# El lenguaje LISP

"LISP is an easy language to learn"
(Henry *et.al.,* 1984)

# Introducción

- Origen: 1958
- John McCarthy, pionero en IA, empezó a trabajar con la primera implementación de Lisp en 1958.
  - » McCarthy, *Recursive functions of symbolic expressions and their computation by machine*, Communications of the ACM, Vol 3, No 4, 1960.

# Introducción

- Uso habitual en Inteligencia Artificial
- LISt Processing
- Características básicas
  - » Lenguaje Interpretado
    - – También posible compilado
  - » Todas las variables son punteros
  - » Liberación automática de memoria (*automatic garbage collection*)
  - » Eficiencia menor que otros lenguajes (causas: es interpretado, liberación automática de memoria, ...).
  - » Adecuado para prototipados y cálculo simbólico.
- Nuevo paradigma de programación: *programación funcional*

# Datos

- Expresiones LISP:
  - » Átomos
    - – Símbolos:
      - simbolo, simbolo3,ejemplo-de-simbolo
      - Especiales: nil, t
    - – Keywords (étiquetas):
      - :ejemplo, :keyword
    - – Strings (distinción minúsculas-mayúsculas)
      - “abc”, “ABC”, “string”
    - – Números:
      - enteros: 33, 4, -5
      - racionales: -3/5, 4/8
      - reales: 3.23, -4.78, 3.33E45
  - » Listas
    - – Colecciones de expresiones
    - – (a b (c (d)) e), (), (3 “abc”)

Listas: Grupo de átomos. Las listas también se pueden agrupar en niveles superiores.

# Evaluación

- Forma LISP: cualquier expresión evaluable (expresión LISP)
- Evaluación
  - » Una expresión es evaluable cuando devuelve un valor
- Distinción entre operadores
  - » Funciones: evalúan todos sus argumentos
  - » Macros: no evalúan todos los argumentos
    - – QUOTE
      - Macro: no evalúa su argumento

        `(quote (a b c)) >>> (A B C)`
      - ‘(a b c) equivale a (quote (a b c))
- Ejemplos de evaluaciones
  - » Átomos
    - – Símbolos:
      - Su evaluación es su valor como variable
      - T y NIL se evalúan a sí mismos
    - – Números, Strings y Keywords
      - Se evalúan a sí mismos
  - » Listas
    - – () se evalúa a NIL
    - – Una lista no vacía evaluable debe tener como primer elemento un símbolo
      - (+ 3 (+ 2 5))
      - (quote (a b c))
      - (setf variable (+ 3 4))

Notación prefijo. El primer elemento de una lista indica qué se desea hacer

# Operadores de manipulación de listas, I

- El macro SETF admite como primer argumento:
  - » Un símbolo:
    - (setf dias '(lunes martes miercoles))
  - » Una posición de memoria:
    - (setf (second dias) 'jueves))
    - dias >>> (LUNES JUEVES MIERCOLES)

      Atención! Lista modificada!
    - (setf (rest dias) '(viernes domingo))
    - dias >>> (LUNES VIERNES DOMINGO)
    - (setf (rest dias) dias) >>> (LUNES LUNES ....)
      - ¡Cuidado!, listas circulares

(setf [<var1> <valor1>] ... [<varn> <valorn>])

Sirve para:
- **Asignar** valores a variables
- **Modificar valores** de componentes de listas, variables, etc.

# Operadores de manipulación de listas, II

- Características comunes:
  - » Al evaluarlos, se obtiene error cuando se aplican sobre datos que no siguen su patrón sintáctico.
  - » La mayoría son funciones (no macros).
- FIRST o CAR
  - » Devuelve el primer elemento de la lista
  - » (first <lista>)
  - » (first ‘(a b c)) >> A
- SECOND, THIRD, ..., NTH
  - » (second <lista>)
    - – (second ‘(a b c)) >>> B
  - » (nth <expresion> <lista>)
    - – (nth 1 ‘(a b c)) >>> B
    - – Devuelve el elemento en la posición <expresion> de la <lista>
    - – Posición inicial: 0
    - – Si <expresion> excede la dimensión de la lista se devuelve NIL

