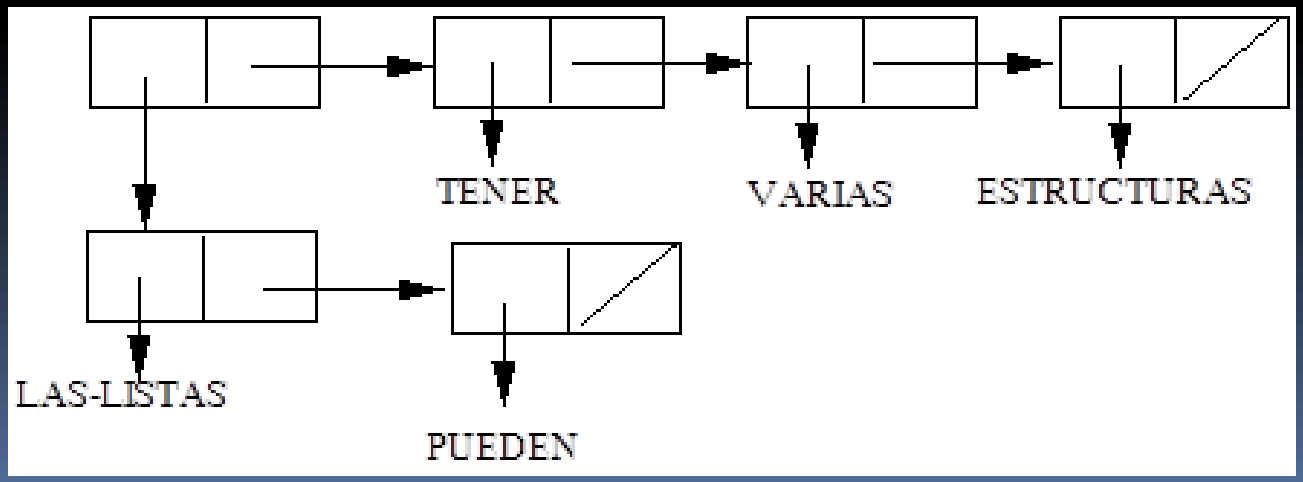
Esta lista no es sintácticamente correcta. Una lista punteada precisa que después del punto aparezca un objeto simple y después un paréntesis.

Ejemplos

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



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



Evaluación en LISP

El *Lisp Listener* es el ciclo que realiza continuamente Lisp, siempre está a la espera de entradas. Cuando recibe una entrada, evalúa la s-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 final de la s-expresión*
3
> “esto es un string” *<return>*
“esto es un string”
>

Podemos decir que cualquier forma que NO es una lista espera un return para poder ser evaluada.
>T *<return>*
T
> A *<return>*
5 (si A tiene el valor 5)

Ejemplos

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

```
> '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 implementaciones actuales de Lisp son hoy en día muy rápidas en sus operaciones matemáticas. Las formas de las expresiones matemáticas son muy familiares para nosotros. Por ejemplo, el *- multiplicación, /- división, <- menor que, >=- mayor o igual que, SQRT- raíz cuadrada, SIN- seno, etc.

Cada una de las funciones incluidas en este tema 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.

Funciones Aritméticas Básicas

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

(+ [número número] ...)

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
```

- (resta): devuelve el resultado de todas las restas sucesivas de los argumentos.

(- [número número] ...)

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 número a cero. Ningún argumento, devuelve error.

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

Funciones Aritméticas Básicas

*** (multiplicación**): devuelve el producto de todos los argumentos dados.

(* [número número] ...)

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

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

(/ [número número] ...)

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. Se devuelve un decimal cuando algún argumento no es entero.

> (/ 30 2 4)
3

> (/ 30 2.0 4)
3.75

Funciones Aritméticas Básicas: Ejemplos

```
>> (+ 2 3 4)          ==>  _________
>> (- 10 7 3)         ==>  _________
>> (- 7)              ==>  _________
>> (* 3 4 0.5)        ==>  _________
>> (/ 100 4 5)        ==>  _________
>> (/ 12 9)           ==>  _________
>> (/ 18 7)           ==>  _________
>> (/ 25.0 10)        ==>  _________
>> (/ 2.0)            ==>  _________
>> (/ 48 24 (* 2 3))  ==>  _________
```

Las operaciones aritméticas no modifican los valores de los argumentos que se les pasa. Únicamente devuelven un valor.

Otras Funciones Aritméticas.

Puede obtenerse un entero como resultado de las funciones **FLOOR, CEILING, TRUNCATE, ROUND** y **REM**.

Truncate: Trunca una expresión tendiendo hacia 0.
Formato: **(truncate <expr> <denominación>)**
Round: Redondea hacia el entero positiva mas cercano.
Formato: **(round <expr> <denominación>)**
Floor: trunca una expresión tendiendo hacia el infinito negativo.
Formato: **(floor <expr> <denominación>)**
Ceiling: trunca una expresión tendiendo hacia el infinito positivo.
Formato: **(ceiling <expr> <denominación>)**

En los cuatro casos, <expr> es una expresión real sobre la cual se aplicará la función, y denominación es un valor que por defecto toma 1.0, que se puede adicionar para dividir el primer argumento (<expr>) por este segundo. El resultado se dará luego de realizada la operación.
Las cuatro operaciones, entregan dos resultados, el primero con el resultado de la operación en sí misma, y el segundo con el remanente de la operación.

Otras Funciones Aritméticas: Ejemplos.

```
> (truncate 3.14)
3
0.14000000000000012
> (round 3.123)
3
0.12300000000000022
> (round 3.723)
4
-0.27700000000000014
> (round 3.723 1)
4
-0.27700000000000014
> (round 3.723 1.5)
2
0.7229999999999999
> (floor 3.123)
3
```

Mas Funciones Aritméticas.

- **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**

Funciones Matemáticas.

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

(INCF PLACE [DELTA])

Place 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 valor PLACE, por defecto es 1.

(DECF PLACE [DELTA])
>> (DECF C 10) **==> -5**
>> C **==> -5**
>> (DECF C) **==> -6**
>> C **==> -6**

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

>> (+ 1 C) **==> 6**
>> C **==> 5**

Funciones Matemáticas.

- **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

Predicados de comparación de números.

Estas funciones se llaman normalmente predicados de números que devuelven 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 números.
  (<= 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**

Mas Funciones Aritméticas.

- 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.

Predicados Numéricos.

Los **predicados** son funciones que devuelven los valores:

NIL => falso **T => verdadero**

Los siguientes predicados numéricos aceptan un único 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

- **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. (algunas funciones iterativas COMO MAPCAR ACEPTA LISTAS DE >< LONGITUDES)
- 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.

Consideraciones.

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)

Consideraciones.

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)**
Nota: algún término del operador puede ser el contenido de una variable y/o resultado de otra función	**> (setq A '(2 -3 4))** **> (setq B '(4 (7) ((2))))** **> (setq C (/ 25 5))** **> (+ (- 7 6) B C A)**	**(12 (10) ((12)))**

Consideraciones.

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))**
Contenido de una variable	Número	> (setq A '(1 2 3)) > (expt A 2)	**(1 2 3)**
Contenido de una variable	**Contenido de una variable**	**> (setq A '(1 2 3)) > (setq B (- 6 4) > (expt A B)**	**(1 2 3)**

FUNCIONES PARA OPERAR SOBRE LISTAS

Objetivos.

- Escribir expresiones como datos y "formas" en código Lisp utilizando la función QUOTE,
- Explicarlo que debe hacer una función, escribir una "forma" con sintaxis correcta de Lisp y que cuando se invoque produzca el resultado deseado.
- Predecir el resultado de las formas que llaman a funciones CAR, CDR y sus abreviados C****R.
- Escribir formas que produzcan una salida dada, utilizando las funciones CAR, CDR y la forma abreviada C****R.
- Predecir correctamente la salida para funciones CONS, LIST y APPEND.
- Explicar como se producen los resultados de las evaluaciones de CONS, LIST y APPEND, en términos de manipulación de las estructuras CONS.
- Dados ciertos datos, escribir la forma con llamadas a función CONS, LIST y APPEND para producir las salidas esperadas.
- Llamar a primitivas que te permitan, dada una lista, determinar su longitud, el último elemento, el elemento de una determinada posición, si un elemento pertenece a la misma, etc.

FUNCIONES PARA OPERAR SOBRE LISTAS

EXPRESIONES SIMBOLICAS Y FORMAS

Los átomos y listas en Lisp son conocidos como **expresiones simbólicas**. Cualquier átomo válido o lista es una expresión simbólica. Por ejemplo,

ATOMOS: **5, A, "A ES UN STRING"**

LISTAS: **(UNA LISTA DE ELEMENTOS)**
(((A)))
(+ 3 (* X (/ 100 (- Y Z))))
(DEFUN ADD2 (X) (+ X 2))

Una **forma** en Lisp es un objeto que puede ser evaluado. A un átomo no se le considera como una forma tipo lista. Algunos ejemplos de los que NO son formas de listas son,

- **3** que se evalúa como **3**,
- **A** que se evalúa según el apuntador del atributo valor del símbolo **A**.

FUNCIONES PARA OPERAR SOBRE LISTAS

Ejemplos de los que sí se considera formas de lista son:

```
>> (+ 2 3)
5
> (+ 8 (* 3 4))
20
```

FUNCION QUOTE

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**.

Por ejemplo, ATOMOS como objetos,

```
> (quote X)
X
> 'X
X
```

FUNCIONES PARA OPERAR SOBRE LISTAS

Las listas son conjuntos ordenados de elementos, átomos o listas, conectados. Para realizar la manipulación de los elementos de una lista es importante tener acceso a sus elementos.

