

# INTRODUCCIÓN A COMMON LISP

**LAURA DE MIGUEL**

**JAVIER FERNANDEZ**

**Dpto. de Estadística, Informática y Matemáticas**

[laura.demiguel@unavarra.es](mailto:laura.demiguel@unavarra.es)

# Instalación de Common Lisp

- LISt Processing
- John McCarthy, 1958
- Programación funcional (Lambda calculus)
- Procesamiento Simbólico
- Common Lisp: Clisp, GCL, CMUCL, Allegro. . .
- Inteligencia Artificial

En Lisp todos son objetos que se representan mediante símbolos.

Cada símbolo tiene asociado 5 componentes que lo caracterizan

Símbolo
Nombre Imprimible
Valor
Definición de función
Lista de propiedades
Paquete

Los tipos de valores que puede tomar un símbolo son:

- **Atom**: números, caracteres, *strings*, otros símbolos
- **Cons**: estructuras que forman las listas
- Una **estructura** definida por el usuario
- **Tipos de datos** propios del usuario

## Atom : las estructuras más básicas de Lisp

En general, un átomo se apunta a sí mismo

- **Números**: se apunta a sí mismo.
  - **Enteros**: representan números enteros

No existe límite de magnitud

Símbolo	
Nombre Imprimible	-2
Valor	-2
Definición de función	
Lista de propiedades	
Paquete	

- Racionales:  $-2/3$ ,  $4/6$ ,  $20/2$ ,  $5465789/43$

Notación::= [signo]  $\{d\}^+ / \{d\}^+$

- Reales (coma flotante):  $0.0$ ,  $-0.5$

Notación::=[signo]  $\{d\}^* . \{d\}$

- Complejos:  $\#C(\text{parte-real parte-imaginaria})$

## Atom : las estructuras más básicas de Lisp

En general, un átomo se apunta a sí mismo

- Caracteres: `#\a` `#\A`
- Caracteres no imprimibles: `#\SPACE`  
`#\NEWLINE`



**Variables:** El atributo valor del símbolo permite que dicho símbolo actúe como una variable. Todo nombre de variable debe ser un símbolo.

En general, LISP intentará aplicar la regla de devolver el valor del símbolo

Constantes especiales

**T** - true

**NIL** - false o lista vacía ()

## CONS type

Lista: conjunto ordenado de elementos separados por espacios.  
Empieza con ( y termina con ).

En general, una lista es una estructura básica de datos dentro del programa Lisp.

Una lista es un conjunto ordenados de elementos de cualquier tipo (incluso de elementos NIL o ()).

Existe también la lista de 0 elementos, que corresponde a una estructura NIL.

(1 2 3 4); lista de números enteros

(a b c d); lista de símbolos

(#\a #\b #\c #\d); lista de caracteres

(4 algo “si”); lista de átomos

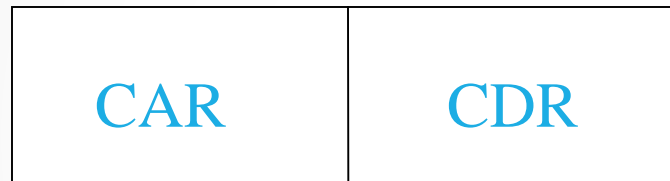
(sqrt 16)

(+ 2 5)

(- 5 3 7)

Todos los programas en LISP se describen mediante funciones que se definen y llaman mediante listas.

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 que está formada por dos partes, **CAR** y **CDR**.

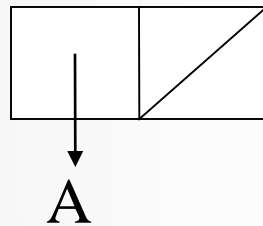


El **CAR** es la primera parte de la estructura y contiene un puntero al primer elemento de la lista. El **CDR** es la segunda parte de la estructura y contiene un puntero a la siguiente estructura **CONS** o si no hay más elementos, apunta a **NIL**.

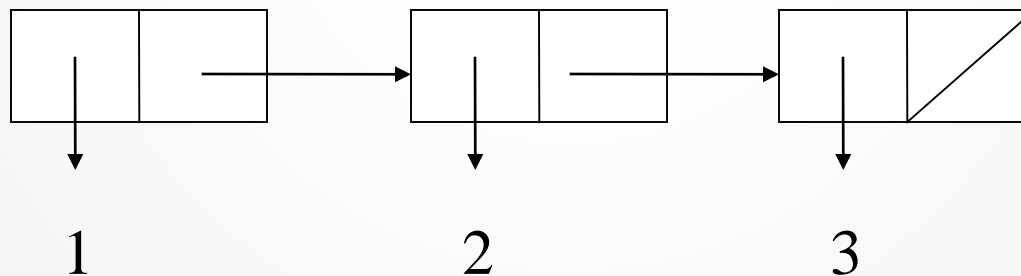
El átomo A no es una estructura CONS

La lista vacía () no tiene estructura CONS

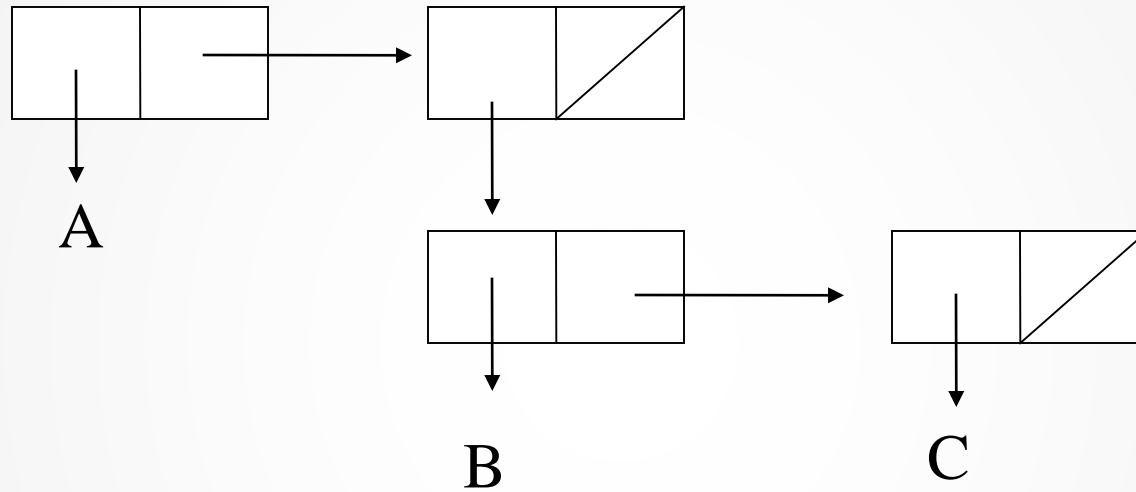
(A)



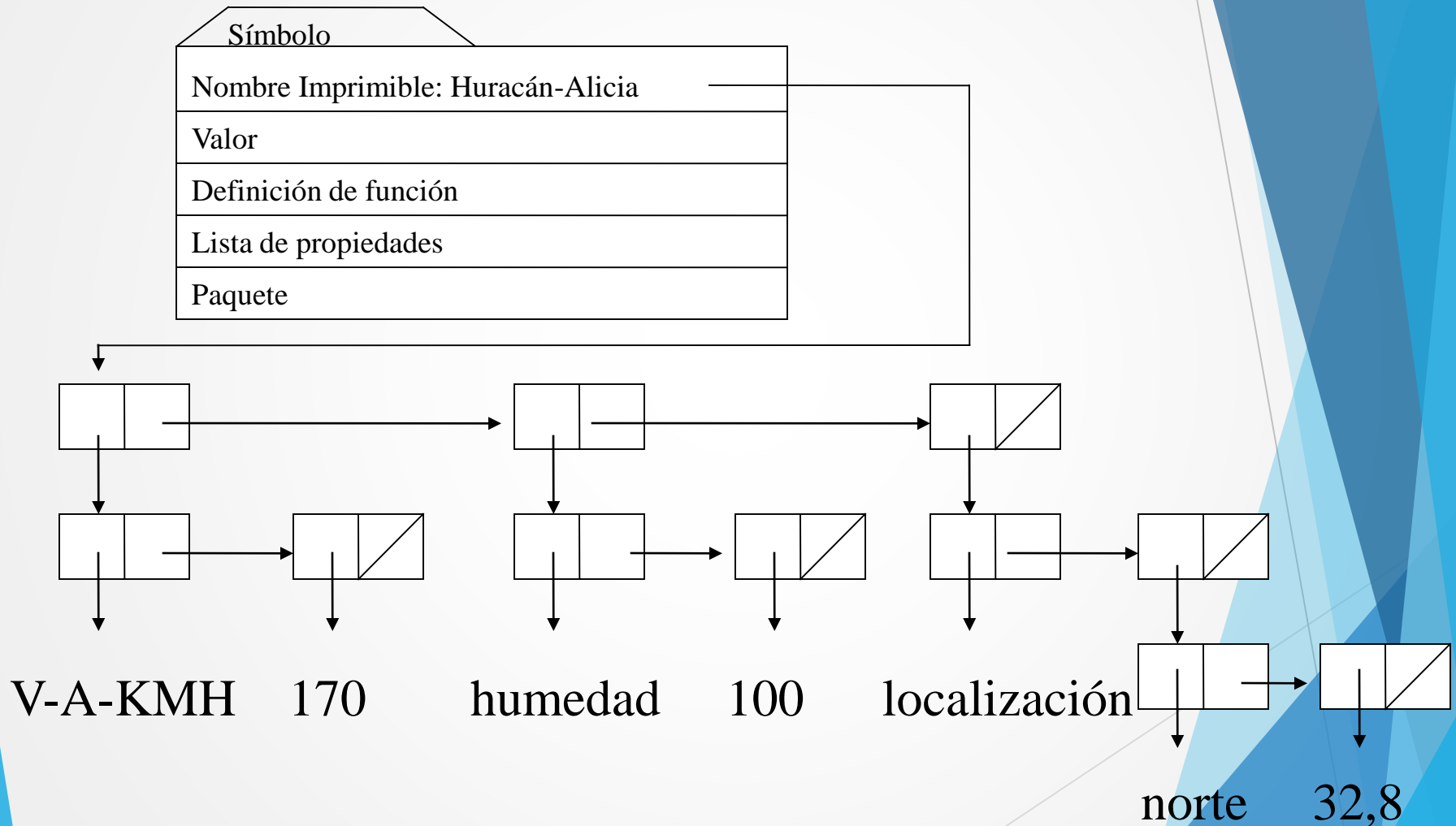
(1 2 3)



