Tema 1: Introducción a Lisp

José L. Ruiz Reina

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Introducción a Lisp

- **Historia: John McCarthy, 1958**
- **LISt Processing**
  - **Lambda cálculo**
- **Lenguaje interpretado**
  - **Bucle lee, evalúa, escribe**
  - **Compilador**
- **Históricamente, uno de los lenguajes de la Inteligencia Artificial**
  - **Procesamiento simbólico**
- **Common Lisp: Clisp, GCL, CMUCL, Allegro...**
  - **Usaremos clisp para las prácticas**

Introducción a Lisp

- **Prototipado Rápido**
- **Eficiencia**
- **Paradigma de programación funcional**
  - **Aunque no es puramente funcional**
- **Un lenguaje de programación flexible**
  - **Lenguaje de programación "programable"**

Una sesión con Lisp

- **Arrancar un intérprete:** `clisp`
- **Interactuar con él:**
  - **El usuario escribe una expresión en el intérprete.**
  - **El intérprete de Lisp reconoce la expresión como la representación escrita de un objeto Lisp.**
  - **El objeto es evaluado. Su valor es un objeto Lisp.**
  - **El intérprete elige una representación escrita del valor obtenido.**
  - **El intérprete escribe dicha representación.**
- **Cerrar el intérprete:** `(exit)`
- **Compilación:** `compile-file`

Introducción a Lisp

- **Expresiones**

```
> 1
1
> (+ 2 3)
5
> (+ (- 5 2) (* 3 3))
12
```

- **Primeras observaciones:**
  - **Notación prefija: función y argumentos. Paréntesis. Expresiones anidadas**
  - **Atomos y listas**
  - **Sintácticamente simple**

Introducción a Lisp

- **Evaluación**

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

- **Regla de evaluación (básica)**
  - **Evaluación de los argumentos**
  - **De izquierda a derecha**
  - **Los valores se pasan a la función**
- **Todo se evalúa**
  - `quote` **para evitar evaluaciones**
- **Las *formas especiales* no siguen la regla básica de evaluación**
  - **Por ejemplo:** `if`, `cond`,...
  - **Se verán más adelante**

Introducción a Lisp

- **La función** `quote`**:**

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

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

- **Otro ejemplo:**

```
> x
*** - EVAL: variable X has no value
1. Break> abort
> 'x
X
> (esto (es) (una lista))
*** - EVAL: undefined function ESTO
1. Break> abort
> '(esto (es) (una lista))
(ESTO (ES) (UNA LISTA))
```

Introducción a Lisp

- **Entrada-Salida frente a evaluación**
- **Salida: función** `format`

```
> (format t "~a mas ~a igual a ~a. ~%" 2 3 (+ 2  3))
2 mas 3 igual a 5.
NIL
```

- **Entrada: función** `read`

```
> (defun pide (frase)
    (format t "~a " frase)
    (read))
PIDE
> (pide "Su edad, por favor:")
Su edad, por favor: 23
23
```

- **Algunas observaciones**
  - **Lectura de expresiones Lisp**
  - **Secuencialidad en programas**
  - **Efectos colaterales**

Introducción a Lisp

- **Variables locales y globales**

```
> (let ((x 1) (y 2))
    (+ x y))
3
> x
*** - EVAL: variable X has no value
> (setf *glob* 3)
*GLOB*
> *glob*
3
```

Introducción a Lisp

- **El paradigma de la programación funcional**
  - **Valores *vs.* Modificaciones**

```
> (setf x '(a b c d))
(A B C D)
> (remove 'b x)
(A C D)
> x
(A B C D)

> (setf x (remove 'b x))
(A C D)
> x
(A C D)
```

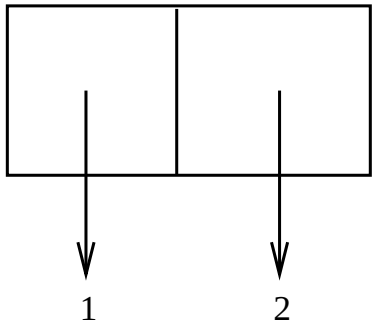
- **La programación sin efectos colaterales es preferible aunque no obligatoria**
  - **Los programas se comportan como funciones en el sentido matemático, dependiendo exclusivamente de sus datos de entrada**
  - **Verificación formal**

Tipos de datos básicos

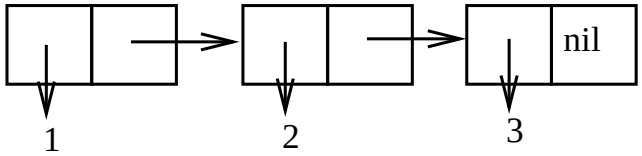
- **Atómicos:**
  - **números:** `27`, `3.14`, ...
  - **símbolos:** `foo`, `FIRST`, `+`, ...
  - **caracteres:** `#\A`, `#\Space`
- **No atómicos:**
  - **Pares punteados y listas:** `(a . b)`, `(a b c d e)`
  - **Cadenas:** `"Buenos días"`
  - **Arrays:** `#(a b 0 nil)`, `#3A(((0 0) (1 1) (2 2)) ((0 1) (1 2) (2 3)))`
  - **Estructuras:** `#S(PERSONA :NOMBRE (ana maria) :ESTADO casado :CALLE (reina mercedes) :CIUDAD sevilla)`
  - **Tablas hash**

