LISP

2/2

SETQ

Liga un valor a un símbolo.

(setq símbolo valor)

```
> (setq x 25)
> (setq var1 :c4)
```

LET

Liga valores locales a uno, varios o ningún símbolo.

(let (ligaduras) expresiones)

> (let ((a 2) (b 4) (c 8)) (+ a (* 3 b) (/ c 4)))

COND

Evalúa expresiones de manera condicional.

(cond (expresión-lógica expresiones*)*)

```
> (let ((a 1) (b 2) (c 3) (d 1))
(cond ((eql a b) 1)
((eql a c) 2)
((eql a d) 3)))
```

QUOTE

- Bloquea la evaluación.
- (quote símbolo) devuelve símbolo
- >(quote (+ 2 3))

CONS

Constructor de un par ordenado. Puede construir listas.

(cons primer-elemento segundo-elemento)

```
> (cons ‘a ‘b)
> (cons ’a ())
```

DEFUN

Sirve para definir una función de usuario y ligarla a un símbolo.

(defun nombre-función lista-de-argumentos cuerpo)

```
(defun pitagoras (a b)
(sqrt (+ (expt a 2.0) (expt b 2.0))))
```

LAMBDA

Devuelve una función sin nombre (sin ligarla a un símbolo).

(lambda lista-de-argumentos cuerpo)

> ((lambda (a b) (+ a b)) 3 4)

STRINGS

Secuencia de caracteres alfanuméricos. Se autoevalúan

> ‘‘mi-cadena’’

PRINT

Convierte una expresión de Lisp en una secuencia de caracteres.

>

(print expresión)

> (print (list 1 2 3))
(1 2 3); lo que imprime en la consola

LOAD

Carga el código fuente contenido en un archivo.

>

(load path-y-nombre-archivo) ; asume la extensión .lsp

> (load ‘‘path/mifichero.lisp’’) T ;
T si lo carga, NIL si no lo carga

Condicionales

IF

(if s_1
s_2
s_3)

Evalúa la sentencia s_1; Si el resultado es diferente de nil ejecuta s_2 y en caso contrario s_3.

El resultado que se arroja es el de la última sentencia evaluada.

Indica si un número es positivo o no
(let ((x -10))
(if (> x 0)
"positivo“
"no positivo"))

WHEN

(when s_1 s_2 ... s_n)

Si la sentencia s_1 devuelve un valor distinto de nil, se ejecutan las instrucciones s_2, ..., s_n.

El resultado que se arroja es el de la última sentencia evaluada.

```
(when (+ 10 20)
(print '(sentencia 1))
(print '(sentencia 2))
'resultado)
```

UNLESS

(unless s_1 s_2 ... s_n)

Si la sentencia s_1 devuelve nil, se ejecutan las instrucciones s_2, ..., s_n

El resultado que se arroja es el de la última sentencia evaluada.

```
(unless (> 5 (+ 2 2))
(+ 3 3)
(+ 3 4))
```

DO

Sirve para poder utilizar varias variables iteradoras a la vez, cada una con su propio incremento.

Su sintaxis es:

```
(do ((lista-inicializacion-1)
(lista-inicializacion-2)
.....
(lista-inicializacion-n))
(condicion (expresiones-verdad))
(expresiones-falso))
```

```
>(DO ((i 3 (- i 1)))
((= i 0)'FIN)
(print i))
```

CASE

(case s $(v_1\ a_1 \ldots a_u)$ $(v_2\ b_1 \ldots b_v)$ ... $(v_m\ z_1 \ldots z_w)$)

Evalúa en primer lugar la sentencia s, Si el resultado es igual a v_i o está contenido en la lista v_i,

Se ejecutan las sentencias que le siguen y termina la ejecución de case devolviendo el resultado de la última sentencia calculada.

Es frecuente utilizar como último valor de comparación el símbolo otherwise para indicar cualquier valor diferente a los especificados más arriba.

```
(let ((e 20))
(case e
(1 "es un uno")
(2 "es un dos")
((3 4 5 6 7 8 9 10)   "es un número grande")
(otherwise "es muuuy grande")))
```

RECURSIVIDAD

- Una función es recursiva cuando se llama a si misma. Una vez que uno se acostumbra a su uso, se comprueba que la recursión es una forma mucho más natural que la iteración de expresar un gran número de funciones y procedimientos.

Función recursiva

Las funciones recursivas se componen de:

- **Caso base:** una solución simple para un caso particular (puede haber más de un caso base).
- **Caso recursivo:** una solución que involucra volver a utilizar la función original, con parámetros que se acercan más al caso base. Los pasos que sigue el caso recursivo son los siguientes:

  1. El procedimiento se llama a sí mismo
  2. El problema se resuelve, tratando el mismo problema pero de tamaño menor
  3. La manera en la cual el tamaño del problema disminuye asegura que el caso base eventualmente se alcanzará

EJEMPLOS

- **FACTORIAL**

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

- **SUMA DE LOS N PRIMEROS NUMEROS**

```
(defun suma (L)
(if (null L) 0 (+ (car L) (suma (cdr L)))))
```

DOLIST

Para iterar sobre los elementos de una lista.

dolist toma una lista de inicialización y un cuerpo.

La lista de inicialización contendrá una variable que tomara los elementos de una lista para cada iteración, la lista de elementos, y una variable que se devolverá como resultado.

DOLIST

(dolist (lista-inicializacion) (cuerpo))

dolist toma una lista de inicialización y un cuerpo.

(dolist (x '(1 2 3)) (print x))

DOTIMES

- Es similar a dolist, solo que la variable iteradora será siempre un numero natural, comenzando por cero.
- (dotimes (lista-inicializacion) (cuerpo))
- Donde (lista-inicialización) tiene la forma: (variable-iteradora valor-máximo resultado)
- El cuerpo se evaluara sucesivamente, comenzando la variable iteradora con un valor de 0 (cero), hasta alcanzar a valor-máximo menos uno.
- (dotimes (x 10) (print x))

READ y WRITE

READ es una operación de lectura de texto

WRITE Escribe objetos que luego pueden ser leídos por read. Strings se escriben con citación doble.

Ejemplo.

```
(defun DobleNumero ()
(print "Ingrese el numero : ")
(setq n1 (read))
(setq doble (* 2.0 n1))
(print "El numero: ")
(write n1)
(print "El doble del numero es : ")
(write doble))
```