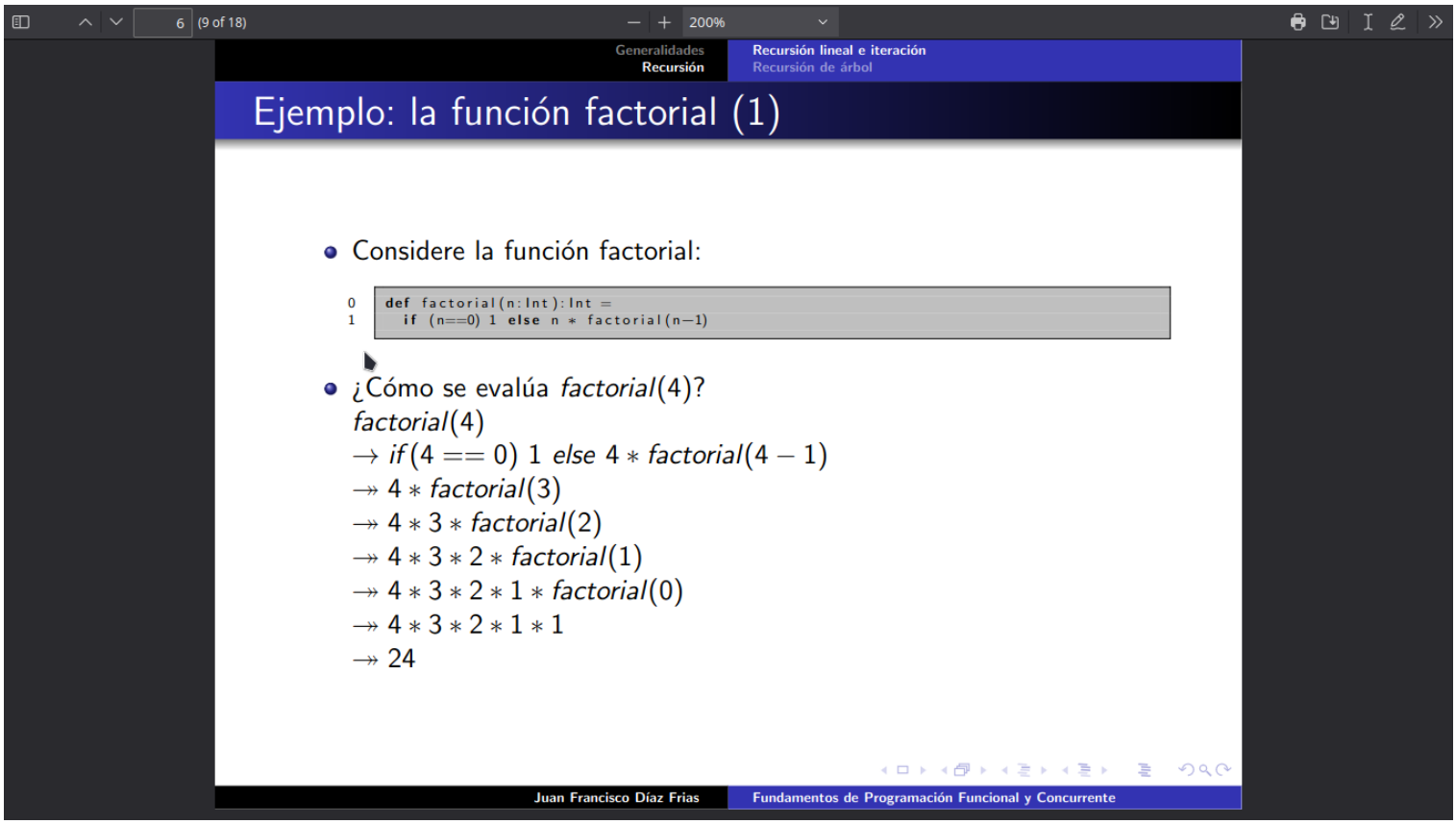
clase_07_09_2023

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

