



### Importante...

Las macros constituyen un poderoso mecanismo para extender el lenguaje *LISP*.

Típicamente las macros de otros lenguajes (`#define` en C) son sólo un mecanismo para substitución de cadenas en tiempo de compilación. Pero las macros de *LISP*, en cambio, proporcionan un sistema completo para evaluación de código bajo diversas condiciones y en tiempo de ejecución...

Dr. Salvador Godoy Calderón

### La idea fundamental...

Recordemos que, en *LISP*, la lista es el formato universal para representar tanto datos como código...

La diferencia se marca con un **quote** que indica que alguna lista debe ser considerada como lista de datos y por lo tanto no debe ser evaluada...

Sin embargo, si pudiéramos aplicar, o no aplicar, el **quote** a cada elemento de una lista, tendríamos más control sobre la evaluación de una expresión ...

Dr. Salvador Godoy Calderón

## Quote selectivo...

Quote invertido (**back quote**):

Indica que, todo elemento no precedido por una coma, no deberá evaluarse; los elementos precedidos por coma sí serán evaluados...

```
>> (setq x 3)
3
>> `(* ,x ,x)
(* 3 3)
```

OJO:  
back quote

Coma (**comma**):

Su función es la de cancelar el efecto de un apóstrofo, pero sólo para el símbolo que va después de la coma.

Dr. Salvador Godoy Calderón

## Ejemplos...

```
>> (setq x 4  y 3.5)
3.5
```

```
>> '(+ x 10 y) ←
(+ x 10 y)
```

Uso de quote normal:  
Toda la lista es de datos...

```
>> `(+ x 10 y) ←
(+ x 10 y)
```

Uso de quote invertido:  
Mismo efecto que quote normal...

```
>> `(+ ,x 10 ,y) ←
(+ 4 10 3.5)
```

Uso de quote selectivo:  
Las variables toman su valor  
de evaluación...

Dr. Salvador Godoy Calderón

## Algunos casos especiales...

```
>> (setq x 4)
4
>> `x
X
>> 'x
X
>> `,x
4
>> ',x
debugger invoked on a SB-INT:SIMPLE-
READER-ERROR in thread
#<THREAD "main thread" RUNNING {102C65}>:
Comma not inside a backquote.
```

```
>> `x
'X
>> `x
`X
>> `,x
'4
```

Dr. Salvador Godoy Calderón

## Definición de macros...

Las macros son funciones especiales que construyen expresiones simbólicas evaluables e inmediatamente las someten al compilador para ser evaluadas en el ámbito global...

```
>> (defmacro cuadrado (x) `(* ,x ,x))
CUADRADO
```

Su definición es muy semejante a la de una función normal, pero con la etiqueta **defmacro**...

Dr. Salvador Godoy Calderón

## Capacidades...

```
>> (cuadrado 4)
16
>> (cuadrado (+ 6/3 #C(4.2 -2.0)) )
#C(34.44 -24.8)
>> (setq x 12.4)
12.4
>> (cuadrado x )
153.76
>> (cuadrado y )
debugger invoked on a UNBOUND-VARIABLE in thread
#<THREAD "main thread" RUNNING {1002C26653}:
  The variable Y is unbound.
```

Dr. Salvador Godoy Calderón

## Possibilidades...

Una macro, al igual que cualquier expresión con quote selectivo puede generar (expandirse en) cualquier expresión simbólica anidada, incluyendo definición de ámbitos, bloques y estructuras de control algorítmico, así como definición de variables y funciones locales.

El principal uso de macros es para definir estructuras algorítmicas muy frecuentes durante un programa y simplificar la sintaxis de las invocaciones a esa estructura...

Dr. Salvador Godoy Calderón

*Limitaciones...*

- ◆ Las macros no son funciones normales y, por tanto, no se pueden usar como argumento para **mapcar** ni **funcall**.
- ◆ La función **macroexpand** permite conocer la expansión de una macro, es decir, la expresión simbólica que será evaluada...