Pares y listas

- **Pares punteados:**
  - **Construtor:** `cons`
  - **Ejemplo:** `(cons 1 2)` **se evalúa al par** `(1 . 2)`



- **Listas**
  - **Un tipo especial de par punteado (pero el tipo de dato no atómico más común)**
  - **Ejemplo: la lista** `(1 2 3)`



Funciones de construcción de listas

- **Funciones básicas de creación**

```
* (CONS X Y)
      (cons 'a 'b)                          =>  (A . B)
      (cons 'a '(b c))                      =>  (A B C)
      (cons 'a (cons 'b (cons 'c '())))     =>  (A B C)
      (cons '(a b) '(c d))                  =>  ((A B) C D)
* (LIST X-1 ... X-N)
      (list 'a 'b 'c)                       =>  (A B C)
      (list '(a b) '(c d))                  =>  ((A B) (C D))
      (list)                                =>  NIL
      (list (list 'a 'b)
            (list 'c 'd 'e))                =>  ((A B) (C D))
* (APPEND L-1 ... L-N)
      (append '(a) '(b) '(c) '(x y))        =>  (A B C X Y)
      (append '(a b) '(c d))                =>  (A B C D)
* (REVERSE L)
      (reverse '(a (b c) d))                =>  (D (B C) A)
```

Funciones de acceso a listas

```
* (FIRST L), (CAR L)
      (first '(a b c))                  =>  A
      (first ())                        =>  NIL

* (REST L), (CDR L)
      (rest '(a b c))                   =>  (B C)
      (rest ())                         =>  NIL

* (SECOND L)
      (second '(a b c d))               =>  B
      (second '(a))                     =>  NIL

* (NTH N L)
      (nth  2 '(a b c d))               =>  C

* (LENGTH L)
      (length '(a (b c) d))             =>  3

* (SUBSEQ L I J)
      (subseq '(a b c d e f g h) 2 5)  => (C D E)
```

Números

- **Existen distintos tipos de números:**
  - **Enteros:** `14`, `0`, `-7`
  - **Racionales:** `4/5`, `-2/1`
  - **Coma flotante:** `4.634`, `-.543`
  - **Complejos:** `#C(3 46)`, `#C(3.2, -5)`

```
> (/ 22 7)
22/7
> (round (/ 22 7))
3 ;
1/7
> (float (/ 22 7))
3.142857
> #c(2 0)
2
```

Funciones aritméticas

```
* (+ X-1 ... X-N)
      (+ 3 7 5)        =>  15

* (- X-1 ... X-N)
      (- 123 7 5)      =>  111
      (- 3)            =>  -3

* (* X-1 ... X-N)
      (* 2 7 5)        =>  70

* (/ X Y)
      (/ 6 2)          =>  3
      (/ 5 2.0)        =>  2.5

* (MOD X Y)
      (mod 7 2)        =>  1

* (EXPT X Y)
      (expt 2 3)       =>  8

* (SQRT X)
      (sqrt 16)        =>  4
```

Valores lógicos: `T` y `NIL`

- **El símbolo `NIL` tiene asociado el valor lógico "falso".**
  - `()` $\equiv$ `'()` $\equiv$ `NIL`
- **El símbolo `T` tiene asociado el valor lógico "verdadero".**
  - **Pero en general, cualquier elemento distinto de `NIL` tiene el valor lógico "verdadero".**
- **Un predicado es una función que devuelve un valor de verdad.**

```
> (listp '(a b c))
T
> (listp 27)
NIL
> (and 3 4 5)
5
```

Predicados aritméticos

- **Predicados aritméticos:**

```
* (= X-1 ... X-N)
      (= 2 2.0 (+ 1 1))  =>  T
      (= 1 2 1)          =>  NIL

* (> X-1 ... X-N)
      (> 4 3 2 1)        =>  T
      (> 4 3 3 2)        =>  NIL

* (>= X-1 ... X-N)
      (>= 4 3 3 2)       =>  T
      (>= 4 3 3 5)       =>  NIL

* (< X-1 ... X-N)

* (<= X-1 ... X-N)
```

Predicados de tipos

```
* (ATOM X)
     (atom 3)           =>  T
     (atom 'hola)       =>  T
     (atom '(1 2 3))    =>  NIL

* (SYMBOLP X)
     (symbolp 'a)       =>  T
     (symbolp 3)        =>  NIL

* (NUMBERP X)
     (numberp 4)        =>  T
     (numberp 3.4)      =>  T
     (numberp '(1))     => NIL

* (CONSP X)
     (consp '(1 . 2))   => T
     (consp nil)        => NIL
     (consp '(2 5))     => T

* (NULL X)
        (null (rest '(a b)))  =>  NIL
        (null (rest '(a)))    =>  T
```