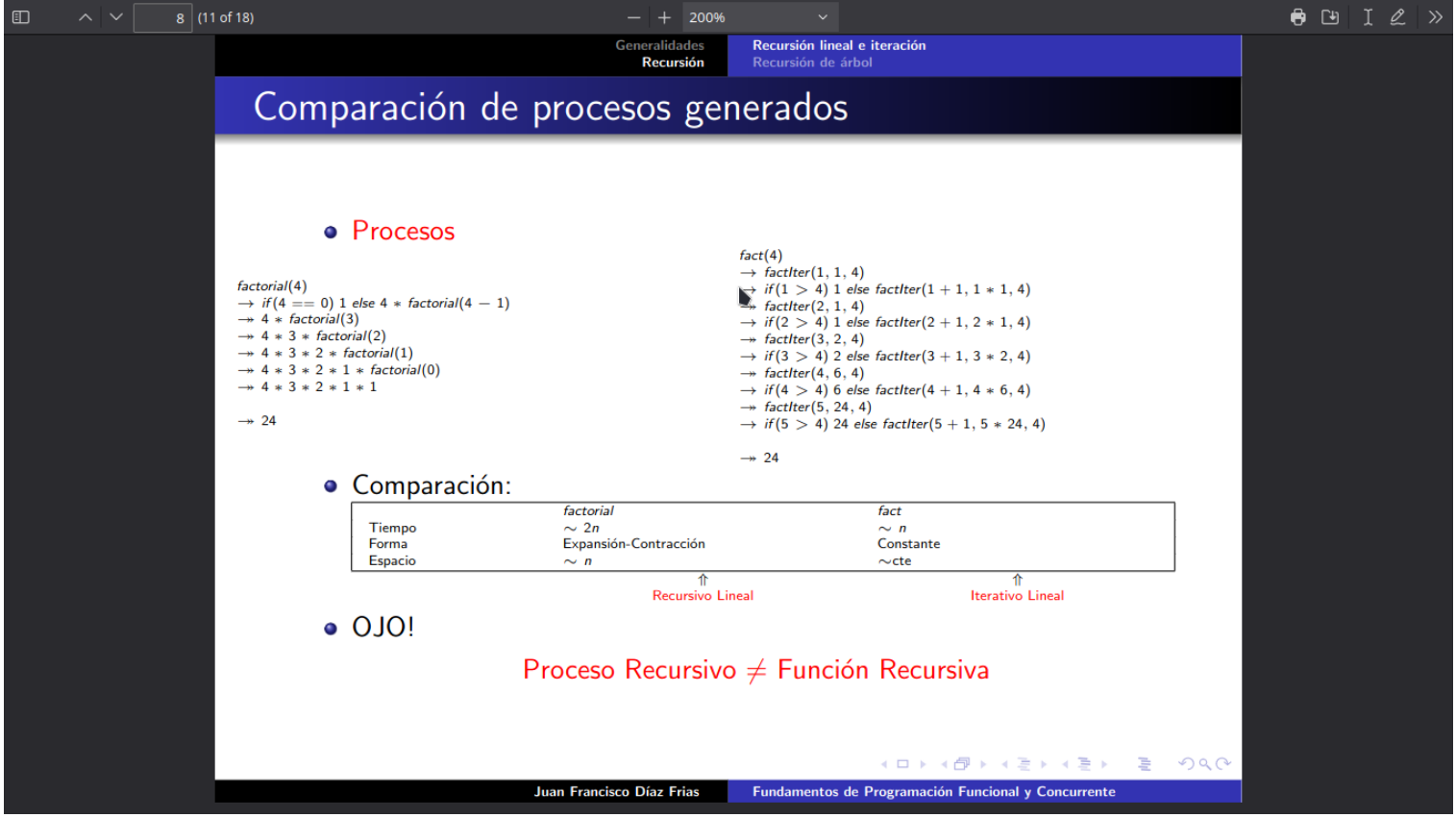
Como en Scala no tenemos while y for, vamos a a hallar formas recursivas para hacer lo mismo.