(A (B C))

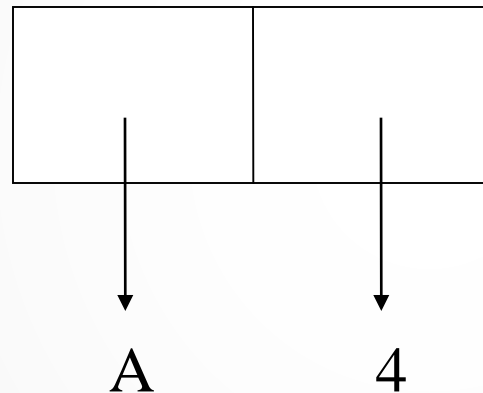


# Huracán-Alicia ((v-a-kmh 170)(humedad 100)(localización (norte 32,8)))



**Listas punteadas.** Cuando el CDR de una estructura CONS apunta a un átomo en lugar de a otra estructura CONS o NIL.

(A.4)



Se libera memoria ya que se ahora una casilla CONS

Se pierde flexibilidad



## Evaluación en LISP

Lisp Listener: ciclo que realiza continuamente Lisp  
(eval form)

### Regla de evaluación básica

Evaluación de los argumentos

De Izquierda a Derecha

Se calculan los valores de las funciones

Todo se evalúa

- Evaluación

```
> (+ (- 5 2) (* 3 3)) <return>  
12
```

## FUNCIONES PREDEFINIDAS SOBRE LISTAS

Expresiones simbólicas. Formas

Función **QUOTE**

La función quote hace que un objeto no sea evaluado

```
> (quote x)
```

**X**

```
> (quote (+ (-5 2) (* 3 3)))
```

```
(+ (- 5 2) (* 3 3))
```

## Función QUOTE

A veces puede ser un poco largo y tedioso estar escribiendo *quote*.

Es por ello que LISP tiene la opción de utilizar el símbolo `'`.

A efectos prácticos `'(A)` es lo mismo que `(quote A)`

```
> (quote (+ (- 5 2) (* 3 3)))
```

```
(+ (- 5 2) (* 3 3))
```

```
> '(+ (- 5 2) (* 3 3))
```

```
(+ (- 5 2) (* 3 3))
```

```
> 'x
```

```
x
```

## Funciones de selección o acceso a los elementos:

**CAR**- devuelve el primer elemento de la lista

**(CAR LISTA)**

> (CAR '(A B C))

A

> (CAR 'A)

ERROR

> (CAR '((A B) C))

(A B)

> (CAR '())

NIL

**CAR=FIRST**

**CDR-** devuelve toda la lista menos el primer elemento

**(CDR LISTA)**

> (CDR '(A B C))  
(BC)

> (CDR '(A B (C D) E))  
(B (CD) E)

> (CDR '((A B) C))  
(C)

> (CDR '((A)))  
NIL

> (CDR 'A)  
ERROR

> (CDR '(A.B))  
B

**CDR=REST**

**C\*\*\*\*R**- permite concatenar funciones CAR y CDR

**(C\*\*\*\*R LISTA)**

> (CADR '(A B C))=(CAR (CDR '(A B C))=(CAR '(B C) )

B

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

C

**LAST** – devuelve la última estructura CONS de la lista  
(LAST LISTA)

> (LAST '(A B C))

(C)

> (LAST '(A (B C)))

((B C))

> (LAST '(1 2 3.5))

(3.5)

>(LAST '())

NIL

**NTH** – devuelve la i-ésima estructura CONS de la lista

**(NTH INT LISTA)** - devuelve la componente int

> (NTH 0 '(A B C))

(A)

> (NTH 3 '(A (B C)))

NIL

> (NTH 1 '(A B C))

(B)

El primer elemento de la lista es considerado el elemento 0

**(NTH 0 LISTA)**



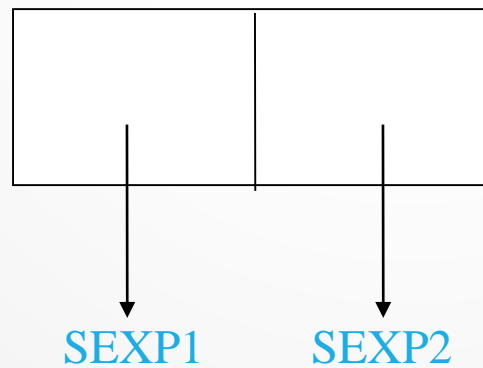
## Funciones de construcción de listas

### Funciones no destructivas

La función CONS crea una estructura CONS

El CAR de la nueva estructura CONS apunta al primer argumento, y el CDR al segundo argumento.

(CONS SEXP1 SEXP2)



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

(A.(B C D))  $\Rightarrow$  (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)

> (SETF a '(1 2))

> (SETF b '(3 4 5))

> (CONS a b)

((1 2) 3 4 5)

> (CONS '(A B) 'C)

((A B).C)

> (CONS 'A 'B)

(A.B)

> (CONS 'NIL 'A)

(NIL.A)

**LIST** - devuelve una lista formada con todos los elementos pasados como argumentos. Los argumentos deberán ser expresiones válidas.

**(LIST SEXP SEXP...)**

> (LIST 'A 'B 'C)

(A B C);

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

((A) (B) (C))

> (LIST 'A '(B C))

(A (B C));

> (LIST (LIST (LIST 'A)))

((A)))

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

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

**APPEND** – concatena los argumentos en una lista. Todos los argumentos, excepto quizás el último, deben ser listas.

El CDR de la última casilla de cada uno de los argumentos apunta al CAR del siguiente argumento.

**(APPEND LISTA LISTA...LISTA SEXP)**

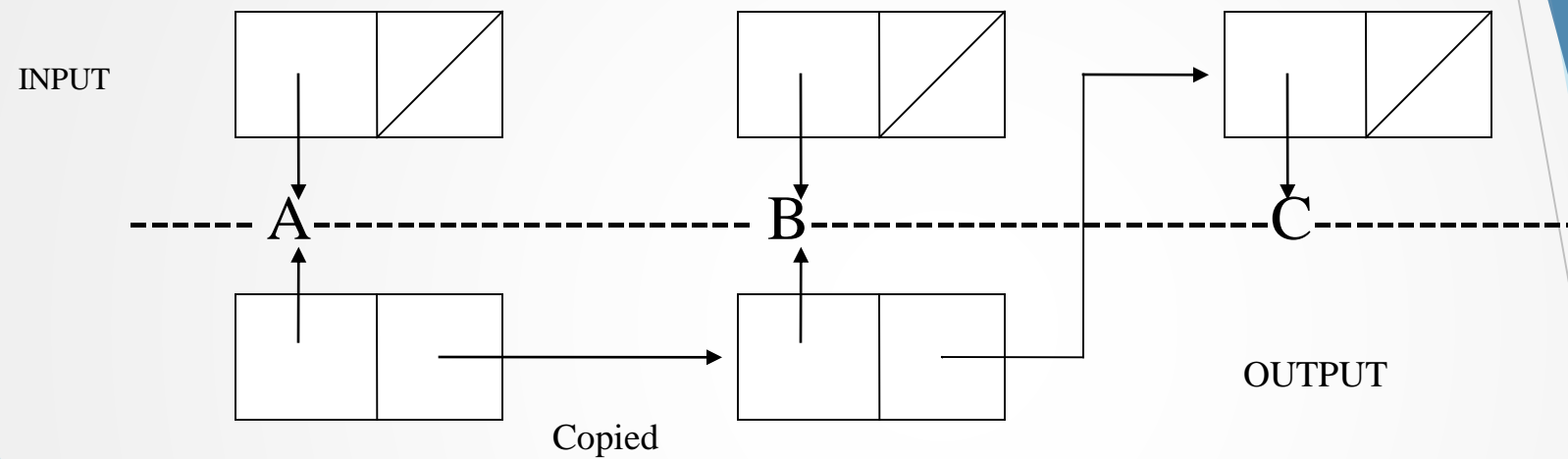
> (APPEND '(A) '(B) '(C))

(A B C)

“Desaparece un nivel o paréntesis”

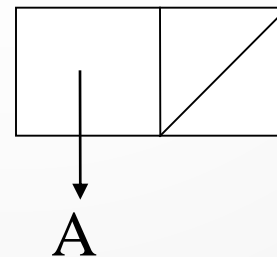
> (APPEND '(A) '(B) '(C))

(A B C)



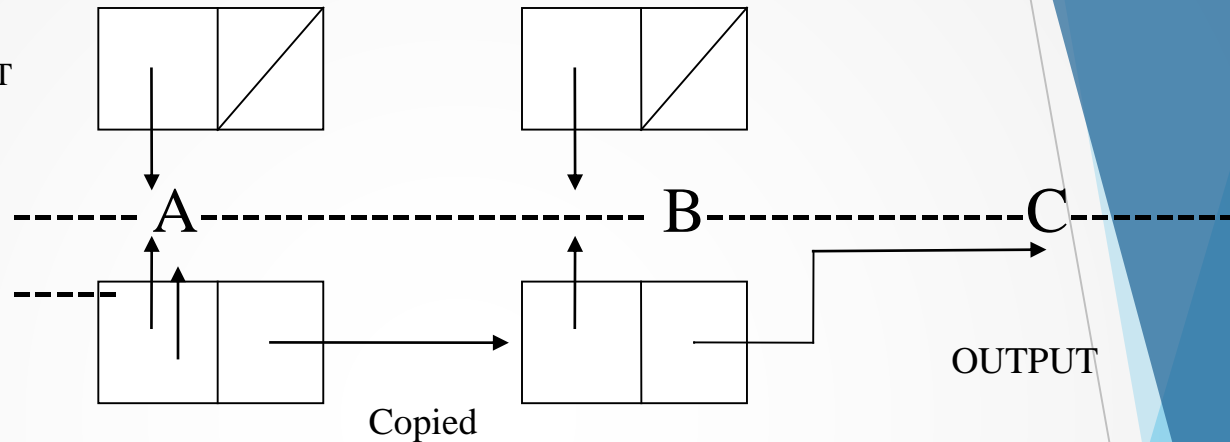
> (APPEND '(A) NIL)

(A)



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

(A B.C) INPUT



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

(A B C)

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

((A) (B) (C))

> (APPEND '(TU PUEDES) '((CREAR (UNA))) '() '(LISTA))

(TU PUEDES (CREAR (UNA) LISTA))

> (APPEND (APPEND (APPEND '(A))))

(A)

> (APPEND '(A) NIL)

(A)

> (APPEND NIL '(B C))

(B C)

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

(A B.C)

> (APPEND 'A '(B C))

ERROR

## Funciones destructivas

En las funciones anteriores los símbolos que representan los parámetros no son modificados.