Existen tres tipos importantes de funciones de selección de elementos:

- **CAR -** devuelve el primer elemento de la lista.
- **CDR** - devuelve toda la lista menos el primer elemento.
- **C****R** - permite la concatenación de funciones CAR Y CDR. Es decir, CADR, CDDR, CADADR, etc.

FUNCIONES PARA OPERAR SOBRE LISTAS

• La función **CAR**, devuelve siempre el primer elemento de una lista. Su sintaxis es (CAR LISTA), o de una manera más técnica sería, (CAR CONS).
Por ejemplo podríamos tener la siguiente llamada

(CAR (1 2 3)) -==> 1

Sin embargo, esto no funciona, ya que Lisp intentará evaluar la lista (1 2 3). Dará un ERROR. Intentará buscar una función definida con el nombre **1** y cuyos argumentos tomen valores **2** y **3** respectivamente. Como ya habíamos comentado y para evitar que LISP evalúe una lista de datos (y en general cualquier objeto) utilizaremos quote **'**. Así para el ejemplo anterior tendremos

(CAR '(1 2 3)) - -> 1

FUNCIONES PARA OPERAR SOBRE LISTAS

Ejemplos de CAR

```
>> (CAR '(A B C) )          ==>   A
>> (CAR '(A B (C D) E) )    ==>   A
>> (CAR '((A B) C) )        ==>   (A B)
>> (CAR '((A B C)) )        ==>   (A B C)
>> (CAR 'A)                 ==>   ERROR
```

Es importante destacar que la función CAR no genera un nuevo valor, sino que da como resultado la dirección de un elemento que existe.

FUNCIONES PARA OPERAR SOBRE LISTAS

• La función **CDR**, para listas propias, de verdad o no punteadas, devuelve la lista sin el primer elemento. Es decir la función CDR devolverá los elementos apuntados por el CDR de la primera estructura CONS de una lista (o CONS). Para una lista punteada, el CDR de una estructura CONS puede ser un átomo. Por tanto, en estos casos se devolverá un átomo y no una estructura CONS. La sintaxis de esta función es (CDR LISTA) o de forma técnica (CDR CONS).

>> (CDR '(A B C)) ==> (B C)
>> (CDR '(A B (C D) E)) ==> (B (C D) E)
>> (CDR '((A B) C)) ==> (C)
>> (CDR '((A B C))) ==> NIL
>> (CDR 'A) ==> ERROR
>> (CDR '(A . B)) ==> B

FUNCIONES PARA OPERAR SOBRE LISTAS

• Las funciones **C****R**, no son más que una abreviación de las formas CAR y CDR. Podemos invocar a las llamadas (C*R LISTA), (C**R LISTA), (C***R LISTA), (C****R LISTA), donde el * podrá sustituirse por A, para invocar a CAR, o por D para invocar a CDR.

Los caracteres incluidos invocan a las funciones respectivas en orden de derecha a izquierda.

>> (CADR '(A B C))
(CAR (CDR '(A B C))) ==> B
>> (CDAR '((A B C) D E))
(CDR (CAR '(A B C))) ==> (B C)
>> (CADDR '(A B C D E)) ==> C
>> (CAR (CDDDDR '(A B C D E))) ==> E

FUNCIONES PARA OPERAR SOBRE LISTAS

- **LAST** LIST. Esta función devolverá la última estructura CONS de la lista. Por ejemplo,

```
>> (LAST '(A B C))        ==>    (C)
>> (LAST '(A (B C)))      ==>    ((B C))
>> (LAST '(1 2 3.5))      ==>    (3.5)
>> (LAST '())             ==>    NIL
>> (LAST '((A B C))       ==>    ((A B C))
```

- **ELT** SECUENCIA INDICE. Esta función devolverá el elemento de la secuencia que está en la posición índice. El primer elemento de la secuencia tiene el índice en la posición 0. Ejemplos,

```
>> (ELT '(A (B C)) 1)                    ==>    (B C)
>> (ELT '(A B) 3)                        ==>    ERROR
>> (SETQ ESTA-LISTA '(A B C D E))        ==>    (A B C D E)
>> (ELT ESTA-LISTA 2)                    ==>    C
```

FUNCION DE ASIGNACION

• La función **SETQ** asigna un valor a una variable. Su sintaxis es, (SETQ {VAR FORM}+). La Q de SETQ es un mnemotécnico para la aplicación de la función quote. Los argumentos impares de la función no son evaluados.

(SETQ VAR FORM VAR FORM …), puede haber un número par de argumentos. Las evaluaciones de las formas y las asignaciones (primero se evalúa y luego se asigna) se realizan secuencialmente, devolviendo el valor de la última evaluación efectuada. Ejemplos,

>> (SETQ X 5) ==> 5
>> (SETQ NOMBRE "MARIA") ==> "MARIA"
>> (SETQ FRUTA (CAR ´(MANZANA PERA))) ==> MANZANA
>> (SETQ LISTA1 ‘(A B C)) ==> (A B C)
>> (SETQ X 5 Y 6 Z 7) ==> 7

FUNCIONES DE CONSTRUCCION DE LISTAS

Las funciones con las que Lisp permite construir listas son CONS, LIST y APPEND. Con ellas deberemos ser capaces de construir listas de elementos, crear estructuras CONS, añadir más elementos a una lista, concatenar listas, etc.

FUNCIONES NO DESTRUCTIVAS

Son funciones que manipulan los valores dados como parámetros para obtener un valor. Los símbolos que representan dichos parámetros no son modificados.

FUNCIONES DE CONSTRUCCION DE LISTAS

- **CONS** SEXP SEXP

La función **CONS**, crea una nueva estructura cons. Sintácticamente podemos expresarlos como **(CONS SEXP SEXP)**, donde SEXP son expresiones simbólicas. El CAR de la casilla CONS apuntará a la primera SEXP y el CDR a la segunda SEXP. Obsérvese que si la segunda SEXP corresponde con un átomo entonces se creará una lista punteada ("impropia"). La función CONS puede utilizarse para añadir un nuevo elemento al principio de la lista.

>> (CONS 'A '(B C D))	**==>**	**(A B C D)**
>> (CONS '(X Y) '(B C D))	**==>**	**((X Y) B C D)**
>> (CONS '((X Y) (W Z)) '((A B)))	**==>**	**(((X Y) (W Z)) (A B))**
>> (CONS 'A NIL)	**==>**	**(A)**
>> (CONS NIL '(A))	**==>**	**(NIL A)**

FUNCIONES DE CONSTRUCCION DE LISTAS

>> (CONS ‘A ‘B) ==> (A . B)
>> (CONS ‘(A B) ‘C) ==> ((A B) . C)

Función CONS	*Ejemplo*	*Resultado*	*Sirve para...*
(CONS átomo lista)	(CONS 1 '(2 3 4))	(1 2 3 4)	Añadir un elemento a una lista
(CONS átomo átomo)	(CONS 1 2)	(1 . 2)	Crear una lista impropia
(CONS lista lista)	(CONS (1 2) (3 4))	((1 2) 3 4)	Añadir una sublista a una lista
(CONS lista átomo)	(CONS '(1 2) 4)	((1 2) . 4)	Añadir sublista y crear lista impropia

FUNCIONES DE CONSTRUCCION DE LISTAS

- **LIST** &REST ARGS

La función LIST devuelve una lista formada con todos los elementos pasados como argumentos. Los argumentos deberán ser expresiones simbólicas, o s-expresiones válidas. La notación &REST ARG implica que se permite cualquier número de argumentos. Es decir, se le puede pasar a la llamada de la función cualquier número de parámetros La sintaxis corresponderá con llamadas del estilo **(LIST SEXP SEXP ...)**. LIST se utiliza para crear listas propias. Las listas impropias sólo pueden crearse a partir de la función CONS.

>> (LIST 'A 'B 'C) ==> (A B C)
>> (LIST ‘(A) ‘(B) ‘(C)) ==> ((A) (B) (C))
>> (LIST ‘A ‘(B C)) ==> (A (B C))

FUNCIONES DE CONSTRUCCION DE LISTAS

- **LIST** &REST ARGS

>> (LIST ‘QUIERO ‘(Y PUEDO) ‘CREAR ‘((ALGUNA LISTA)) ‘DIVERTIDA)

==> (QUIERO (Y PUEDO) CREAR ((ALGUNA LISTA)) DIVERTIDA)

>> (LIST (LIST (LIST ‘A))) ==> (((A)))

Como se ha podido observar Lisp construye nuevos datos pero sólo en un nivel, el nivel superior.

FUNCIONES DE CONSTRUCCION DE LISTAS

- **APPEND** &REST ARGS

Permite crear una nueva lista concatenando dos o más listas dadas. Mientras que LIST y CONS aceptan listas y átomos como argumentos, APPEND sólo acepta listas, excepto en el último argumento. También acepta un número indefinido de argumentos.

La función **APPEND** concatena los argumentos en una lista. Todos los argumentos, excepto el último deben ser listas. Los argumentos que no sean listas no serán incluidos. Hablando técnicamente, las casillas CONS de todos excepto el último son copiados en el orden que preserve el orden de los argumentos. Así en estas copias, el CDR de la última casilla de cada uno de los argumentos se enlaza con el siguiente. Sintácticamente lo indicaríamos como **(APPEND LISTA LISTA … LISTA SEXP)**.