Recursión

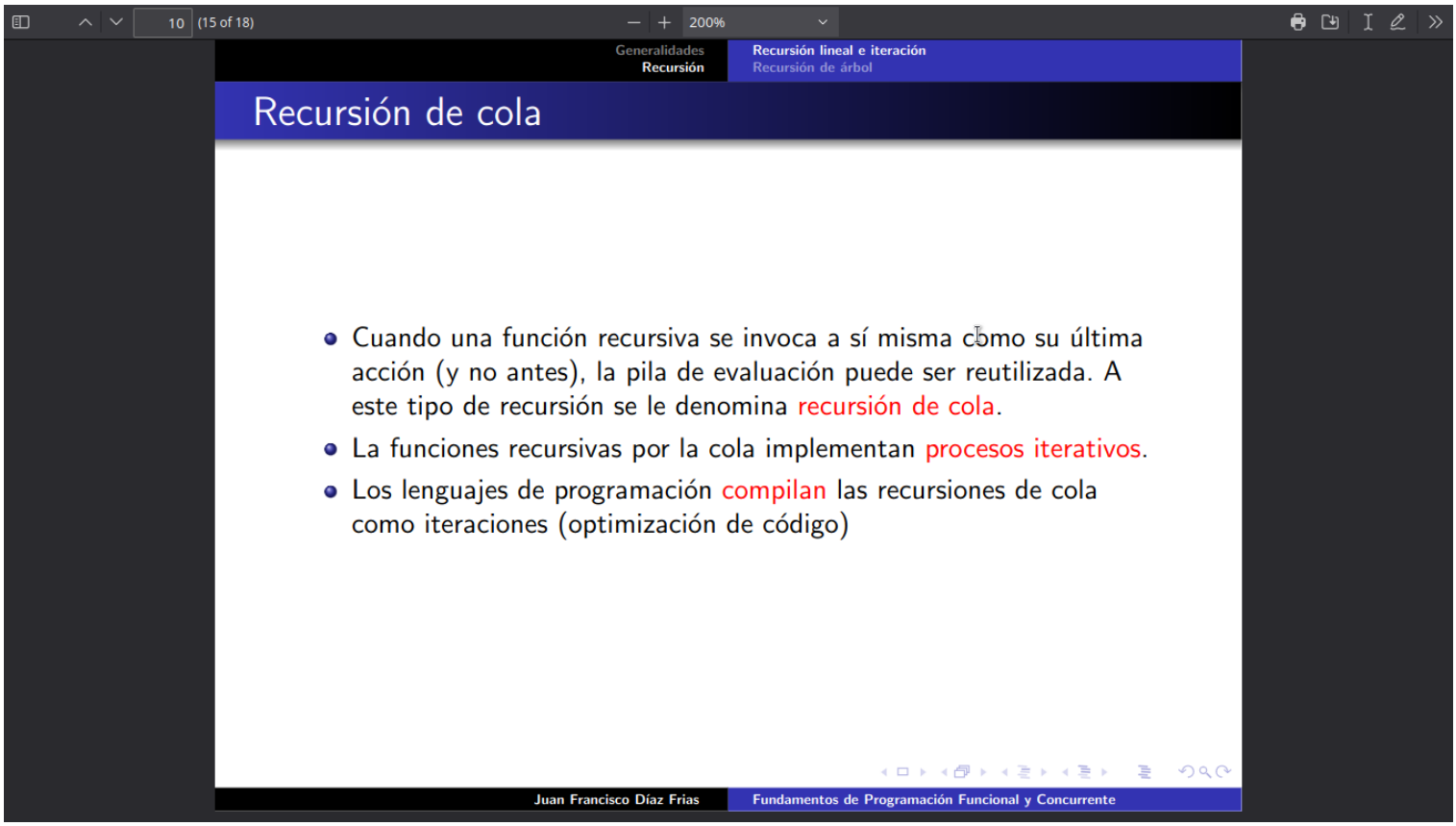
Recursión lineal e iteración



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)*.