Predicados de igualdad

```
* (EQ X Y)
      (eq 3 3)                                  =>  T
      (eq 3 3.0)                                =>  NIL
      (eq 3.0 3.0)                              =>  NIL
      (eq (first '(a b c)) 'a)                  =>  T
      (eq (cons 'a '(b c)) '(a b c))            =>  NIL

* (EQL X Y)
      (eql 3.0 3.0)                             =>  T
      (eql (cons 'a '(b)) (cons 'a '(b)))       =>  NIL

* (EQUAL X Y)
      (equal (cons 'a '(b)) (cons 'a '(b)))     =>  T
```

Expresiones condicionales

- **La forma especial `IF`:**

```
(IF <condición>
    <consecuencia>
    <alternativa>)
```

- **Mecanismo de evaluación de un `IF` (forma especial):**
  - **Evalúa `<condición>`:**
  - **Si el valor es distinto de `NIL` ("verdadero") devuelve el valor de `<consecuencia>`,**
  - **en caso contrario devuelve el valor de `<alternativa>`.**
- **Ejemplo:** `(if (= y 0) 9999999 (/ x y))`

Expresiones condicionales

- **La forma especial `COND`:**
  `(COND <pares>)`

  - **`<pares>`: sucesión de expresiones de la forma**
    `(<condicion><consecuencias>)`
    siendo `<consecuencias>` una sucesión de expresiones Lisp.

- **Evaluación de un `COND` (forma especial):**

  - **Evalúa los elemento `<condicion>` hasta que aparece un valor distinto de `NIL` ("verdadero").**
  - **Evalúa la correspondiente `<consecuencias>`. Evalúa cada una de las expresiones que la componen, devolviendo como el valor el último obtenido.**
  - **Ejemplo:**

```
(cond
 ((< x 3) (format t "Es pequeño") x)
 ((< x 5) (format t "Es mediano") (* x 2))
 (t (* x x)))
```

Condicionales

```
* (IF TEST ENTONCES [EN-CASO-CONTRARIO])
      (if (> 2 1)   1 2)     =>  1
      (if (> 1 2) 1)         =>  NIL

* (WHEN TEST E-1 ... E-N)
      (when (= 1 1) 1 2 3)  =>  3
      (when (= 1 2) 1 2 3)  =>  NIL

* (UNLESS TEST E-1 ... E-N)
      (unless (= 1 1)   1 2 3)  =>  NIL
      (unless (= 1 2) 1 2 3)  =>  3

* (COND L-1 ... L-N)
      > (defun notas (n)
          (cond ((< n 5) 'suspenso)
                ((< n 7) 'aprobado)
                ((< n 9) 'notable)
                (t 'sobresaliente) ))
      NOTAS
      > (notas 8)
      NOTABLE
```

Operadores lógicos

```
* (NOT X)
      (not (= (+ 1 1) 2))          =>  NIL
      (not (= (+ 1 1) 3))          =>  T

* (OR E-1 ... E-N)
      (or nil 2 3)                 =>  2
      (or (eq 'a 'b) (eq 'a 'c))   =>  NIL

* (AND E-1 ... E-N)
      (and 1 nil 3)                =>  NIL
      (and 1 2 3)                  =>  3
```

Predicado de pertenencia y listas de asociación

- **Pertenencia:**

```
* (MEMBER E L [:TEST #'PREDICADO])
        (member 'x '(a x b x c))          => (X B X C)
        (member 'x '(a (x) b))            => NIL
        (setf l '((a b) (c d)))           => ((A B) (C D))
        (member '(c d) l)                 => NIL
        (member 2.0 '(1 2 3))             => NIL
        (member '(c d) l :test #'equal)   => ((C D))
        (member 2.0 '(1 2 3) :test #'=)   => (2 3)
        (member 2.0 '(1 2 3) :test #'<)   => (3)
```

- **Listas de asociación:**

```
* (ASSOC ITEM A-LISTA [:TEST PREDICADO])
        (assoc 'b '((a 1) (b 2) (c 3)))                    =>  (B 2)
        (assoc '(b) '((a 1) ((b) 1) (c d)))                =>  NIL
        (assoc '(b) '((a 1) ((b) 1) (c d)) :test #'equal)  =>  ((B) 1)
```

Variables locales

```
* (LET ((VAR-1 VAL-1)...(VAR-M VAL-M)) E-1 ... E-N)
        (setf a 9 b 7)                        =>  7
        (let ((a 2)(b 3)) (+ a b))            =>  5
        (+ a b)                               =>  16
        (let ((x 2)(y (+ 1 x))) (+ x y))      =>  Error

* (LET* ((VAR-1 VAL-1) ... (VAR-N VAL-N)) E-1 ... E-M)
        (let* ((x 2)(y (+ 1 x))) (+ x y))     =>  5
```

Definiendo funciones

- **La forma especial** `DEFUN`

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

  - `<nombre>`**: símbolo al que se asocia la función.**
  - `<parámetros>`**: sucesión de símbolos a los que se asignará, temporalmente, el valor de los argumentos con los que se utilice la función.**
  - `<documentación>`**: cadena en la que se describe la función. Es opcional.**
  - `<cuerpo>`**: sucesión de expresiones que se evalúan cuando se utiliza la función.**