FUNCIONES DE CONSTRUCCION DE LISTAS

- **APPEND** &REST ARGS

```
>> (APPEND '(A) '(B) '(C))                    ==>   (A B C)
>> (APPEND '(A) '(B C))                       ==>   (A B C)
>> (APPEND '((A)) '((B)) '((C)) )             ==>   ((A) (B) (C))
>> (APPEND '(PODRA) '((CREAR (UNA))) '() '(LISTA))
                ==>   (PODRA (CREAR (UNA)) LISTA)
>> (APPEND (APPEND (APPEND '(A) )))           ==>   (A)
>> (APPEND '(A) NIL)                          ==>   (A)
>> (APPEND NIL '(B C))                        ==>   (B C)
>> (APPEND 'A '(B C))                         ==> ERROR
```

NIL es una lista y por tanto cumple los requerimientos de la definición de APPEND.

```
>> (APPEND '(A) '(B) 'C)                      ==>   (A B . C)
```

FUNCIONES DE CONSTRUCCION DE LISTAS

Otras funciones de listas

(BUTLAST lista &optional n), devuelve una lista con los n últimos elementos de la lista pasada como parámetro eliminados. No se modifica el contenido de la lista original

Ejemplos,

>> (setq b '(4 5 6))
>> (butlast b 2)
<< (4)
>> b
<< (4 5 6)

Si quisiéramos que el resultado se asigne a una nueva lista, haríamos:

(setq lista (butlast lista n))

FUNCIONES DE CONSTRUCCION DE LISTAS

- **NTH <n> <list>**

Permite extraer el elemento "n" de una lista "list".

Ejemplos,

```
>> (setq b '(4 5 6))
>> (nth 1 b)
<< 5
>> (nth 0 b)
<< 4
```

- **NTHCDR <n> <list>**

Permite extraer el cdr de una lista "list" a partir del elemento "n".

Ejemplos,

```
>> (setq b '(4 5 6 7 8))
>> (nthcdr 3 b)
<< (7 8)
>> (nthcdr 1 b)
<< (5 6 7 8)
```

FUNCIONES DE CONSTRUCCION DE LISTAS

- **REVERSE <list>**

  Permite obtener el reverso de una lista.

  Ejemplos,

```
>> (setq b '(4 5 6 7 8))
>> (reverse b)
>> (8 7 6 5 4)
>> b
<< (4 5 6 7 8)
```

FUNCIONES DESTRUCTIVAS

FUNCIONES DESTRUCTIVAS

Las funciones **destructivas** son aquellas que modifican el contenido de algún parámetro.

- **(RPLACA lista elem)**, sustituye el car de la lista con elem.

Ejemplos:

>> (setq a '(1 2 3))
<< (1 2 3)
>> (rplaca a 7)
<< (7 2 3)
>> a
<< (7 2 3)

FUNCIONES DESTRUCTIVAS

FUNCIONES DESTRUCTIVAS

- **(RPLACD lista elem),** sustituye el resto (cdr) de la lista con elem.

Ejemplos:

```
>> a
<< (7 2 3)          ;(1)
>> (rplacd a '(9))
<< (7 9)            ;(2)
>> a
<< (7 9)
>> (rplacd a 6)
<< (7.6)            ;(3)
>> a
<< (7.6)
```

FUNCIONES DESTRUCTIVAS

• **(NCONC lista1 … listan),** a diferencia de APPEND, NCONC devuelve su valor modificando el símbolo que expresa **lista1, y modifica el valor de las sucesivas listas hasta n-1.**

-> **(setq lista1 (APPEND lista1… listan)).**

Ejemplos:

>> (setq a '(1 2 3))
<< (1 2 3)
>> (setq b '(4 5 6))
<< (4 5 6)
>> (nconc a b)
<< (1 2 3 4 5 6)
>> a
<< (1 2 3 4 5 6)
>> b
<< (4 5 6)

FUNCIONES DESTRUCTIVAS

- **(PUSH item lista),** añade un elemento a **lista**
  -> **(setq lista (cons item lista)).**

Ejemplos:

>> (push 9 b)
<< (9 4 5 6)
>> b
<< (9 4 5 6)

- **(POP lista),** elimina el primer elemento de **lista**
  -> **(setq lista (cdr lista)).**

>> (pop b)
<< 9
>> b
<< (4 5 6)

OTRAS FUNCIONES

- **LENGTH** SECUENCIA. Devuelve el número de elementos existentes en el nivel superior de la lista (o secuencia).

Por ejemplo,

>> (LENGTH ‘(A B C)) ==> 3

>> (LENGTH ‘(A B . C)) ==> 2 (en este caso devuelve 2, porque el segundo elemento es uno solo llamado “Par punteado”).

>> (LENGTH ‘()) ==> 0

OTRAS FUNCIONES

- **MEMBER** ITEM LISTA {&KEY {:TEST / :TEST-NOT / :KEY}}.

La función MEMBER busca en el nivel superior de la LISTA un elemento que sea igual, **EQL**, que el ITEM. Si se encuentra un elemento que cumple esta condición, la función devolverá una lista cuyo CAR es el elemento buscado y el CDR es el resto de la lista. Por el contrario si no se encuentra ningún elemento se devolverá NIL. Por defecto, el test que se realiza es EQL. Sin embargo, se puede expresar otro tipo de comparación en la opción KEY de la lista de parámetros. La opción :TEST permite realizar comparaciones sucesivas del ITEM con cada uno de los elementos de la lista a través del operador señalado después de :TEST, hasta encontrar el primer elemento que cumple dicha condición, es decir, que de NO-NIL.

OTRAS FUNCIONES

La opción :TEST-NOT es semejante a la anterior, salvo que se detiene al encontrar el primer elemento en cuya evaluación se obtenga NIL. Finalmente la opción :KEY hace lo mismo que las anteriores pero aplica la función indicada en clave.
Veamos algunos ejemplos:

```
>> (MEMBER 'C '(A B C D E F))                    ==> (C D E F)
>> (MEMBER 'Z '(A B C D E F))                    ==> NIL
>> (MEMBER 'J '(A B (I J) F))                    ==> NIL
>> (MEMBER '(X Y) '(A B C (X Y) D))              ==> NIL
        ; Su valor es NIL por que compara con EQL
>> (MEMBER '(X Y) '(A B C (X Y) D) :TEST #'EQUAL)
                                                 ==> ((X Y) D)
>> (MEMBER '7 '(3 5 7 9) :TEST-NOT #'>)          ==> (7 9)
```

OTRAS FUNCIONES

FMAKUNBOUND <sym>
Permite desligar un símbolo de función.

>> **(fmakunbound mi-suma)** ; desliga la función mi-suma
>> **mi-suma** **==>** Error: The variable MI-SUMA is unbound

MAKUNBOUND <sym>
Permite desligar un símbolo de su valor.

>> **(setq a 4)** ; liga el símbolo a con 4
>> A **==>** 4
>> **(makunbound 'a)** ; desliga el símbolo a
>> A **==>** Error: The variable A is unbound.

OTROS PREDICADOS

BOUNDP <sym>

El predicado boundp chequea si el símbolo sym, tiene un valor ligado a el. Retorna T si tiene un valor ligado, o NIL en caso contrario.

>> **(setq a 1)** ; liga la variable a con el valor 1
>> **(boundp 'a)** ; retorna T – el valor es 1

FBOUNDP <sym>

El predicado fboundp chequea si el símbolo sym, tiene una definición de función ligada a el. Retorna T si tiene un valor ligado, o NIL en caso contrario.

>> (defun f1 (x) (print x)) ; set up function F1
>> (fboundp 'f1) → T
>> (fboundp 'car) → T
>> (fboundp 'f2) → Nil

PREDICADOS Y OPEADORES LOGICOS P/LISTAS

Predicados sobre tipos de datos.

Todos los predicados que se muestran a continuación devuelven T si son ciertos o NIL sino lo son.

- **ATOM** OBJETO. Devuelve true si el objeto NO es una construcción CONS.
- **CONSP** OBJETO. Devuelve cierto si el objeto es CONS.
- **LISTP** OBJETO. Devuelve cierto si el objeto es CONS o NIL (la lista vacía).
- **NULL** OBJETO. Será cierto si objeto es NIL.
- **TYPEP** OBJETO TIPO-ESPECICADO. Es cierto cuando el objeto pertenece al tipo definido en tipo especificado.

Predicados de igualdad.

Los objetos Lisp pueden verificarse a diferentes niveles de igualdad:

Igualdad numérica- **=**

Identidad referencial – **EQ.**

Identidad representacional- **EQL**. (acepta solo dos argumentos)

Igualdad estructural- **EQUAL** (acepta solo dos argumentos)

Igualdad de valores – **EQUALP**.

PREDICADOS DE IGUALDAD

EQ X Y.

La función EQ testea si X e Y están referenciados por punteros iguales.

Solo se utiliza para caracteres, simbolos y enteros pequeños menores o iguales a 255.

Sintácticamente tendríamos (**EQL SEXPR SEXPR**), donde la sexpr se deberá evaluar como un número o variable.

>> **(eq 'a 'a)** → T

>> **(eq 1 1)** → T

>> (eq 256 256) → Nil (solo hasta 255)

>> (eq 1 1.0) → Nil

>> (eq ''a'' ''a'') → Nil

PREDICADOS DE IGUALDAD

EQL X Y.

La función EQL testea si X e Y representan el mismo valor. Esta función puede devolver false aunque el valor imprimible de los objetos sea igual. Esto sucede cuando los objetos apuntados por variables parecen los mismos pero tienen diferentes posiciones en memoria. Este predicado no conviene utilizarlo con strings, ya que su respuesta no será fiable. Por el contrario se deberá utilizar con variables y números cuando sea preciso conocer si tienen la misma posición en memoria.

Sintácticamente tendríamos (**EQL SEXPR SEXPR**), donde la sexpr se deberá evaluar como un número o variable.

PREDICADOS DE IGUALDAD

EQL X Y.

