

INSTITUTO POLITÉCNICO NACIONAL
Centro de Investigación en Computación

*Taller de programación
en Common Lisp*

04

*Recursividad,
mapeos y filtros...*





Dr. Salvador Godoy Calderón



Recursividad...

Recordando...

Un **Conjunto** se puede especificar de tres formas distintas:

- ◆ Por extensión: $A = \{a, b, c, d, e\}$
 $B = \{3, 5, 7, 11, 13\}$
- ◆ Por intención: $C = \{x \in \mathbb{R} \mid 0 \leq x \leq 3.1415\}$
- ◆ Por inducción: Combinación de las dos anteriores...

Potencias de a :

Caso base: $a^0 = 1$

Inducción: $a^{1+n} = a(a^n)$

Factores de a :

Caso base: $a0 = 0$

Inducción: $a(1+n) = a + a(n)$

Dr. Salvador Godoy Calderón

Aclaración...

Formalmente, una especificación por inducción consta de tres partes:

- ◆ *Casos base:* Respuesta final para los casos en que se puede calcular de forma inmediata...
- ◆ *Casos de inducción:* Fórmula que permite calcular nuevos elementos a partir de los elementos ya conocidos...
- ◆ *Caso de cerradura:* Ningún otro elemento es parte del conjunto especificado...

Dr. Salvador Godoy Calderón

Concepto...

A un conjunto especificado de forma inductiva se le llama *Conjunto Inductivo*.

Una función se llama *Recursiva*, si y sólo si, calcula o construye un conjunto inductivo.

Se reconoce una función recursiva por alguna de las siguientes situaciones:

- ◆ Se invoca a sí misma,
- ◆ Invoca a otra función que, a su vez, invoca a la primera
(en la misma rama de procesamiento).

Dr. Salvador Godoy Calderón

Al programar...

Una función recursiva debe seguir la estructura de la especificación por inducción...

Para algunos argumentos de entrada, la respuesta se puede dar de forma inmediata (*casos base*)...

Cuando no es posible caer en algún caso base, se requiere calcular la respuesta, SIEMPRE combinando una parte conocida con otra parte desconocida (*casos de inducción*)...

Dr. Salvador Godoy Calderón

Lo clásico...

Calcular el factorial de un número:

Caso base:

$$1! = 1$$

Inducción:

$$x! = x * (x-1)!$$

```
(defun factorial (x)
  (cond ((= x 1) 1)
        (T (* x (factorial (- x 1))))) )
```

Parte conocida

x

Parte desconocida

$$(x-1)!$$

Forma de combinar
multiplicación

Dr. Salvador Godoy Calderón

Desarrollo...

```
(defun factorial (x)
  (cond ((= x 0) 1)
        (T (* x (factorial (- x 1))))) )
```

```
>> (factorial 4)  
24
```

La invocación (**factorial 4**) se desarrolla así:

valor de x: parte conocida: parte desconocida:

4

$x = 4$

(factorial 3)

$$4 * 6 = 24$$

3

$x = 3$

(factorial 2)

$$3 * 2 = 6$$

2

$x = 2$

(factorial 1)

$$2 * 1 = 2$$

1

respuesta = 1

(caso base)

Dr. Salvador Godoy Calderón

Suma de una lista numérica...

Caso base:

Lista vacía: $(\) \rightarrow 0$ Lista unitaria: $(n) \rightarrow n$

Inducción:

 $(n \ x \ \dots) \rightarrow n + \text{Suma}(x \ \dots)$ 

```
(defun suma-lista (lista)
  (cond ((null lista) 0)
        ((null (rest lista)) (first lista))
        (T (+ (first lista) (suma-lista (rest lista))))))
```



Dr. Salvador Godoy Calderón

Suma de una lista numérica...

```
>> (defun suma-lista (lista)
      (cond ((null lista) 0)
            ((null (rest lista)) (first lista))
            (T (+ (first lista) (suma-lista (rest lista))))))
```

SUMA-LISTA

```
>> (suma-lista '(2 4 6 8))
20
>> (suma-lista '(1.1 2.2 3.3))
6.6
>> (suma-lista '(3/9 4 1.17))
5.50333336
```

```
>> (suma-lista '(17))
17
>> (suma-lista '())
0
```

Dr. Salvador Godoy Calderón

Emular funciones LISP...

Número de celdas en una lista (**LENGTH**):

Caso base:

Lista vacía: $(\) \rightarrow 0$

Inducción:

$(x \dots) \rightarrow 1 + \text{Conteo}(\dots)$

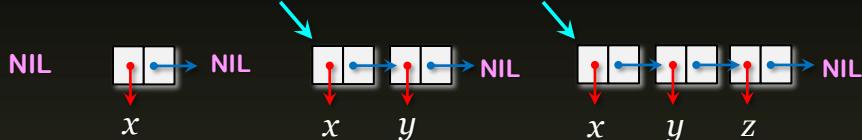
```
(defun num-elementos (lista)
  (cond ((null lista) 0 )
        (T (+ 1 (num-elementos (rest lista))))))
```

combinación conocida desconocida

Dr. Salvador Godoy Calderón

Formas de combinar...