Dr. Salvador Godoy Calderón

*Error común...*

```
>> (defmacro Cuadrado (x) `(* ,x ,x))
Cuadrado
```

```
>> (Cuadrado (expt 3 2))
81

>> (macroexpand '(Cuadrado (expt 3 2)))
(* (expt3 2) (expt3 2))
t
```

La función **expt** del argumento se evalúa dos veces, cuando debiera evaluarse sólo una vez...

Dr. Salvador Godoy Calderón

**Variables...**

Para solucionar ese problema debemos recordar que, en *LISP* (a diferencia de *C*), la capacidad completa de evaluación de expresiones está disponible en tiempo de compilación, por tanto, podemos usar variables locales:

```
>> (defmacro Cuadrado2 (x) `(let ((tmp ,x))
                                (* tmp tmp)) )
CUADRADO2
```

```
>> (macroexpand '(Cuadrado2 (expt 3 2)))
(let ((tmp (expt 3 2)))
  (* tmp tmp))
t
```

Dr. Salvador Godoy Calderón

**Error...**

Al usar **let** y **let\*** para definir variables locales, debemos asegurarnos que no existe conflicto entre esas variables y otras previamente definidas en el programa ...

```
>> (defmacro Suma-cuadrado (x y)
      `(let* ((uno ,x)
              (dos ,y)
              (result (+ uno dos)))
        (* result result)))
SUMA-CUADRADO
```

```
>> (suma-cuadrado 1 2)
9
```

Dr. Salvador Godoy Calderón

## No funciona...

La macro de la lámina anterior parece no tener problema alguno, sin embargo, las variables locales pueden chocar con variables previamente definidas, ya que la expansión ocurre en tiempo de compilación...

```
>> (setq uno 9)
9

>>(suma-cuadrado 1 uno)
4
!!!
```

Dr. Salvador Godoy Calderón

## Explicación...

El error está en la instancia de **uno** que se toma en cuenta para la expansión...

```
>> (macroexpand '(suma-cuadrado 1 uno))
(let* ((uno 1) (dos uno) (result (+ uno dos)))
  (* result result))
```

T



La variable local **dos** se inicializa con el valor de la variable local **uno**, en lugar del valor de **uno** que se recibió como parámetro...

Dr. Salvador Godoy Calderón

**Solución...**

Es necesario crear variables locales únicas en el programa que se está ejecutando, para ello usamos la función **gensym**...

**gensym** crea un nuevo símbolo, con garantía de nombre único y recibe como argumento opcional un prefijo para el nombre del nuevo símbolo...

```
>>(setq símbolo (gensym) )
#:G781

>>(setq símbolo (gensym "otro-") )
#:|otro-782|
```

Dr. Salvador Godoy Calderón

**Ahora sí funciona...**

Así, podemos usar **gensym** para generar símbolos únicos que sirvan como variables locales de nuestra macro:

```
>>(defmacro suma-cuadrado (x y)
  (let ((uno (gensym "uno-"))
        (dos (gensym "dos-"))
        (result (gensym "result-")))
    `(let* ((,uno ,x)
           (,dos ,y)
           (,result (+ ,uno ,dos)))
      (* ,result ,result))) )
```

**SUMA-CUADRADO**

Dr. Salvador Godoy Calderón

*Expansión...*

Ahora, la expansión de la macro es diferente:

```
>> (setq uno 9)
9

>> (macroexpand '(suma-cuadrado 1 uno))
(let* ((#:|uno-787| 1)
       (#:|dos-788| 9)
       (#:|result-789| (+ #:|uno-787| #:|dos-788|)))
  (* #:|result-789| #:|result-789|))

t

>> (suma-cuadrado 1 uno)
100
```