Ejemplos

```
>> (EQL 5 5)                          ==> T
>> (EQL 5 5.0)                        ==> NIL
>> (SETQ A ´WORD)
>> (SETQ B ´WORD)
>> (EQL A B)                          ==> NIL
>> (SETQ L ´(A B C))
>> (EQL ´(A B C) L)                    ==> NIL (NO FIABLE)
>> (SETQ M L)
>> (EQL L M)                          ==> T (dos variables apuntan al
                                             mismo sitio )
>> (SETQ N ´(A B C))
>> (EQL L N)                          ==> NIL
>> (EQUAL L A)                         ==> NIL
```

PREDICADOS DE IGUALDAD

EQUAL X Y.

Verifica si los objetos X e Y, son estructuralmente iguales. Es decir, los valores imprimibles de los objetos son iguales. Sintácticamente tendríamos que expresarlo (EQUAL SEXPR SEXPR).

Ejemplos

```
>> (EQUAL 5.0 5)                    ==> NIL
>> (EQUAL NIL ())                   ==> T
>> (SETQ A ´WORD  B ´WORD)
>> (EQUAL A B)                      ==> T
>> (SETQ L ´(A B C))
>> (EQUAL ´(A B C) L)               ==> T
>> (SETQ M L)
>> (EQUAL L M)                      ==> T
>> (SETQ N ´(A B C))
>> (EQUAL L N)                      ==> T
>> (EQUAL L A)                      ==> NIL
```

PREDICADOS DE IGUALDAD

EQUALP X Y.

La función EQUALP testea si X e Y representan el mismo valor, pero a diferencia de EQUAL, esta función se denomina Case Insensitive. A diferencia de Eql, puede aplicarse a los string.

```
>> (EQUALP 5 5)                    ==> T
>> (EQUALP 5 5.0)                  ==> T
>> (EQUALP  ‘’a’’ ‘’A’’)           ==> T
```

PREDICADOS DE IGUALDAD

Resumen:

- **Eq** – Punteros idénticos. Palabras con caracteres, símbolos y pequeños enteros.
- **Eql** – Solo números del mismo tipo.
- **Equal** – Listas y Strings.
- **Equalp** – Caracteres y strings con identificación de mayúsculas., números de diferentes tipos, arreglos.

PREDICADOS DE IGUALDAD

	funcion fn			
Argumentos	**eq**	**eql**	**equal**	**equalp**
(fn 'a 'a)	T	T	T	T
(fn 1 1)	T	T	T	T
(fn 1 1.0)	NIL	NIL	NIL	T
(fn 1.0 1.0)	NIL	T	T	T
(fn "a" "a")	NIL	NIL	T	T
(fn '(a b) '(a b))	NIL	NIL	T	T
(setq a '(a b))				
(setq b a)				
(setq c '(a b))				
(fn a b)	T	T	T	T
(fn a c)	NIL	NIL	T	T
(fn '(a b) '(A B))	NIL	NIL	T	T
(fn '(a b) '(c d))	NIL	NIL	NIL	NIL
(fn "a" "A")	NIL	NIL	NIL	T
(fn "abc" "abcD")	NIL	NIL	NIL	NIL

OPERADORES LOGICOS

Lisp tiene tres de los operadores lógicos más comunes como primitivas, AND, OR y NOT.
Los demás pueden crearse a partir de estos. Los funciones AND y OR devuelven un valor cuando las condiciones son ciertas, en otro caso devuelven nil.
Estas funciones no evalúan necesariamente todos los argumentos. Una vez que se verifica la condición se devuelve NO-NIL o T.
Por ello es importante el orden en el que se colocan los argumentos.

OPERADORES LOGICOS

AND {FORM}*.
La función AND evalúa sus argumentos en orden. Si cualquiera de los argumentos se evalúa a NIL, se detiene la evaluación y se devuelve el valor NIL. Por el contrario si todos los argumentos son NO-NIL, devolverá el resultado de la última evaluación.
Sintácticamente (AND SEXPR SEXPR …), está permitido cualquier número de argumentos.
Ejemplos,

```
>> (AND T  T  T  (* 2 5))                          ==> 10
>> (SETQ X 3 CONT 0)
>> (INCF CONT)
>> (AND (<= CONT 10) (NUMBERP X) (* 2 X))          ==> 6
>> (AND (EVENP X) (/ X 2))                         ==> NIL
```

OPERADORES LOGICOS

OR {FORM}*.

La función OR evalúa sus argumentos en orden. Si cualquiera de los argumentos se evalúa a NO-NIL, se detiene la evaluación y se devuelve es valor. Por el contrario si todos los argumentos son NIL, la función OR devolverá NIL. Sintácticamente (OR SEXPR SEXPR …), está permitido cualquier número de argumentos.

Ejemplos,

>> (OR NIL NIL (NULL ´(A B)) (REM 23 13)) ==> 10

>> (SETQ X 10)

>> (OR (< 0 X) (DECF X)) ==> T

>> (SETQ X 10)

>> (OR (CONSP X) (ODDP X) (/ X 2)) ==> 5

OPERADORES LOGICOS

NOT FORM.

La función NOT evalúa un único argumento. La función NOT devuelve T o NIL. Si los argumentos se evalúan a NIL, entonces devuelve T, en otro caso NIL. Sintácticamente (NOT SEXPR), está permitido sólo un argumento.

Ejemplos:

```
>> (NOT NIL)                                    ==> T
>> (NOT T)                                      ==> NIL
>> (NOT (EQUAL ´A ´A) )                         ==> NIL
>> (SETQ X ´(A B C) )
>> (AND (NOT (NUMBERP X)) (NOT (ATOM X)) "ESTO ES
     CONS")                                     ==> "ESTO ES CONS"
```

SECUENCIA DE ACCIONES

Se llama ***bloque*** a una secuencia de sentencias que deben ser ejecutadas secuencialmente en el orden en el que aparecen.

LISP proporciona diferentes formas de crear estos bloques.

Algunos de ellos se crean de forma implícita, como veremos mas adelante (loop y return).

LISP también proporciona mecanismos para crear bloques de forma explícita.
Para ello utilizaremos los bloques creados por la familia prog.

SECUENCIA DE ACCIONES

progn

permite crear un bloque de sentencias que serán evaluadas secuencialmente. Se toma el valor devuelto por la última de ellas como valor de retorno de la forma progn completa.

(progn [< sentencia 1 > ... < sentencia n>])

Donde hay que observar que se podría crear un bloque sin sentencias.

Proporciona una forma de agrupar una serie de expresiones LISP. Cada una corresponde con una forma determinada y serán evaluadas en orden.

PROGN {((VAR {INIT})/ VAR*)} DECLARACIONES CUERPO).

SECUENCIA DE ACCIONES

El valor devuelto por PROGN corresponde al de la última forma evaluada.

PROG1 y PROG2, devuelven el valor de la primera y segunda forma respectivamente, mientras se evalúan el resto de las formas.

Si en la declaración de un PROGN se incluyen variables locales, la asignación de sus valores se hace en paralelo, al igual que sucedía con LET. LISP da la posibilidad de que la asignación de valores a las variables se realice secuencialmente mediante PROG*
(PROG* ...)
La devolución de un valor para una forma puede hacerse explícito mediante RETURN.
(RETURN resultado)

SECUENCIA DE ACCIONES

Ejemplos:

>> (progn (setq a 23) (setq b 35) (setq c (* a b))) → 805

>> (prog1 (setq a 23) (setq b 35) (setq c (* a b))) → 23

>> (prog2 (setq a 23) (setq b 35) (setq c (* a b))) → 35

>> c → 805

FUNCIONES DEFINIDAS POR EL USUARIO, LIGADURA, ASIGNACION Y AMBITO.

Una **función** es un objeto Lisp que puede invocarse como un procedimiento. Una función puede tomar una serie de argumentos y devolver uno o más valores, pero al menos uno. Lisp tiene un gran número de funciones predefinidas, a menudo llamadas *primitivas*, aunque también permite la definición de funciones propias.

Lisp es un lenguaje funcional en el que todos los programas se escriben como colecciones de funciones y procedimientos. La definición de la función se almacena en memoria como un atributo/objeto del símbolo. Por su parte una función se referenciará por el nombre imprimible de un símbolo cuando se realice una llamada a dicha función.

FUNCIONES DEFINIDAS POR EL USUARIO, LIGADURA, ASIGNACION Y AMBITO.

Por ejemplo la llamada a la función de suma **(+ 2 3)**. La llamada a la función es una lista cuyo primer elemento es el nombre de la función y el resto formarán los parámetros de dicha función.

Lisp comprobará que efectivamente el primer elemento de la lista se corresponde con alguna función predefinida o del usuario. Los argumentos de la llamada a la función serán igualmente evaluados antes de pasárselos como información a la función (este procedimiento se usa siempre en las llamadas a menos que la función sea una macro o una forma especial).

FUNCIONES DEFINIDAS POR EL USUARIO, LIGADURA, ASIGNACION Y AMBITO.

Si el argumento de una función es a su vez otra llamada a función, entonces se evalúa primero la función del argumento y el resultado de la misma se pasa a la función principal.

Por ejemplo:

ENTRADA:	**(+ 3 (* 2 4))**
EVALUACION DE ARGUMENTOS:	**(+ 3 8)**
SALIDA:	**11**

DEFINICION DE FUNCIONES.

DEFUN (DEfine FUNction), permite al usuario definir sus propias funciones. En general una función consta de tres partes:
nombre de la función, argumentos y ***cuerpo*** (lo que se suele llamar lambda expresión)**.**

(DEFUN nombre-función lambda-lista {declaration: string-doc} cuerpo)

Donde nombre-función debe ser el nombre de un símbolo atómico, como para una variable, la lambda-lista es el conjunto de argumentos que se pueden pasar (hay varias opciones en las lambda-listas), de las declaraciones y documentación hablaremos un poco más adelante, y cuerpo es cualquier forma a evaluarse en secuencia cuando se invoca al nombre de la función.