Última celda de una lista (**LAST**):



```
(defun último (lista)
  (cond ((null lista) NIL )
        ((null (rest lista)) lista)
        (T (último (rest lista))))))
```

conocida ? combinación ? desconocida

Dr. Salvador Godoy Calderón

*Emular funciones LISP ...*Invertir una lista (**REVERSE**):

Casos base:

$$\begin{aligned} () &\rightarrow () \\ (x) &\rightarrow (x) \end{aligned}$$

Inducción:

$$(x y \dots) \rightarrow \text{append} ((y \dots)^R, (x))$$

```
(defun invierte (lista)
  (cond ((null lista) NIL)
        ((null (rest lista)) lista))
        (T (append (invierte (rest lista))
                    (list (first lista))))))
```

Dr. Salvador Godoy Calderón

Emular funciones LISP ...

```
>> (defun invierte (lista)
      (cond ((null lista) NIL)
            ((null (rest lista)) lista))
            (T (append (invierte (rest lista))
                        (list (first lista))))))
```

INVIERTE

```
>> (invierte '(2 4 6 8))
(8 6 4 2)
```

```
>> (invierte '(a b (c d) e))
(E (C D) B A)
```

```
>> (invierte '((() () ())))
(NIL NIL NIL)
```

```
>> (invierte '((a b) 1 (x y) 2 () 3))
(3 NIL 2 (X Y) 1 (A B))
```

Dr. Salvador Godoy Calderón

Rastreando...

La macro **TRACE** permite rastrear las invocaciones recursivas de una función...

```
>> (trace invierte)  
(INVIERTE)
```

```
>> (invierte '(a b c))  
0:(invierte (a b c))  
1:(invierte (b c))  
2:(invierte (c))  
3:(invierte nil)  
3:returned nil  
2:returned (c)  
1:returned (c b)  
0:returned (c b a)  
(c b a)
```

Al final usar:

```
>> (untrace invierte)  
T
```

Dr. Salvador Godoy Calderón

**Optimización del
código recursivo...**



Identificando el problema...

Concatenar dos listas (**APPEND**):

```
(defun pega-listas (lista1 lista2)
  (cond ((null lista1) lista2)
        (t (cons (first lista1) (pega-listas (rest lista1) lista2)))))
```

Invertir una lista (forma no-optimizada):

```
( defun invierte-lento (lista)
  (cond ((null lista) NIL)
        ((null (rest lista)) lista)
        (T (pega-listas (invierte-lento (rest lista))
                      (list (first lista)))) )
```

Dr. Salvador Godoy Calderón

Rastreo...

Usando **TRACE** podemos rastrear su desempeño:

```
>> (trace invierte-lento)
(INVIERTE-LENTO)
```

```
>> (invierte-lento '(a b c))
0:(Invierte-lento (a b c))
 1:(invierte-lento (b c))
 2:(invierte-lento (c))
 3:(invierte-lento nil)
 3:returned nil
 2:returned (c)
 1:returned (c b)
 0:returned (c b a)
(C B A)
```

```
>> (trace pega-listas )
(pega-listas)
```

```
>> (invierte-lento '(a b c))
0:(invierte-lento (A B C))
 1:(invierte-lento (B C))
 2:(invierte-lento (C))
 2:invierte-lento returned (C)
 2:(pega-listas (C) (B))
  3:(pega-listas NIL (B))
  3:pega-listas returned (B)
  2:pega-listas returned (C B)
 1:invierte-lento returned (C B)
 1:(pega-listas (C B) (A))
  2:(pega-listas (B) (A))
  3:(pega-listas NIL (A))
  3:pega-listas returned (A)
  2:pega-listas returned (B A)
 1:pega-listas returned (C B A)
 0:invierte-lento returned (C B A)
(C B A)
```



Dr. Salvador Godoy Calderón

Optimización...