# Operadores de manipulación de listas, III

- REST o CDR
  - » Devuelve la lista sin el primer elemento
  - » (rest <lista>)
  - » (rest ‘(a b c)) >> (B C)
- NTHCDR
  - » (nthcdr <expresion> <lista>)
  - » (nthcdr (+ 1 1) ‘(a b c)) >> (C)
- (car (cdr (car ‘((a b c) d)))) >> B
  - » Es equivalente (cadar ‘((a b c) d))
  - » C_ _ _ R _ = A (first) o D (rest)
- Cómo crear listas?
  - » cons
  - » append
  - » list

# Operadores de manipulación de listas, IV

- CONS
  - » (cons <expresion> <lista>)
  - » Crea una lista cuyo primer elemento es <expresion> y cuyo resto es <lista>
    - – (cons ‘(a b) ‘(c d e)) >>> ((A B) C D E)

(cons ‘a (cons ‘b nil)) : construir una lista, concatenando elementos a una lista vacía

- APPEND
  - » (append <lista>*)
  - » (append ‘(a b) ‘(c d)) >>> (A B C D)
  - » (append '(a b) '(c (d)) '(e f)) >>> (A B C (D) E F)
  - » No destructivo. Crea copia (a primer nivel) de todos los argumentos menos el último
  - » (append ‘a ‘b ‘c) >>> ? ERROR
- LIST
  - » (list <expresion>*)
  - » (list ‘(a b) ‘(c d)) >>> ((A B) (C D))
  - » (list '(a b) '(c (d)) '(e f)) >>> ((A B) (C (D)) (E F))
  - » (list ‘a ‘b ‘c) >>> ? (A B C)

# Operadores de manipulación de listas, V

- Operadores destructivos:
  - » Suelen ser macros
  - » (setf a ‘(a b)); (setf b ‘(c d)); (setf c ‘(e f))
    - – NCONC (append). Función.
      - (nconc a b) >>> (A B C D)
      - a >> (A B C D) ; b >>> (C D)

      Modifica la primera lista
    - – PUSH (cons). Macro
      - (push ‘r b) >>> (R C D)
      - b >>> (R C D)

      Introduce elemento. Modifica la lista
    - – POP. Macro
      - (pop c) >>> E
      - c >>> (F)

      Saca elemento. Modifica la lista

# Operadores de manipulación de listas, VI

- Otras funciones:
  - LAST
    - (last <lista>)
    - (last ‘(a b c)) >>> (C)
    - (last ‘((A B) (C D))) >>> ((C D))

    Devuelve el último elemento (del primer nivel de anidamiento)
  - REVERSE
    - (reverse <lista>)
    - (reverse ‘(a b c)) >>> (C B A)
    - (reverse '((A B) (C D))) >>> ((C D) (A B))

    Da la vuelta al primer nivel de la lista
  - LENGTH
    - (lenght <lista>)
    - (lenght ‘(a b c)) >>> 3

    Longitud de la lista, del primer nivel
  - SUBST
    - (subst <nuevo> <viejo> <lista>)
    - (subst ‘a ‘b ‘(a b c)) >>> (A A C)

    Sustituir con <nuevo> todas las apariciones de <viejo> en <lista>

# Otras funciones LISP, I

- PROGN, PROG1
  - » Permiten escribir instrucciones compuestas (como “{“ y “}” en el lenguaje C)
  - » Evalúan todas las sentencias que contienen y devuelven la última o primera sentencia.
  - » (progn (+ 3 4) (+ 4 5)) >>> 9
  - » (prog1 (+ 3 4) (+ 4 5)) >>> 7

(progn [<sentencia1>]… [<sentenciaN>])

(prog1 <sentencia1> [<sentencia2>]…
[<sentenciaN>])

progn puede no recibir ningun argumento, entoces devuelve NIL
prog1 tiene que recibir al menos una sentencia

# Otras funciones LISP, II

- PRINT
  - » (PRINT <forma>)
  - » Mostrar información al usuario.
  - » Si el valor de “<forma>” lo llamamos “A”, el valor es A. Adicionalmente al ejecutarse, también se imprime A
  - » > (progn (setf x (print ‘A)) (+ 3 4))

    A

    >>> 7
  - » > x

    >>> A
- READ
  - » (SETF variable (READ))
  - » Leer información del usuario

    – CL-USER(1): (setf variable (read))
    – HOLA!  (lo introduce usuario y pulsa intro)
    – HOLA!
    – CL-USER(2): variable
    – HOLA!
    – CL-USER(3):