Definiendo funciones

- **Ejemplo:**

```
> (defun cuadrado (x) "Cuadrado de un numero" (* x x))
CUADRADO
```

- **`(cuadrado 4)` se evalúa a `16`:**
  - **Se evalúa `cuadrado`: función definida por el usuario.**
  - **Se evalúa `4`: `4`**
  - **Se asocia al parámetro de `cuadrado`, `x`, el valor `4`.**
  - **Se evalúa la expresión que compone el cuerpo de `cuadrado`:**
  - **Se evalúa `*`: función primitiva de Lisp para calcular el producto.**
  - **Se evalúa `x`: `4`**
  - **Se evalúa `x`: `4`**
  - **Se calcula el producto con argumentos `4` y `4`: `16`**
  - **Se elimina la asociación del parámetro.**

Definiciones recursivas

- **Recursión**

```
> (defun factorial (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))))
FACTORIAL
> (factorial 3)
6
```

Traza de una recursión

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

- **En general, `TRACE` (y `UNTRACE`) son herramientas útiles para depuración**

Patrones comunes de recursión

- **Descendente numérico:**

```
(defun f
  (... n ...)
  (if (= 0 n)
      <...>
      <... (f ... (- n 1) ...) ...>))
```

- **Ejemplo:**

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

Patrones comunes de recursión

- **Listas lineales:**

```
(defun f
  (... l ...)
  (if (endp l)
      <...>
      <... (first l)
           (f ... (rest l) ...) ...>))
```

- **Ejemplo:**

```
(defun suma-lista
  (l)
  (if (endp l)
      0
      (+ (first l)
         (suma-lista (rest l)))))
```

Ejemplos de recursión en Lisp

- **Ejemplo: función** `subconjunto`

```
;;; (SUBCONJUNTO L1 L2)
;;; > (subconjunto () '(s 3 e 4))
;;; T
;;; > (subconjunto '(4 3) '(s 3 e 4))
;;; T
;;; > (subconjunto '(4 a 3) '(s 3 e 4))
;;; NIL

(defun subconjunto (l1 l2)
  (if (endp l1)
      t
    (and (member (first l1) l2)
         (subconjunto (rest l1) l2))))
```

Ejemplos de recursión en Lisp

- **Ejemplo: función** `elimina-uno`

```
;;; (ELIMINA-UNO X L)
;;; > (elimina-uno 3 '(a b c d))
;;; (A B C D)
;;; > (elimina-uno 'b '(a b c d))
;;; (A C D)
;;; > (elimina-uno 'b '(a b c b d))
;;; (A C B D)

(defun elimina-uno (x l)
  (cond ((endp l) l)
        ((equal x (first l)) (rest l))
        (t (cons (first l) (elimina-uno x (rest l))))))
```

Ejemplos de recursión en Lisp

- **Ejemplo: función** `permutacion`

```
;;; (PERMUTACION L1 L2)
;;; > (permutacion '(x 1 3 4) '(3 2 x 4))
;;; NIL
;;; > (permutacion '(x 2 3 4) '(3 2 x 4))
;;; T
;;; > (permutacion '(x 2 3 4) '(3 2 x 4 4))
;;; NIL

(defun permutacion (l1 l2)
  (cond ((endp l1) (endp l2))
        ((member (car l1) l2)
         (permutacion (rest l1)
                      (elimina-uno (car l1) l2)))
        (t nil)))
```

Ejemplos de recursión en Lisp

- **Recursión cruzada: funciones `par-p` e `impar-p`**

```
;;; (PAR-P N), (IMPAR-P N)
;;; > (par-p 4)
;;; T
;;; > (impar-p 10)
;;; NIL

(defun par-p (n)
  (cond  ((= n 0) t)
         ((= n 1) nil)
         (t (impar-p (- n 1)))))

(defun impar-p (n)
  (cond ((= n 0) nil)
        ((= n 1) t)
        (t (par-p (- n 1)))))
```

Ejemplos de recursión en Lisp

- **Ejemplo: función** `comprime`

```
;;; (COMPRIME LISTA)
;;; > (comprime '(a a b c c c a d d d d d e))
;;; ((2 A) B (3 C) A (5 D) E)
;;; > (comprime '(a b c c a d e))
;;; (A B (2 C) A D E)

(defun comprime (l)
  (if (consp l) (comprime-aux (first l) 1 (rest l)) l))

(defun comprime-aux (x n l)
  (if (endp l)
      (list (n-elementos x n))
    (let ((siguiente (car l)))
      (if (eql x siguiente)
          (comprime-aux x (+ n 1) (rest l))
        (cons (n-elementos x n)(comprime-aux siguiente 1 (rest l)))))))

(defun n-elementos (x n) (if (> n 1) (list n x) x))
```

El operador básico de iteración: DO

- **Ejemplo:**

```
(defun lista-cuadrados (ini fin)
       (do ((i ini (1+ i)))
           ((> i fin) 'terminado)
         (format t "~a ~a~%" i (* i i))))
```

