

# Fundamentos de Programación Funcional y Concurrente

Funciones y los procesos que ellas generan

Juan Francisco Díaz Frias

Profesor Titular (1993-hoy)  
juanfco.diaz@correounivalle.edu.co  
Edif. B13 - 4009



**Universidad del Valle**

Octubre 2022

# Plan

## 1 Generalidades

## 2 Recursión

- Recursión lineal e iteración
- Recursión de árbol

# Plan

## 1 Generalidades

## 2 Recursión

- Recursión lineal e iteración
- Recursión de árbol

# Generalidades

- Funciones v.s. **Procesos computacionales**:
  - Una función especifica la evolución de un proceso computacional
  - Las reglas de evaluación de una función determinan el siguiente estado del proceso computacional.
- Objetivo: Hacer observaciones globales sobre el comportamiento de un proceso.
- Procesos más comunes:
  - **Recursión lineal e iteración**
  - **Recursión en árbol**

# Generalidades

- Funciones v.s. **Procesos computacionales**:
  - Una función especifica la evolución de un proceso computacional
  - Las reglas de evaluación de una función determinan el siguiente estado del proceso computacional.
- Objetivo: Hacer observaciones globales sobre el comportamiento de un proceso.
- Procesos más comunes:
  - **Recursión lineal e iteración**
  - **Recursión en árbol**

# Generalidades

- Funciones v.s. **Procesos computacionales**:
  - Una función especifica la evolución de un proceso computacional
  - Las reglas de evaluación de una función determinan el siguiente estado del proceso computacional.
- Objetivo: Hacer observaciones globales sobre el comportamiento de un proceso.
- Procesos más comunes:
  - **Recursión lineal** e **iteración**
  - **Recursión en árbol**

# Recordar evaluación: Modelo de sustitución

La aplicación de funciones con parámetros ( $f(e_1, \dots, e_n)$ ) se evalúa así:

- Evalúe los argumentos  $e_1, \dots, e_n$  de izquierda a derecha. Denotemos  $v_1, \dots, v_n$  los resultados de esas evaluaciones.
- Sustituya la aplicación de la función por su cuerpo (lado derecho del  $=$ ) y,
- Sustituya en ese cuerpo, los parámetros formales por los argumentos actuales  $v_1, \dots, v_n$
- Evalúe esta nueva expresión

Por ejemplo:

*sumOfSquares*(3, 2 + 2)

→ *sumOfSquares*(3, 4)

→ *square*(3) + *square*(4)

→ 3 \* 3 + *square*(4)

→ 9 + *square*(4)

→ 9 + 4 \* 4

→ 9 + 16

→ 25

# Plan

## 1 Generalidades

## 2 Recursión

- Recursión lineal e iteración
- Recursión de árbol



# Ejemplo: la función factorial (1)

- Considere la función factorial:

```
0 def factorial(n: Int): Int =  
1   if (n==0) 1 else n * factorial(n-1)
```

- ¿Cómo se evalúa *factorial*(4)?

*factorial*(4)

→ *if*(4 == 0) 1 *else* 4 \* *factorial*(4 - 1)

→ 4 \* *factorial*(3)

→ 4 \* 3 \* *factorial*(2)

→ 4 \* 3 \* 2 \* *factorial*(1)

→ 4 \* 3 \* 2 \* 1 \* *factorial*(0)

→ 4 \* 3 \* 2 \* 1 \* 1

→ 24

# Ejemplo: la función factorial (2)

- Considere la función factorial:

```
0 def factIter(cont:Int, prod:Int, n:Int):Int =  
1   if (cont > n) prod else factIter(cont+1, cont*prod, n)  
2  
3 def fact(n:Int)= factIter(1, 1, n)
```

- ¿Cómo se evalúa *fact(4)*?

*fact(4)*

→ *factIter(1, 1, 4)*

→ *if(1 > 4) 1 else factIter(1 + 1, 1 \* 1, 4)*

→ *factIter(2, 1, 4)*

→ *if(2 > 4) 1 else factIter(2 + 1, 2 \* 1, 4)*

→ *factIter(3, 2, 4)*

→ *if(3 > 4) 2 else factIter(3 + 1, 3 \* 2, 4)*

→ *factIter(4, 6, 4)*

→ *if(4 > 4) 6 else factIter(4 + 1, 4 \* 6, 4)*

→ *factIter(5, 24, 4)*

→ *if(5 > 4) 24 else factIter(5 + 1, 5 \* 24, 4)*

→ 24

# Comparación de procesos generados


## ● Procesos


```
factorial(4)
→ if (4 == 0) 1 else 4 * factorial(4 - 1)
→ 4 * factorial(3)
→ 4 * 3 * factorial(2)
→ 4 * 3 * 2 * factorial(1)
→ 4 * 3 * 2 * 1 * factorial(0)
→ 4 * 3 * 2 * 1 * 1
→ 24
```

```
fact(4)
→ factIter(1, 1, 4)
→ if (1 > 4) 1 else factIter(1 + 1, 1 * 1, 4)
→ factIter(2, 1, 4)
→ if (2 > 4) 1 else factIter(2 + 1, 2 * 1, 4)
→ factIter(3, 2, 4)
→ if (3 > 4) 2 else factIter(3 + 1, 3 * 2, 4)
→ factIter(4, 6, 4)
→ if (4 > 4) 6 else factIter(4 + 1, 4 * 6, 4)
→ factIter(5, 24, 4)
→ if (5 > 4) 24 else factIter(5 + 1, 5 * 24, 4)
→ 24
```

