

Recursión!

Ideas:

- Repetir: Encierra un proceso repetitivo.
- Función que se llama a si misma (auto-referenciación).
- Metodología de solución, donde resolvemos problemas complejos a partir de la resolución de problemas más simples.
- Ejemplo:
 1. Ejemplo del factorial.
 2. Fibonacci.
 3. Recorrido y procesamiento de datos en una lista.
- Racket:
Definición de listas (conceptual)

List (Num) es:

- empty (caso base)
- (cons Num **List (Num)**) (caso recursivo o constructivo)

Recordamos que:

- empty y cons son los **constructores** de la lista,
- predicados: empty?, cons?

- first y rest son los **selectores** de las lista.

Definición de Suma de elementos de una lista de números List (Num))

```
;suma-lista: List(Num) -> Num
; Calcula la suma de todos los elementos de la lista.
(define (suma-lista lista)
  (cond [ (empty? lista) 0]
        [ (cons? lista) (+ (first lista)
                             (suma-lista (rest
lista)))) ]
Ej:
(cons 1 (cons 45 (cons 65 empty)))
```

Atenti!

- Poner bien el caso base
- Cuestiones de Eficiencia (mas adelante volveremos a este tema)
- Problemas de terminación si no está bien diseñada, la memoria se agota al llamarse “infinitamente”.

Números Naturales

0, 1, 2,, n, n+1,

Un **nat** es:

- 0 (caso Base)

- $\text{suc}(\text{nat})$ (caso Constructivo), suc : sucesor, siguiente, sumar 1

Los constructores son: 0 (cero), y la operación suc (sucesor)

El $\text{suc}(n) = n+1$

Si quiero para un número **m distinto del caso base**, sabemos que:

$$m = n+1$$

¿Cómo puedo proyectar el valor de n ? analicemos

$$m = n+1$$

$$m - 1 = n$$

El valor de n lo podemos extraer mediante el predecesor (selector) restar uno.

Ejemplo: Construcción del número “tres” y el “cinco”

“tres” = 3 = $\text{suc}(\text{suc}(\text{suc}(0)))$

“cinco” = 5 = $\text{suc}(\text{suc}(3))$

= $\text{suc}(\text{suc}(\text{suc}(\text{suc}(\text{suc}(0)))))$

Pensamos la función de descomposición de un número:

descomposicion (3) $n \neq 0$, $n = 3$, $n = n'+1$, n' ? no sé si es cero o distinto de cero, si sé que $n' = 3 - 1$

↓

“ $\text{suc}(\text{descomposicion}(2))$ ” $n \neq 0$, $n = 2$, $n = n'+1$, n' ? no sé si es cero o distinto de cero, si sé que $n' = 2-1$

↓
 “suc (“ + “suc (“ descomposicion (1) ”)” +”)”, n! =0, n= 1,
 n = n' +1, n'? no se si es cero o es distinto de cero, si sé
 que n' = 1-1