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



Recursión de árbol

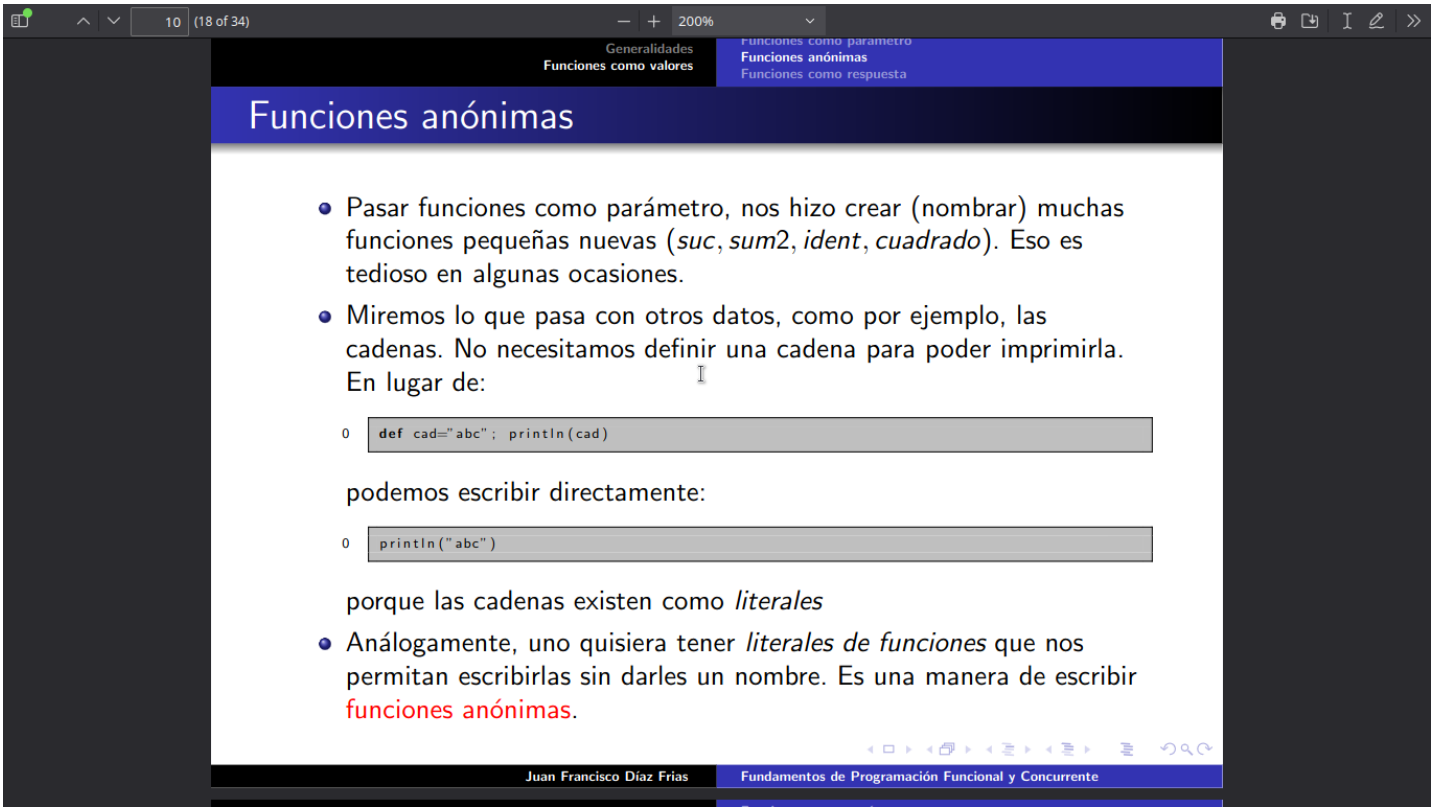
Este tipo de recursión divide mi problema en varias partes. Se puede decir que es lamenos eficiente de todas, pero hay ocasiones en las que **hay que** usar este tipo de approach