## ● Comparación:

	<i>factorial</i>	<i>fact</i>
Tiempo	$\sim 2n$	$\sim n$
Forma	Expansión-Contracción	Constante
Espacio	$\sim n$	$\sim \text{cte}$

  
 Recursivo Lineal

  
 Iterativo Lineal

## ● OJO!

Proceso Recursivo  $\neq$  Función Recursiva

# Ejercicio

Considere las siguientes dos versiones de la suma de dos números enteros:

- Versión 1:

```
0 def pred(a: Int) = a-1
1 def suc(a: Int) = a+1
2 def suma(a: Int, b: Int): Int = if (a==0) b else suc(suma(pred(a),b))
3 suma(4,5)
```

- Versión 2:

```
0 def pred(a: Int) = a-1
1 def suc(a: Int) = a+1
2 def suma(a: Int, b: Int): Int = if (a==0) b else suma(pred(a),suc(b))
3 suma(4,5)
```

- Ilustrar el proceso generado por cada procedimiento al evaluar *suma(4,5)*. ¿Cómo son estos procesos?
- [Socratica]

# Recursión de cola

- Cuando una función recursiva se invoca a sí misma como su última acción (y no antes), la pila de evaluación puede ser reutilizada. A este tipo de recursión se le denomina **recursión de cola**.
- Las funciones recursivas por la cola implementan **procesos iterativos**.
- Los lenguajes de programación **compilan** las recursiones de cola como iteraciones (optimización de código)

# Recursión de cola

- Cuando una función recursiva se invoca a sí misma como su última acción (y no antes), la pila de evaluación puede ser reutilizada. A este tipo de recursión se le denomina **recursión de cola**.
- Las funciones recursivas por la cola implementan **procesos iterativos**.
- Los lenguajes de programación **compilan** las recursiones de cola como iteraciones (optimización de código)

# Recursión de cola

- Cuando una función recursiva se invoca a sí misma como su última acción (y no antes), la pila de evaluación puede ser reutilizada. A este tipo de recursión se le denomina **recursión de cola**.
- Las funciones recursivas por la cola implementan **procesos iterativos**.
- Los lenguajes de programación **compilan** las recursiones de cola como iteraciones (optimización de código)

# Plan

## 1 Generalidades

## 2 Recursión

- Recursión lineal e iteración
- Recursión de árbol



# La función Factorial, en recursión de árbol (1)



$$n! = \underbrace{1 * 2 * 3 * \dots * (k-1)}_{p_1} * \underbrace{k * \dots * (n-1)}_{p_2} * n$$

Podríamos pensar entonces en dividir ese cálculo en dos.



Considere la función *Producto* definida así:

$$\text{Producto}(a, b) = \begin{cases} 1 & \text{Si } a \geq b \\ a & \text{Si } a = b - 1 \\ \text{Producto}(a, m) * & \\ \text{Producto}(m, b) & \text{Sino, y } m = a + (b - a)/2 \end{cases}$$

$$n! = \text{Producto}(1, n + 1)$$



Definida en Scala:

```
0 def producto(i: Int, j: Int): Int = {
1   // dado i < j, devuelve i*(i+1)*...*(j-1)
2   // dado i > j devuelve 1
3   if (i >= j) 1
4   else if (i == j - 1) i
5   else {
6     val m = i + (j - i) / 2
7     producto(i, m) * producto(m, j)
8   }
9 }
10 def fact(n: Int) = producto(1, n + 1)
```

# La función Factorial, en recursión de árbol (2)

- Proceso para evaluar  $fact(4)$ :

$fact(4)$

→  $producto(1, 5)$

→  $if\ (1 \geq 5)\ 1\ else\ if\ (1 == 4)\ 1\ else\ \{val\ m = 1 + (5 - 1)/2;\ producto(1, m) * producto(m, 5)\}$

→  $producto(1, 3) * producto(3, 5)$

→  $producto(1, 2) * producto(1, 3) * producto(3, 5)$

→  $1 * producto(2, 3) * producto(3, 5)$

→  $1 * 2 * producto(3, 5)$

→  $1 * 2 * producto(3, 4) * producto(4, 5)$

→  $1 * 2 * 3 * 4$

→ 24

- En general, se genera un árbol de evaluaciones