↓
 “suc (“ + “suc (“ + “suc (“ descomposicion (0) “)” +”)” +”)”,
 n = 0

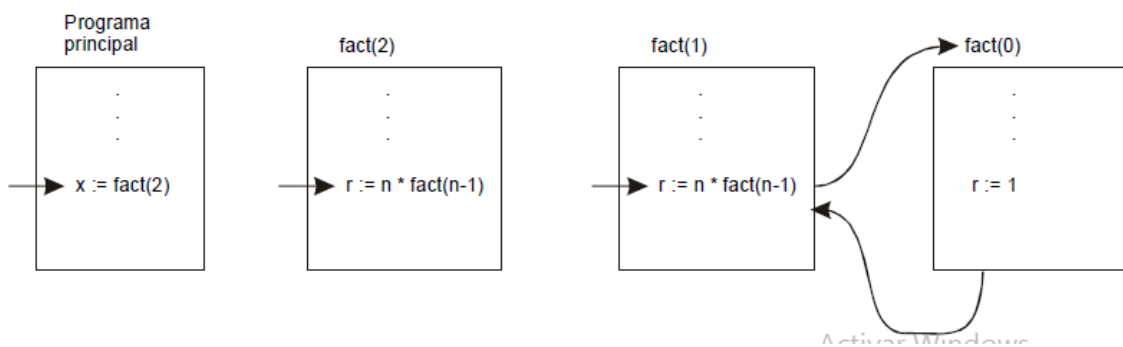
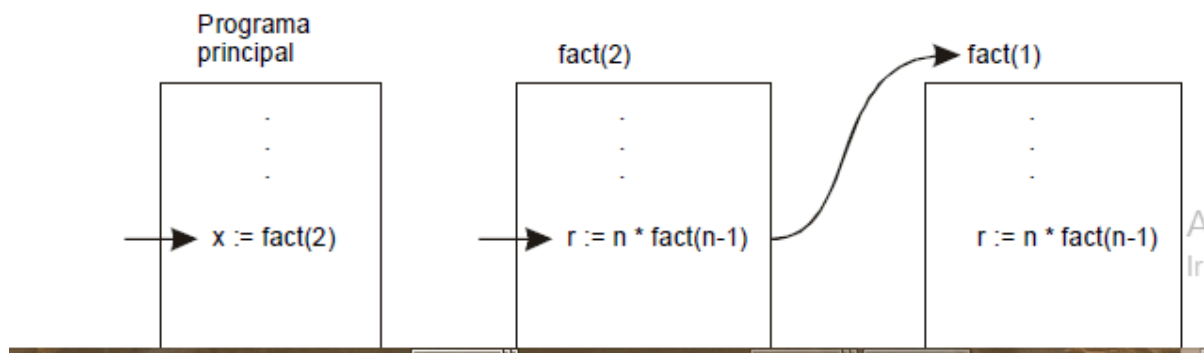
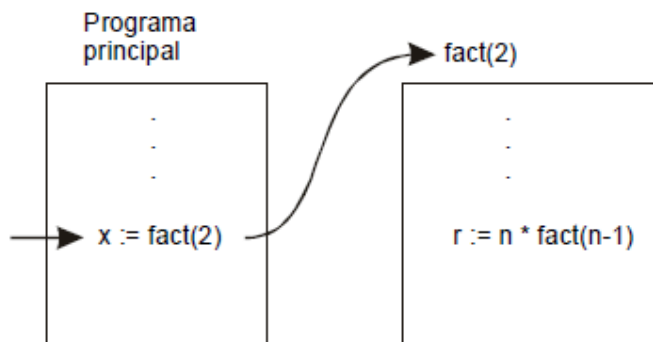
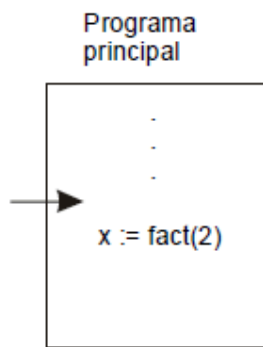
↓
 “suc (“+”suc (“+ “suc (“ + “0”+”)”+”)”+”)”+”)”,
fin!

