

Fundamentos de Programación Funcional y Concurrente

Elementos de Programación Funcional

Juan Francisco Díaz Frias

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



Universidad del Valle

Septiembre 2023

Plan

- 1 Elementos básicos
 - Generalidades
 - El Read-Eval-Print Loop
 - Expresiones y evaluación
- 2 Estrategias de evaluación y terminación
 - El modelo de substitución
 - Evaluación y terminación
- 3 Condicionales y definición de valores
 - Expresiones condicionales
 - Definiciones de valores
- 4 El método de Newton para calcular la raíz cuadrada
 - La tarea
 - El método
 - Implementación en Scala
- 5 Bloques y alcance léxico
 - Bloques
 - Alcance Léxico

Generalidades

Todo lenguaje de programación provee:

- **Expresiones primitivas** representando los elementos más sencillos
- Maneras de **combinar expresiones**
- Maneras de **abstraer expresiones**, permitiendo nombrarlas y luego referirse a ellas por su nombre

El Read-Eval-Print Loop

Se puede descargar este proyecto

<https://github.com/cardel/talleres-funcional-template>

- La programación funcional es similar a usar una calculadora
- Una capa interactiva (El Read-Eval-Print Loop o **REPL**) le permite escribir expresiones a las que responde con su valor
- El REPL de Scala puede iniciarse desde:
 - En Windows (CMD):
`gradlew repl --console plain`
 - En Linux (Consola):
`./gradlew repl --console plain`

Expresiones

Interacciones sencillas con REPL:

- Como calculadora

```
0 scala> 87+145  
1 val res3: Int = 232
```

- Es más que las calculadoras pues permite definir valores y calcular con ellos

```
0 scala> def size=2  
1 def size: Int  
2  
3 scala> 5*size  
4 val res4: Int = 10
```

Evaluación de expresiones

- Toda **expresión no primitiva** (aplicar un operador a unos operandos) se evalúa de la siguiente manera:
 - Identifique el operador principal de la expresión
 - Evalúe sus operandos, de izquierda a derecha
 - Aplique el operador a los operandos
- Un **nombre** se evalúa substituyéndolo por lo que hay en el lado derecho de su definición.
- El proceso de evaluación **termina** una vez se tiene un valor (un valor es, por ahora, un número).

Ejemplo de evaluación de una expresión sencilla

¿Cómo se evaluaría la expresión: $(2 * Size) * Size$?

- Identificar el operador principal: $(2 * Size) * Size$
- Evaluar el primer operando $(2 * Size)$
 - Identificar el operador principal: $2 * Size$
 - Evaluar el primer operando 2
 - Evaluar el segundo operando $Size$. Como es un nombre, devuelve el lado derecho: 2
 - Aplicar el operador $*$ a los operandos 2 y 2 devolviendo 4
- Evaluar el segundo operando $Size$. Como es un nombre, devuelve el lado derecho: 2
- Aplicar el operador $*$ a los operandos 4 y 2 devolviendo 8

Parámetros

Las definiciones pueden tener parámetros:

```
0 scala> def square(x:Double)= x*x
1 def square(x: Double): Double
2 scala> square(2)
3 val res5: Double = 4.0
4 scala> square(5+4)
5 val res6: Double = 81.0
6 scala> square(square(4))
7 val res7: Double = 256.0
8 scala> def sumOfSquares(x:Double, y:Double)= square(x) + square(y)
9 def sumOfSquares(x: Double, y: Double): Double
```

Los tipos de los parámetros de las funciones se escriben después de `:`.
El tipo devuelto por la función se escribe después de `:`, posterior a la lista de parámetros.

```
0 scala> def sumOfSquares(x:Double, y:Double):Double= square(x) + square(y)
1 def sumOfSquares(x: Double, y: Double): Double
```

Los tipos son como en Java, pero con mayúscula:

```
0 Int, Double, Boolean
```


Evaluación de aplicación de funciones definidas por el programador

La aplicación de funciones con parámetros se evalúa de manera similar a las expresiones no primitivas:

- Evalúe los argumentos de izquierda a derecha
- Reemplace la aplicación de la función por su cuerpo (lado derecho del `=`) y,
- Reemplace en ese cuerpo, los parámetros formales por los argumentos actuales evaluados
- Evalúe esta nueva expresión

Por ejemplo:

```
0 sumOfSquares(3, 2+2)
1 sumOfSquares(3, 4)
2 square(3) + square(4)
3 3*3 + square(4)
4 9 + square(4)
5 9 + 4*4
6 9 + 16
7 25
```

El modelo de substitución

Este esquema de evaluación de expresiones se denomina **El modelo de substitución**

- La idea subyacente a este modelo es que evaluar es **reducir** la expresión a un valor.
- Este modelo puede ser aplicado a todas las expresiones, siempre que no tengan efectos de borde.
- El modelo de substitución se formaliza con el **cálculo λ** , el cual es el fundamento de la programación funcional.

