

# Linguagens de Programação

---

Fabio Mascarenhas - 2013.1

<http://www.dcc.ufrj.br/~fabiom/lp>

# Funções de alta ordem

---

- Em uma linguagem funcional, uma função é um valor como qualquer outro
- Isso quer dizer que funções podem ser passadas como parâmetros para outras funções e retornadas de outras funções
- Passagem e retorno de funções nos dá uma ferramenta poderosa para composição de programas
- Funções que recebem ou retornam outras funções são chamadas de *funções de alta ordem*

# Exemplo

---

- Seja uma função que calcula a soma dos inteiros entre  $a$  e  $b$ :
  - `def somaInt(a: Int, b: Int): Int = if (a > b) 0 else a + somaInt(a + 1, b)`
- Seja agora uma função que calcula a soma dos *quadrados* dos inteiros entre  $a$  e  $b$ :
  - `def quadrado(x: Int) = x * x`
  - `def somaQuad(a: Int, b: Int): Int = if (a > b) 0 else quadrado(x) + somaQuad(a + 1, b)`

# Exemplo

---

- Seja agora uma função que soma os *fatoriais* dos inteiros entre  $a$  e  $b$ :
  - `def somaFat(a: Int, b: Int): Int = if (a > b) 0 else fat(a) + somaFat(a + 1, b)`
- Todas essas funções são muito parecidas! Elas são casos especiais de:

$$\sum_{n=a}^b f(n)$$

# Somatório

---

- Vamos definir:

- `def soma(f: Int => Int, a: Int, b: Int): Int = if (a > b) 0  
else f(a) + soma(f, a + 1, b)`

- Agora podemos escrever:

- `def somaInt(a: Int, b: Int) = soma(id, a, b)`

- `def somaQuad(a: Int, b: Int) = soma(quadrado, a, b)`

- `def somaFat(a: Int, b: Int) = soma(fat, a, b)`

- Onde `id` é `def id(x: Int) = x`

# Tipos de função

---

- Repare no tipo de f:
  - $\text{Int} \Rightarrow \text{Int}$
- Um tipo  $A \Rightarrow B$  é o tipo de uma *função* que recebe um argumento do tipo A e retorna um resultado de tipo B, logo  $\text{Int} \Rightarrow \text{Int}$  é uma função que recebe um inteiro e retorna um inteiro

# Funções anônimas

---

- Passar funções como parâmetros leva à criação de muitas funções pequenas
- Às vezes queremos denotar uma função sem precisar dar um *nome* para ela
  - Do mesmo jeito que usamos literais como 2, 3, “foo” e expressões como `List(1, 2, 3)`
- Scala fornece uma sintaxe de *funções anônimas* para isso

# Sintaxe de funções anônimas

---

- Exemplo: uma função que eleva seu argumento ao quadrado:

- $(x: \text{Int}) \Rightarrow x * x$

- O termo  $(x: \text{Int})$  dá a lista de parâmetros da função anônima, e  $x * x$  é o seu corpo

- Em algumas ocasiões o tipo do parâmetro pode ser omitido!

- Funções anônimas com vários parâmetros são possíveis:

- $(x: \text{Int}, y: \text{Int}) \Rightarrow x + y$



# Somatório com funções anônimas

---

- Podemos definir as funções de somatório usando soma e funções anônimas:
  - `def somaInt(a: Int, b: Int) = soma(x => x, a, b)`
  - `def somaQuad(a: Int, b: Int) = soma(x => x * x, a, b)`
  - `def somaFat(a: Int, b: Int) = soma(fat, a, b)`
- Repare que não precisamos dizer o tipo do parâmetro `x` das funções anônimas, pois o compilador sabe que soma precisa de uma função `Int => Int`
- Se não precisamos dizer o tipo, e a função anônima tem apenas um parâmetro, então podemos omitir os parênteses também

# Exercícios

---

- Escreva uma função produto que calcula o produto dos valores de uma função de inteiros para inteiros em determinado intervalo
- Escreva a função que calcula o fatorial em termos de produto
- É possível generalizar tanto soma quanto produto?

# Voltando aos somatórios

---

- Relembrem as funções de somatório:
  - `def somaInt(a: Int, b: Int) = soma(x => x, a, b)`
  - `def somaQuad(a: Int, b: Int) = soma(x => x * x, a, b)`
  - `def somaFat(a: Int, b: Int) = soma(fat, a, b)`
- Reparem que os parâmetros a e b são passados sem mudanças para a função soma
- Podemos nos livrar deles fazendo soma retornar uma função!