Separamos la función invierte-lento en dos funciones. La primera, no-recursiva, sólo invoca a la segunda. La función auxiliar sí es recursiva y agrega un parámetro extra:

```
>> (defun invierte (lista)
  (invierte-aux lista nil))
INVIERTE
```

```
>> (defun invierte-aux (lista acumulado)
  (cond ((null lista) acumulado)
        (T (invierte-aux (rest lista)
                           (cons (first lista) acumulado))))
```

Dr. Salvador Godoy Calderón

Verificando...

```
>> (trace invierte)
(INVIERTE)

>> (trace invierte-aux)
(INVIERTE-AUX)
```

```
>> (invierte '(a b c d e f) )
0: (invierte (a b c d e f))
1: (invierte-aux (a b c d e f) nil)
2: (invierte-aux (b c d e f) (a))
3: (invierte-aux (c d e f) (b a))
4: (invierte-aux (d e f) (c b a))
5: (invierte-aux (e f) (d c b a))
6: (invierte-aux (f) (e d c b a))
7: (invierte-aux nil (f e d c b a))
7: invierte-aux returned (f e d c b a)
6: invierte-aux returned (f e d c b a)
5: invierte-aux returned (f e d c b a)
4: invierte-aux returned (f e d c b a)
3: invierte-aux returned (f e d c b a)
2: invierte-aux returned (f e d c b a)
1: invierte-aux returned (f e d c b a)
0: invierte returned (f e d c b a)
(F E D C B A)
```

Dr. Salvador Godoy Calderón

Por supuesto...

Conviene definir la función auxiliar como función local a la función principal...

Sólo recordar que esto se hace cuando ya se validó el código, porque ya no será posible rastrear la función auxiliar...

```
>> (defun invierte (lista)
  (labels((invierte-aux (lista acumulado)
    (cond ((null lista) acumulado)
      (T (invierte-aux (rest lista)
        (cons (first lista) acumulado))))))
  (invierte-aux lista nil)))
```

Dr. Salvador Godoy Calderón

Un problema clásico...

Un problema clásico es calcular la serie de *Fibonacci*, definida de la siguiente forma:

$$Fib(x) = \begin{cases} x & \text{si } x < 2 \\ Fib(x-1) + Fib(x-2) & \text{en otro caso} \end{cases}$$

```
>> (defun Fibonacci (x)
  (cond ((< x 2) x)
    (T (+ (Fibonacci (- x 1)) (Fibonacci (- x 2))))))
```

Dr. Salvador Godoy Calderón

Verificando...

```
>> (trace Fibonacci)
(FIBONACCI)
```

Evidentemente la ejecución
no es óptima...

```
>> (Fibonacci 4 )
0:(Fibonacci 4)
1:(Fibonacci 3)
2:(Fibonacci 2)
3:(Fibonacci 1)
3:Fibonacci returned 1
3:(Fibonacci 0)
3:Fibonacci returned 0
2:Fibonacci returned 1
2:(Fibonacci 1)
2:Fibonacci returned 1
1:Fibonacci returned 2
1:(Fibonacci 2)
2:(Fibonacci 1)
2:Fibonacci returned 1
2:(Fibonacci 0)
2:Fibonacci returned 0
1:Fibonacci returned 1
0:Fibonacci returned 3
3
```

Dr. Salvador Godoy Calderón

Más grave aún...

```
>> (untrace Fibonacci)
T
```

```
>> (time (Fibonacci 40))

Evaluation took:
[4.225 seconds of real time]
4.224835 seconds of total run time
(4.224835 user, 0.000000 system)
100.00% CPU
[12,987,812,460 processor cycles]
0 bytes consed
```

102334155

Dr. Salvador Godoy Calderón

Optimizando...

```
>> (defun Fib (x)
  (Fib-aux x 0 1))
```

```
>> (defun Fib-aux (x acum1 acum2)
  (cond ((= x 0) acum1)
        (T (Fib-aux (- x 1) acum2 (+ acum1 acum2)))) )
```

Dr. Salvador Godoy Calderón

Verificando...

```
>> (trace Fib)
(FIB)
>> (trace Fib-aux)
(FIB-AUX)
```

```
>> (Fib 4)
0: (Fib 4)
1: (Fib-aux 4 0 1)
2: (Fib-aux 3 1 1)
3: (Fib-aux 2 1 2)
4: (Fib-aux 1 2 3)
5: (Fib-aux 0 3 5)
5: Fib-aux returned 3
4: Fib-aux returned 3
3: Fib-aux returned 3
2: Fib-aux returned 3
1: Fib-aux returned 3
0: Fib returned 3
3
```

Dr. Salvador Godoy Calderón

Lo mejor de todo...

```
>> (time (Fibonacci 45))

Evaluation took:
46.825 seconds of real time
46.820822 seconds of total run time
(46.820819 user, 0.000003 system)
99.99% CPU
143,959,049,860 processor cycles
0 bytes consed

1134903170
```



```
>> (time (Fib 45))

Evaluation took:
0.000 seconds of real time
0.000003 seconds of total run time
(0.000003 user, 0.000000 system)
100.00% CPU
4,288 processor cycles
0 bytes consed

1134903170
```

Dr. Salvador Godoy Calderón

Recomendaciones...