El modelo de substitución

Este esquema de evaluación de expresiones se denomina **El modelo de substitución**

- La idea subyacente a este modelo es que evaluar es **reducir** la expresión a un valor.
- Este modelo puede ser aplicado a todas las expresiones, siempre que no tengan efectos de borde.
- El modelo de substitución se formaliza con el **cálculo λ** , el cual es el fundamento de la programación funcional.

¿Toda expresión reduce a un valor (en un número finito de pasos)?

Cambiando la estrategia de evaluación

En lugar de reducir los argumentos a valores antes de aplicar la función, se podría aplicar primero la función a los argumentos, sin reducir:

```
0  sumOfSquares(3, 2+2)
1  square(3) + square(2+2)
2  3*3 + square(2+2)
3  9 + square(2+2)
4  9 + (2+2)*(2+2)
5  9 + 4*(2+2)
6  9 + 4 * 4
7  9+16
8  25
```

A esta estrategia de evaluación se le denomina **evaluación por nombre (call-by-name)**, CBN, en contraposición al modelo de sustitución que se le denomina **evaluación por valor (call-by-value)**, CBV.

CBV vs CBN

- Ambas estrategias de evaluación reducen al mismo valor final siempre y cuando:
 - La expresión sea puramente de funciones, y
 - Ambas evaluaciones terminen
- La evaluación por valor tiene la ventaja que **evalúa cada argumento una sola vez**.
- La evaluación por nombre tiene la ventaja que si un argumento no es usado en el cuerpo de la función, **dicho argumento no será evaluado nunca**.

CBV vs CBN, eficiencia

Considere la siguiente definición:

```
0  def test(x:Int, y:Int) = x*x
```

Para cada caso a continuación, analice cuál estrategia es más eficiente, o si son iguales en número de pasos de la reducción:

```
0  test(2, 3)
1  test(3+4, 8)
2  test(7, 2*4)
3  test (3+4, 2*4)
```

CBN

$$0) \text{ test}(2, 3)$$

$$1) 2 \times 2$$

$$2) 4$$

$$0) \text{ test}(2+4, 8)$$

$$1) (2+4) \times (2+4)$$

$$2) 6 \times (2+4)$$

$$3) 6 \times 6$$

$$4) 36$$

$$0) \text{ test}(7, 2+4)$$

$$1) 7 \times 7$$

$$2) 49$$

∴)

$$0) \text{ test}(3+4, 2 \times 4)$$

$$1) (3+4) \times (3+4)$$

$$2) 7 \times (3+4)$$

$$3) 7 \times 7$$

$$4) 49$$

CVB

$$0) \text{ test}(2, 3)$$

$$1) 2 \times 2$$

$$2) 4$$

$$0) \text{ test}(2+4, 8)$$

$$1) \text{ test}(6, 8)$$

$$2) 6 \times 6$$

$$3) 36$$

∴)

$$0) \text{ test}(7, 2+4)$$

$$1) \text{ test}(7, 6)$$

$$2) 7 \times 7$$

$$3) 49$$

$$0) \text{ test}(3+4, 2 \times 4)$$

$$1) \text{ test}(7, 2 \times 4)$$

$$2) \text{ test}(7, 8)$$

$$3) 7 \times 7$$

$$4) 49$$

CBV vs CBN, terminación

Ya sabemos que si ambas estrategias terminan, ambas reducen al mismo valor.

Pero, ¿qué pasa si no está garantizada la terminación?

- Si CBV termina para una expresión e , entonces CBN termina para esa expresión e $CBV \rightarrow CBN$
- Para el otro lado, no se puede garantizar eso. **Escriba un caso en que CVN termine pero CBV no**

$CBN \nrightarrow CBV$

CBV vs CBN, ejemplo de no terminación

Considere la siguiente definición:

```
0  def loop:Int = loop
1  def first(x:Int, y:Int) = x
```

¿Cuál es el resultado de evaluar

```
0  first(1, loop)
```

usando CBV y usando CBN?

