

Linguagens de Programação

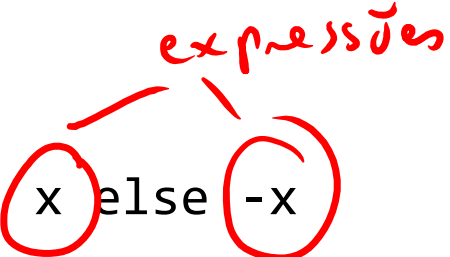
Fabio Mascarenhas - 2017.2

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

Expressões condicionais

- Scala tem uma expressão if-else para expressar escolha entre alternativas que se parece muito com a estrutura de controle de Java, mas é usado com expressões ao invés de comandos (e é uma expressão, ou seja, avalia para um valor)

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



- A condição de uma expressão if-else deve ter tipo Boolean

Expressões booleanas

- Expressões booleanas podem ser
 - Constantes `true` e `false`
 - Negação: `!b`
 - Conjunção (e): `a && b`
 - Disjunção (ou): `a || b`
 - Operadores relacionais: `e1 <= e2`, `e1 == e2`, `e1 != e2`, `e1 >= e2`, `e1 < e2`, `e1 > e2`

Avaliação de expressões booleanas

- A avaliação de expressões booleanas segue as seguintes *regras de reescrita* (a expressão do lado esquerdo é substituída pela do lado direito, onde *e* é uma expressão qualquer):

`!true --> false`

`!false --> true`

`true && e --> e`

`false && e --> false`

`true || e --> true`

`false || e --> e`

curto circuito

- Note que `&&` e `||` são operadores de “curto-circuito”, ou seja, às vezes eles não precisam avaliar ambos os operandos

Avaliação do if-else

- As regras de avaliação de uma expressão if-else são intuitivas:

`if(true) e1 else e2 --> e1`

`if(false) e1 else e2 --> e2`

- Naturalmente, primeiro é preciso avaliar a expressão condicional até se obter seu valor booleano!

val vs. def

- Até agora usamos def para definir tanto valores quanto funções, mas para valores o normal em Scala é usar val

```
val raio = 10
```

- A diferença entre def e val para valores é a mesma entre parâmetros CBN e CBV, com val vamos sempre avaliar o lado direito da definição, e o valor resultante é usado
- Fica óbvio se o lado direito da definição é uma expressão que não termina!

Exemplo: raiz quadrada

- Vamos definir uma função para calcular a raiz quadrada de um número, usando o método de Newton (aproximações sucessivas)

```
def raiz(x: Double) = ...
```

- Começamos com uma *estimativa* y para a raiz de x (por ex., $y = 1$), obtemos a média entre y e x/y para ter uma nova estimativa, e repetimos o processo até o grau de precisão desejável
- Exemplo para $x = 2$

$$\begin{aligned} \frac{(1) + 2}{2} &\Rightarrow \frac{1.5 + 2/1.5}{2} \Rightarrow \frac{1.417 + 2/1.417}{2} \\ &\Downarrow \\ &1.414213562 \end{aligned}$$

Implementação Raiz Quadrada

- Função recursiva que computa um passo do processo

```
def raizIter(est: Double, x: Double): Double =  
    if (suficiente(est, x)) est else raizIter(melhora(est, x), x)
```

- Já temos precisão suficiente

```
def suficiente(est: Double, x: Double) =  
    abs(quadrado(est) - x) < 0.001
```

- Melhora a estimativa

```
def melhora(est: Double, x: Double) = (est + x / est) / 2
```