# Funções retornando funções

---

- Vamos reescrever soma:

```
def soma(f: Int => Int): (Int, Int) => Int = {  
  def somaF(a: Int, b: Int): Int =  
    if (a > b) 0  
    else f(a) + somaF(a + 1, b)  
  somaF  
}
```

- Ela agora é uma função que retorna outra função: repare no seu tipo de retorno

# Definições parciais

---

- Agora podemos redefinir os somatórios como:
  - `def somaInt(a: Int, b: Int) = soma(x => x)`
  - `def somaQuad(a: Int, b: Int) = soma(x => x * x)`
  - `def somaFat(a: Int, b: Int) = soma(fat)`
- Podemos até usar soma diretamente: `soma(quadrado)(1, 10)`

# Várias listas de parâmetros

---

- Funções retornando funções são tão comuns em programação funcional que Scala tem um atalho para escrevê-las:

```
def soma(f: Int => Int)(a: Int, b: Int): Int =  
  if (a > b) 0 else f(a) + soma(f)(a + 1, b)
```

- Esse estilo de uso de funções é chamado de *currying*, é a maneira normal de definir funções com múltiplos argumentos em algumas linguagens funcionais
- Existem linguagens funcionais em que *todas* as funções recebem apenas um argumento!
- O tipo de soma é `(Int => Int) => (Int, Int) => Int`

# Exemplo: pontos fixos

---

- Em matemática,  $x$  é um *ponto fixo* da função  $f$  se  $f(x) = x$
- Para algumas funções, uma maneira de achar um ponto fixo é começando com uma *estimativa* e repetidamente aplicando  $f$ :
  - $x, f(x), f(f(x)), f(f(f(x))), f(f(f(f(x))))$ , ...
- Até que a variação entre um valor e o próximo seja pequena o suficiente

# Processo de achar ponto fixo

---

- A definição do slide anterior nos dá uma receita para escrever uma função Scala que acha um ponto fixo de outra função:

```
def pontoFixo(f: Double => Double)(est: Double): Double = {  
  val erro = 0.000001  
  def suficiente(est1: Double, est2: Double) =  
    abs(est2 - est1) < erro * est1  
  def loop(est: Double): Double = {  
    val prox = f(est)  
    if (suficiente(est, prox)) prox  
    else loop(prox)  
  }  
  loop(est)  
}
```



# De volta às raízes quadradas

---

- Vamos pensar na *especificação* da função *raiz*:
  - $raiz(x)$  = um número  $y$  tal que  $y * y = x$
- Dividindo ambos os lados da equação  $y * y = x$  por  $y$ :
  - $raiz(x)$  = um número  $y$  tal que  $y = x / y$
- Ou seja uma raiz quadrada de  $x$  é um ponto fixo para a função  $f(y) = x / y$ !

# Raiz quadrada via ponto fixo

---

- Em Scala:
  - `def raiz(x: Double) = pontoFixo(y => x / y)(1.0)`
- Vamos testar!

# Raiz quadrada via ponto fixo

---

- Em Scala:
  - `def raiz(x: Double) = pontoFixo(y => x / y)(1.0)`
- Vamos testar!
- Oops...
- Vamos debugar com *println* e ver o que está acontecendo

# Convergência

---

- A *raiz*( $x$ ) é um número  $y$  tal que  $y = x / y$
- Se adicionarmos  $y$  a ambos os lados da equação e simplificarmos temos:
  - *raiz*( $x$ ) = um número  $y$  tal que  $y = (y + x / y)/2$
- O que acontece se tentamos achar o ponto fixo dessa segunda função em  $y$ ?

# Raiz quadrada, take 3: Newton

---

- Uma terceira maneira de achar uma raiz quadrada de um número  $x$  é achando uma solução da equação  $y^2 - x = 0$
- Um método numérico de achar raízes de funções é o *método de Newton*, que consiste em, para uma equação dada por  $g(x) = 0$ , achar o *ponto fixo* da função:
  - $f(x) = x - g(x)/g'(x)$ , onde  $g'(x)$  é a primeira derivada de  $g(x)$
- Já temos quase todas as ferramentas, só precisamos da derivada! Mas isso é fácil, é só lembrar que:
  - $g'(x) = (g(x + dx) - g(x)) / dx$ , para um  $dx$  pequeno o bastante