

# Linguagens de Programação

---

Fabio Mascarenhas - 2015.2

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

# Casamento de padrões

---

- Nós vimos que podemos criar listas em Scala usando `List` e `::` e podemos *desmontar* listas usando `isEmpty`, `head` e `tail`
- Várias funções em listas que escrevemos começam com um padrão simples:
  - `if (l.isEmpty) ... else ... l.head ... l.tail`
- Mas usar as funções de desmontar a lista para isso não é *idiomático*
- Scala (e outras linguagens funcionais) prefere que usemos a sua sintaxe de *casamento de padrões*

# Match

---

- O casamento de padrões é uma espécie de *switch*, mas em cima de expressões envolvendo os construtores de alguma estrutura de dados
- Usa a palavra chave `match` seguindo a expressão que queremos casar, seguida de um *bloco de casos*
- Cada caso associa uma *padrão* com uma expressão

```
def tamanho[T](l: List[T]): Int = 1 match {  
  case Nil => 0  
  case h :: t => 1 + tamanho(t)  
}
```

padrões

objeto


# Padrões

---

- Um padrão usa:
  - *Construtores*, como `List`, `Nil` e `::`
  - *Variáveis*, como `h`, `t`, `foo`, etc.
  - O *coringa* `_`
  - *Constantes*, como `1`, `“foo”`, `true`
- Uma variável só pode aparecer uma vez em um padrão, já que um padrão *define* variáveis

# Casando um padrão

---

- Um padrão como  $List(p_1, \dots, p_n)$ , casa uma lista que pode ser construída com o construtor `List` e argumentos que casam com os padrões  $p_1, \dots, p_n$
- `Nil` casa com a lista vazia
- $p_1 :: p_2$  casa com uma lista não vazia se  $p_1$  casar com a cabeça da lista e  $p_2$  com a cauda
-  Uma variável casa com qualquer valor, e é associada a esse valor dentro da expressão associada ao padrão
- Uma constante casa com um valor igual a ela (ou seja, ela mesma)
- `_` também casa com qualquer valor, mas pode ser usado várias vezes

# Exemplos

---

- `List(x, 2, y)` casa com uma lista de três elementos se o segundo elemento for igual a 2, e associa `x` ao primeiro elemento e `y` ao terceiro

- `x :: 2 :: y :: Nil` é equivalente ao padrão acima

*x :: 2 :: y*

- `h :: t` casa com qualquer lista não vazia e associa `h` à cabeça e `t` à cauda

- `_ :: x :: _` casa com qualquer lista com pelo menos dois elementos, e associa `x` ao segundo

# Avaliando *match*

---

- Uma expressão e `match { case p1 => e1 ... case pn => en }` primeiro avalia e até obter um valor
- Depois tenta casar esse valor com cada padrão  $p_1, \dots, p_n$  em sequência
- Se casar com o padrão  $p_i$  então avalia-se a expressão  $e_i$  depois de substituir as ocorrências das variáveis que foram associadas pelo padrão
- Se nenhum padrão casa o resultado é um erro

*code -*

# Padrões com *val* (*destructuring bind*)

---

- Podemos usar um padrão como lado esquerdo de um *val*

```
val h :: t = List(1,2,3)
```

- h é associado a 1, t a List(2,3)
- Tenta casar o valor obtido com o lado direito com o padrão, se não conseguir dá erro



# Tuplas

---

- Listas são sequências com um número arbitrário de elementos do mesmo tipo
- Uma *tupla* é uma sequência com um número *fixo* de elementos de diferentes tipos
  - Generalização de par ordenado
- Um tipo tupla é  $(T_1, \dots, T_n)$ , onde  $T_1, \dots, T_n$  são tipos quaisquer
- Uma constructor de tuplas é  $(e_1, \dots, e_n)$ , onde  $e_i$  são expressões
- Para acessar os elementos de uma tupla usamos casamento de padrões com seu construtor

$(Int, Strings)$   
 $(Int, "foo")$

# Zipper

---

- Andar para a “frente” em uma lista é fácil, mas como andamos de volta para “trás”?

# Zipper

---

- Andar para a “frente” em uma lista é fácil, mas como andamos de volta para “trás”?
- Um *zipper* de uma lista é uma estrutura de dados para isso
- A ideia central é manter uma lista com os elementos que foram visitados
- Um zipper para uma  $List[T]$  é uma tripla  $(List[T], T, List[T])$  onde o elemento do meio é o *foco* (o elemento na posição atual), e as listas à esquerda à direita são os elementos à esquerda e direita do foco, *na ordem na qual eles aparecem*

$List(1, 2, 3, 4, 5)$   
 $(List(2, 1), 3, List(4, 5))$

# Zipper

---

```
def zipper[T](l: List[T]): (List[T], T, List[T]) = match l {  
  case Nil => error("lista vazia")  
  case h :: t => (Nil, h, t)  
}
```

```
def paraFrente(z: (List[T], T, List[T])):  
  (List[T], T, List[T]) = match z {  
  case (e, f, Nil) => error("final do zipper")  
  case (e, f, h :: t) => (f :: e, h, t)  
}
```

```
def paraTras(z: (List[T], T, List[T])):  
  (List[T], T, List[T]) = match z {  
  case (Nil, f, d) => error("início do zipper")  
  case (h :: t, f, d) => (t, h, f :: d)  
}
```

# 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(a) + 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:

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

$$f(x) = x$$

$$f(x) = x^d$$

$$f(x) = x!$$

# 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:

Int => 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

$(\text{Int}, \text{Int}) \Rightarrow \text{Int}$

# 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: Int) => x * x`

- O termo `(x: 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: Int, y: Int) => 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