  - **La primera parte del `do` especifica las variables del bucle (`i` en este caso), su valor inicial (`ini`) y cómo se actualiza en cada iteración (se incrementa en uno)**
  - **La segunda parte especifica la condición de parada (`(>i fin)`) y lo que se devuelve (`'terminado`)**
  - **Finalmente, la expresión final especifica qué hacer en cada iteración (escribir una línea con `i` y con el cuadrado de `i`)**

El operador básico de iteración: DO

- **Sesión:**

```
> (lista-cuadrados 1 5)
1 1
2 4
3 9
4 16
5 25
TERMINADO
```

- **Otro ejemplo, en el que se manejan varias variables**

```
(defun factorial (n)
       (do ((j n (1- j))
            (f 1 (* j f)))
           ((= j 0) f)))
```

  - **Nota: no hay asignaciones explícitas**

El operador básico de iteración: DO

- **Otro ejemplo:**

```
;;; (ESPECULAR LISTA)
;;; > (especular '(a b c c b a))
;;; T
;;; > (especular '(a b c g h h g c d a))
;;; NIL

(defun especular (l)
       (let ((long (length l)))
         (and (evenp long)
              (do ((del 0 (1+ del))
                   (atr (1- long) (1- atr)))
                  ((or (> del atr)
                       (not (eq (nth del l) (nth atr l))))
                   (> del atr))))))
```

Las macros DOLIST y DOTIMES

- **Ejemplo (con `dolist` y `return`):**

```
> (dolist (x '(a b c d e))
    (format t "~a " x)
    (if (eq x 'c) (return 'ya)))
A B C
YA
```

- **Ejemplo (con `dotimes`):**

```
> (dotimes (x 5 x)
     (format t "~a " x))
0 1 2 3 4
5
```

Bucles con LOOP

- **Ejemplos de uso de LOOP:**

```
> (loop for x from 1 to 7 collect (* x x))
(1 4 9 16 25 36 49)
> (loop for x from 1 to 7
        when (evenp x)
        collect (* x x))
(4 16 36)
> (loop for x from 1 to 7
        when (evenp x)
        summing (* x x))
56
> (loop for x from 1 to 7 by 2 collect (* x x))
(1 9 25 49)
> (loop for x in '(1 3 5) summing x)
9
> (loop for l in
          '((a b c) (d e f) (g h i))
          append l)
 (A B C D E F G H I)
```

Bucles con LOOP

- **Ejemplos:**

```
> (defun factor (x)
    (or (loop for i from 2 to (sqrt x)
              thereis (when (= (mod x i) 0)
                            i))
        x))
FACTOR
> (factor 35)
5
> (defun es-primo (x)
    (= x (factor x)))
ES-PRIMO
> (es-primo 7)
T
> (es-primo 35)
NIL
> (loop for x from 2 to 100
        count (es-primo x))
25
```

Bucles con LOOP

```
> (defun primos-gemelos (x)
    (when (and (es-primo x)
               (es-primo (+ x 2)))
          (list x (+ x 2))))
PRIMOS-GEMELOS
> (loop for i from 200 to 2000
        thereis (primos-gemelos i))
(227 229)
> (loop for x from 2 to 100
        count (primos-gemelos x))
8
```

Bucles con LOOP

- **Opciones iniciales:**

```
(LOOP FOR <VARIABLE> FROM <INICIO> TO <FIN> ...)
(LOOP FOR <VARIABLE> FROM <INICIO> TO <FIN>
                     BY <INCREMENTO> ...)
(LOOP FOR <VARIABLE> IN <LISTA> ...)
(LOOP WHILE <CONDICION> ...)
(LOOP UNTIL <CONDICION> ...)
```

- **Opciones centrales:**

```
(LOOP ... WHEN <CONDICION> ...)
```

- **Opciones finales:**

```
(LOOP ... DO <EXPRESION>)
(LOOP ... COLLECT <EXPRESION>)
(LOOP ... APPEND <EXPRESION>)
(LOOP ... THEREIS <EXPRESION>)
(LOOP ... COUNT <EXPRESION>)
(LOOP ... SUMMING <EXPRESION>)
```

Escritura

- **La función** `format`**:**

```
(FORMAT DESTINO CADENA-DE-CONTROL X-1 ... X-N)
```

```
> (format t "~&Linea 1 ~%Linea 2")
Linea 1
Linea 2
NIL
> (format t "~&El cuadrado de ~a es ~a" 3 (* 3 3))
El cuadrado de 3 es 9
NIL
> (setf l '(a b c))
(A B C)
> (format t
        "~&La longitud de la lista ~a es ~a"
        l (length l))
La longitud de la lista (A B C) es 3
NIL
```

Escritura

- **Algunas directivas de la función `format`:**

```
~a      escribe el siguiente argumento

~&      comienza nueva línea,
        si no está al comienzo de una

~%      comienza nueva línea

~Na     escribe el siguiente argumento
        más los espacios suficientes para
        ocupar N caracteres de ancho
```

Lectura

- **La función `read`**

  `(READ)`

```
> (read)
a
A
> (setf a (read))
2
2
> a
2
> (* (+ (read) (read))
3
4
7
> (read)
(+ 2 3)
(+ 2 3)
```

