

- Iteración es el uso de while y for.
- Recursión es un programa que se llama a sí mismo

Como en Scala no tenemos while y for, vamos a a hallar formas recursivas para hacer lo mismo.

Recursión

Recursión lineal e iteración

The screenshot shows a presentation slide with a dark blue header and footer. The header contains the text 'Generalidades Recursión' and 'Recursión lineal e iteración Recursión de árbol'. The main content area has a title 'Ejemplo: la función factorial (1)' and a list of bullet points. The first bullet point is 'Considere la función factorial:' followed by a code block containing a Scala function definition. The second bullet point is '¿Cómo se evalúa factorial(4)?' followed by a series of steps showing the recursive evaluation of factorial(4) down to factorial(0) and back up to the final result 24. The footer contains the name 'Juan Francisco Díaz Frias' and the course title 'Fundamentos de Programación Funcional y Concurrente'.

6 (9 of 18) 200%

Generalidades 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

Juan Francisco Díaz Frias Fundamentos de Programación Funcional y Concurrente

Este programa se llama a sí mismo, no termina hasta que `n == 0` (Punto de parada, donde ya no se llama así mismo, sino que solamente suelta 1).

8 (11 of 18) Generalidades Recursión Recursión lineal e iteración Recursión de árbol

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:

	factorial	fact
Tiempo	$\sim 2n$	$\sim n$
Forma	Expansión-Contracción	Constante
Espacio	$\sim n$	$\sim cte$

↑
Recursivo Lineal
↑
Iterativo Lineal

- OJO!

Proceso Recursivo \neq Función Recursiva

Juan Francisco Díaz Frias Fundamentos de Programación Funcional y Concurrente

La función de la izquierda es un **Proceso recursivo**, mientras que el de la derecha es un **Proceso iterativo**, la función de la izquierda es más ineficiente, ya que en cada paso la memoria se está expandiendo, se van **acumulando** procesos, en cambio en el de la derecha la memoria se mantiene de forma constante (en cada paso hace la operación, y se ahorra el trabajo de acumular y acumular números como en el de la izquierda), en cada paso hace un proceso y así se evita tanta acumulación

10 (15 of 18) Generalidades Recursión Recursión lineal e iteración Recursión de árbol

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)

Juan Francisco Díaz Frias Fundamentos de Programación Funcional y Concurrente

Recursión de árbol

Este tipo de recursión divide mi problema en varias partes. Se puede decir que es lamenos eficiente de todas, pero hay ocasiones en las que **hay que** usar este tipo de approach

Funciones de alto Orden

Funciones anónimas

- así como puedo llamar cosas sin declararlas, también puedo llamar funciones sin darles un nombre, esto tiene el nombre de función anónima.

The screenshot shows a presentation slide titled "Funciones anónimas". The slide content includes:

- Pasar funciones como parámetro, nos hizo crear (nombrar) muchas funciones pequeñas nuevas (*suc*, *sum2*, *ident*, *cuadrado*). Eso es tedioso en algunas ocasiones.
- Miremos lo que pasa con otros datos, como por ejemplo, las cadenas. No necesitamos definir una cadena para poder imprimirla. En lugar de:

```
0 def cad="abc"; println(cad)
```


podemos escribir directamente:

```
0 println("abc")
```


porque las cadenas existen como *literales*
- Análogamente, uno quisiera tener *literales de funciones* que nos permitan escribirlas sin darles un nombre. Es una manera de escribir **funciones anónimas**.

The slide footer shows "Juan Francisco Díaz Frías" and "Fundamentos de Programación Funcional y Concurrente".