# Otras funciones LISP, III

- EVAL (en desuso)
  - » Evalúa dos veces una expresión
  - » (eval ‘(+ 3 4)) >> 7
  - » (setf a 'b)
  - » (setf b ‘c)
  - » a >> b
  - » b >> c
  - » **(eval a) >> c**
- Operadores matemáticos
  - » +, -, *, /, ABS
  - » EXPT
    - – (expt 2 4) >>> 16
  - » MAX, MIN
    - – (max 2 3 4 5) >>> 5
  - » FLOAT
    - – (float 1/2) >>> 0.5
  - » ROUND
    - – (round 3.2) >> 3

# Condicionales, I

- El valor lógico de una forma LISP se considera “falso” si su valor LISP es NIL. En caso contrario, el valor lógico es “verdadero”.

Condicionales:

* if * when

* unless *cond

- IF (macro)
  - » Sintaxis:
    - – (if <expresion> <forma1> [forma2])
    - – Si el valor lógico de <expresión> es “verdadero”, devuelve el valor de <forma1>. Si es “falso”, devuelve el valor de <forma2>.
  - » (if (> 5 (+ 2 2))

    (+ 3 3)

    (+ 3 4))

    >>> 6

Si se desea que se evalúen varias formas cuando sea cierta o falsa la condición, qué hacemos?

Usamos progn o prog1

# Condicionales, II

- WHEN (macro)
  - » Sintaxis:
    - – (when <expresion>
      <forma-1> <forma-2> ...<forma-n>)
    - – Si el valor lógico de <expresión> es “verdadero”, ejecuta las formas <forma-1>, <forma-2>, ... <forma-n> y devuelve como valor final el valor de <forma-n>. Si es “falso”, devuelve NIL.
  - » (when (> 5 (+ 2 2))
    (+ 3 3)
    (+ 3 4))
    >>> 7

Se ejecuta todo, no hace falta poner progn o prog1

No hay parte else!

**(when <exp> <forma>+) ⇔ (if <exp> (progn <forma>+))**

# Condicionales, III

- UNLESS (macro)
  - » Sintaxis:
    - – (unless <expresion>
      <forma-1> <forma-2> ...<forma-n>)
    - – Si el valor lógico de <expresión> es “falso”, ejecuta las formas <forma-1>, <forma-2>, ... <forma-n> y devuelve como valor final el valor de <forma-n>. Si es “verdadero”, devuelve NIL.
  - » (unless (> 5 (+ 2 2))
    (+ 3 3)
    (+ 3 4))
    >>> NIL

unless es el contrario a when

Cuando se devuelve NIL, no se ha ejecutado nada en este caso

# Condicionales, IV

- COND (macro)
  - » Sintaxis:
    - – (cond (<condición-1>
      <forma-11> <forma-12> ...<forma-1-a1>)
      (<condición-2>
      <forma-21> <forma-22> ...<forma-2-a2>)
      ...
      (<condición-n>
      <forma-n1>
      <forma-n2> ...<forma-n-an>))
    - – Cuando encuentra una <condicion-i> cuyo valor lógico de es “verdadero”, ejecuta las formas <forma-i1>, <forma-i2>, ... <forma-i-ai> y devuelve como valor final el valor de <forma-i-ai>.
  - » (cond ((> 5 7) (+ 3 4) (+ 4 5))
    ((> 5 3) (+ 5 6) (+ 6 7))
    (t (+ 7 8) (+ 8 9)))
    >>> 13

# Predicados

- Procedimientos que devuelven T o NIL
- Usual calcular el valor lógico de expresiones LISP construidos con ellos.
- Tipos de datos:
  - » (<predicado> <expresion>)
  - » ATOM, STRINGP, NUMBERP, SYMBOLP, LISTP
  - » (atom ‘a) >>> T ; (atom NIL) >>> T
  - » (listp ‘a) >>> NIL ; (listp NIL) >>> T

Para saber de qué tipo es un dato

- Numéricos:
  - » Los operandos deben ser números
  - » >, <, <=, >=, ZEROP, PLUSP, MINUSP, EVENP, ODDP

