

# Linguagens de Programação

---

Fabio Mascarenhas – 2017.2

<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]`

Java

- `(vector-ref a 42)`

- `a[42]`

- `a[42]`

# 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?

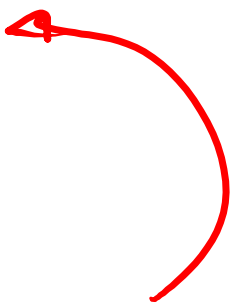

# Como estudar semântica?

---

- 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 instruções 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



# 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
- No IDE IntelliJ IDEA, temos um REPL usando o recurso de *worksheets* (File->New->Scala Worksheet)
- Em uma worksheet, as definições e expressões ficam do lado esquerdo, e os resultados do lado direito

# Avaliação de Expressões

---

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

# Exemplo

---

```
def pi = 3.14159
```

```
def raio = 10
```

```
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
```