Linguagens de Programação

Fabio Mascarenhas - 2013.1

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

Introdução

- Quais das expressões abaixo têm o mesmo significado?
  - a[42]
  - (vector-ref a 42)
  - a[42]
  - a[42]

Introdução

- Quais das expressões abaixo têm o mesmo significado?
  - a[42]
  - (vector-ref a 42)
  - a[42]
  - a[42]

Java

Introdução

- Quais das expressões abaixo têm o mesmo significado?
  - a[42] Java
  - (vector-ref a 42) Scheme
  - a[42]
  - a[42]

Introdução

- Quais das expressões abaixo têm o mesmo significado?
  - a[42] — Java
  - (vector-ref a 42) — Scheme
  - a[42] — C
  - a[42]

Introdução

- Quais das expressões abaixo têm o mesmo significado?
  - a[42] Java
  - (vector-ref a 42) Scheme
  - a[42] C
  - a[42] Haskell

Introdução

- Quais das expressões abaixo têm o mesmo significado?
  - a[42] — Java ←
  - (vector-ref a 42) — Scheme ←
  - a[42] — C
  - a[42] — Haskell

Introdução

- Quais das expressões abaixo têm o mesmo significado?
  - a[42] — Java ←
  - (vector-ref a 42) — Scheme ←
  - a[42] — C
  - a[42] — Haskell

- Nesse curso vamos estudar o *significado* dos programas, e os diferentes *paradigmas* de programação

- Outro nome para significado é *semântica*

Como estudar semântica?

- Precisamos de uma linguagem pra descrever semântica
- Técnicas matemáticas?

Como estudar semântica?

- Precisamos de uma linguagem pra descrever semântica
- Técnicas matemáticas?
  - Denotacional
  - Operacional
  - Axiomática

Como estudar semântica?

- Precisamos de uma linguagem pra descrever semântica
- Técnicas matemáticas?
  - Denotacional
  - Operacional
  - Axiomática
- Não, vamos usar *interpretadores* escritos no paradigma de *programação funcional*

Paradigmas de Programação

- Paradigmas de programação descrevem uma maneira de se programar, e de se *raciocinar* sobre programas
  - Programação imperativa
  - Programação funcional
  - Programação lógica
  - Programação OO
- Os paradigmas não são totalmente independentes

Programação Imperativa

- É o paradigma mais usado, e a maneira mais comum de se usar o paradigma OO
  - Programas são sequências de *comandos*
  - Mutação de variáveis (atribuição)
  - Laços são as estruturas básicas de controle

Programação Imperativa e a Máquina

- Há uma correspondência entre os conceitos da programação imperativa e a linguagem de máquina
  - Variáveis mutáveis são células na memória
  - Acessos às variáveis são instruções de leitura (*load*)
  - Atribuições são instrucões de escrita (*store*)
  - Estruturas de controle e laços são saltos

Programação Funcional

- Na programação funcional, o modelo básico é mais distante da máquina, e mais próximo da matemática
  - Programas são *expressões* que eventualmente retornam um valor
  - Programação com valores imutáveis e operações envolvendo esses valores
  - Sem variáveis mutáveis, sem atribuição, sem laços e outras estruturas de controle imperativas
  - Funções como mecanismo básico de abstração, e funções como valores que podem ser produzidos, consumidos e combinados
- A proximidade com a matemática torna a programação funcional uma maneira natural de estudar a semântica de linguagens via interpretadores

Scala

- Em nosso curso vamos usar *Scala* como exemplo de linguagem para programação funcional, e como linguagem para escrever interpretadores
- Scala é uma linguagem multi-paradigma, mas vamos nos ater a seus aspectos de programação funcional, e evitar suas partes imperativas/OO
- Scala é uma linguagem que roda na JVM (Java Virtual Machine), e através dela temos acesso a todo o acervo de bibliotecas disponível para Java
- Veja nosso site (http://www.dcc.ufrj.br/~fabiom/lp) para instruções de como baixar e usar o compilador Scala

Elementos Básicos da Programação Funcional

- Scala, e qualquer outra linguagem de programação funcional, oferece:
  - expressões primitivas representando os elementos mais simples da linguagem
  - operações que permitem *combinar* expressões
  - maneiras de *abstrair* expressões, dando um nome e parâmetros para uma expressão de modo que ela possa ser reutilizada

REPL

- Uma maneira comum de interagir com Scala é através de seu REPL (Read-Eval-Print-Loop, ou laço leitura-avaliação-exibição), uma espécie de “linha de comando” para a linguagem

- O REPL permite escrever expressões e examinar seus valores de maneira interativa

- A maneira mais fácil de iniciar o REPL Scala é com `sbt console` na linha de comando

Avaliação de Expressões

- Uma expressão não primitiva é avaliada da seguinte maneira
  - Pegue o operador de menor precedência mais à direita
  - Avalie seus operandos, primeiro o esquerdo, depois o direito
  - Aplica a operação aos operandos
- Um nome é avaliado substituindo o nome pelo lado direito de sua definição

Exemplo

```
> def pi = 3.14159
pi: Double
> def raio = 10
raio: Int
> 2 * pi * raio
res0: Double = 62.8318
```

Parâmetros

- Definições podem ter parâmetros, definindo *funções*
  - `> def quadrado(x: Double) = x * x`
  - `> def somaDeQuadrados(x: Double, y: Double) = quadrado(x) + quadrado(y)`
- Notem que precisamos dizer os *tipos* dos parâmetros, mas normalmente o tipo que a função retorna é opcional (o compilador Scala consegue deduzi-lo na maior parte dos casos)
- Tipos primitivos são como os de Java, mas escritos com a primeira letra maiúscula: Int, Double, Boolean

Avaliando Chamadas de Função

- Uma chamada (ou *aplicação*) de função é avaliada de modo parecido com um operador
  - Avalia-se os argumentos da função, da esquerda para a direita
  - Avalia-se o lado direito da definição da função, substituindo os parâmetros pelos valores dos argumentos

```
> somaDeQuadrados(3, 2+2)
res0: Double = 25.0
```

Call-by-value vs. Call-by-name

- Scala avalia chamadas de função primeiro avaliando os argumentos, mas esta é apenas uma das estratégias de avaliação
- Outra estratégia é substituir os parâmetros pelos argumentos sem primeiro avaliá-los
- A primeira estratégia é a *call-by-value* (CBV), e a segunda é a *call-by-name* (CBN)
- Se as expressões são funções puras, e se ambas produzem um valor, é garantido que as duas estratégias produzem os mesmos resultados
- Mas uma expressão pode produzir um valor avaliada por CBN, mas não via CBV!

Não-terminação

- Sejam as definições
  - `def loop: Double = loop`
  - `def primeiro(x: Double, y: Double) = x`
- Agora vamos avaliar `primeiro(1, loop)` usando as estratégias CBV e CBN
- Scala usa CBV por padrão, mas podemos forçar uma estratégia CBN parâmetro a parâmetro usando => antes do seu tipo
  - `def primeiro(x: Double, y: => Double) = x`

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 se 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 --> false`
- 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

Implementação Raiz Quadrada

- `def raizIter(est: Double, x: Double): Double = if (suficiente(est, x)) est else raizIter(melhora(est, x), x)`
  - Função recursiva que computa um passo do processo
- `def suficiente(est: Double, x: Double) = abs(quadrado(est) - x) < 0.001`
  - Já temos precisão suficiente
- `def melhora(est: Double, x: Double) = (est + x / est) / 2`
  - Melhora a estimativa

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
}
```