DEFINICION DE FUNCIONES.

Ejemplo,

```
>> (defun PRIMERO (L)
                (CAR L) )          ==> PRIMERO

>> (PRIMERO ´(A B C))              ==> A

>> (defun RESTO (L)
                (CDR L) )          ==> RESTO
>> (RESTO ´(A B C))                ==> (B C)
```

El parámetro L de la definición se corresponde con una variable local, es decir, sólo es accesible desde el interior de la función. Si hay una variable externa con igual nombre, la L que hace referencia dentro del cuerpo se refiere a la del argumento.

DEFINICION DE FUNCIONES.

Como decíamos se permite hacer declaraciones e incluir documentación en las funciones. Las **declaraciones** se utilizan para proporcionar información extra al sistema. A menudo son mensajes utilizados por el compilador. Las declaraciones también pueden aparecer al principio del cuerpo de ciertas formas como en las funciones. Los **string de documentación** se utilizan para describir comandos y pueden imprimirse al invocar la función de documentación de la función. Ejemplo:

```
>> (defun CUADRADO (x)
     "Esta función devuelve el cuadrado de un número"
     (* x x) ; fin de CUADRADO
           )                          ==> CUADRADO
```

Esta función utiliza un string de documentación y un comentario justo antes del último paréntesis de la función.

```
>> (CUADRADO 3)                      ==> 9
```

DEFINICION DE FUNCIONES.

¿Qué sucede al definir la función CUADRADO?

- Lisp crea un símbolo CUADRADO
- incorpora en la *definición de función* el valor correspondiente a dicha función.

Y ¿cómo se evalúa y activa una función?

La evaluación de una función, cuando esta se define, consiste en el nombre de la propia función. Por ejemplo, al definir en LISP la función CUADRADO, devuelve como valor de la misma **CUADRADO**.

```
>> (defun CUADRADO (x)
        (* x x) )                    ==> CUADRADO
```

DEFINICION DE FUNCIONES.

Por otro lado, cuando se llama a una función con ciertos argumentos, el valor que devuelve dicha función se corresponde con la última forma ejecutada en el cuerpo de DEFUN. En la función CUADRADO, corresponde con la forma (* x x)

Ejemplos

```
>> (cuadrado 3)                    ==> 9
>> (cuadrado 2)                    ==> 4
>> (cuadrado (cuadrado 3))         ==> 81
>> (cuadrado (+ -4 5 (/ 3 1)))     ==> 16
```

DEFINICION DE FUNCIONES.

La definición de una función puede incluir tantos argumentos como se estimen necesarios. Por ejemplo:

```
>> (defun SUMA-CUADRADO (x y)
                    (+ (cuadrado x) (cuadrado y)) )
            ==> SUMA-CUADRADO
>> (suma-cuadrado 2 3) ==> 13
```

Se pueden definir también funciones sin parámetros:

```
>> (defun SALUDO ()
                    (print "hola a todos") ) ==> SALUDO
>> (saludo)              ==> "hola a todos"
"hola a todos"
```

La respuesta de LISP al llamar a la función SALUDOS, es mediante dos strings que corresponderán:

El primer string a la acción del comando PRINT,

El segundo es el valor devuelto por la función saludo.

DEFINICION DE FUNCIONES.

La definición general de una función con nombre es:

(defun nomb (p1 ... pn) cuerpo)

donde (p1 ... pn) = ({var}*

[&optional {var / (var [inic]) / (var [inic] [flag])}*]
[&rest var]
[&key {var / (var [inic])}*]
[&aux {var / (var [inic])}*)

Los usos y significados de cada una de estas opciones son las siguientes.:

DEFINICION DE FUNCIONES.

&OPTIONAL {VAR / (VAR INIC) / (VAR INIC FLAG)}*. Los parámetros opcionales no precisan argumentos en la llamada. Los parámetros que no tienen correspondencia en la llamada se identificarán con NIL o con el valor que se indique en la inicialización. Las variables **flag** pueden utilizarse para indicar si un parámetro opcional se ha pasado como argumento en la llamada.

>> (defun ejemplo-opcional (a &optional (b 2 c) (d 3))
(list a b c d)) ==> ejemplo-opcional
>> (ejemplo-opcional 5) ==> (5 2 NIL 3)
>> (ejemplo-opcional 5 6) ==> (5 6 T 3)

Los elementos dados como valores de inicialización se evalúan. Por ejemplo, cambiando el último parámetro opcional d,

>> (defun ejemplo-opcional (a &optional (b 2 c) (d inicial-d))
(list a b c d)) ==> ejemplo-opcional
>> (setq inicial-d 4) ==> 4
>> (ejemplo-opcional 5) ==> (5 2 NIL 4)

DEFINICION DE FUNCIONES.

&REST VAR.

Sigue a los parámetros opcionales y sólo tiene un parámetro. En la llamada a la función primero se identifican los argumentos necesarios y luego los opcionales, cualquier otro argumento se agrupa en una lista que se identifica con VAR.

Por tanto VAR se identificará con NIL si no hay más argumentos.

Por ejemplo

>> (defun ejemplo-opcional-rest (a &optional (b 2) &rest x)
(list a b x)) ==> ejemplo-opcional-rest

>> (ejemplo-opcional-rest 5 6) ==> (5 6 NIL)

>> (ejemplo-opcional-rest 5 6 7 8 9) ==> (5 6 (7 8 9))

DEFINICION DE FUNCIONES.

&KEY {VAR / (VAR INIC)}*.

La lista de claves son variables especificadas después de &KEY y se les puede dar valores de inicialización. La forma de acceder a las claves en las llamadas de la función es a través de los ":" seguidos del nombre de la clave y en su caso de una forma para darle un valor. Cualquier clave a la que no se hace referencia en la llamada se identifica con NIL.

```
>> (defun ejemplo-op-rest-key (a &optional (b 2) &key c (d 2))
        (list a b c d) )                    ==> ejemplo-op-rest-key
>> (ejemplo-op-rest-key 5 6 :c 10)    ==> (5 6 10 2 )
>> (ejemplo-op-rest-key 5 6 :d 10)    ==> (5 6 nil 10 )
>> (ejemplo-op-rest-key :d 15)        ==> (:d 15 nil 2)
```

La combinación de opciones &key y &rest en la misma definición de función, da problemas con algunas llamadas.

DEFINICION DE FUNCIONES.

&AUX {VAR / (VAR INIC) / (VAR INIC FLAG)}*.
No son parámetros de la función, y por tanto no se aceptan como argumentos de la llamada a la función.
La opción &AUX se utiliza para establecer variables locales de la función a las que se puede asignar un valor inicial.

```
>> (defun eje-op-aux (a &optional (b 2 c) (d 3) &aux (z 3) (u 5))
                     (list a b c d z u) )
```

>> (eje-op-aux 5 6 7) → **(5 6 T 7 3 5)**

>> (eje-op-aux 5 6 7 8) → **Error: too many arguments**

FUNCIONES ESPECIALES SIN NOMBRE O FUNCIONES LAMBDA.

En algunas ocasiones, y lo veremos más adelante, es deseable crear funciones sin nombre. Estas funciones se conocen con el nombre de lambda-expresiones y tienen una estructura semejante a las de las funciones.

((lambda (p1 ... pn) cuerpo) ‘(a1 … an))

Además, el comportamiento es semejante al de la función.
Ejemplos,

```
>> ((lambda (x y)
(cons (car x) (cdr y)) )
'(a b) '(c d))                    → (a d)

>> (funcall (lambda (a b) (* a b)) 4 8)    → 32
```

LIGADURA DE PARAMETROS EN LA FUNCION.

Las variables creadas en la lista de argumentos de una DEFUN (parámetros formales) son normalmente locales para la función. La lista de parámetros organiza un espacio en memoria para el ámbito de definición de la función, que será utilizado por los valores que se pasen a dicha función en la llamada.

```
>> (defun simple (string)
        string )                    ==> SIMPLE
>> (simple "vale")                  ==> "vale"
>> string                           ==> ERROR, UNBOUND VARIABLE
```

*Esta variable sólo tiene sentido dentro de la función simple, no tiene acceso desde fuera y por tanto si pretendemos tener información de dicha variable nos dará un **ERROR.***

LIGADURA DE PARAMETROS EN LA FUNCION.

La lista de parámetros apunta a los mismos valores que los correspondientes argumentos en la expresión de llamada. Cuando se termina la ejecución de la función se desligan los valores de la lista de parámetros. Las variables de la lista de parámetros no afectan a las variables que se encuentran fuera del procedimiento. Veamos el efecto del ejemplo anterior.

Las variables ligadas, de la lista de argumentos de una función, no pueden ser accedidas por otras funciones llamadas en ejecución desde la misma (a no ser que estos valores se pasen como argumentos de la nueva función llamada).

LIGADURA DE PARAMETROS EN LA FUNCION.

Un ejemplo:

```
>> (defun menos-simple (string)
        (simple) )                          ==> MENOS-SIMPLE
>> (defun simple ()
        string )                            ==> SIMPLE
>> (menos-simple "algo")                    ==> ERROR, UNBOUND
                                                 VARIABLE
```

Al hacer una llamada a la función MENOS-SIMPLE con un argumento (asignándole el valor "algo") la llamada interna a la función SIMPLE me dará un **ERROR**. Esta es la norma general, sin embargo puede suceder en algunas implementaciones que la función interna tenga acceso a las ligaduras de la función padre.

FUNCIONES DE ASIGNACION.

La función **SETF**, es utilizada como las SETQ para realizar asignación de valores a variables. Su sintaxis es

(SETF {LUGAR FORM}+).