---

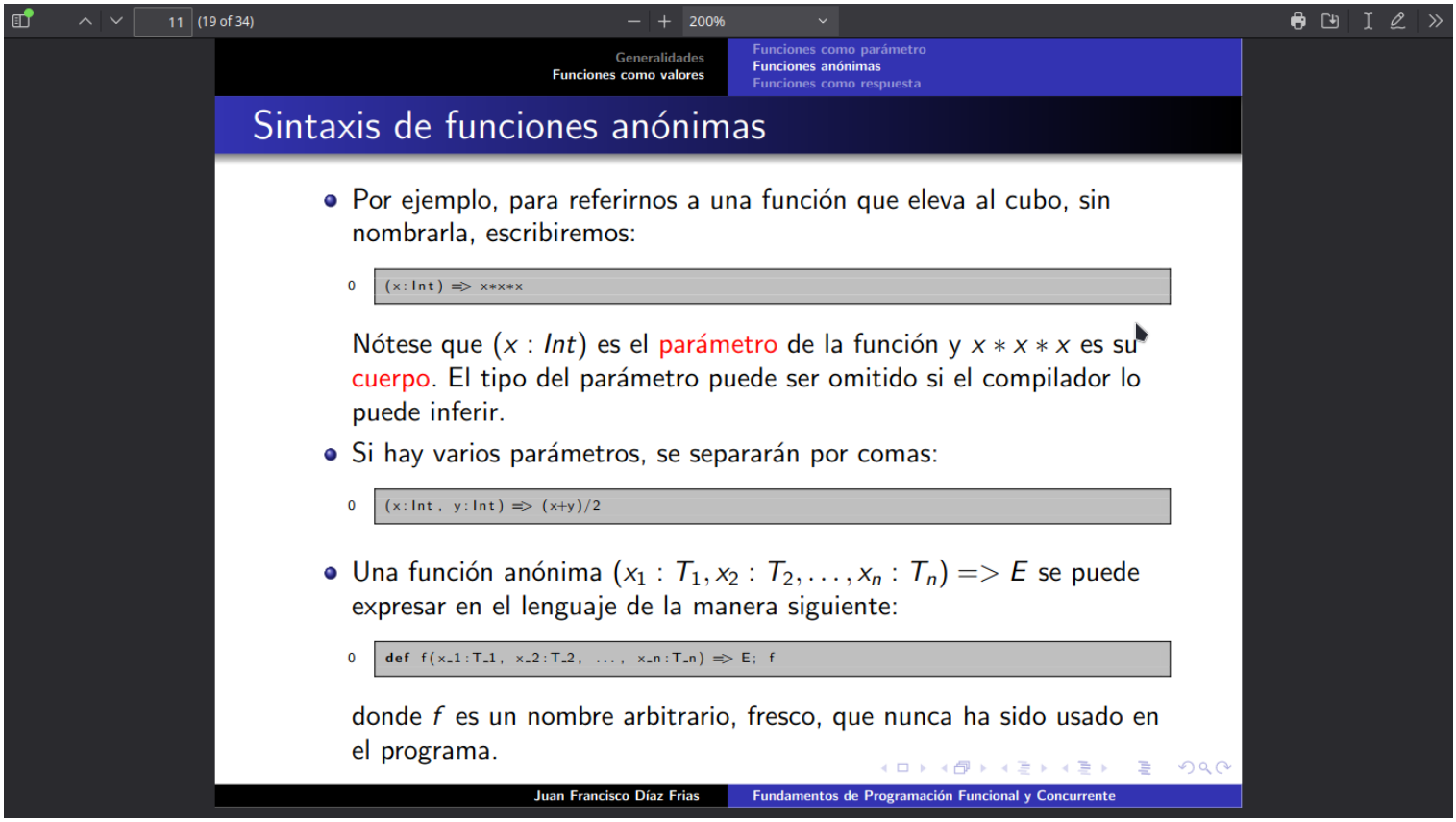
Funciones de alto Orden

Funciones anónimas

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



- 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



Funciones que devuelven funciones