En las funciones destructivas se modifica el contenido de algún parámetro.

**RPLACA** – reemplaza el CAR de la lista por el elemento dado.

(RPLACA LISTA ELEMENTO)



> (Setf a '(1 2 3))

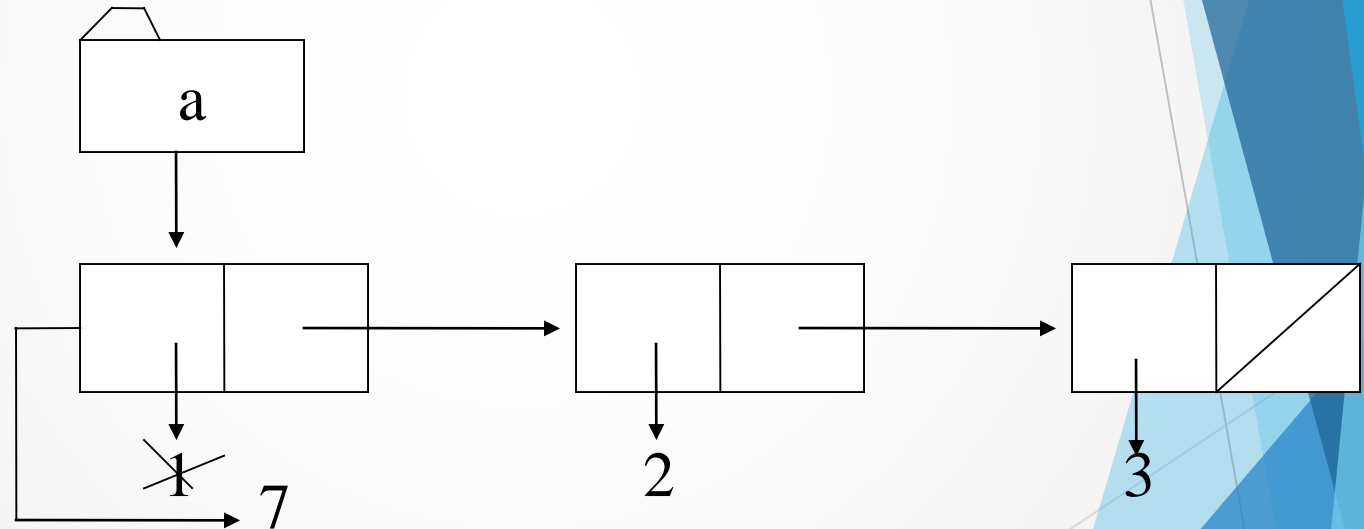
(1 2 3)

> (RPLACA a 7)

(7 2 3)

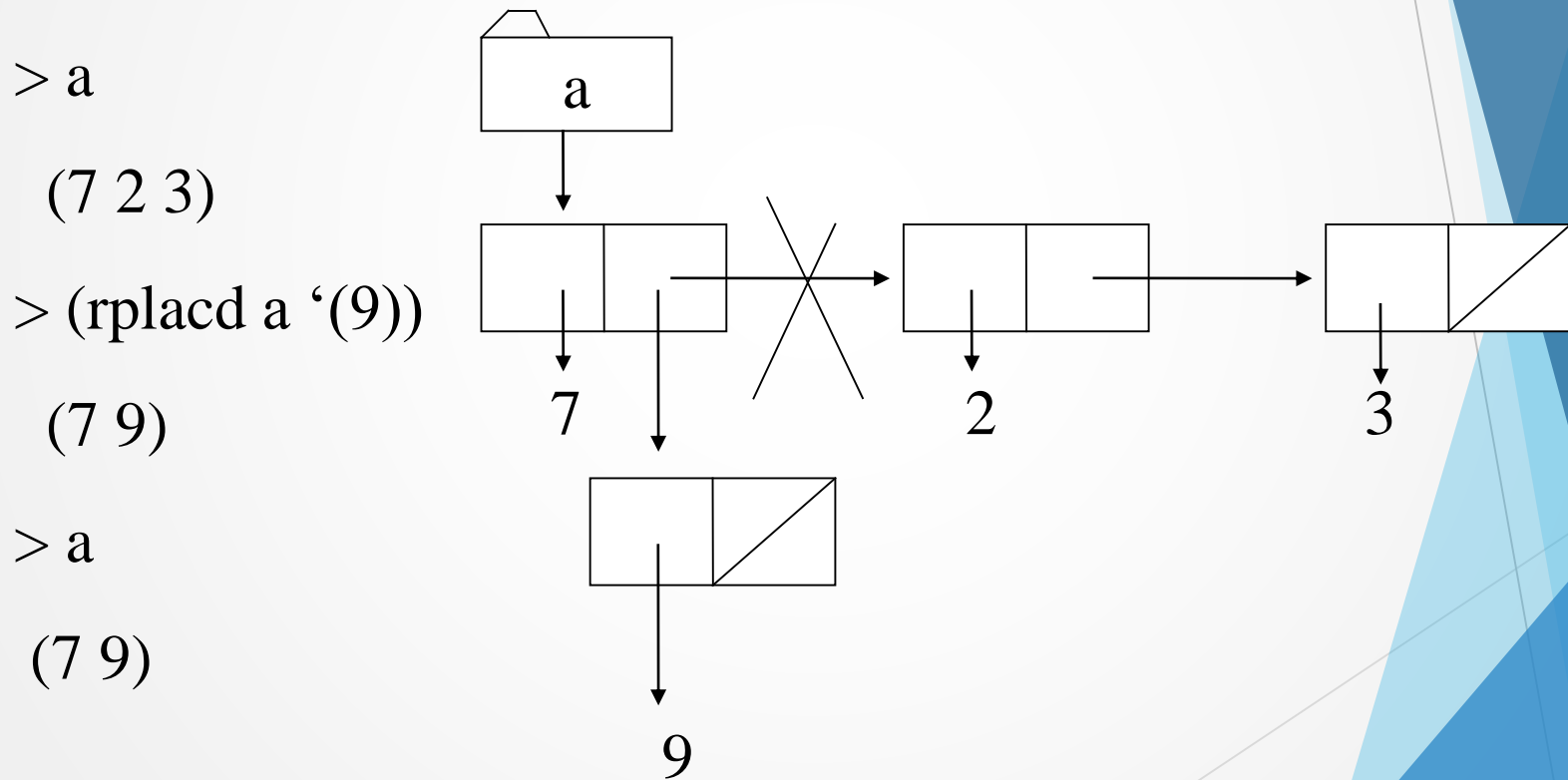
> a

(7 2 3)



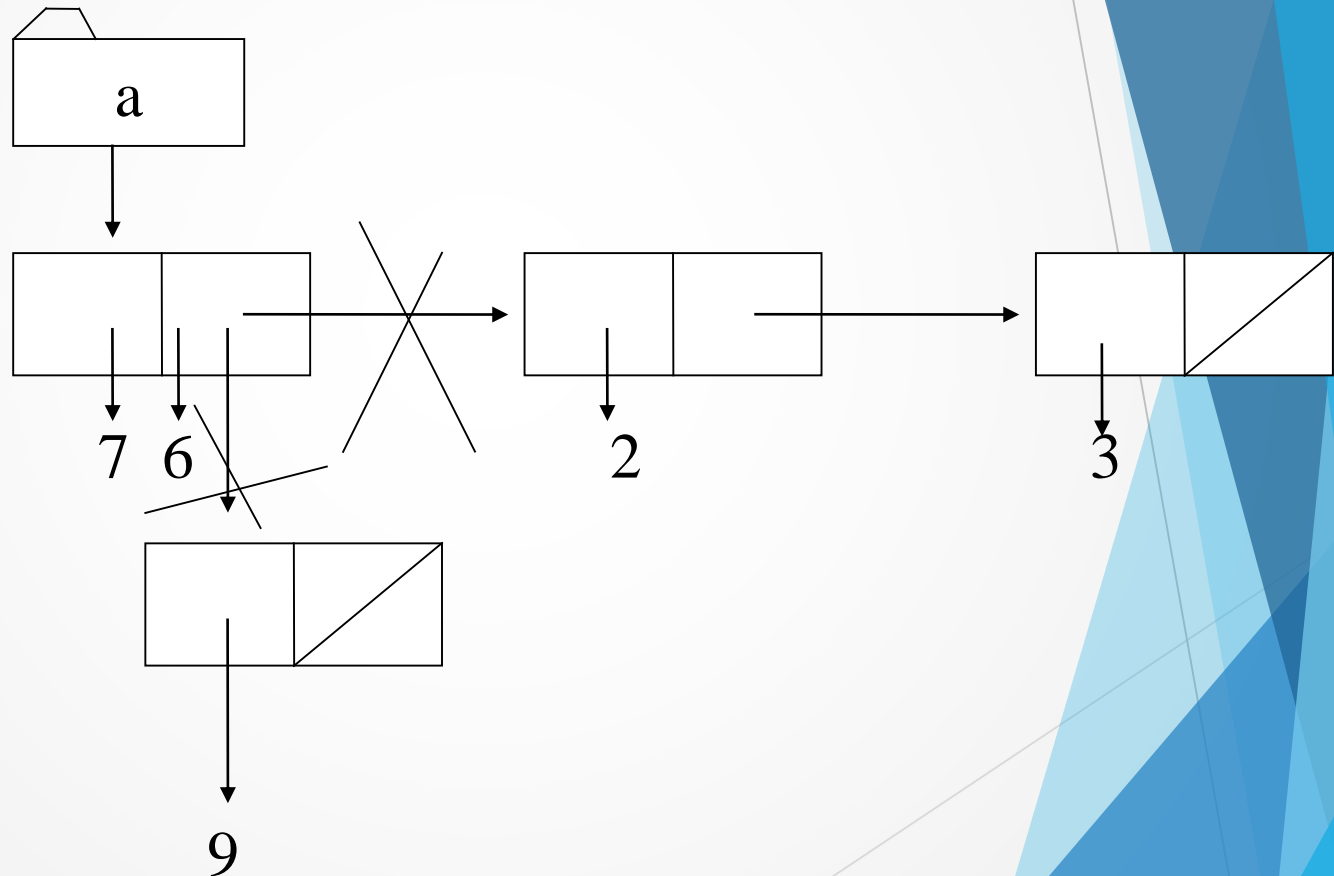
**RPLACD**- reemplaza el CDR de la lista por el elemento dado.

