Linguagens de Programação

Fabio Mascarenhas - 2015.2

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

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
  - 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)
```