Hoja 1, ejercicio 3

- Identificador listas vacías:
  - » NULL
    - – ¿Es la lista vacía? .¿Es su valor lógico “falso”?
    - – (null (rest ‘(a))) >>> T

Hoja 1, ejercicio 5

# Predicados de igualdad, I

- EQ, EQL, EQUAL, =
- =
  - » (= <numero> <numero>)
  - » (= 3 (+ 2 1)) >>> T; (= 3 3.0) >>> T
- EQ — Determina si dos variables ocupan la misma posición de memoria
  - » Comprueba igualdad a nivel de punteros
  - » Ejemplos:
    - – (setf x '(a b 2)) ; (setf y '(a b 2)); (setf z x)
    - – (eq x y) >>> NIL ;
    - – (eq (first x) (first y)) >>> T
    - – (eq z x) >>> T
- EQL — Comprueba igualdad de átomos
  - » (eql <exp1> <exp2>)
    - – Es T cuando:
      - i. Cuando <exp1> y <exp2> son átomos y
      - ii. (eq <exp1> <exp2>) se evalúa a T
  - » Ejemplos:
    - 1) (eql (cons 'a nil) (cons 'a nil)) >>> NIL (falla i)
    - 2) (setf x (cons 'a nil)) >>> (A)
      (eql x x) >>> T

# Predicados de igualdad, II

- EQUAL
  - » Comprueba a nivel simbólico si dos expresiones son iguales (al imprimirlas)
  - » (equal x y) >>> T
  - » (equal (first x) (first y)) >>> T
  - » (equal z x) >>> T
  - » (equal 3 3.0) >>> NIL

# Operadores lógicos, I

- AND (macro)
  - » (and <exp1> <exp2> ... <expn>)
  - » Si el valor lógico de todas las <expi> es “verdadero”, devuelve el valor de la última (<expn>). En caso contrario, devuelve NIL.
  - » Cuando encuentra alguna <expi> con valor lógico “falso”, ya no sigue evaluando el resto de las <exp>.
  - » (and (evenp 2) (plusp -3) (print 3)) >>> NIL
- OR (macro)
  - » (or <exp1> <exp2> ... <expn>)
  - » Si el valor lógico de alguna de las <expi> es “verdadero”, devuelve su valor. En caso contrario, devuelve NIL.
  - » Cuando encuentra alguna <expi> con valor lógico “verdadero”, ya no sigue evaluando el resto de las <exp>.
  - » (or (evenp 2) (plusp -3) (print 3)) >>> T

# Operadores lógicos, II

- NOT
  - » (not <exp>)
  - » Si el valor lógico de <exp1> es “verdadero”, devuelve NIL. En caso contrario devuelve T.
  - » (not (oddp 2)) >>> T
  - » (not (list 2 3)) >>> NIL
  - » (not (oddp 3) >>> NIL

Hoja 1, ejercicio 6

# Variables locales y globales, I

- (setf dias ‘(lunes martes miercoles))
  - » Es una variable global (puede ser llamada por el resto de instrucciones)
- Las variables globales en LISP se suelen denominar con “*”:
  - » *dias-de-la-semana*

# Variables locales y globales, II

- Variables locales
  - » LET (macro)
    - – Ejemplo:

```
>(prog1
      (let ((x 'a) (y 'b) z)
         (setf z (list x y))
         (list z z))
      (list 'final 'evaluacion))
  >>> ((A B) (A B))
```

    - – Sintaxis:

```
(let ((<var-1> <valor-inicial-1>)
      (<var-2> <valor-inicial-2>)
       ...
      (<var-n> <valor-inicial-n>))
   <forma-1>
    <forma-2>
   ....
   <forma-m>)
```

LET: asignación de valores en paralelo
LET*: asignación de valores de forma secuencial
Se puede: (let* ( (x 'a) (z x)) …

# Definición de funciones, I

- Ejemplo:

```
>(defun factorial (numero)
    (cond ((= numero 0) 1)
          (t (* numero (factorial (- numero 1))))))
 >>> FACTORIAL
> (factorial 3) >>> 6
```

Si la sintaxis está bien, LISP devuelve el nombre de la función.