(RPLACD LISTA ELEMENTO)



> (rplacd a 6)

(7.6)



```
> (setf a '(1 2 3 4))
```

```
(1 2 3 4)
```

```
> (rplacd a 9)
```

```
(1.9)
```

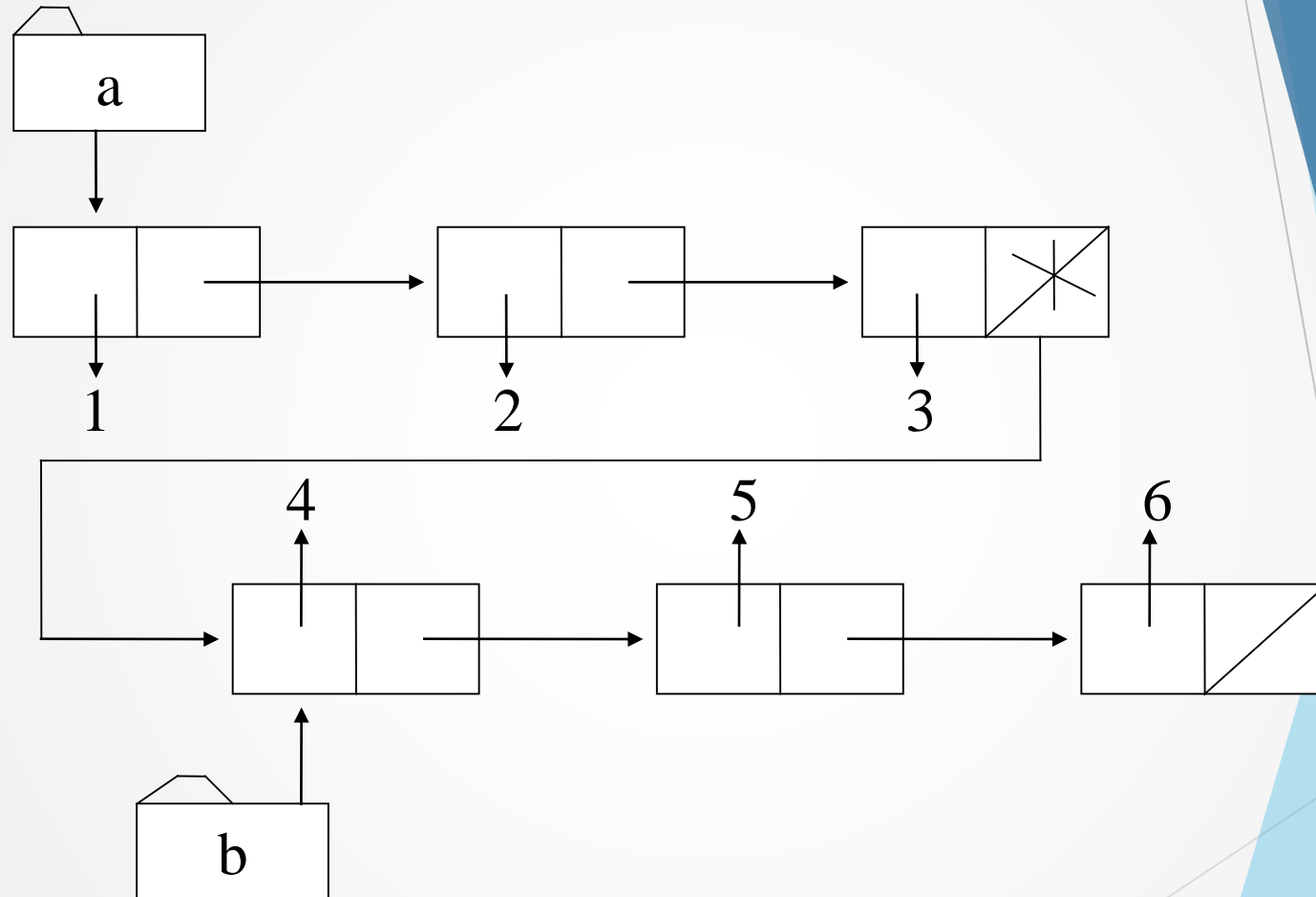
```
> a
```

```
(1.9)
```

**NCONC**- concatena las listas, pero modificando el valor de la primera lista dada como argumento de entrada.

**(NCONC LISTA LISTA...)**

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



**PUSH** – añade el elemento como CAR de la lista  
(PUSH ELEMENTO LISTA)

```
> (setf b '(4 5 6))
```

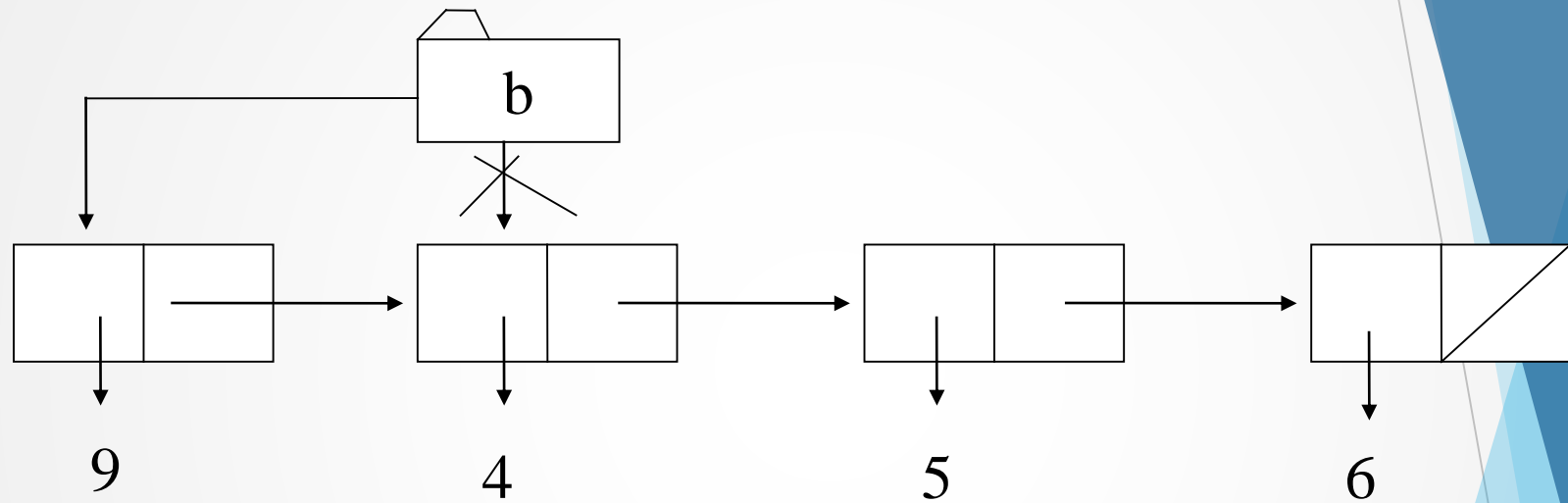
```
(4 5 6)
```

```
> (push 9 b)
```

```
(9 4 5 6)
```

```
> b
```

```
(9 4 5 6)
```





**POP** – elimina el primer elemento (CAR) de la lista  
(POP LISTA)

```
> (setf b '(9 4 5 6))
```

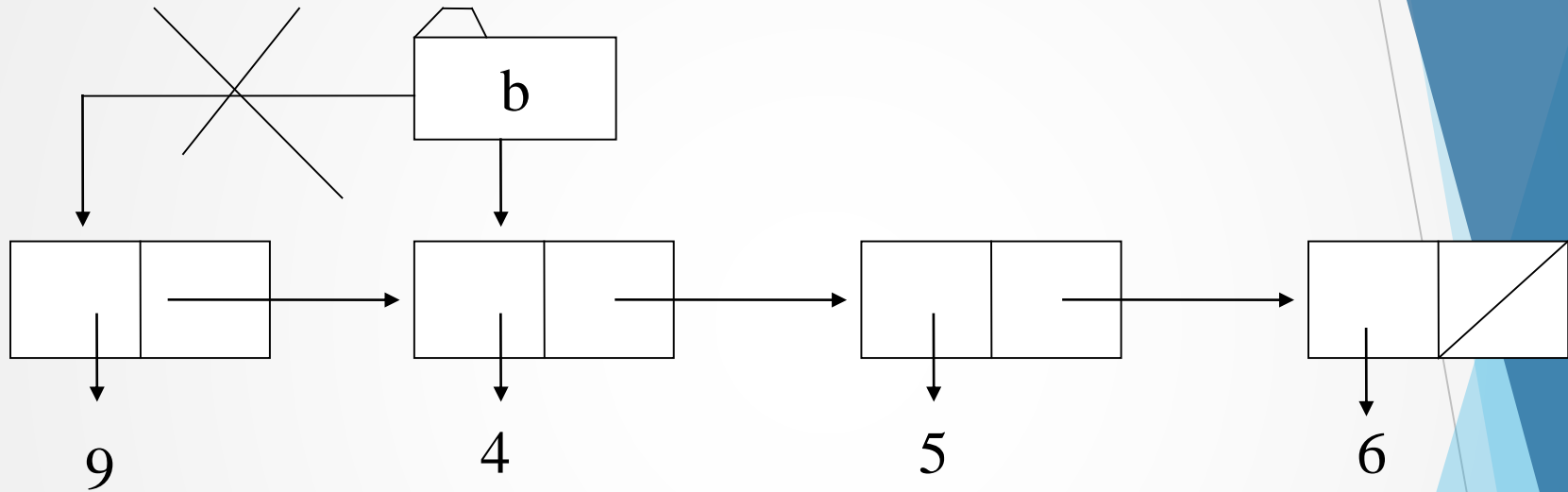
```
(9 4 5 6)
```

```
> (pop b)
```

```
9
```

```
> b
```

```
(4 5 6)
```