Lectura

- `read` **no evalúa. Función** `eval`**:**

```
(EVAL EXPRESION)

  > (eval (read))
  (+ 2 2)
  4
```

- **Ejemplo:**

```
(defun calcula-cuadrados ()
  (loop
   (let ((aux (read)))
     (if (numberp aux)
         (format t "~&El cuadrado de ~a es ~a~%"
                   aux (* aux aux))
         (return 'fin)))))
```

Matrices

- **Creación:**

```
(MAKE-ARRAY DIMENSIONES  [:INITIAL-CONTENTS EXPRESION])
```

```
> (make-array '(2 2))
#2A((NIL NIL) (NIL NIL))
> (make-array '(2 2 1))
#3A(((NIL) (NIL)) ((NIL) (NIL)))
> (make-array  '(2 2) :initial-element 2)
#2A((2 2) (2 2))
> (make-array '(3 3)
              :initial-contents '((a b c)
                                  (1 2 3)
                                  (x y z)))
#2A((A B C) (1 2 3) (X Y Z))
> (setf *matriz*
    (make-array '(3 3)
                :initial-contents '((a b c)
                                    (1 2 3)
                                    (x y z))))
#2A((A B C) (1 2 3) (X Y Z))
```

Matrices

- **Acceso:**

```
(AREF MATRIZ INDICE-1 ... INDICE-N)
```

```
> *matriz*
#2A((A B C) (1 2 3) (X Y Z))
> (aref *matriz* 0 0)
A
> (aref *matriz* 1 1)
2
> (aref *matriz* 2 2)
Z
```

- **Modificación:**

```
(SETF (AREF MATRIZ INDICE-1 ... INDICE-N) EXPRESION)
```

```
> *matriz-2*
#2A((1 2 3) (8 H 4) (7 6 5))
> (setf (aref *matriz-2* 1 2) 'h)
H
> (setf (aref *matriz-2* 1 1) 4)
4
> *matriz-2*
#2A((A B C) (1 4 H) (X Y Z))
```

Matrices

- **Ejemplo**

```
;;; (SUMA-COLUMNAS MATRIZ)
;;; > (setf mat
;;;       (make-array '(3 3)
;;;                   :initial-contents
;;;                   '((1 2 3) (4 5 6) (7 8 9))))
;;; #2A((1 2 3) (4 5 6) (7 8 9))
;;; > (suma-columnas mat)
;;; #(12 15 18)

(defun suma-columnas (a)
  (let* ((dim (array-dimensions a))
         (f (first dim))
         (c (second dim))
         (res (make-array (list c))))
    (loop for i from 0 to (- c 1)
          do (setf (aref res i)
                   (loop for j from 0 to (- f 1)
                         summing (aref a j i))))
    res))
```

Estructuras (ejemplo)

```
> (defstruct persona
    (nombre nil)
    (estado 'casado)
    (calle  nil)
    (ciudad 'Sevilla))
PERSONA
> (setf ejemplo-1
        (make-persona :nombre 'ana :calle '(Reina Mercedes)))
#S(PERSONA :NOMBRE ANA :ESTADO CASADO :CALLE (REINA MERCEDES) :CIUDAD SEVILLA)
> (setf ejemplo-2
        (make-persona :nombre 'pepe :ciudad 'Huelva))
#S(PERSONA :NOMBRE PEPE :ESTADO CASADO :CALLE NIL :CIUDAD HUELVA)
> (persona-ciudad ejemplo-1)
SEVILLA
> (persona-nombre ejemplo-2)
PEPE
> (setf (persona-nombre ejemplo-1) '(Ana Maria))
(ANA MARIA)
> ejemplo-1
#S(PERSONA :NOMBRE (ANA MARIA) :ESTADO CASADO
           :CALLE (REINA MERCEDES) :CIUDAD SEVILLA)
```

Estructuras (ejemplo)

```
> (setf ejemplo-3 ejemplo-1)
#S(PERSONA :NOMBRE (ANA MARIA) :ESTADO CASADO
           :CALLE (REINA MERCEDES) :CIUDAD SEVILLA)
> (setf (persona-calle ejemplo-3) '(tetuan))
(TETUAN)
> ejemplo-3
#S(PERSONA :NOMBRE (ANA MARIA) :ESTADO CASADO
           :CALLE (TETUAN) :CIUDAD SEVILLA)
> ejemplo-1
#S(PERSONA :NOMBRE (ANA MARIA) :ESTADO CASADO
           :CALLE (TETUAN) :CIUDAD SEVILLA)
> (setf ejemplo-4 (copy-persona ejemplo-2))
#S(PERSONA :NOMBRE PEPE :ESTADO CASADO :CALLE NIL :CIUDAD HUELVA)
> (setf (persona-ciudad ejemplo-4) 'cadiz)
CADIZ
> ejemplo-4
#S(PERSONA :NOMBRE PEPE :ESTADO CASADO :CALLE NIL :CIUDAD CADIZ)
> ejemplo-2
#S(PERSONA :NOMBRE PEPE :ESTADO CASADO :CALLE NIL :CIUDAD HUELVA)
```