- Sintaxis:

```
(defun <nombre-funcion> (<lista-argumentos>)
    [(let/let* …]   para definir variables locales
      <forma-1>
      <forma-2>
      ...
      <forma-n>)
```

- Comentarios:
  - El valor de la función es el de la última forma (<forma-n>). No se puede utilizar “return” para devolver el valor de retorno (como en el lenguaje C).
  - Los argumentos son también variables locales.
  - Valor de un símbolo como variable y como función (una variable y una función pueden tener el mismo nombre).

# Definición de funciones, II

- Argumentos
  - » Número fijo de argumentos:
    - – (defun f (x y) (list x y))
    - – (f ‘a ‘b ‘c) >>> error
  - » Número variable de argumentos (&rest):
    - – Función LIST (ejemplo de función que tiene &rest en su implementación)
    - – (defun f (x y &rest z) (list x y z))
    - – (f ‘a ‘b ‘c ‘d) >>> (A B (C D))
    - – (f ‘a ‘b) >>> ? (A B NIL) z tiene NIL
  - » Argumentos opcionales con nombre y valor por defecto (&key):
    - – (defun f (x y &key (z ‘a) u (v ‘b)) (list x y z u v))
    - – (f ‘a ‘b) >>> (A B A NIL B)
    - – (f ‘a ‘b :z ‘c :v ‘d) >>> (A B C NIL D)

&key (<arg1>[<valor1>]) … [(<argN>[<valorN>])]

# Definición de funciones, III

- Un mismo símbolo se puede utilizar para definir una variable y una función.
  - » (setf f ‘a)
  - » (defun f (x y) (list x y))
- LISP permite referirse a las funciones o bien por su nombre o bien por la expresión que las define (caso lambda)
- LISP manipula las funciones como un tipo más de datos.
  - » Las funciones pueden recibir funciones como argumento
  - » Las funciones pueden devolver como valor de retorno funciones.

Hoja 1, ejercicio 7, 8, 10-13

# Definición de funciones, IV

- Operadores de iteración
  - » DOTIMES
    - – (dotimes (i 3 (list 'final)) (list i i)) >>> (FINAL)
    - – Sintaxis:

      (dotimes (<var> <num-iter> [<valor-retorno])
      <forma-1> ... <forma-n>)
  - » DOLIST
    - – (dolist (i (list ‘a ‘b) (list 'final)) (list i i)) >> (FINAL)
    - – Sintaxis:

      (dolist (<var> <lista> [<valor-retorno])
      <forma-1> ... <forma-n>)
  - » WHILE
    - – Sintaxis:

      (while <condicion> <forma-1> ... <forma-n>)
  - » LOOP
    - – Sintaxis (la más sencilla):

      (loop <forma-1> ... <forma-n>)
  - » En todos: (return <expresion>), abandona la iteración devolviendo <expresion>.

# Definición de funciones, V

- Ejercicios de iteración

```
>> (dotimes (i 3 (list 'final)) (print  i))
   0
   1
   2
   (FINAL)
```

» Cómo sería con while?:

```
(progn
   (let ((i 0))
   (while (< i 3)
       (print i)
      (setf i (+ i 1)))
  (list 'final)))
```

» Cómo sería con loop?

```
(progn
  (let ((i 0))
  (loop (print i) (setf i (+ i 1))
       (if (= i 3)  (return)))
  (list 'final)
))
```

# Definición de funciones, VI

- Funciones lambda (anónimas o sin nombre)
  - » Un subprograma de carácter tan auxiliar que no es necesario darle nombre: lambda.

```
(lambda ( <lista-argumentos> )
        <forma-1>
        <forma-2>
        ...
        <forma-n>)
```

  - » Todos los componentes mismo significado que en defun.

Contar los elementos de una lista que cumplan

```
(count-if
    #'(lambda (x) (eql (first x) 'pantalon) )
    '( (medias 20) (falda 10)
       (pantalon 40) (medias 1)
       (pantalon 2) (total 73 )) )
>>> 2
```

Valores entre los que contar

# Operadores sobre funciones, I

- FUNCALL
  - » (funcall <funcion> <arg1> ... <argn>)
  - » Son equivalentes:

    (f ‘a ‘b) >>> (A B)

    (funcall #’f ‘a ‘b)  Con funcall necesario #’

    function <expresion> ⇔ #’ <expresion>
  - » (progn (setf funcíon-f #’f) (funcall funcíon-f ‘a ‘b))
  - » (funcall #’(lambda (x y) (list x y)) ‘a ‘b)
- APPLY
  - » (apply <funcion> (<arg1> ... <argn>))
  - » Son equivalentes:

    (f ‘a ‘b) >>> (A B)

    (apply #’f ‘(a b))

Diferencia entre funcall y apply?