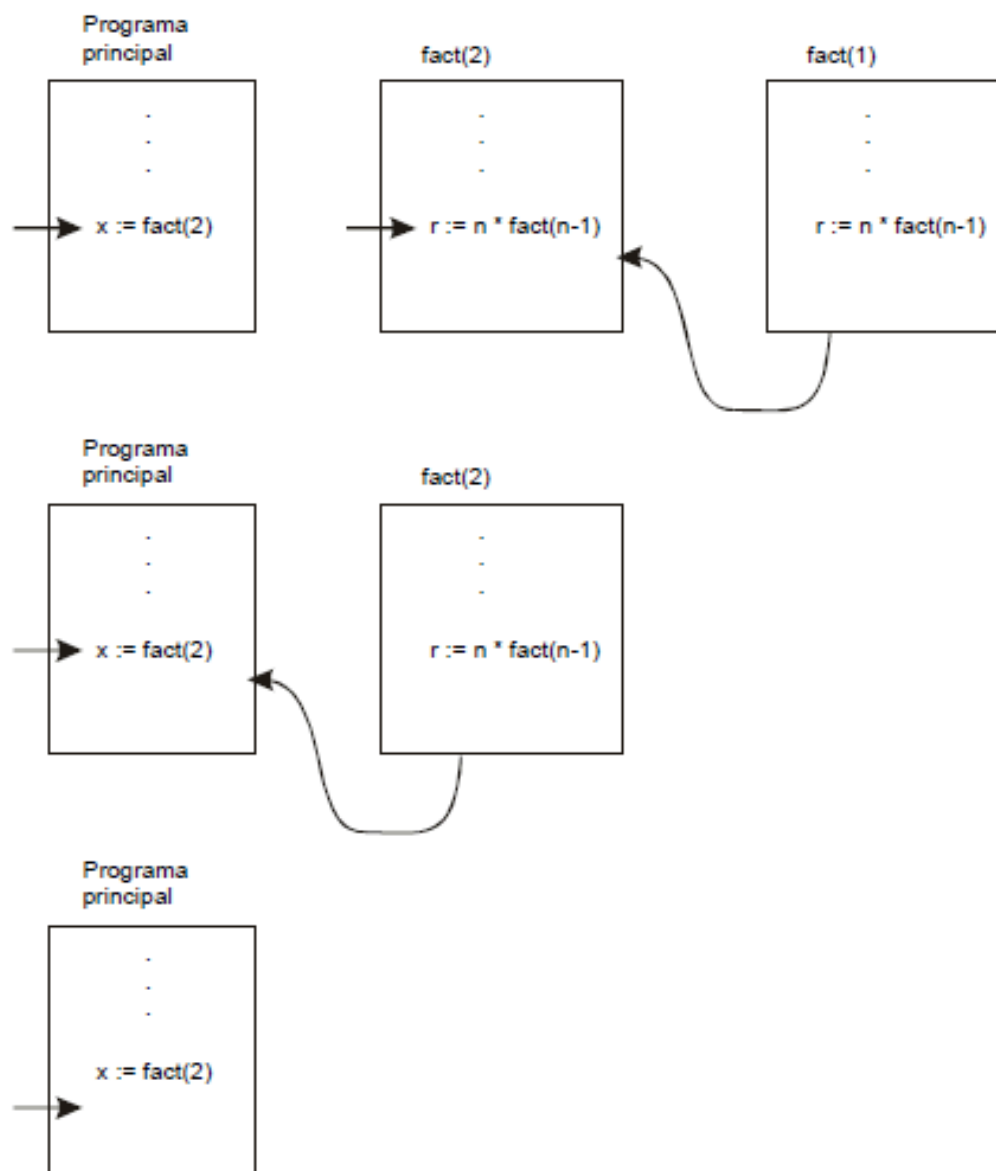
Ejemplo del factorial:

fact (n) = 1 , si n = 0
 n* **fact**(n-1), si n >0

n! = 1 , si n = 0
 n* (n-1)!, si n > 0

¿Cómo se ejecuta en memoria esta función al ser invocada en fact(2)?





fact (2), n!=0

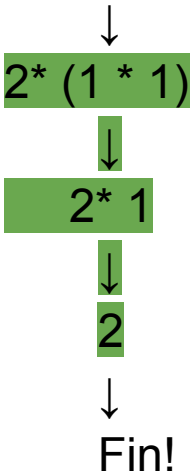


2* fact(1), n!=0,



2* (1 * fact (0)), n = 0

Corresponde al unfolding de la invocación recursiva (se expanden las invocaciones a las funciones hasta llegar al caso base)

 <p>↓ 2* (1 * 1) ↓ 2* 1 ↓ 2 ↓ Fin!</p>	<p>Corresponde al collect de la recursión donde se procesa el cálculo. No hay llamas recursivas, es simplemente procesar la información respetando los paréntesis.</p>
---	--

Este tipo de recursión se llama “**recursión lineal no final**”.