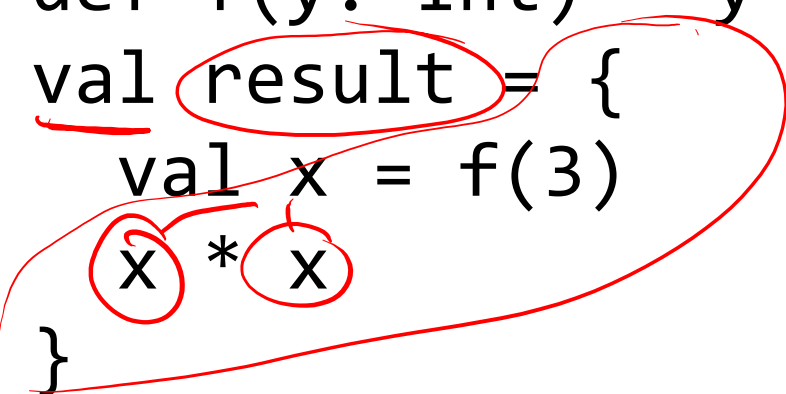

Blocos

- As funções auxiliares que fazem parte da implementação de raiz (raizIter, suficiente, melhora) não precisam ficar visíveis para o programa todo
- Podemos defini-las dentro de raiz usando um *bloco* como corpo de raiz
- Um bloco é delimitado por `{ }`, e é uma expressão que contém uma sequência de definições e expressões
- O *último* elemento do bloco deve ser uma expressão que vai dar o valor de todo o bloco
- As definições em um bloco só são visíveis dentro desse bloco

Exercício: blocos e escopo

- Qual o valor de result no programa abaixo?

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



Recursão Final

- Sejam as duas funções abaixo

```
def mdc(a: Int, b: Int): Int = if (b == 0) a else mdc(b, a % b)
```

```
def fat(x: Int): Int = if (x < 2) 1 else x * fat(x - 1)
```

- Vamos avaliar mdc(14, 21) e fat(4) passo a passo
- Qual a diferença entre as duas sequências?

Recursão Final

- Se o tamanho do termo sendo avaliado permanece em uma faixa constante durante o processo de avaliação, então deve ser possível implementar o processo de avaliação em uma quantidade constante de memória!
 - A recursão em mdc (e em raizIter) não precisa “estourar a pilha”
 - Esse tipo de chamada de função tem o nome de recursão final (*tail recursion*), ou *chamada final* (*tail call*)
- Geralmente linguagens funcionais implementam chamadas finais dessa forma, mas Scala, por limitações da JVM, não faz isso por padrão

Recursão Final em *Scala*

- Se uma função recursiva usa recursão final, você pode anotar sua definição com a anotação `@tailrec`, e o compilador Scala vai otimizar a chamada recursiva
- Se a chamada não for final o compilador vai reclamar

`@tailrec`

`def mdc(a: Int, b: Int): Int = if (b == 0) y else mdc(b, a % b)`

Listas

- Em Scala, o tipo `List[T]` é o tipo de listas imutáveis para algum tipo `T`
- Uma lista imutável é uma estrutura de dados recursiva que pode ser:
 - `List()`
 - Uma lista vazia (`Nil`), ou
 - Um par de um elemento do tipo `T` (a cabeça, ou *head*, da lista), e outra lista do tipo `List[T]` (a cauda, ou *tail*, da lista)

Construindo Listas

- Uma maneira de construir uma lista é através do operador `::` (*cons*)
 - `1 :: (2 :: (3 :: Nil))` constrói uma `List[Int]` com os elementos 1, 2 e 3
 - `::` é associativo a **direita**, então `1 :: 2 :: 3 :: Nil` é o mesmo que `1 :: (2 :: (3 :: Nil))`
 - O operando esquerdo é sempre um elemento de um tipo `T`, e o direito uma lista de tipo `List[T]`
- Um atalho para construir uma lista é a função `List(...)`, que recebe um número arbitrário de argumentos de um tipo `T` e constrói uma `List[T]` com eles
- `List(1, 2, 3)` constrói a mesma lista que a expressão acima

Desconstruindo Listas

- Scala tem diversas funções que operam em listas, as três primeiras que vamos usar são
 - `l.isEmpty`, que retorna `true` se `l` é uma lista vazia (`Nil`) ou `false` se não for
 - `l.head`, que retorna a cabeça de `l` (seu primeiro elemento)
 - `l.tail`, que retorna a cauda de `l` (uma lista com o segundo elemento em diante, que pode ser vazia)
- Vamos usar essas funções para definir uma função `concat[T](l1: List[T], l2: List[T]): List[T]` que retorna a *concatenação* das listas `l1` e `l2`

Concat

```
def concat[T](l1: List[T], l2: List[T]): List[T] =  
  if (l1.isEmpty)  
    l2  
  else  
    l1.head :: concat(l1.tail, l2)
```

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

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

Handwritten examples in red:
`Nil`
`List()`
`h::t`
`List(x1, x2, x3)`
`x1::x2::x3::Nil`
`1::2::t`

Casando um padrão

- Um padrão como $\text{List}(p_1, \dots, p_n)$, 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
- `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* $\{ \text{case } p_1 \Rightarrow e_1 \dots \text{case } p_n \Rightarrow e_n \}$ 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

Case V \Rightarrow — ✓ —

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