Otras funciones importantes:

**REVERSE** – invierte la lista

No-destructiva

**(REVERSE LISTA)**

```
> (setf a '(1 2 3))
```

```
(1 2 3)
```

```
> (reverse a)
```

```
(3 2 1)
```

```
> a
```

```
(1 2 3)
```

**NREVERSE**- invierte la lista

Destructiva

**(NREVERSE LISTA)**

```
> (setf a '(1 2 3 4))
```

```
(1 2 3 4)
```

```
> (nreverse a)
```

```
(4 3 2 1)
```

```
> a
```

```
(4 3 2 1)
```

**LENGTH**- devuelve el número de elementos existentes

**(LENGTH LISTA)**

> (length '(a b c))

3

> (length '(a b.c))

2

> (length '(a (b c) d (e f)))

4

> (length '())

0

## MEMBER (primera definición)

**MEMBER** –busca en el nivel superior de la lista un elemento “igual” al elemento dado. Si lo encuentra la función devuelve una lista cuya CAR es el elemento buscado y el CDR el resto de la lista. Si no encuentra el elemento, la función devuelve NIL.

(MEMBER ELEMENTO LISTA)

> (MEMBER 'c '(a b c d e f))

(c d e f)

> (MEMBER 'z '(a b c d e f))

NIL

## MEMBER (segunda definición)

**MEMBER** –busca en el nivel superior de la lista un elemento que devuelva “true” con la función test dada. Si lo encuentra la función devuelve una lista cuyo CAR es el elemento buscado y el CDR el resto de la lista. Si no encuentra el elemento, la función devuelve NIL.

(MEMBER ELEMENTO LISTA)

> (MEMBER 2.0 '(1 2 3) :TEST #'=)

(2 3)

> (MEMBER 2.0 '(1 2 3) :TEST #'<)

(3)

## Funciones para operar con números

### Operaciones matemáticas

+ **SUMA** – devuelve la suma de todos los argumentos

– **RESTA** – devuelve el resultado de restar al primer elemento todos los demás argumentos

\* **MULTIPLICACIÓN** – devuelve el producto de todos los argumentos

/ **DIVISIÓN** – devuelve el resultado de dividir al primer elemento todos los demás argumentos. Permite el caso de un solo argumento (que devuelve el resultado de dividir 1 entre el argumento)



> (+ 2 3 4)

9

> (- 10 7 3)

0

> (-7)

-7

> (\* 3 4 0.5)

6.0

> (/ 100 4 5)

5

> (/ 12 9)

4/3

> (/ 18 7)

18/7

> (/ 25.0 10)

2.5

> (/ 7)

1/7

> (/ 2.0)

0.5

> (/ 48 24 (\* 2 3))

1/3

## Destructivas

**INCF**- incrementa en una unidad el valor del elemento.

(INCF ELEMENTO)

```
> (setf c 5)
```

```
> (incf c)
```

```
6
```

```
> c
```

```
6
```

Fíjese que la operación es la misma que `(+ 1 c)` pero la suma no es destructiva.

## Destructivas

**INCF**- incrementa en una cantidad DELTA el valor del elemento.

(INCF ELEMENTO [DELTA])

```
> (setf c 5)
```

```
> (incf c 10)
```

```
15
```

```
> c
```

```
15
```

## Destructiva

**DECF** -sustraе una cantidad **DELTA** (opcional) al valor del elemento.

**(DECF ELEMENTO [DELTA])**

```
> (setf c 5)
```

```
> (decf c 10)
```

```
-5
```

Si no se pone argumento DELTA se sustraе 1 unidad.

Fíjese que la operación es la misma que `(- 1 c)` pero la resta no es destructiva

(EXPT x y) – devuelve el valor  $x^y$

> (EXPT 2 3)

8

(EXP n) – devuelve el valor  $e^n$

> (EXP 3)

20,085537  $\approx e^3$

(ABS x) – devuelve el valor absoluto del número x

(SQRT x) – devuelve la raíz cuadrada de x

(REM x y)

(MOD x y)

devuelven el resto de la división entera

## Funciones para comparar números

El resultado es NIL o NO-NIL (T)

Igualdad (= NUM NUM NUM...)

> (= 3 3.0)

T

Distinto (/= NUM NUM...)

> (/= 3 3.0)

NIL

**MENOR QUE** - devuelve verdad si es una secuencia de números estrictamente creciente.

**(< NUM NUM ...)**

> (< 3 5 8)

T

> (< 3 5 4)

NIL

**MAYOR QUE** - devuelve verdad si es una secuencia de números estrictamente decreciente.

(> NUM NUM ...)

> (> 8 5 4)

T

> (> 8 3 5)

NIL



## MENOR O IGUAL QUE

( $\leq$  NUM NUM...)

> ( $\leq$  5 5 8)

T

> (< 9 9 4)

NIL

## MAYOR O IGUAL QUE

( $\geq$  NUM NUM...)

> ( $\geq$  9 77)

T

> ( $\geq$  8 9 8)

NIL

**MAX** – devuelve el máximo.

**(MAX NUM NUM...NUM)**

> (max 2 5 3 1)

5

**MIN** – devuelve el mínimo.

**(MIN NUM NUM ...NUM)**

> (min 2 5 3 1)

1

## PREDICADOS NUMÉRICOS

Funciones Booleanas devuelven verdadero (T) o falso (NIL)

**NUMBERP** – devuelve T si el objeto es un número.

**(NUMBERP OBJETO)**

> (numberp 7)

T

> (numberp 'man)

NIL

## PREDICADOS NUMÉRICOS

Funciones Booleanas que devuelven verdadero (T) o falso (NIL)

**ODDP** - devuelve T si num es un número entero impar.

**(ODDP NUM)**

> (ODDP -7)

T

> (oddp 5.8)

El argumento debe ser un número entero.

## PREDICADOS NUMÉRICOS

Funciones Booleanas que devuelven verdadero (T) o falso (NIL)

**EVENP** – devuelve T si el num es un número entero par.

**(EVENP NUM)**

> (EVENP 8)

T

> (EVENP 'hi)

El argumento debe ser un número entero.

## PREDICADOS NUMÉRICOS

Funciones Booleanas que devuelven verdadero (T) o falso (NIL)

**ZEROP** – devuelve T si el número es 0.

**(ZEROP NUM)**

> (zerop 0.0)

T

> (zerop 'what)

El argumento debe ser un número.

## OPERADORES LÓGICOS

- Predicados sobre el tipo de datos.

**CONSP** – devuelve T si el objeto es una estructura CONS.

(CONSP OBJETO)

**ATOM** – devuelve T si el objeto es un átomo (a efectos prácticos para nosotros, si el objeto no es una estructura CONS).

(ATOM OBJETO)

## OPERADORES LÓGICOS

- Predicados sobre el tipo de datos

**LISTP** –devuelve T si el objeto es CONS o la lista vacía ().

(LISTP OBJETO)

**NULL** –devuelve T si el objeto es NIL

(NULL OBJETO)



> (atom 'moon)

T

> (atom "string")

T

> (atom ())

T

> (atom nil)

T

> (consp '(a list))

T

> (consp '())

NIL

> (listp NIL)

T

> (null 5)

NIL

> (null (\* 2 3))

NIL

## OPERADORES LÓGICOS

- Operadores AND/OR/NOT

**AND** –Esta función admite múltiples argumentos.

- Si alguno de los argumentos es evaluado como NIL, se para la ejecución y la función devuelve **NIL**.
- Si ninguno de los argumentos es evaluado como NIL, devuelve el resultado de la **última evaluación** (o último argumento).

**(AND SEXP SEXP ...)**

```
> (AND T T T (* 2 5))
```

```
10
```

```
> (setf x 3)
```

```
> (setf cont 0)
```

```
> (incf cont)
```

```
> (AND (<= cont 10)(numberp x)(* 2 x))
```

```
6
```

```
> (AND (evenp x)(/ x 2))
```

```
NIL
```

## OPERADORES LÓGICOS

- Operadores AND/OR/NOT

**OR** –Esta función admite múltiples argumentos.

- Si todos los argumentos son evaluados como NIL, devuelve **NIL**.
- Si alguno de los argumentos no es evaluado como NIL, se para la ejecución y se **devuelve ese valor**.

**(OR SEXP SEXP ...)**

```
> (OR NIL NIL (NULL '(A B)) (REM 23 13))
```

10

```
> (setf x 10)
```

```
> (OR (< 0 x)(decf x))
```

T

```
> (setf x 10)
```

```
> (OR (CONSP x)(ODDP x)(/ x 2))
```

5

## OPERADORES LÓGICOS

- Operadores AND/OR/NOT

**NOT** – Solo admite un argumento. Si el argumento es evaluado como NIL, devuelve **T**. En cualquier otro caso, devuelve **NIL**.