La forma de cómo se presentan los datos sobre los que se aplica la función, con apply los datos van dentro de ( y )

# Operadores sobre funciones, II

- MAPCAR
  - » (mapcar <funcion> <lista1> ... <listan>)
  - » Aplicación de funciones a elementos de listas

```
> (mapcar #'(lambda (x y) (list x x y y))
                '(a b c)
                '(d e f))
>>> ((A A D D) (B B E E) (C C F F))
```

Qué ocurre si las listas son de distintos tamaños?

Hace lo común

- REMOVE
  - » (remove <elemento> <lista>
    [:test <funcion-igualdad>])
  - » Obtener lista a partir de otra, borrando los elementos que cumplan una condición
  - » (remove 'b '(a b c)) >>> (A C)
    - – Equivale a (remove 'b '(a b c) :test #'eql)
  - » (remove '(a b) '(a (a b) c)) >>> (A (A B) C)
    - – (remove '(a b) '(a (a b) c) :test #'equal) >>> (A C)

Necesario :test para borrar cuando hay sublistas

Hoja 1, ejercicio 9

# Operadores sobre funciones, III

- DELETE
  - » Versión destructiva de REMOVE
  - » (setf lista ‘(a b c))
  - » (remove ‘b lista) >>> (A C)
  - » lista >>> (A B C)
  - » (delete ‘b lista) >>> (A C)
  - » lista >>> (A C)

QUÉ OCURRE:
(setf l '(a (a b) c) )
(delete '(a b) l)
(delete '(a b) l :test #'equal)

- MEMBER
  - » Sintaxis:

    (member <elemento> <lista>
    [:test <funcion-gualdad>])
  - » (member ‘b ‘(a b c)) >>> (B C)
    - – Equivale a (member ‘b ‘(a b c) :test #’eql)
  - » (member ‘(a b) ‘(a (a b) c)) >>> NIL

    >(member ‘(a b) ‘(a (a b) c) :test #’equal)

    >>> ((A B) C)

# Operadores sobre funciones, IV

- Otros operadores cuyos argumentos son funciones:
  - » COUNT-IF, FIND-IF, REMOVE-IF, REMOVE-IF-NOT, DELETE-IF, DELETE-IF-NOT
- Observación:
  - » Al utilizar #'<f>, si <f> es un macro se tienen resultados inesperados. No usarlos, por tanto.

# Funciones sobre strings

- Más corriente el uso de símbolos
- Funciones:
  - » LENGTH, REVERSE
  - » STRING= (sensible a mayúsculas),
    STRING-EQUAL
  - » (read-from-string <string>) >>>
    <simbolo>
    Devuelve el string y su nº de caracteres hasta el primer blanco
  - » (string <simbolo>) >>> <string>
    Convierte simbolo a string
  - » (prin1-to-string <numero>) >>>
    <string>
    Convierte número a string
  - » (search <string1> <string2>)
    Busca parte de un string
  - » (subseq <string> <posicion>) >>>
    substring a partir de <posicion>
  - » (concatenate <string>+)

# Arrays

- Colección n dimensional de elementos. Se puede recorrer con un índice.
- Acceso no secuencial a elementos
  - » En listas, el acceso a los elementos es secuencial
  - » Más eficiente el acceso a elementos en arrays que en listas
- Operadores: MAKE-ARRAY, AREF
  - » > (setf mi-array (make-array '(2 3)))
    >>> #2A((NIL NIL NIL) (NIL NIL NIL))
  - » > (setf (aref mi-array 0 1) 'a)
    >>> A
  - » > mi-array
    >>> #2A((NIL A NIL) (NIL NIL NIL))

Crear array:
(make-array <lista-dimensiones>)
Acceso a los elementos:
(aref <array> <indice1> … <indiceN>)