- Sinceramente no entendí muy bien el punto de esto, pero básicamente puedes crear el nombre de una función (*como lo harías con una variable*), pero sin especificar lo que va a hacer, solo el tipo. Esto idealmente se hace para funciones chiquitas, de una línea, de más es tedioso

The screenshot shows a presentation slide titled "Sintaxis de funciones anónimas". The slide content includes:

- Por ejemplo, para referirnos a una función que eleva al cubo, sin nombrarla, escribiremos:

```
0 (x: Int) => x*x*x
```


Nótese que $(x: Int)$ es el **parámetro** de la función y $x * x * x$ es su **cuerpo**. El tipo del parámetro puede ser omitido si el compilador lo puede inferir.
- Si hay varios parámetros, se separarán por comas:

```
0 (x: Int, y: Int) => (x+y)/2
```
- Una función anónima $(x_1: T_1, x_2: T_2, \dots, x_n: T_n) \Rightarrow E$ se puede expresar en el lenguaje de la manera siguiente:

```
0 def f(x_1:T_1, x_2:T_2, ..., x_n:T_n) => E; f
```


donde f es un nombre arbitrario, fresco, que nunca ha sido usado en el programa.

The slide footer shows "Juan Francisco Díaz Frías" and "Fundamentos de Programación Funcional y Concurrente".

Funciones que devuelven funciones

Funciones que devuelven funciones

- Reescribamos *suma* de la siguiente manera:

```

0  def suma2(f: Int => Int, prox: Int => Int): (Int, Int) => Int = {
1    def sumaF(a: Int, b: Int): Int = if (a > b) 0
2                                     else f(a) + sumaF(prox(a), b)
3    sumaF
4  }

```

suma ahora es una **función que devuelve otra función (*sumaF*) como resultado**.

- Podemos calcular las funciones *sumaEnteros2*, *sumaCuadrados2*, *sumaAlternada2* así:

Disclaimer: Este va a ser uno de los temas principales en el **Taller 2**

Básicamente es eso, como lo dice el nombre. También podemos asignar esa misma función a un valor, tipo:

```
val s: Int => Int = (x: Int) => x * 2
```

- Aquí se usa lo de funciones anónimas:

```

val s: Int => Int = (x: Int) => {
  // Aquí defines la función
  val resultado = x * 2
  resultado // El resultado de la función es la última expresión evaluada
}

// Uso de la función almacenada en s
val resultado = s(5) // resultado contendrá 10

```

Clausura: Elemento conceptual con el cual se guardan las funciones como valores.

Currying

Coger una función de dos o más argumentos, y volverlo un solo argumento

```

def sumac(a: Int): Int => Int = {
  (b: Int) => a + b
}

```

```
def suma_v3(a:Int)(b:Int) = a + b
def suma_v3(a:Int) (b:Int): Int
```

The screenshot shows a presentation slide titled "Ejemplo sencillo de currificación". The slide content is as follows:

- Sea $\text{suma} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ la función de dos argumentos, tal que $\text{suma}(a, b) = a + b$.
- Entonces $\text{sumac} : \mathbb{N} \rightarrow \mathbb{N}^{\mathbb{N}}$ es la función tal que $\text{sumac}(a) = g_a$ y $g_a : \mathbb{N} \rightarrow \mathbb{N}$ es tal que $g_a(b) = \text{suma}(a, b) = a + b$.
O sea, $\text{sumac}(a)$ es la función que le suma a a cualquier número natural.
- $$\text{suma}(a, b) = (\text{sumac}(a))(b)$$

pero sumac no sólo permite calcular suma sino también otras funciones:
 - $\text{sumac}(2)$ es la función que suma 2: $\text{sumac}(2)(b) = 2 + b$
 - $\text{sumac}(5)$ es la función que suma 5: $\text{sumac}(5)(b) = 5 + b$

The slide footer includes the name "Juan Francisco Díaz Frias" and the course "Fundamentos de Programación Funcional y Concurrente".

si quiero evaluar un solo valor para `suma_v3`, puedo hacerlo, y solo me devolverá la evaluación con dichos valores

```
suma_v3(1)(_) // -> 1
suma_v3(2)(3) // -> 5
```

Una función que recibe una función *la cual retorna un double*. Esta función retornará otra función que retorne un double (sí, ya sé que suena algo enredado, pero no sé cómo más describirlo)

```
def derivada (f:Double => Double, dx:Double): Double => Double =
  (x: Double) => (f(x+dx) - f(x))/dx
  derivada _
def cube(x: Double) = x*x*x
def cubeD = derivada(cube, 0.0001)
cubeD(1)
cubeD(2)
cubeD(3)
```

recibo la función y devuelvo la derivada