Estructuras

- **Creación de estructuras con `defstruct`:**

```
(DEFSTRUCT (NOMBRE (:CONSTRUCTOR FUNCION-CONSTRUCTURA)
                   (:CONC-NAME PREFIJO-)
                   (:PRINT-FUNCTION FUNCION-DE-ESCRITURA))
  CAMPO-1
  ...
  CAMPO-N)
```

- **Ejemplo (puntos en el plano):**

```
(defstruct (punto (:constructor crea-punto)
                  (:conc-name coordenada-)
                  (:print-function escribe-punto)
   x
   y)
```

- **Ejemplo (función de escritura):**

```
(defun escribe-punto (punto &optional (canal t) profundidad)
  (format canal "Punto de abcisa ~a y ordenada ~a"
          (coordenada-x punto)
          (coordenada-y punto)))
```

Estructuras (ejemplo)

```
> (setf *punto-1* (crea-punto :x 2 :y 3))
Punto de abcisa 2 y ordenada 3
> (coordenada-y *punto-1*)
3
> (setf (coordenada-y *punto-1*) 5)
5
> *punto-1*
Punto de abcisa 2 y ordenada 5
> (punto-p *punto-1*)
T
> (punto-p '(2 5))
NIL
```

Estructuras (ejemplo)

```
> (setf *punto-2* (copy-punto *punto-1*))
Punto de abcisa 2 y ordenada 5
> (equal *punto-1* *punto-2*)
NIL
> (equalp *punto-1* *punto-2*)
T
> (setf (coordenada-y *punto-2*) 3)
3
> *punto-2*
Punto de abcisa 2 y ordenada 3
> (setf *punto-3* (crea-punto :y 3 :x 2))
Punto de abcisa 2 y ordenada 3
> (equalp *punto-2* *punto-3*)
T
```

Listas, matrices y tablas hash

- **Las matrices no son tan flexibles como las listas, pero permiten acceso rápido a sus componentes y ahorran espacio**
  - **Ventajas: acceso y actualización en tiempo constante**
  - **Inconveniente: al crear un array necesitamos reservar el espacio de antemano**
- **Una buena solución que permite utilización dinámica del espacio y acceso y actualizaciones rápidas son las tablas hash**
  - **Las tablas hash almacenan asociaciones `(clave,valor)`**
  - **La clave puede ser de cualquier tipo (números, símbolos, funciones,...)**

Tablas hash

- **Creación (MAKE-HASH-TABLE):**

```
> (setf *tabla* (make-hash-table))
#S(HASH-TABLE EQL)
```

- **Acceso y actualización (GETHASH):**

```
> (gethash 'color *tabla*)
NIL ;
NIL
> (setf (gethash 'color *tabla*) 'rojo)
ROJO
> (setf (gethash 'talla *tabla*) 'grande)
GRANDE
> (gethash 'color *tabla*)
ROJO ;
T
```

Tablas hash

- **Borrado de asociaciones (REMHASH):**

```
[13]> (gethash 'talla *tabla*)
GRANDE ;
T
[14]> (remhash 'talla *tabla*)
T
[15]> (gethash 'talla *tabla*)
NIL ;
NIL
```

- **Iterando sobre tablas hash (MAPHASH):**

```
> (setf (gethash 'forma *tabla*) 'redonda
(gethash 'precio *tabla*) 150)
150

> (maphash #'(lambda (k v) (format t "~a = ~a~%" k v)) *tabla*)
PRECIO = 150
FORMA = REDONDA
COLOR = ROJO
NIL
```

Argumentos clave y opcionales

- **Argumentos opcionales**

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

(factorial 3)                              => 6
(factorial 3 4)                            => 24
```

- **Argumentos clave**

```
(defun f (&key (x 1) (y 2))
   (list x y))
(f :x 5 :y 3)                              =>  (5 3)
(f :y 3 :x 5)                              =>  (5 3)
(f :y 3)                                   =>  (1 3)
(f)                                        =>  (1 2)
```

Valores de un símbolo

- **Hemos visto:**

```
DATOS EN LISP:
- Números: 3, 2.5, 14.3, ...
- Cadenas: "hola", "el perro come", ...
- Símbolos: adios, fun, fact, cuadrado, ...
- Listas: (1 s 3), (1 (a (b))), ...
- Estructuras
- Matrices, etc.
```

- **Un nuevo tipo de dato: funciones.**
  - **Expresadas mediante** `lambda`

```
(lambda (x) (* x x))

(lambda (x y) (cond ((> x 0) 1)
                    ((< x 0) -1)
                    (t 0)))
```

Valores de un símbolo

- **Estos datos funcionales se pueden asignar a símbolos:**

```
> (setf (symbol-value 'cuadrado) 8)
8
> (setf (symbol-function 'cuadrado)
        (lambda (x) (* x x)))
#<CLOSURE :LAMBDA (X) (* X X)>
```

- **Usualmente:**

```
> (setf cuadrado 8)
8
> (defun cuadrado (x) (* x x))
CUADRADO
```

Evaluación en Lisp

- **Constantes que no sean listas: el valor representado**

  `Ejemplos: 1, 3, "hola"`

- **Lambdas: la función representada**

  `Ejemplo: (lambda (x) (* 2 x))`

- **Símbolos (sin quote): su valor como *variable***

  `Ejemplo: cuadrado => 8`

Evaluación en Lisp (continuación)

- **Listas** `(E1 E2 ... En)`

```
* Se evalúa el VALOR FUNCIONAL de E1.
   - Si es un símbolo: valor funcional.
   - Una lambda-expresión: la función que representa.
* Se evalúan E2 ... En (los argumentos) recursivamente.
* Se "aplica" la función obtenida en primer lugar a los
  valores de los argumentos.