*Dr. Salvador Godoy Calderón*

*Otro ejemplo...*

Construir un **If-numérico** que evalúa una expresión asumiendo que su resultado es numérico y dispone de tres opciones de flujo: número positivo, negativo y cero:

```
>> (defmacro numIf (expr pos neg zero)
      (let ( (c (gensym "cond-")))
        `(let* ( (,c ,expr) )
           (cond ((plusp ,c) ,pos)
                 ((zerop ,c) ,zero)
                 (t ,neg)))) )
numIf
```

*Dr. Salvador Godoy Calderón*

*Otro ejemplo...*

```
>> (numIf (+ 2 7)
           (print "positivo")
           (print "negativo")
           (print "zero") )
positivo
positivo
```

```
>> (numIf (+ 2 -7)
           (print "positivo")
           (print "negativo")
           (print "zero") )
negativo
negativo
```

Dr. Salvador Godoy Calderón

*Deconstruyendo los argumentos...*

Al formar expresiones con apóstrofo invertido (back-quote), es común manipular listas y no siempre es claro cómo manejar de forma discriminada los elementos de esas listas:

Para lograr el mismo efecto que **cons** se requiere tener acceso a cada elemento de la lista, en lugar de a la lista completa...

```
>> (setq inicio 'a)
A
>> (setq resto '(b c d) )
(B C D)
>> `,(,inicio ,resto) )
( A (B C D))
```

Dr. Salvador Godoy Calderón

*Deconstruyendo los argumentos...*

Cuando colocamos una lista como argumento de una macro se puede tener acceso individual a cada elemento de la lista, siempre y cuando esos parámetros se indiquen dentro de un paréntesis en la invocación de la macro...

```
>> (defmacro suma-terna ((uno dos tres))
`(+ ,uno ,dos ,tres))
```

```
>> (suma-terna (1 2 3))
6
>> (suma-terna (10 100 1000))
1110
```

Dr. Salvador Godoy Calderón

*Un operador para separar...*

Para esas ocasiones en que se desea deconstruir un parámetro lista, contamos también con el operador @ que separa la lista en sus elementos constituyentes...

```
>> (setq inicio 'a)
A
>> (setq resto '(b c d))
(B C D)
>> `,(inicio ,@resto)
(A B C D)
```

```
>> (setq inicio '(a b c))
(A B C)
>> (setq resto '(d e f))
(D E F)
>> `,(,inicio ,@resto)
((A B C) D E F)
>> `,(,@inicio ,@resto)
(A B C D E F)
```

Dr. Salvador Godoy Calderón

*Destructivo y no-destructivo ...*

El operador ,@ es no-destructivo. Si se desea usar un operador destructivo, usar ,.:

```
>> (setq inicio '(a b))
(A B)
>> (setq resto '(c d e))
(C D E)
```

```
>> `(` ,@inicio ,@resto)
(A B C D E)
>> inicio
(A B)
>> resto
(C D E)
```

```
>> `(` ,.inicio ,@resto)
(A B C D E)
>> inicio
(A B C D E)
>> resto
(C D E)
```

Dr. Salvador Godoy Calderón

*¿Cuándo usar y cuándo no?...*

La parte más complicada de las macros es decidir cuándo usarlas y cuándo no usarlas ...

```
>> (defmacro Mas1-AlCuadrado (x)
      (let ((y (gensym)))
        ` (let ((y (+ ,x 1))) (* y y)) )
MAS1-ALCUADRADO
```

```
>> (defun Mas1-AlCuadrado (x)
      (let ((y (+ ,x 1))) (* y y)) )
MAS1-ALCUADRADO
```

Dr. Salvador Godoy Calderón

## Situación...

En el ejemplo anterior la macro, aunque correcta, no es adecuada por las siguientes razones:

- 1) La invocación de la macro y de la función toman exactamente el mismo número de caracteres; por tanto, la ganancia es nula...
- 2) La macro no sirve para mapeos:

```
>> (mapcar #'Mas1-AlCuadrado '(1 2 3))
Error: Mas1-AlCuadrado is not fboundp -- bad arg for function.
[condition type: program-error]
```