(NOT SEXP)

> (NOT NIL)

T

> (NOT T)

NIL

> (NOT (\* 2 3))

NIL

## Predicados de IGUALDAD

- Igualdad de números (**=**)
- Igualdad de estructuras (**EQUAL**)
- Igualdad de objetos (**EQ**)
- Igualdad de representación (**EQL**)

## Predicados de IGUALDAD

- Igualdad de números (=)

= (solo se puede usar en números)- devuelve T si los valores de los números son iguales.

(= NUM1 NUM2)

> (= 3 3)

T

> (= 3 2.5)

NIL

> (= 5 5.0)

T



## Predicados de IGUALDAD

- Igualdad de estructuras (**EQUAL**)

**EQUAL** –devuelve T si los objetos X e Y son estructuralmente iguales. Es decir, si los valores imprimibles son iguales.

**(EQUAL X Y)**

> (equal 5 5.0)

NIL

> (equal nil ())

T

> (setf a 'home)

> (setf b 'home)

> (equal a b)

T

## Predicados de IGUALDAD

- Igualdad de estructuras (**EQUAL**)

**EQUAL** –devuelve T si los objetos X e Y son estructuralmente iguales. Es decir, si los valores imprimibles son iguales.

**(EQUAL X Y)**

> (setf l '(a b c))

> (equal l '(a b c))

T

## Predicados de IGUALDAD

- Igualdad de objetos (**EQ**)

**EQ** –devuelve T si los objetos X e Y son iguales en la representación de la memoria.

**(EQ X Y)**

> (setf 1 '(a b c))

> (EQ 1 '(a b c))

NIL

El uso del comando EQ es complejo, pero es importante decir que no se debe usar para comparar números.

```
> (EQ 3 3)
```

```
T
```

```
> (EQ 3.0 3.0)
```

```
NIL
```

## Predicados de IGUALDAD

- Igualdad de representación (**EQL**)

**EQL** – Su comportamiento depende del tipo de argumento.

- Números - devuelve T si son iguales en valor y tipo.
- Caracteres – devuelve T si son el mismo carácter o son EQ.

**(EQL X Y)**

> (EQL 3.7 3.7)

T

> (EQL 5 5.0)

NIL

No se debe usar el comando EQL en Strings

	ÁTOMOS		LISTAS
	Números	Símbolos	
=	¿Mismo valor?	No permitido	No permitido
EQ	NO USAR	NO USAR	¿Mismo objeto físicamente? (Apunta a la misma lista)
EQL	¿Mismo valor y tipo?	¿Mismo símbolo?	
EQUAL			¿Mismo aspecto? ¿Mismos elementos?

## Números

> (= 3 3)

T

> (= 5 5.0)

T

> (EQUAL 3 3)

T

> (EQUAL 5 5.0)

NIL

## Símbolos

> (setf a 'hi)

> (setf b 'hi)

> (EQL a b)

T

> (EQUAL a b)

T

## Listas

> (SETF 1 '(a b) )

> (EQUAL 1 '(a b))      (Mismo aspecto)

T

> (EQ 1 '(a b))      (Mismo puntero)

NIL



## ESTRUCTURAS CONDICIONALES

**IF** – si la condición es evaluada distinta de NIL, se ejecuta la sentencia1, sino la sentencia2.

(if condición sentencia1 sentencia2)

*En Pascal, corresponde a:*

*if condición then sentencia1  
else sentencia2;*

Sentencia1 y sentencia2 solo pueden ser una instrucción y no una lista de instrucciones.

```
> (setf x 5)
```

```
> (if (> y x) y x)
```

```
> (setf y 4)
```

```
5
```

**PROGN** proporciona una forma de agrupar una lista de instrucciones. Es muy útil en la instrucción **IF**.

Al finalizar el comando if, Lisp devuelve lo último evaluado.

```
> (if (> y x) y (progn (print (+ x y))(print 'hola))))
```

```
9
```

```
HOLA
```

```
HOLA
```

```
> (if (listp '(a b c)) (+ 1 2) (+ 5 6))
```

3

```
> (if (listp 4) (+ 1 2) (+ 5 6))
```

11

## ESTRUCTURAS CONDICIONALES

**COND**- Se ejecuta la lista de sentencias de la primera condición evaluada **NO-NIL**. El resto de condiciones se ignoran.

No es necesario usar **PROGN**

**(COND (cond1 sentencias1) (cond2 sentencias2)...) (COND**

*En Pascal corresponde al CASE of IF-anidado:*

*(if cond1 then sentencias1*

*else if cond2 then sentencias2*

*else if cond3....*

```
> (setf x 5)
```

```
5
```

```
> (setf y 4)
```

```
4
```

```
> (cond ((= x y)(print 'iguales))
```

```
      ((> x y) (print 'mayor) (* x y))
```

```
      (t (print 'menor)))
```

```
MAYOR
```

```
20
```

```
> (cond ((* 2 6) (print 'hola))  
      ((= x y) (print 'iguales))  
      ((< x y) (print 'menor))  
      (t (print 'mayor)))
```

HOLA

HOLA

## ESTRUCTURAS CONDICIONALES

**WHEN**- Si la condición es evaluada NO-NIL se ejecutan las sentencias. Se devuelve lo último evaluado. Si la condición es evaluada NIL, se devuelve NIL.

(**WHEN** condición sentencias)

```
> (setf x 5)
```

```
5
```

```
> (setf y 4)
```

```
4
```

```
> (when (> x y) (print 'SI) (+ x 2) (- 2 y))
```

```
SI
```

```
-2
```

## ESTRUCTURAS CONDICIONALES

**WHEN**- Si la condición es evaluada NO-NIL se ejecutan las sentencias. Se devuelve lo último evaluado. Si la condición es evaluada NIL, se devuelve NIL.

(**WHEN** condición sentencias)

```
> (setf x 5)
```

```
5
```

```
> (setf y 4)
```

```
4
```

```
> (when (< x y) (print 'SI) (+ x 2) (- 2 y))
```

```
NIL
```

```
> (when (* 2 3) (print 'ok))
```

```
OK
```

```
OK
```



## ESTRUCTURAS CONDICIONALES

**UNLESS**- Si la condición es evaluada NIL se ejecutan las sentencias. Se devuelve lo último evaluado. Si la condición es evaluada NO-NIL, se devuelve NIL.

(UNLESS condición sentencias)

```
> (unless (= 1 1) 1 2 3)
```

NIL

```
> (unless (= 1 2) 1 2 3)
```

3

## ESTRUCTURAS ITERATIVAS

(LOOP

sentencias)

Ejecuta las sentencias del interior del bucle reiteradamente.  
Se sale del bucle con la orden RETURN.

(RETURN) puede encontrarse solo o

(RETURN instrucción)

```
> (setf y 0)
```

```
0
```

```
> (setf l '())
```

```
NIL
```

```
> (loop
```

```
  (if (= y 5) (return (reverse l))
```

```
    (progn (push y l) (incf y))))
```

```
(0 1 2 3 4)
```

```
(loop
  (print "Dame tu nombre:")
  (setf nombre (read))
  (print "Dame tu apellido:")
  (setf apellido (read))
  (print "¿Es correcto s/n?")
  (setf s/n (read))
  (if (equal s/n 's)(return ((print cons nombre apellido))))
" Dame tu nombre:" Yon
" Dame tu apellido : " Arrieta
"¿Es correcto s/n? " s
(YON . ARRIETA)
```

## ESTRUCTURAS ITERATIVAS

```
( DO ( (var1 inicialización [crecimiento])  
      (var2 inicialización [crecimiento])  
      ..... )  
      (condición de parada [instrucción de salida] )  
      sentencias )
```

*En Pascal corresponde al FOR*

## ESTRUCTURAS ITERATIVAS

( DO .... )

NOTA: para la condición de parada no se ejecuta el cuerpo (en Pascal para el FOR se ejecuta).

NUNCA se debe modificar la condición de parada dentro del cuerpo.

```
> (do ((i 0 (incf i))) ((= i 2) 'adiós)  
    (print 'hola))
```

HOLA  
HOLA  
ADIÓS

```
> (do ((i 0 (incf i)))((> i 5) 'adiós)
    (do ((j 0 (incf j)))(> j i)
        (princ j))
    (terpri))
0
01
012
0123
01234
012345
ADIÓS
```

## ESTRUCTURAS ITERATIVAS

( **DOLIST** (variable lista [instrucción de salida] )  
cuerpo )

Ejecutará un bucle donde la variable va tomando todos los valores de lista. Se ejecutan las instrucciones del cuerpo una vez para cada valor de la variable.



```
> (dolist (x '(1 2 3 4 5) 'adiós )  
    (print x))
```