SETF utiliza la forma lugar para determinar una localización o dirección en la que hay que realizar la asignación. El lugar queda restringido a variables o alguna forma de función aceptable que normalmente evalúa un lugar al que asignar un valor. Tal y como la hacía SETQ, realiza las evaluaciones y asignaciones secuencialmente, devolviendo el valor de la última evaluación realizada. Ejemplos,

>> (SETF NOMBRE "MARIA") → **"MARIA"**

>> (SETF X ´(A B C)) → **(A B C)**

>> (SETF (CAR X) (CAR ´(1 2 3))) → **1**

>> X → **(1 B C)**

FUNCIONES DE ASIGNACION.

La Funcion **Let** permite crear variables locales interiores a una función. Su forma general consta de dos partes ***asignación de variables*** y ***cuerpo*** , y lo podemos expresar:

(LET ({VARIABLE}* / {VARIABLE FORMA}*) {DECLARATIONS}* CUERPO)

La *asignación de variables* es una lista de listas donde cada lista consta de un nombre de variable y una forma a evaluar. Cuando un LET es llamado las formas se evalúan asignando paralelamente valores a todas las variables. Por su parte el *cuerpo* del LET es como el cuerpo de un DEFUN. Cualquier valor previo que tuvieran variables con el mismo nombre se guardarán y se restaurarán al finalizar el LET.

Las operaciones incluidas en el LET son evaluadas en el entorno del LET. Y por otra parte las variables creadas en el LET son locales (al LET) y por tanto no pueden ser accedidas desde el exterior (tal y como sucede con DEFUN).

FUNCIONES DE ASIGNACION. LET vs SET.

Los parámetros del LET se ligan sólo dentro del cuerpo del LET. Las asignaciones de SET dentro de un parámetro LET no crean ni modifican variables globales. Las asignaciones realizadas por el SET en variables libres crearán o cambiarán valores globales.

Veamos algunos ejemplos.

```
>> (SETF X 5)
>> (SETF E 0)
>> (SETF TRIPLE 0)
>> (LET ((D 2) (E 3)
         (CUADRUPLE))
         (SETF DOBLE (* D X))
         (SETF TRIPLE (* E X))
         (SETF CUADRUPLE (* 4 X)) )
<< 20
```

FUNCIONES DE ASIGNACION. LET vs SET.

>> D

<< ERROR - UNBOUND VARIABLE ; variable no ligada

>> E

<< 0

>> DOBLE

<< 10

>> TRIPLE

<< 15

>> CUADRUPLE

<< ERROR - UNBOUND VARIABLE

Los valores de las variables libres pueden modificarse en la ejecución del LET (efecto lateral en la ejecución del LET).

DECLARACIONES.

Las variables, a menos que sean declaradas de otra forma, son de ámbito léxico. Lo que se ha visto hasta ahora con respecto a la ligadura para los argumentos de una función es que sólo pueden verse dentro del texto de dicha función (tienen ámbito léxico).
Las variables pueden declararse como especiales y en estos casos se dice que son de ámbito dinámico. Es decir son variables libres que pueden ser accedidas desde cualquier parte del programa a partir del punto en el que han sido definidas.
Para declarar variables especiales para todas las funciones, sin necesidad de declararlas separadamente en cada una de ellas, usamos:

- DEFVAR y DEFPARAMETER
- DEFCONSTANT

DECLARACIONES.

Esta es la idea de **variable global** utilizada en otros lenguajes de programación. La forma sintácticamente correcta de las mismas es la siguiente:

- **(DEFVAR nom-var [valor])**, declara una variable global de nombre nom-var accesible desde cualquier parte del programa (desde cualquier función).

- **(DEFPARAMETER nom-var [valor])**, tiene las mismas características que DEFVAR, excepto que será usado como parámetro y no como variable.

- **(DEFCONSTANT nom-var valor)**, el valor permanecerá constante en nom-var, si se intenta cambiar su valor dará error.

DECLARACIONES.

Ejemplo.

```
>> (defconstant c 36)
        << c
>> c
        << 36
>> (setf c 40)
        <<Error: C has constant value 36 and cannot be changed
        to 40
```

ESTRUCTURAS CONDICIONALES.

Los condicionales son formas que permiten tomar decisiones. Verifica y ramifica: verifica devolviendo NIL o NO-NIL (como cualquier función), y ramifica es el comando a ejecutarse posteriormente basándose en el resultado de la verificación.
Las condicionales se implementan como macros y formas especiales.
Existen diferentes condicionales en Lisp, igual que en otros lenguajes: IF, CASE, COND, WHEN, UNLESS, TYPECASE.

- **IF** TEST THEN [ELSE]. La forma condicional IF puede expresar opcionalmente su parte THEN. Dos formas sintácticamente correctas serían: (IF FORMA-COND FORMA-THEN) y (IF FORMA-COND FORMA-THEN FORMA-ELSE).

ESTRUCTURAS CONDICIONALES.

```
>> (defun dame-numero (obj)
        (if (numberp obj)
                "Es un número. Dame más"
                "Esto no es un número") )
>> DAME-NUMERO
>> (dame-numero 4)          → "Es un número. Dame más"
>> (dame-numero 'a) → "Esto no es un número"
```

• **COND** {(TEST {FORMA}+) }+. Permite expresar varias condiciones, ejecutándose la primera que se cumple. La forma COND acepta cualquier número de listas cuyos elementos son formas. Estas listas están formadas normalmente por un test seguido de cero o más formas consecuentes. Estas formas se ejecutaran si y solo si el test devuelve un valor NO-NIL.

ESTRUCTURAS CONDICIONALES.

La cabeza de las listas se evalúa secuencialmente hasta encontrar una que devuelva un valor NO-NIL, entonces las formas siguientes a la cabeza de esa lista se evalúan y se sale de la forma COND. Si hubiera listas más adelante cuyas condiciones (cabeza de lista) fueran también ciertas éstas no se evalúan y por tanto tampoco se ejecutan las formas incluidas en el resto. COND devolverá el resultado de la última forma evaluada de la lista con un test NO-NIL. Si todos los test devuelven NIL, entonces COND devolverá NIL. El ejemplo general para la sintaxis sería:

(COND (TEST1 FORMA11 FORMA12 … FORMA1N)

(TEST2) ; no tiene ningún consecuente y se devolverá el valor NO-NIL del test

(TEST3 FORMA31 FORMA32 … FORMA 3M)

…

(TESTR …))

ESTRUCTURAS CONDICIONALES.

```
>> (setq a 4)                                    ==> 4
>> (cond        ((numberp a) "Esto es un número")
                (t "Esto no es un número") )
==> "Esto es un número.
==> mi-función

>> (defun verificar (x)
        (cond ((< x 0) (print "NEGATIVO"))
                ((= x 0) (print "CERO"))
                ((<= X 10) (print "VALOR EN RANGO"))
                (T (print "VALOR MUY GRANDE") x)
        ) ;fin-cond
    ) ;fin-mi-función
==> mi-función
```

ESTRUCTURAS CONDICIONALES.

WHEN TEST FORMA1 ... FORMAN. Es equivalente a la forma IF sin opción de incluir la forma else.

Ejemplo,

>> (setq a 4)

>> (when (equal a 4) (print "cierto")) ==> "cierto"

"cierto"

>> (when (equal a 3) (print "cierto")) ==> nil

UNLESS TEST FORMA1 ... FORMAN. Esta forma condicional difiere de las vistas hasta ahora en que se evaluarán las formas siempre que el resultado de la evaluación del test sea NIL. Hasta ahora la evaluación de las formas asociados a un test estaba condicionada a que éstas dieran como resultado un valor NO-NIL. Ejemplos,

>> (unless (equal a 4) (print "cierto")) ==> nil

>> (unless (equal a 3) (print "cierto")) ==> "cierto"

"cierto"

ESTRUCTURAS CONDICIONALES.

CASE KEYFORM {(LIST { FORM}+)}+.
Permite realizar ciertas acciones cuando una determinada forma tiene ciertos valores, expresados a través de una lista.
Por ejemplo,

```
>> (setq mes 'may)
>> may
>> (case mes
        ((ene mar may jul ag oct dic) 31)
        ((ab jun sep nov) 30)
        (feb (if (zerop (mod año 4)) 29 28) ) )
==> 31
```

ESTRUCTURAS CONDICIONALES.

TYPECASE KEYFORM {(TYPE {FORMA}+)}+. Permite como en el caso anterior realizar una serie de acciones siempre que un determinado objeto sea de la clase indicada.

Ejemplo,

```
>> (SETQ X 2) ==> 2
>> (TYPECASE X
        (string "es un string")
        (integer "es un entero")
        (symbol (print '(es un símbolo)))
        (otherwise (print "operador")(print "desconocido") ) )
==> "es un entero"
"es un entero "
```

ESTRUCTURAS ITERATIVAS.

• **LOOP** {FORMA}*. Expresa una construcción iterativa que cicla continuamente, a menos que se le indique explícitamente que pare, evaluando las formas en secuencia. La forma de indicar la terminación del ciclo es mediante RETURN. En general la evaluación de la sentencia RETURN provocará la salida de cualquier sentencia iterativa.

Ejemplo,

```
(loop
        (format t "escribe tu nombre, por favor.")
        (setq nombre (read))
        (format t "escribe tu apellido.")
        (setq apellido (read))
        (format t "¿es correcta la información ?")
        (setq s/n (read))
        (if (equal s/n ‘s) (return (cons nombre apellido) ) )
) ;loop
```

ESTRUCTURAS ITERATIVAS.

- **DO** ({PARAM}* / {(PARAM VALOR)}* / {(PARAM VALOR INCRE)}*) (TEST-EVALUACION {FORMAS}*) {FORMAS}*.