nciones 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
```

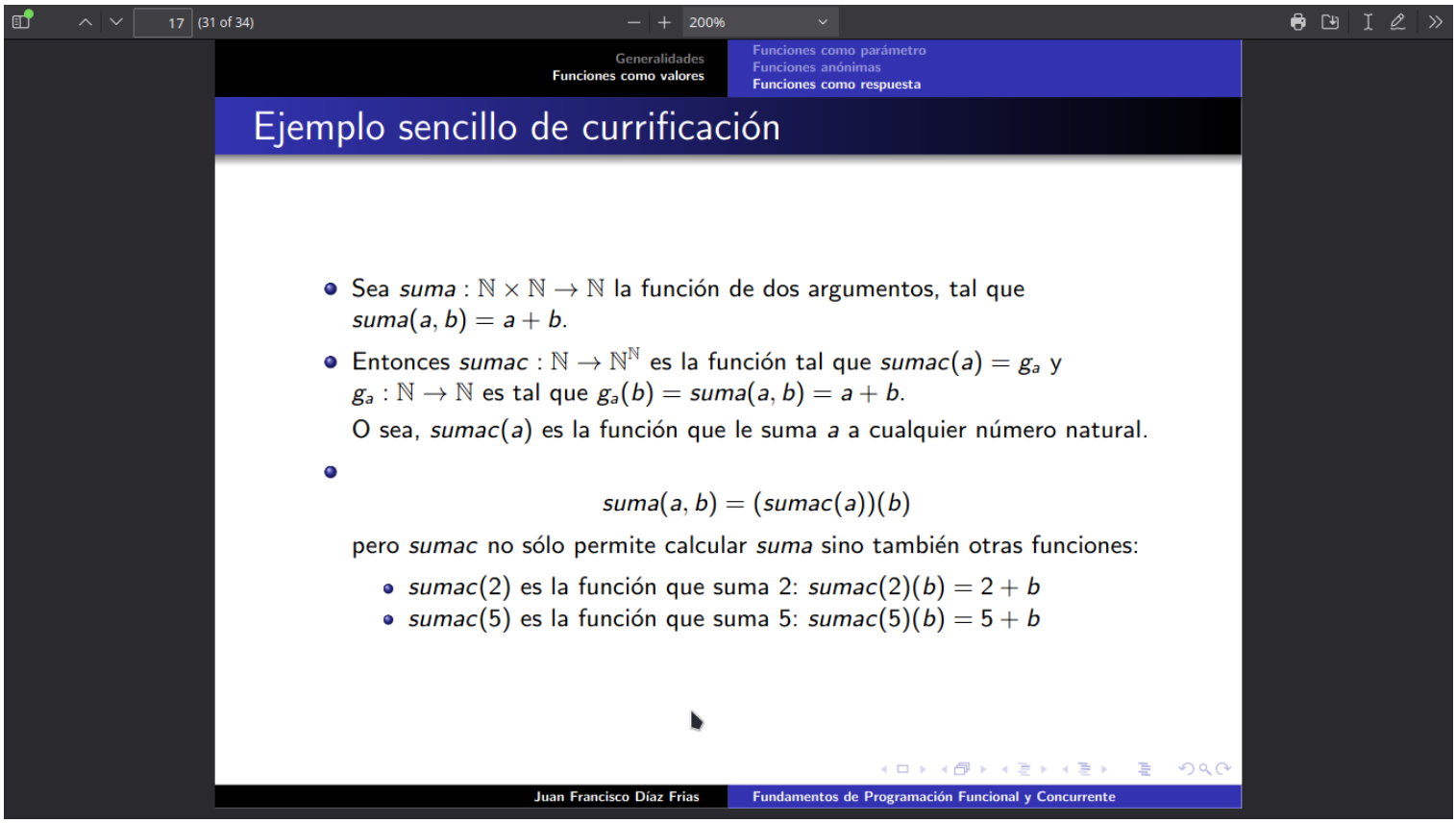
Clausura: Elemento conceptual con el cual se guardan las funciones como valores.

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
```



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