1

2

3

4

5

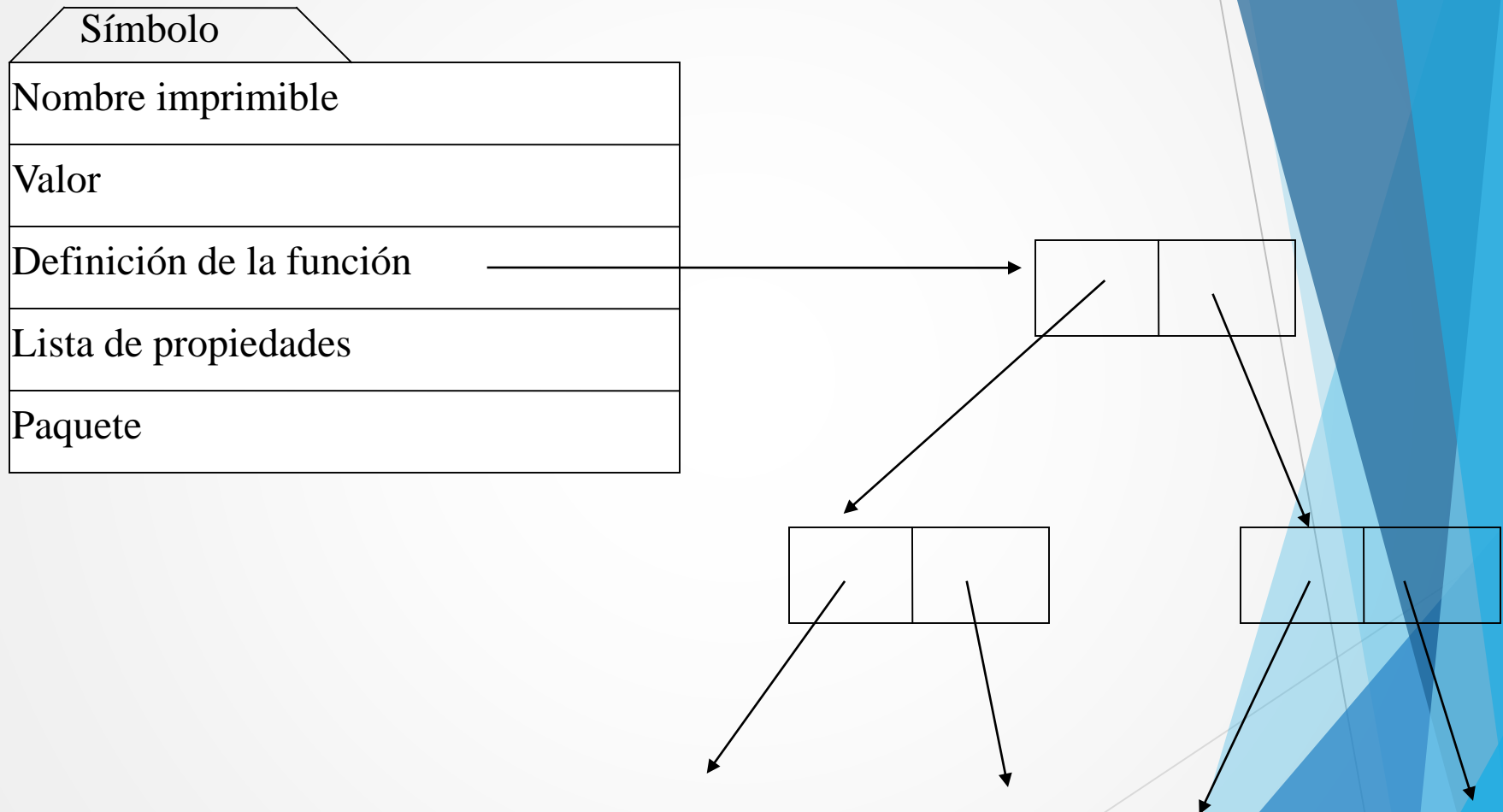
ADIÓS

```
> (setf l '(1 2 3 4 5 6 7 8))  
> (dolist (x l 'test)  
  (if (= x 3) (return 'si)(print x)))  
1  
2  
SI
```

```
> (setf p '(1 2 3))  
> (setf q '(4 5 6 7 8))  
> (setf r '())  
> (dolist (a p (reverse r))  
      (dolist (b q)  
        (push (+ a b) r)))  
(5 6 7 8 9 6 7 8 9 10 7 8 9 10 11)
```

## FUNCIONES DEFINIDAS POR EL USUARIO

- Todos los programas en LISP se describen mediante funciones (LISP es un lenguaje funcional).
- Todos los programas se llaman mediante listas.
- Cuando el intérprete de LISP encuentra un “(“ los trata como el comienzo de una función cuyo nombre es el primer argumento.



## FUNCIONES DEFINIDAS POR EL USUARIO

(DEFUN nombre (lista de argumentos)  
cuerpo)

Es posible escribir una función sin argumentos de entrada, pero se deben incluir los paréntesis vacíos.

(DEFUN nombre ()  
cuerpo)

```
> (defun ejemplo ()  
    (print 'hola))
```

EJEMPLO

```
> (defun cubo (x)  
    (* x x x))
```

CUBO

```
> (defun area-rectangulo (x y)  
    (* x y))
```

AREA-RECTANGULO

```
> (defun perimetro_rectangulo (x y)  
    (+ (* 2 x) (* 2 y)))
```

PERIMETRO\_RECTANGULO

## FUNCIONES DEFINIDAS POR EL USUARIO

Llamada a las funciones:

(nombre\_función argumentos)

Si la función no tiene argumentos:

(nombre\_función)



> (ejemplo)

HOLA

HOLA

> (cubo 3)

27

> (area-rectangulo 3 4)

12

> (perimetro\_rectangulo 3 4)

14

## ERRORES MÁS FRECUENTES AL DEFINIR FUNCIONES

1. Añadir comillas o paréntesis en la lista de argumentos de entrada de la función.

```
(defun area-rectangulo ('x 'y)
  (list 'el 'area 'es (* x y)))
```

```
(defun area-rectangulo ((x) (y))
  (list 'el 'area 'es (* x y)))
```

## ERRORES MÁS FRECUENTES AL DEFINIR FUNCIONES

2. Añadir comillas o paréntesis a las variables en el cuerpo de la función.

```
(defun area-rectangulo (x y)
  (list 'el 'area 'es (* 'x 'y)))
```

```
(defun area-rectangulo (x y)
  (list 'el 'area 'es (* (x) (y))))
```

## ERRORES MÁS FRECUENTES AL DEFINIR FUNCIONES

3. No poner comillas en el cuerpo de la función.

```
(defun area-rectangulo (x y)
  (list the area is (* x y)))
```

## OTRAS FUNCIONES

### MAPCAR

- Aplica la función introducida como argumento a los consecutivos CAR de las listas introducidas como argumentos.
- Devuelve una nueva lista con el resultado de aplicar la función componente a componente.
- La función introducida como argumento debe tener tantos argumentos como listas introducidas como argumentos.

(MAPCAR #'función lista1...listan)

> (mapcar #'(7 8 9) '(1 2 3))

(8 10 12)

> (mapcar #'oddp '(7 8 9))

(T NIL T)

> (mapcar #'cubo '(3))

(27)

> (mapcar #'cubo '(2 3 4))

(8 27 64)

> (mapcar #'\* '(1 2 3) '(4 5 6))

(4 10 18)

## OTRAS FUNCIONES

### APPLY

- Permite pasar los argumentos de las funciones como una lista.
- No devuelve una lista, sino solo el átomo calculado.

(APPLY #'función LISTA\_de\_argumentos)

```
> (apply #'cubo '(3))  
27
```

```
> (apply #'area-rectangulo '(2 4))  
8
```

## OTRAS FUNCIONES

### SYMBOL-FUNCTION

- Permite dar el nombre de la función como una variable (es equivalente a #')

(APPLY (symbol-function nombre\_función) lista\_de\_argumentos)



> (apply (symbol-function 'area-rectangulo) '(4 5))

20

> (setf rectangulo '(area-rectangulo perimetro\_rectangulo))

(AREA-RECTANGULO PERIMETRO\_RECTANGULO)

> (apply (symbol-function (car rectangulo)) '(2 8))

16

> (apply (symbol-function (car (cdr rectangulo))) '(2 8))

20

## OTRAS FUNCIONES

### FUNCALL

- Igual que APPLY pero los argumentos no van en una lista.
- No devuelve una lista.

(FUNCALL función argumentos)

```
(funcall (symbol-function (car rectangulo)) 2 5)  
10
```

```
> (defun miaplica (lista n1 n2)
  (dolist (p lista)
    (princ (funcall (symbol-function p) n1 n2))
    (princ " " )))
```

MIAPLICA

```
> (miaplica rectangulo 2 4)
8 12
```

- Observaciones:

- El primer argumento de `apply` y `funcall` debe ser algo cuyo valor sea una función.
- Para forzar a obtener el valor funcional se usa `#'`.
- Si el nombre de la función viene dado en una variable, será necesario usar `SYMBOL-FUNCTION`.

## CARACTERÍSTICAS DE LAS FUNCIONES

- 1.- El nombre de una función es siempre un símbolo.
- 2.- Los parámetros de la definición son variables locales.
- 3.- El paso de los parámetros es por valor.

> (setf x 5)

5

> (defun modifica (x)  
                  (incf x))

MODIFICA

> (modifica x)

6

> x

5

## CARACTERÍSTICAS DE LAS FUNCIONES

4.- Normalmente, las variables que se declaran dentro de la función con `setf` son globales.

```
> (defun test (x y)
      (setf z 3)
      (+ x y z))
```

TEST

```
> (test 3 4)
10
> z
3
```

## CARACTERÍSTICAS DE LAS FUNCIONES

5.- Se pueden crear variables locales con la orden **LET**.

```
(LET ((var1 inicialization)
      (var2 inicialization)
      .....))
      ámbito_de_validez_variables )
```



- Las variables dentro de **LET** que no son inicializadas, por defecto LISP las inicializa a NIL.
- Si dentro del cuerpo LET se declara una variable con SETF esta es global para el programa, salvo en el caso de que sea una variables declaradas en LET.

```
> (defun ejemplo ()  
  (let ((x 0) (y 3) (z) )  
    (setf x (+ x y))  
    (setf z "hola")))
```

EJEMPLO

```
> (ejemplo)
```

HOLA

```
> z
```

EVAL: variable z has no value

```
> (defun suma_4 (x)  
  (let ((y 1) (z 3))  
    (+ x y z)))
```


SUMA\_4

```
> (suma_4 10)
```

14

- **LET** asigna los valores en paralelo.

```
> (defun suma_algo (x)
    (let ((x (+ x 2))(y (+ x 1)))
      y))
```

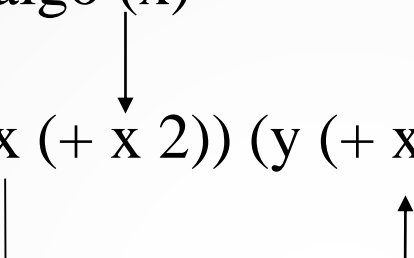


SUMA\_ALGO

```
> (suma_algo 3)
4
```

- **LET\*** asigna los valores secuencialmente

```
>(defun suma_algo (x)
  (let* ((x (+ x 2)) (y (+ x 1)))
    y))
```



SUMA\_ALGO

```
> (suma_algo 3)
6
```

## CARACTERÍSTICAS DE LAS FUNCIONES

6.- Mientras no se diga lo contrario, las funciones siempre devuelven lo último evaluado.

```
> (defun test (x y)
      (+ x y)
      (* x y))
```

TEST

```
> (test 2 5)
10
```

## CARACTERÍSTICAS DE LAS FUNCIONES

7.- Los argumentos de entrada son evaluados antes de pasarlos como información a la función.

```
> (defun test_2 (x)
      (+ 10 x))
TEST_2
```

```
> (test_2 (+ 99 1))= (test_2 (100))=110
```

## CARACTERÍSTICAS DE LAS FUNCIONES

### 8.- Recursividad

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

FACTORIAL

```
> (factorial 3)
6
```

En la recursividad puede ser interesante el uso de la función **TRACE**.