La estrategia de evaluación de Scala

Scala usa por defecto CBV

Pero, si el tipo de un parámetro de una función es anotado con \Rightarrow , se usará CBN para evaluar ese argumento.

Por ejemplo, considere:

```
0 def constOne(x: Int, y:  $\Rightarrow$  Int) = 1
```

¿Cuál es el resultado de evaluar

```
0 ConstOne(1+2, loop)
```

y

```
0 ConstOne(loop, 1+2)
```

? [Socratic]

Expresiones condicionales

Para expresar escogencia entre dos alternativas, Scala tiene una expresión condicional

```
0 if-else
```

Es similar al de Java, pero se usa para expresiones (es decir siempre devuelve algo) no para declaraciones.

Por ejemplo:

```
0 scala> def abs(x: Int) = if (x >= 0) x else -x
1 def abs(x: Int): Int
```

$(x \geq 0)$ es un **predicado** de tipo Boolean.

Expresiones Booleanas

Las expresiones booleanas son:

```
0  true  false  // Constantes
1  !b     // Neg
2  b && b    // Conj
3  b || b    // Disy
```

Y las operaciones típicas de comparación:

```
0  e <= e, e >= e, e < e, e > e, e == e, e != e
```

Conjunciones && o &

- && es un operador de CORTO-CIRCUITO
- & no es un operador de CORTO-CIRCUITO

Evaluación de expresiones booleanas y condicionales

Reglas de evaluación de las expresiones booleanas:

```
0  !true      → false
1  !false     → true
2  true && e   → e
3  false && e  → false
4  true || e   → true
5  false || e → e
```

Nótese que el operando a la derecha no necesita ser evaluado siempre

Reglas de evaluación de la expresión condicional:

Se evalúa primero el predicado, y luego

```
0  if true e1 else e2 → e1
1  if false e1 else e2 → e2
```

Definiciones de valores

- Así como los parámetros de función pueden ser evaluados por valor o por nombre, así sucede con las definiciones.
- La evaluación de *def* es por nombre: el lado derecho es evaluado cada vez que se usa.
- Existe una manera de hacerlo por valor: *val*

```
0  scala> val x=2
1  val x: Int = 2
2
3  scala> val y = square(x)
4  val y: Double = 4.0
```

El lado derecho de una definición *val* se evalúa en el mismo momento de la definición. Después, el nombre se refiere al valor.

Definición de valores y terminación

- La diferencia entre *val* y *def* es evidente cuando la evaluación del lado derecho no termina.
- Suponga que tiene:

```
0 scala> def loop: Boolean = loop
1           ^
2           warning: method loop does nothing other than call itself recursively
3 def loop: Boolean
```

- Una definición:

```
0 scala> def x = loop
1 def x: Boolean
```

está bien, pero una definición:

```
0 scala> val x = loop
```

se queda en un ciclo infinito.

Hacer una función para calcular \sqrt{x}

- Ilustrar:
 - Definición matemática vs Definición procedimental
 - Qué (declaración) vs Cómo (definición de función)

- Declaración:

$$\sqrt{x} = y : y \geq 0, \text{ y } y^2 = x$$

- ¿Cómo calcular \sqrt{x} ?

Utilizar el método de Newton

- Para calcular \sqrt{x} :
 - Empiece con una estimación inicial y (digamos $y = 1$)
 - Iterativamente mejore la estimación, tomando el promedio entre y y x/y
- Por ejemplo, para calcular $\sqrt{2}$:

Estimación	Cociente	Promedio
1	$\frac{2}{1} = 2$	$\frac{2+1}{2} = 1,5$
1.5	$\frac{2}{1,5} = 1,3333$	$\frac{1,3333+1,5}{2} = 1,4167$
1.4167	$\frac{2}{1,4167} = 1,4118$	$\frac{1,4118+1,4167}{2} = 1,4142$
1.4142

x al cual quieren sacar una raiz cuadrada = a
delta que es el error que aceptamos

$$\sqrt{x - a^2} \leq d$$

$$Q = 1$$

$$Q = \frac{\frac{x}{Q} - a}{2}$$

Implementación en Scala (1)

```
0  /* Primera version */
1  def abs(x:Double)= if (x>0) x else -x
2  def mejorar(estim:Double, x:Double) = (estim + x/estim)/2
3  def esBuenaEstim(estim:Double, x:Double) = abs(estim*estim - x) < 0.001
4  def raizCuadlter(estim:Double, x:Double): Double =
5      if (esBuenaEstim(estim, x)) estim else raizCuadlter(mejorar(estim, x), x)
6
7  def raizCuad(x:Double) = raizCuadlter(1,x)
```