# Estructuras, I

- Se asocia un nombre a un conjunto de propiedades o componentes (de distintos tipos) a los que se les puede asignar valores.
- Ejemplo:

```
> (defstruct nombre-persona
    nombre
    (alias 'sin-alias)
    apellido)
```

```
(defstruct <nombre-estructura>
  <nombre-componentei>
  (<nombre-componentej> <valorj>)
)
```

```
>>> NOMBRE-PERSONA
> (setf persona1
        (make-nombre-persona :nombre 'juan))
>>> #S(NOMBRE-PERSONA
              NOMBRE JUAN ALIAS SIN-ALIAS
              APELLIDO NIL)
> (setf persona2
         (make-nombre-persona
              :nombre 'pedro :alias 'perico))
>>>  #S(NOMBRE-PERSONA
              NOMBRE PEDRO ALIAS PERICO
              APELLIDO NIL)
```

**make-**nombreestructura :
Crear instancia y dar valores a las partes

# Estructuras, II

nombreestructura-campoestructura :
Acceder a un campo de la estructura

```
> (setf (nombre-persona-alias persona1)
        'juanito)
>>> JUANITO
> persona1
#S(NOMBRE-PERSONA
        NOMBRE JUAN ALIAS JUANITO
        APELLIDO NIL)
```

- Funciones definidas automáticamente por LISP:
  - » MAKE-NOMBRE-PERSONA
  - » NOMBRE-PERSONA -NOMBRE, NOMBRE-PERSONA -ALIAS, NOMBRE-PERSONA -APELLIDO

make-nombreestructura : Crear instancia y dar valores a las partes
(setf <instancia> (<make-nombreestructura> <nomcomponentei><valori> …))

nombreestructura-campoestructura : Acceder a un campo de la estructura
(setf (<nombreestructuracompontei><instancia>)<valori>)

# Alternativa a estructuras

- Las listas son una alternativa al uso de estructuras:

```
> (setf persona1 '((:NOMBRE JUAN)
                   (:ALIAS JUANITO)
                   (:APELLIDO NIL)))
>>> ((:NOMBRE JUAN)
     (:ALIAS JUANITO)
     (:APELLIDO NIL))
> (assoc :alias persona1)
>>> (:ALIAS JUANITO)
> (first
    (member :alias persona1
                :test #'(lambda (x y)
                          (eql x (first y)))))
>>> (:ALIAS JUANITO)
```

- ASSOC utiliza por defecto EQL

# DEBUGGING

- TRACE
  - » Función recursiva:
    - – (defun factorial (n)
      (if (= n 0) 1 (* n (factorial (- n 1)))))
  - » (trace factorial)

```
> (factorial 2)
FACTORIAL [call 5 depth 1] with arg: 2
FACTORIAL [call 6 depth 2] with arg: 1
FACTORIAL [call 7 depth 3] with arg: 0
FACTORIAL [call 7 depth 3] returns value: 1
FACTORIAL [call 6 depth 2] returns value: 1
FACTORIAL [call 5 depth 1] returns value: 2
```

- UNTRACE
  - » (untrace), (untrace factorial)
- Uso de PRINT, DESCRIBE, etc
  - » (describe ‘factorial)
- No es posible uso de “breakpoints”, “steps”, etc

# Otros temas en LISP, I

- Comentarios
  - » Después de ";" y hasta final de línea
- Macros
  - » Permiten definir operadores que no evalúan sus argumentos:
    - – COND, SETF, DEFUN son macros
    - – (defmacro defun
      (nombre-funcion lista-argumentos
      &rest formas)
      .....)

      Hoja 1, ejercicio 14, 15
  - » Uso de backquote (`) y de arroba (@)
    - – Operadores adicionales al quote (')
- Muchas funciones ya implementadas en LISP:
  - » SORT, LOOP generalizado, etc
- CLOS (programación orientada a objetos)

Hoja 2, ejercicios 1,2, 3, 8, 11, 12

# Otros temas en LISP, II

- Dotted-pairs (en desuso)

  > (cons 'a 'b) >>> (A . B)

- Edición, carga compilación:
  - » COMPILE-FILE, LOAD-FILE
- Entrada-salida:
  - » PRINT, READ, FORMAT, READ-LINE, READ-CHAR
- Otros operadores de iteración:
  - » DO, DO* (su diferencia es equivalente a la de LET y LET*)
- (declare (special <variable>))
- Argumentos opcionales (&optional)
- “Property-names” de símbolos
- Copias de listas: COPY-TREE
- Funciones sobre conjuntos: UNION, INTERSECTION, etc. Paquetes