**TRACE** - muestra la traza de la operaciones que va realizando el intérprete de LISP.

(TRACE F-1 ... F-N)

**UNTRACE** - finaliza la orden trace.

(UNTRACE F-1 ... F-N)

```
> (trace factorial)
(FACTORIAL )
> (factorial 2)
1. Trace: (FACTORIAL '2)
2. Trace: (FACTORIAL '1)
3. Trace: (FACTORIAL '0)
3. Trace: FACTORIAL ==> 1
2. Trace: FACTORIAL ==> 1
1. Trace: FACTORIAL ==> 2
2
> (untrace FACTORIAL)
```



```
> (defun cuenta-atomos (unalista)
  (cond
    ((null unalista) 0)
    ((atom (car unalista)) (+ 1 (cuenta-atomos (rest unalista))))
    (t (cuenta-atomos (rest unalista)))))
Cuenta-atomos
```

```
> (cuenta-atomos '(1 2 3 4))
4
```

```
> (cuenta-atomos '((a b) "hola" (1 2 3 4)))
1
```

## CARACTERÍSTICAS DE LAS FUNCIONES

### 9.- Funciones sin nombre.

Son útiles si:

- Cuando hay mucho código;
- Se van a utilizar pocas veces (solo una);
- Ahorran bucles.

(**LAMBDA** (parametros)

cuerpo)

> (lambda (x)

(\* x x))

#<CLOSURE :LAMBDA (X) (\* X X)>

```
> (mapcar #'(lambda (x) (* x x)) '(2 3 4))  
(4 9 16)
```

```
> (lambda (x) (+ x 100))  
#<CLOSURE :LAMBDA (X) (+ X 100)>
```

```
> ((lambda (x) (+ x 100)) 4)  
104
```

```
> (funcall #'(lambda (x) (+ x 100)) 4)  
104
```

## CARACTERÍSTICAS DE LAS FUNCIONES

### 10.- Modificadores de parámetros.

**&OPTIONAL**= parámetros opcionales.

- LISP acepta lo que se llaman parámetros opcionales.
- No precisan que se introduzcan como argumentos de entrada.
- Los parámetros que no tiene correspondencia en la llamada se identifican con NIL.

```
> (defun area (a &optional b)
      (if (null b) (* a a) (* a b)))
```

AREA

```
> (area 3)
```

9

```
> (area 3 4)
```

12

## CARACTERÍSTICAS DE LAS FUNCIONES

**&REST**= el resto de parámetros.

- En la llamada primero se identifican los argumentos de entrada necesarios (obligatorios) y los demás, el intérprete los uno en una lista.

```
> (defun list (x &rest y)
      (append x y))
```

LIST

```
> (list '(1 2) 3 4 5 6 7)
(1 2 3 4 5 6 7)
```

## CARACTERÍSTICAS DE LAS FUNCIONES

**&KEY**= parámetros con nombre.

- Se pasan en cualquier orden.
- 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 valor.
- Cualquier clave en la que no se hace referencia en la llamada se identifica con NIL.

```
> (defun f (&key (x 1) (y 2))  
      (list x y))
```

F

```
> (f)  
(1 2)
```

```
> (f :x 5 :y 3)  
(5 3)
```

```
> (f :y 3)  
(1 3)
```



## CARACTERÍSTICAS DE LAS FUNCIONES

**&AUX**= valores auxiliares de la función. No son parámetros.

- Es obligatorio inicializarlos.

```
> (defun mas_uno (x &aux (y 1))  
      (+ x y))
```

MAS\_UNO

```
> (mas_uno 2)
```

3

```
> (mas_uno 2 3)
```

\*\*\* too many arguments given to MAS\_UNO

## Variables especiales: variables globales de otros lenguajes

**DEFVAR** – declara una variable global `nombre_variable` que es accesible desde cualquier parte del programa.

**(DEFVAR nombre\_variable [valor])**

**DEFPARAMETER** - declara un parámetro global `nombre_parametro` que es accesible desde cualquier parte del programa.

**(DEFPARAMETER nombre\_parametro [valor])**

**DEFCONST** – Será una constante para todo el programa. Si se cambia dará un error.

**(DEFCONST nombre\_constante valor)**

## MACROS

### DEFMACRO

- Al igual que las funciones son formas que permiten agrupar acciones y darles un nombre.
- Expenden su código en tiempo de interpretación (las funciones lo hacen en tiempo de ejecución).
- **NO EVALÚAN LOS ARGUMENTOS.**

## (DEFMACRO NOMBRE (LISTA\_PARAMETROS) CUERPO)

```
> (defmacro f (a)
    (cond
      ((numberp a) (print "numero"))
      ((listp a) (print "lista"))))
```

F

```
> (f (+ 4 5))
LISTA
LISTA
```

```
> (defun f (a)
  (cond
    ((numberp a) (print "numero"))
    ((listp a) (print "lista"))))
```

F

```
> (f (+ 4 5))
NUMERO
NUMERO
```

## DOCUMENTACIÓN

Se pueden hacer comentarios en cualquier parte del programa utilizando “;”, de tal forma que no se ejecuta lo que sigue al ;

;;;;, y un espacio en blanco, al comienzo del fichero para indicar su contenido.

;;, y un espacio en blanco, para comentarios sobre una función . Debe aparecer en la línea posterior a la definición de la función (cabecera).

;;, y un espacio en blanco, se utiliza para comentar una línea de código.

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

```
(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"); es una chica que vuela
    ((Mortadelo) "Filemon"); Ambos son agente de la T.I.A:
    (t "no tiene pareja")
    );Fin de selección
  );Fin de la función
```

;;; Esperamos que haya encontrado la pareja. Si no, modifique la

;;; función siempre que sea necesario

;;; Fin del fichero

## ENTRADA/SALIDA SIMPLE Y STREAMS

- Funciones de entrada

**READ** – lee un objeto desde teclado y devuelve dicho objeto. Detiene la ejecución del programa mientras no se presione el enter o return. La función no evalúa.

(**READ** &optional stream )

**READ-CHAR** –Lee un carácter. lo devuelve como un objeto tipo carácter.

(**READ-CHAR** &optional stream )



- Funciones de salida

**PRINT** —toma un objeto y lo escribe en una nueva línea con un blanco detrás.

(PRINT objeto &optional stream)

**PRINC** —Evita la repetición de mostrar el último comando evaluado.

(PRINC objeto &optional stream)

**TERPRI** — Escribe y realiza un salto de línea.

(TERPRI &optional stream)

**FORMAT** –forma de escribir diversos tipos de variables.

(FORMAT destination control-string &rest arguments)

Salida:

t, indica la salida por defecto

NIL, se creará un string con las características indicadas y éste será lo que devuelve forma

stream, indica el dispositivo de salida

A nuestros efectos prácticos, nosotros podemos utilizar siempre t.

Control-string: Contiene el texto a escribir y las directivas.

Se utilizan distintas directivas para distintos tipos de datos.

~A o ~a: imprime el siguiente argumento sin comillas.

~S o ~s: imprime el siguiente argumento con comillas.

~D o ~d: imprime el siguiente número en base 10.

~E o ~e: imprime el siguiente número real en notación científica.

~F o ~f: imprime el siguiente número real en coma flotante.

~%: salta de línea. ~& idem (pero si ya está en una nueva línea no salta).

~T o ~t: tabulador.

~num@A: deja tantos espacios en blanco como se indique en “num”

```
> (setf x "hola")
```

```
> (setf y “que”)
```

```
> (setf z “tal”)
```

```
> (format t “Esto es una prueba: ~A ~A ~A” x y z)
```

```
Esto es una prueba: hola que tal
```

```
NIL
```

> (format t "~% ~% el numero ~D es un entero." 3)  
EL NUMERO 3 ES UN ENTERO.  
NIL

> (format t "~% el numero ~%~E es un real." 3.73)  
EL NUMERO 3,73 ES UN REAL.  
NIL

> (format t "~% ~% el numero ~F es un real." 3.75)  
EL NUMPERO 3,75 ES UN REAL.  
NIL

```
(format t "esto es una prueba~10@A" "hola")  
ESTO ES UNA PRUEBA      HOLA  
NIL
```

## 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 a un terminar inactivo.

## NOTAS SOBRE FICHEROS

- Sugerimos el uso del programa Notepad ++ para escribir los programas en LISP.
- Hay que guardarlos como  $\Rightarrow$  `nombre.lsp`.
- Para cargar el archivo se usa la orden  $\Rightarrow$   
`(load “nombre.lsp”).`

```
> (load "example.lsp")  
;; Loading file example.lsp ...  
Write a possitive odd number:7  
1 3 5  
;; Loaded file example.lsp  
T
```



## MÁS ADELANTE VEREMOS

- Estructuras
- Matrices
- Ordering
- ...

## BIBLIOGRAFÍA

- Graham, P. ANSI Common Lisp (Prentice Hall, 1995).
- Winston, P.R. y Horn, B.K. LISP (3a. ed.) (Addison Wesley, 1991).
- David S. Touretzky. COMMON LISP: A Gentle Introduction to Symbolic Computation (The Benjamin/Cummings Publishing Company, 1990).