La estructura DO tiene tres partes: ***lista de parámetros, test de final*** y ***cuerpo***. La lista de parámetros liga las variables con sus valores iniciales y en cada ciclo se asignan nuevos valores dependiendo de la función de incremento indicada. En la estructura DO, primero se activa la ligadura de los parámetros y después se evalúa el test de final, y si no se cumple se ejecuta el cuerpo.
Esta estructura podemos encontrarle semejanza con la de while ... do utilizada en otros lenguajes.

```
(DO ((var1 inic1 [paso1])...
(varn inicn [pason]))          ; Asignación de variables en paralelo
(test-final resultado)         ; Test-final se evalúa antes que cuerpo
        declaraciones
        cuerpo)
```

ESTRUCTURAS ITERATIVAS.

Ejemplos:

```
>> (defun exponencial (m n)
        (do     ((resultado 1)
                        (exponente n))
                ((zerop exponente) resultado)
        (setq resultado (* m resultado))
        (setq exponente (1- exponente)) ) )
==> EXPONENCIAL

>> (defun nueva-exp (m n)
        (do     ((resultado 1 (* m resultado))
                (exponente n (1- exponente)))
                ((zerop exponente) resultado) ) )
==> NUEVA-EXP
```

ESTRUCTURAS ITERATIVAS.

Ejemplos:

```
>> (defun nuevo-reverse (lista)
        (do     ((l lista (cdr l))
                 (resultado nil (cons (car l) resultado)))
                 ((null l) resultado) ) )
==> NUEVO-REVERSE
```

DO* … A diferencia del anterior realiza la asignación de variables secuencialmente.

```
>> (defun exponente (m n)
        (do*    ((resultado m (* m resultado))
                 (exponente n (1- exponente))
                 (contador (1- exponente) (1- exponente)) ) ;fin-pa
                 ((zerop contador) resultado) ) ;fin-do ) ;fin-defun
==> EXPONENTE
```

ESTRUCTURAS ITERATIVAS.

DOTIMES permite escribir procedimientos sencillos con iteración controlada por un contador. Ejecuta el cuerpo un número de veces determinado.

(DOTIMES (var forma-limite-superior [forma-resultado])
declaraciones
cuerpo)

Cuando comienza el ciclo se evalúa la forma límite superior, produciendo un valor n. Entonces desde el valor 0 incrementándose en uno hasta n-1, se asigna a la variable var.

Para cada valor de la variable se ejecuta el cuerpo y al final, la ligadura con la variable se elimina y se ejecuta la forma resultado dando valor al DOTIMES. Si no tiene forma-resultado, su propósito es realizar efectos secundarios.

Ejemplo:

ESTRUCTURAS ITERATIVAS.

```
>> (dotimes (cont 4) (print cont))
0
1
2
3
NIL

>> (defun exponencial-con-times (m n)
        (let ((resultado 1))
        (dotimes (cuenta n resultado)
        (setf resultado (* m resultado) ) ) ) )
==> exponencial-con-times

>> (exponencial-con-times 4 3)
```

➔ 64

ESTRUCTURAS ITERATIVAS.

DOLIST repite el cuerpo para cada elemento de una lista y como en el caso anterior al finalizar evaluará la forma resultado, siendo ésta el valor de DOLIST.

```
(DOLIST (var lista [resultado])
                declaraciones
                Cuerpo)
>> (dolist (x '(a b c))
        (print x) )
A
B
C
NIL
```

FUNCIONES ITERATIVAS PREDEFINIDAS.

Son precisamente en este tipo de funciones predefinidas, donde pueden tener sentido la definición de funciones sin nombre o lambda expresiones (como párametro).

(MAPCAR función lista1 ... listan)
Va aplicando la función a los sucesivos car's de las listas.
La función debe tener tantos argumentos como lista tiene el mapcar.
Devuelve una lista con los resultados de las llamadas a la función.
Utiliza LIST para construir el resultado.

>> (mapcar #´+ ´(7 8 9) ´(1 2 3)) → **(8 10 12)**
>> (mapcar #´oddp ´(7 8 9)) → **(T NIL T)**

FUNCIONES ITERATIVAS PREDEFINIDAS.

(MAPLIST función lista1 ... listan)

Se aplica a los CDR´s sucesivos de las listas.
Devuelve una lista con los resultados de las llamadas a la función.
Utiliza LIST para construir el resultado.

>> (maplist #´+ ´(7 8 9) ´(1 2 3)) ➔ **((8 10 12) (10 12) (12))**

>> (maplist #´(lambda (x) (cons ´123 x)) ´(a b c))

==> ((123 a b c) (123 b c) (123 c))

FUNCIONES ITERATIVAS PREDEFINIDAS.

Ejemplos:

```
>> (setq x '(1 2 3))
<< (1 2 3)
>> (mapcar #'(lambda (arg) (print (list arg arg))) x)
(1 1)
(2 2)
(3 3)
<< ((1 1) (2 2) (3 3))
>> x
<< (1 2 3)
>> (maplist #'(lambda (arg) (print (list arg arg))) x)
((1 2 3) (1 2 3))
((2 3) (2 3))
((3) (3))
<< (((1 2 3) (1 2 3)) ((2 3) (2 3)) ((3) (3)))
```

ENTRADA/SALIDA SIMPLE .

Lisp tiene primitivas de funciones de entrad/salida. Las más comúnmente utilizadas son: READ, PRINT y FORMAT, aunque también veremos otras. Los dispositivos por defecto son, el de entrada el teclado y salida la pantalla, pero se puede redireccionar a otros dispositivos. Es decir, todas estas funciones toman un argumento opcional llamado input-stream o output-stream. Si este valor no es suministrado o es nil, se tomará por defecto el valor contenido en la variable *standard-input* que será teclado o pantalla respectivamente. Si el valor es t se usará el valor de la variable *terminal-io*.

ENTRADA/SALIDA SIMPLE .

Función de entrada.

- **READ**, lee un objeto Lisp del teclado y devuelve dicho objeto. Esta función detiene la ejecución del programa mientras no se termine con un RETURN. Sintácticamente tendríamos
- **(READ &optional stream)**

Ejemplos:

>> (read)
>> 35
<< 35
>> (setq a (read))
>> HOLA
<< HOLA
>> a
<< Hola

ENTRADA/SALIDA SIMPLE .

Funciones de salida.

- **PRINT**, toma un objeto Lisp como argumento y lo escribe en una nueva línea con un blanco por detrás (introduce <RT> y blanco después del objeto). Escribe los objetos por su tipo, tal y como serían aceptados por un READ ->
- **(PRINT objeto &optional stream).**

Ejemplos:

```
(print 'a) ;        imprime A con #\Newline
(print '(a b)) ;    imprime (A B) con #\Newline
(print 99) ;        imprime 99 con #\Newline
(print "hi") ;      imprime "hi" con #\Newline
```

ENTRADA/SALIDA SIMPLE .

Funciones de salida.

PRIN1, toma un objeto Lisp como argumento y lo escribe con un blanco por detrás. Escribe los objetos por su tipo, tal y como serían aceptados por un READ, sin salto de linea ->

(PRIN1 objeto &optional stream).

Ejemplos:

```
(prin1 'a) ;       imprime A sin #\Newline
(prin1 '(a b)) ;   imprime (A B) sin #\Newline
(prin1 2.5) ;      imprime 2.5 sin #\Newline
(prin1 "hi") ;     imprime "hi" sin #\Newline
```

ENTRADA/SALIDA SIMPLE .

Funciones de salida.

PRINC, toma un objeto Lisp como argumento y lo escribe con un blanco por detrás, y en el caso de los string, imprime sin las comillas. No escribe los objetos por su tipo.

(PRINC objeto &optional stream).

Ejemplos:

```
(princ 'a) ;        imprime A sin #\Newline
(princ '(a b)) ;    imprime (A B) sin #\Newline
(princ 99) ;        imprime 99 sin #\Newline
(princ "hi") ;      imprime hi sin #\Newline
```

ENTRADA/SALIDA SIMPLE .

Funciones de salida.

PPRINT, toma un objeto Lisp como argumento y lo escribe con un blanco por detrás, introduce solo un Return. No realiza la segunda evaluacion.

(PPRINT objeto &optional stream).

Ejemplos:

(pprint 'a) ; imprime A returns NIL
(pprint "abcd") ; imprime "abcd" returns NIL

ENTRADA/SALIDA SIMPLE .

Funciones de salida.

FORMAT, aparece como un mecanismo más generalizado de dar la salida. Se indican directivas que son como variables situadas entre el string y comienzan con ~. Muchas directivas miran el siguiente argumento de la lista del format y lo procesan de acuerdo a sus propias reglas. Su forma es

(FORMAT destino control-string &rest argumentos)

- destino = nil/ t/ stream. Si se indica el nil, se creará un string con las características indicadas y éste será lo que devuelva el format. Si se indica t, será la salida por defecto y sino será otro dispositivo indicado en stream.

- control-string = contiene el texto a escribir y "directivas"

- argumentos = lista con los parámetros para las "directivas"

ENTRADA/SALIDA SIMPLE .

Funciones de salida.

Se usan diferentes directrices para procesar diferentes tipos de datos e incluirlos en el string:

~A - Imprime un objeto cualquiera.,

~D - Imprime un número en base 10

~S - Imprime una expresión simbólica.

Además se puede utilizar el símbolo @ junto con la directiva ~A, para justificar a la derecha valores numéricos. No todas la directrices utilizan argumentos: ~

% - Inserta una nueva línea y

~| - Nueva página.

Ejemplos,

>> (FORMAT T "STRING DE SALIDA")

==> STRING DE SALIDA

==> NIL

ENTRADA/SALIDA SIMPLE .