  Ejemplo: (cuadrado 4) => 16
           (cuadrado cuadrado) => 64
           ((lambda (m n) (+ m n))
                    (cuadrado 2) cuadrado)  =>  12
```

- **Quote** `('Exp)`**: el valor representado por** `Exp`

```
Ejemplo: '(cuadrado 4) => (cuadrado 4)
```

- **La función** `function` **(**`#'Exp`**): valor funcional de** `Exp`

```
Ejemplos: #'cuadrado => #<CLOSURE :LAMBDA (X) (* X X)>
          #'(lambda (x) (* 2 x)) =>
                        #<CLOSURE :LAMBDA (X) (* 2 X)>
```

Funciones como argumentos de entrada: FUNCALL y APPLY

- **La función** `FUNCALL`

```
(FUNCALL FN E-1 ... E-N)

  (funcall #'+ 1 2 3)                    => 6
  (funcall #'cuadrado 4)                 => 16
  (funcall #'(lambda (x) (* 2 x)) 3)     => 6
```

- **La función** `APPLY`

```
(APPLY FN ARGS)

  (apply #'+ '(1 2 3))                          => 6
  (apply #'max '(4 5 7))                        => 7
  (apply #'(lambda (x y) (* 2 x y)) '(3 4))     => 24
```

- **Observaciones:**
  - **El primer argumento de** `funcall` **y de** `apply` **debe ser algo cuyo valor sea una función**
  - **Uso de** `#'` **para forzar a obtener el valor funcional**
  - **A veces es necesario** `symbol-function`

Funciones como argumentos de entrada (otras)

```
* (MAPCAR FN L)

  (mapcar #'(lambda (x) (* x x)) '(1 2 3 4 5))   => (1 4 9 16 25)
  (mapcar #'atom '(a (b) 3))                     => (T NIL T)

* (REMOVE-IF-NOT PREDICADO L), (REMOVE-IF PREDICADO L)

  (remove-if-not #'atom '((a) b (c) d))          => (B C)
  (remove-if #'atom '((a) b (c) d))              => ((A) (C))

* (COUNT-IF PREDICADO L), (COUNT-IF-NOT PREDICADO L)

  (count-if #'atom '((a) b (c) d e))             => 3
  (count-if-not #'atom '((a) b (c) d e))         => 2

* (FIND-IF PREDICADO L), (FIND-IF-NOT PREDICADO L)

  (find-if #'atom '((a) b (c) d e))              => B
  (find-if-not #'atom '((a) b (c) d e))          => (A)
```

Funciones como argumentos de entrada (otras)

* (POSITION-IF PREDICADO L), (POSITION-IF-NOT PREDICADO L)

```
(position-if #'atom '((x) b d (a) (c)))      => 1
(position-if-not #'atom '(b d (a) (c)))      => 2
```

* (FIND-IF PREDICADO LISTA), (FIND-IF-NOT PREDICADO LISTA)

```
(find-if #'atom '((a) b (c) d e))              => B
(find-if-not #'atom '((a) b (c) d e))          => (A)
```

Funciones como argumentos de entrada (otras)

```
* (EVERY PREDICADO L), (EVERY PREDICADO L1 ... LN)

  (every #'atom '(1 a))                          => T
  (every #'atom '((1) a))                        => NIL
  (every #'<= '(1 2) '(1 3))                     => T
  (every #'< '(1 2) '(1 3))                      => NIL

* (SOME PREDICADO L), (SOME PREDICADO L1 ... LN)

  (some #'atom '((1) a))                         => T
  (some #'atom '((1) (a)))                       => NIL
  (some #'< '(1 2) '(1 3))                       => T
  (some #'< '(1 2) '(1 0))                       => NIL
```

Bibliografía

- Graham, P. ***ANSI Common Lisp*** (Prentice Hall, 1995).
- Norvig, P. ***Paradigms of Artificial Intelligence Programming: Cases Studies in Common Lisp*** (Morgan Kaufmann, 1992).
- Seibel, P. ***Practical Common Lisp*** (Apress).
  `http://www.gigamonkeys.com/book/`
- Steele, G.L. ***Common Lisp the Language, 2nd edition*** (D. P., 1990).
  `http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/-ai/html/cltl/cltl2.html`
- Touretzky, D. ***Common Lisp: A Gentle Introduction to Symbolic Computation*** (Benjamin-Cummings Pub Co).
  `http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/user/dst/www/LispBook/`
- Winston, P.R. y Horn, B.K. ***LISP (3a. ed.)*** (Addison–Wesley, 1991).