Nótese que:

- *raizCuad* lanza el cálculo invocando a *raizCuadlter* con el **estado inicial**
- *raizCuadlter* está **definida recursivamente** (en el lado derecho se invoca a sí misma)
- La definición de *raizCuadlter* **explicita el tipo que devuelve** (obligatorio por ser recursiva; opcional si no)
- Aunque la definición es recursiva, **implementamos una computación iterativa**, sin uso de estructuras de control como *while*, *for*, *repeat*, ...
- ¿Qué pasaría si el *if* se evaluara como una expresión normal?

Funciones como cajas negras

- El problema de calcular \sqrt{x} fué descompuesto en subproblemas:
$$\text{raizCuad} \left\{ \begin{array}{l} \text{raizCuadIter} \left\{ \begin{array}{l} \text{mejorar} \\ \text{esBuenaEstim} \end{array} \right\} \text{abs} \end{array} \right.$$
- Crucial: división en tareas modulares.

⇒ Funciones como cajas negras

Funciones anidadas

- Es un buen estilo de programación funcional, definir una tarea como la **composición** de varias funciones más sencillas
- Los nombres de las funciones como *raizCuadIter*, *esBuenaEstim*, *mejorar*, ... son útiles para la implementación de *raizCuad* pero no para nadie más.
- Típicamente, nadie más usará esas funciones
- Solución: estructura de bloques

```
0  /* Segunda version */
1  def raizCuad(x: Double) = {
2      def abs(x: Double) = if (x > 0) x else -x
3
4      def mejorar(estim: Double, x: Double) = (estim + x / estim) / 2
5
6      def esBuenaEstim(estim: Double, x: Double) = abs(estim * estim - x) < 0.001
7
8      def raizCuadIter(estim: Double, x: Double): Double =
9          if (esBuenaEstim(estim, x)) estim else raizCuadIter(mejorar(estim, x), x)
10
11      raizCuadIter(1, x)
12  }
```

Bloques en Scala

- Un *Bloque* está delimitado por corchetes `{...}`

```
0 scala> {val y= raizCuad(2)
1           y*y
2         }
3 val res2: Double = 2.0000060073048824
```

- Un bloque contiene una secuencia de definiciones o expresiones
- El último elemento de un bloque es una expresión que define su valor
- Esta expresión de retorno, puede estar precedida de definiciones auxiliares
- Los bloques son, ellos mismos, expresiones, y por tanto, pueden ser usados en todo lugar donde una expresión se puede usar.

Bloques y visibilidad

- Las definiciones dentro de un bloque, sólo son visibles dentro de ese bloque.
- Las definiciones dentro de un bloque ocultan las definiciones del mismo nombre por fuera del bloque

```
0  val x=0
1  def f(y: Int) = y+1
2  val result = {
3      val x=f(3)
4      x*x
5  }
```

- Qué valor calcula la siguiente expresión en *result*:

```
0  val x=5
1  def f(y: Int) = y+1
2  val result = {
3      val x=f(3)
4      x*x
5  } + x
```


Alcance léxico

- Las definiciones de bloques externos, son visibles dentro de un bloque a menos que ellas hayan sido ocultadas
- Por tanto, podemos simplificar *raizCuad*, eliminando ocurrencias redundantes de *x*, donde éste signifique lo mismo:

```
0  /* Tercera version */
1  def raizCuad(x: Double): Double = {
2      def raizCuadIter(estim: Double): Double = {
3          if (esBuenaEstim(estim)) estim else raizCuadIter(mejorar(estim))
4      }
5      def mejorar(estim: Double) = (estim + x/estim)/2
6      def esBuenaEstim(estim: Double) = abs(estim*estim - x) < 0.001
7      def abs(x: Double) = if (x > 0) x else -x
8      raizCuadIter(1.0)
9  }
```