>> (SETQ VAR 5)

==> 5

>> (FORMAT T "STRING QUE INCLUYE ~A" VAR)

==>STRING QUE INCLUYE 5

==> NIL

>> (FORMAT NIL "STRING QUE INCLUYE ~A" VAR)

==> STRING QUE INCUYE 5

>> (FORMAT T "ESCRIBE EL ARGUMENTO ~10@A JUSTIFICADO A LA DERECHA CON 10 ESPACIOS." 1000)

==> ESCRIBE EL ARGUMENTO 1000 JUSTIFICADO A LA DERECHA CON 10 ESPACIOS.

==> NIL.

>> (FORMAT T "~%LA PRIMERA RESPUESTA ES ~5@A ~%Y LA SEGUNDA RESPUESTA ES ~3@A " (* 5 3 2) (/ 3 5))

==> LA PRIMERA RESPUESTA ES 30

==> Y LA SEGUNDA RESPUESTA ES 0.6

==> NIL

ENTRADA/SALIDA SIMPLE .

TERPRI realiza un salto de línea
(TERPRI &optional stream)

Ejemplos de salida:
>> (princ "hola") ==> hola
"hola"
>> (print "hola") ==> "hola"
"hola"
>> (prin1 "hola") ==> "hola"
"hola"
>> (format t "hola") ==> hola
NIL
>> (format nil "hola") ==> "hola"
>> (format nil "hola soy ~a, ~a" "yo" "Bonn")
==> "hola soy yo, Bonn"

RECURSIVIDAD EN LISP.

Se dice que un proceso es recursivo cuando esta basado en su propia definición. En programación, la recursividad, no es una estructura de datos, sino una técnica de programación que permite que un bloque de instrucciones se ejecute una n cantidad de veces. En muchas ocasiones reemplaza a las estructuras repetitivas.
En LISP, es común que las funciones se llamen a si mismas en el cuerpo de la función, esto es lo que se llama una función recursiva.

CARACTERISTICAS DE LA PROGRAMACION RECURSIVA

- Implementación intuitiva y elegante.
- La traducción de la solución recursiva de un problema (caso base y caso recursivo) a código Lisp es prácticamente inmediata.
- Útil cuando hay varios niveles de anidamiento. La solución para un nivel es válida para el resto.
- La interpretación y comprensión del código puede ser compleja.

RECURSIVIDAD EN LISP.

Ejemplos.

1. **Función Factorial (n!)**

```
(defun factorial (n)
        (if (= n 0) 1
                (* n (factorial (- n 1)))
        )
)
```

2. **Función Potencia (m^n)**

```
(defun potencia (x m)
        (if (= m 0) 1
                (* x (potencia x (- m 1)))
        )
)
```

RECURSIVIDAD EN LISP.

Ejemplos.

3. Obtener el valor correspondiente a la posición n de una serie de Fibonacci.

Método Recursivo

```
(defun fib (x)
      (if (< x 2)
          x
          (+ (fib (1- x)) (fib (- x 2)))))
```

Metodo Iterativo

```
(defun fibi (n)
            (do       ((i 1 (1+ i))
                       (fib-i-1 0 fib-i)
                       (fib-i   1 (+ fib-i fib-i-1)))
                      ((= i n) fib-i)))
```

SEGUIMIENTO DE FUNCIONES (Trace)

Cuando se necesita saber exactamente *cómo* una función está trabajando en un punto particular en el código, hacer Break y usar el depurador de Lisp son herramientas indispensables. Pero es trabajoso y lento (por lo menos en relación con la ejecución normal del programa).
A veces, es suficiente saber que una determinada función se ha llamado y devolvió un valor. TRACE nos brinda esta capacidad imprimiendo el nombre de la función y sus argumentos a la entrada, y el nombre de la función y de sus valores a la salida. Todo esto ocurre sin cambiar el código fuente de la función.

(trace [<sym>...])

Ejemplo
(trace factorial fibo potencia)
>> (FACTORIAL FIBO POTENCIA)

La ejecución del trace permanecerá activa hasta que se ejecute la macro UNTRACE. Si se ejecuta sin argumentos, detendrá el traceo de todas las funciones.

(untrace [<sym>...])

SEGUIMIENTO DE FUNCIONES (Trace)

Si se ejecuta la macro trace sin argumentos, mostrara una lista con todas las funciones que estan siendo trazeadas en su ejecución.

```
(trace)
(FACTORIAL FIBO POTENCIA)

>> (fibo 4)

Entering: FIBO, Argument list: (4)
 Entering: FIBO, Argument list: (3)
  Entering: FIBO, Argument list: (2)
  Exiting: FIBO, Value: 1
  Entering: FIBO, Argument list: (1)
  Exiting: FIBO, Value: 1
 Exiting: FIBO, Value: 2
 Entering: FIBO, Argument list: (2)
 Exiting: FIBO, Value: 1
Exiting: FIBO, Value: 3
3
```

STREAMS

Es un tipo de datos que mantiene la información sobre el dispositivo (fichero) al que está asociado. Hará que las E/S sean independientes del dispositivo.
Un stream puede estar conectado a un fichero o a un terminal interactivo.

OPEN Esta función devuelve un stream.

(OPEN nomfich &key :direction)

donde nomfich será un string/ pathname/ stream;
:direction permitirá señalar si el dispositivo va a ser de lectura-entrada, :input, salida-escritura, :output, o ambos, :io.

STREAMS

Ejemplo.

(setq f (open 'mine :direction :output)) ; crea un archivo llamado MINE

(print "hola" f) ;	retorna "hola"
(print "Segundo hola" f) ;	retorna "Segundo hola"
(close f) ;	archiva "hola" <newline>
(setq f (open 'mine :direction	:input)) ; abre el archive comor input
(read f) ;	lee y devuelve "hola"
(read f) ;	lee y devuelve "Segundo hola"
(close f) ;	cierra el archivo

STREAMS

Algunos predicados con streams serían:

- (**STREAMP** Objeto), dará T si objeto es un stream
- (**OUTPUT-STREAM-P** stream), dará T si stream es de salida.
- (**INPUT-STREAM-P** stream), será T si stream es de entrada.

CLOSE, Cierra un stream, desaparece la ligadura entre el fichero y el stream (en caso de estar ligado a él).

(CLOSE stream)

Algunos streams estandar

standar-input

standar-output

error-output

terminal-io

Funciones de Entrada de STREAMS

READ-CHAR lee un carácter y lo devuelve como un objeto de tipo carácter. Su forma sintácticamente correcta es, (READ-CHAR &optional stream)

READ-LINE lee caracteres hasta encontrar un carácter #\newline. Y devuelve la línea como un string de caracteres sin #\newline -> (READ-LINE &optional input-stream)

PEEK-CHAR lee un carácter del stream de entrada sin eliminarlo del stream. La próxima vez que se acceda para leer se leerá el mismo carácter -> (PEEK-CHAR &optional peek-type input-stream) Si peek-type es **t** se saltarán los caracteres en blanco. Si su valor es **nil** no.

Funciones de Manipulación de STREAMS

(**RENAME-FILE** fichero nuevo-nombre), cambia de nombre un fichero. Si fichero es un nombre de stream, el stream y el fichero a él asociado resultan afectados.

(**DELETE-FILE** fichero), borra un fichero. Si fichero es un nombre de stream, el stream y el fichero a él asociado resultan afectados.

(**FILE-LENGTH** stream-de-fichero), devuelve la longitud de un fichero. Si esta no puede ser determinada devolverá nil.

(**LOAD** fichero &key :verbose :print :if-does-not-exist), carga un fichero al interprete Lisp. A partir de ese momento el entorno Lisp contará con los objetos Lisp definidos en ese fichero.

Funciones de Manipulación de STREAMS

Las claves incluidas en la forma LOAD corresponden con:

:verbose Si es t permitirá al cargar el fichero, sacar un comentario sobre el *standar-output* indicando las características del fichero que está siendo cargado.

:print Si es t se saca un valor de cada expresión cargada sobre *standar-output*.

:if-does-not-exist Si es nil devuelve nil;
si es t el error correspondiente en caso de que lo haya.

DOCUMENTACION

Se puede hacer comentarios en cualquier lugar del programa siempre que se utilice el ";", de tal forma que después de la coma comienza el comentario. Por convenio se utilizará:

;;;; 4 ";" y un blanco, al comienzo del fichero para indicar su contenido.

;;; 3 ";" y un blanco, para realizar comentarios sobre una función y aparecerá en la línea posterior a la de definición de función (cabecera).

;; 2 ";" y un blanco, se utiliza para comentar una línea de código. En la línea inmediatamente seguida a esta.

;sólo un ; y sin blanco, se utiliza para explicar una línea de código en la misma línea.

DOCUMENTACION

Ejemplo de un fichero comentado:

```
;;;;
**************************************************
;;;; Fichero: Ejemplo-comentario.lisp
;;;; Fecha-de-creación: 20/02/2014
;;;; Modulo: Apuntes de lisp.
;;;; Comentarios: Este fichero ha sido creado con fines didácticos.
;;;; Autores: Los profesores de Paradigmas
;;;; UNNE
;;;;
**************************************************
```

DOCUMENTACION

Ejemplo de un fichero comentado:

```
(defun pareja (x)
;;; Esta función devuelve la pareja del personaje que se pasa como valor del parámetro x.
(case x
;; se realiza la selección.
((Peter-pan) "Campanilla") ;Campanilla es una chica que vuela.
((Mortadelo) "Filemón") ;Ambos son agentes de la T.I.A.
( t "no tiene pareja")
) ;Fin de selección
) ;Fin de la función.
;;; Esperamos que haya encontrado su pareja con esta función. Si no modifíquela siempre que lo crea necesario.
;;;; Fin del fichero.
```