- 1) Cada proceso recursivo separarlo en dos funciones,
- 2) La función principal sólo invoca a la auxiliar y no es recursiva;
- 3) La función auxiliar (recursiva) argrega un argumento acumulador por cada rama de recursión requerida...
- 4) En el caso base de la función auxiliar, se entrega como respuesta el valor acumulado en el parámetro acumulador...

Dr. Salvador Godoy Calderón



Mapeos...

Un **Mapeo** es una función que aplica un mismo procesamiento a todos los elementos de varias listas y en las posiciones correspondientes...

(**mapcar** <*función*> <*lista1*> <*lista2*> ...)

```
>> (mapcar #'+ '(1 2 3) '(7 8 9))  
(8 10 12)  
>> ( mapcar #'max '(1 3 5) '(2 4 6) '(5 4 3) )  
(5 4 6)
```

Dr. Salvador Godoy Calderón

Expresiones Lambda...

En ocasiones se desea aplicar un filtro o un mapeo pero usando una función de usuario, sin embargo, no se desea tener que definirla por separado previamente.

```
( lambda (<argumentos>) <cuerpo> )
```

```
>> (mapcar #'(lambda (x) (append x x)) '((a b c)) )
(A B C A B C)

>> (mapcar #'(lambda (x y z) (list z y x))
            '(a b c) '(x y z) '(1 2 3) )
((1 X A) (2 Y B) (3 Z C))
```

Dr. Salvador Godoy Calderón

Otros ejemplos...

```
>> (mapcar #'(lambda (x)(* x x)) '(1 2 3 4 5) )
(1 4 9 16 25)

>> (mapcar #'(lambda (x)(* x 2)) '(1 2 3 4 5) )
(2 4 6 8 10)

>> (mapcar #'(lambda (x)(and (> x 0) (< x 10)))
            '(1 2 4 5 -9 8 23))
(T NIL T NIL T NIL)
```

Dr. Salvador Godoy Calderón



Filtros...

Un *Filtro* es una función que examina todos los elementos de una lista, aplicando una prueba a cada uno y eliminando aquellos que pasan la prueba aplicada.

```
(defun filtra-pares (lista)
  (cond ((null lista) nil)
        ((evenp (first lista)) (filtra-pares (rest lista)))
        (t (cons (first lista) (filtra-pares (rest lista))))))
```

```
>> (filtra-pares '(-3 -2 -1 0 1 2 3 4 5 6 7 8 9))
(-3 -1 1 3 5 7 9)

>> (filtra-pares '(10 100 1000 20 200 2000))
NIL
```

Dr. Salvador Godoy Calderón

Ejemplos...

```
( defun filtra-pares (lista)
  (cond ((null lista) nil)
        ((evenp (first lista)) (filtra-pares (rest lista)))
        (T (cons (first lista) (filtra-pares (rest lista)))) ))
```

```
( defun filtra-ceros (lista)
  (cond ((null lista) nil)
        ((zerop (first lista)) (filtra-ceros (rest lista)))
        (T (cons (first lista) (filtra-ceros (rest lista)))) )))
```

Ambos códigos son casi idénticos , excepto por la función de verificación (`evenp` o `zerop`). ¿Cómo podemos construir un filtro general que sirva para cualquier tipo de prueba?

Dr. Salvador Godoy Calderón

La familia FUNCALL...

Para lograrlo se requiere que la prueba a realizar sea un argumento de la función y poder ejecutar ese argumento:

```
( funcall <función> <arg1> <arg2> <arg3> ...)
```

```
(funcall #'+ 5 4) -----> 9
(funcall #'first '(a b c))-----> A
(apply #'+ '(2 3)) -----> 5
(apply #'first '((a b c))) -----> A
(eval '(+ 7 3)) -----> 10
```

Dr. Salvador Godoy Calderón

La familia FUNCALL...

```
(funcall #'+ 5 4) -----> 9
(funcall #'first '(a b c))-----> A
(apply #'+ '(2 3)) -----> 5
(apply #'first '((a b c)))-----> A
(eval '(+ 7 3)) -----> 10
```

Todas las funciones de este grupo hacen lo mismo, evaluar alguna función con los argumentos dados...

La única diferencia entre ellas es la forma en que reciben sus argumentos...

La función **eval** resulta ser la opuesta de **quote**...

Dr. Salvador Godoy Calderón

Generalización...

```
(defun filtra (lista función)
  (cond ((null lista) nil)
        ((funcall función (first lista)) (filtra (rest lista) función))
        (t (cons (first lista) (filtra (rest lista) función)))))
```

```
>> (filtra '(-3 -2 -1 0 1 2 3) #'plusp )
(-3 -2 -1 0)

>> (filtra '(1 2 3 4 5 6 7 8 9 0) #'evenp )
(1 3 5 7 9)

>> (filtra '(2 0 0 5 6 7 0 3 4 0) #'zerop )
(2 5 6 7 3 4)
```

Dr. Salvador Godoy Calderón