Dr. Salvador Godoy Calderón

## Sin embargo...

Las macros tienen mucha utilidad extendiendo el lenguaje que usamos para programar, por ejemplo, en lugar de escribir:

`(lambda (x) ...)`

podríamos sólo escribir:

`(// (x) ...)`

Para ello, basta con programar la macro:

```
>> (defmacro // (&rest expr) `'(lambda ,@expr) )
//
```

Dr. Salvador Godoy Calderón

**Otros usos...**

A algunas personas les incomoda el uso de **mapcar** por considerarlo de obscura sintaxis... en lugar de escribir:

```
>> (mapcar (lambda (x) ...)  
           lista)
```

prefieren escribir:

```
>> (for x :in lista  
        ... )
```

*Dr. Salvador Godoy Calderón*

**Solución...**

Basta con escribir la macro:

```
>> (defmacro for (listaspec exp)  
    (cond ((and (= (length listaspec) 3)  
                (symbolp (first listaspec))  
                (eq (second listaspec) ':in))  
           ` (mapcar (lambda (,(first listaspec)) ,exp)  
                     ,(third listaspec)))  
    (t (error "mal formado: %s" `(for ,listaspec ,exp))))
```

```
>> (for (i :in '(a b c))  
        (print i))  
A  
B  
C  
(A B C)
```

*Dr. Salvador Godoy Calderón*

*Un ciclo con números primos...*

Deseamos escribir una estructura de ciclo, que itere sobre números primos sucesivos...

Necesitamos un predicado que nos diga si un número es primo o no, así como una función que dado un primo regrese el siguiente primo...

```
>> (defun primo? (n)
  (when (> n 1)
    (loop for divisor from 2 to (isqrt n)
          never (zerop (mod n divisor))))) )
```

```
>> (defun sig-primo (n)
  (loop for num from (+ n 1)
        when (primo? num) return num) )
```

*Dr. Salvador Godoy Calderón*

*Construcción de la macro...*

Deseamos construir una macro cicla-primos que permita realizar cualquier tipo de ciclo sobre números primos, por ejemplo,

```
>> (cicla-primos (var 0 15)
  (print var))
2
3
5
7
11
13
NIL
```

Que ciclará una vez por cada número primo mayor o igual a **0** y menor o igual que **15**

*Dr. Salvador Godoy Calderón*

### Solución ...

La macro **cicla-primos** queda de la siguiente forma:

```
>> (defmacro cicla-primos ((var inicio fin) &body cuerpo)
   `(do ( ,var (siguiente-primo ,inicio) (siguiente-primo ,var)))
        ( (> ,var ,fin) )
        ,@cuerpo))
```

```
>> (cicla-primos (i 0 10)
                  (print i))
2
3
5
7
NIL
```

```
>> (cicla-primos (x 1325 1370)
                  (print x))
1327
1361
1367
NIL
```

Dr. Salvador Godoy Calderón

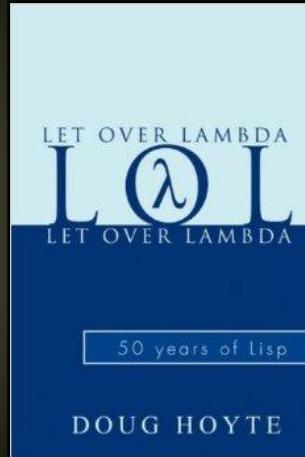
3

### Recursos adicionales...



Consultar...

Doug Hoyte  
*Let Over Lambda*  
50 Years of Lisp (2008)  
Lulu.com



Dr. Salvador Godoy Calderón

Consultar...

*The Common Lisp Cookbook – Macros and Backquote*  
<http://cl-cookbook.sourceforge.net/macros.html>

*Common Lisp Macro Examples*  
<http://c2.com/cgi/wiki?CommonLispMacroExamples>

Dr. Salvador Godoy Calderón

