

# Linguagens de Programação

---

Fabio Mascarenhas - 2013.1

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

# Busca combinatória

---

- Vimos como expressar com *map*, *reduce*, *filter* e *fold* computações para as quais usaríamos laços em linguagens imperativas, mas até agora nos limitamos a um único laço
- Vamos pensar no seguinte problema: dadas duas listas de números inteiros, achar todos os pares no produto cartesiano dessas listas que são primos entre si
- Para isso, podemos gerar uma lista com todos os pares no produto cartesiano, e filtrar o resultado

# Produto cartesiano

---

- A primeira tentativa de gerar o produto cartesiano de duas listas:

```
l1.map(x => l2.map(y => (x, y)))
```

- Para cada elemento da primeira lista, geramos os pares daquele elemento com os elementos da segunda lista
- Mas isso não dá bem o que queremos: o resultado do *map* externo é um `List[List[(Int,Int)]]`, e não um `List[(Int,Int)]`!
- Uma alternativa é fazer um `foldRight(List[(Int,Int)]())((l1, l2) => l1 ++ l2)` que concatenaria todas essas sublistas

# flatten e flatMap

---

- Concatenar as sublistas de uma `List[List[T]]` em uma `List[T]` é uma operação tão comum que ela também é pré-definida em Scala: *flatten*
- Existe também uma versão de *map* combinada com *flatten*, que é mais eficiente: *flatMap*

```
def flatMap[T,U](l: List[T], f: T => List[U]): List[U] = l match {  
  case Nil => Nil  
  case h :: t => f(h) ++ flatMap(t, f)  
}
```

- Nosso problema de obter os pares do produto cartesiano que são primos entre si pode ser resolvido com:

```
l1.flatMap(x => l2.map(y => (x, y))).filter(  
  { case (x, y) => mdc(x, y) == 1 })
```

# Expressões *for*

---

- Com uma combinação de maps e filtros conseguimos expressar diversas buscas combinatórias, mas a sintaxe atrapalha
- Por isso Scala (e outras linguagens funcionais) tem um açúcar sintático para esse tipo de expressão
- Em Scala esse açúcar é a *expressão for*
- Uma expressão *for* em nada se parece com um laço *for* de uma linguagem imperativa, seu modelo de funcionamento é mais parecido com a notação de conjuntos da matemática:

$$\{ (x, y) \mid x \in I_1 \wedge y \in I_2 \wedge \text{mdc}(x, y) = 1 \}$$

# Expressões *for*

---

- Uma expressão *for* tem o formato:

```
for {  
    <gerador ou filtro>  
    ...  
    <gerador ou filtro>  
} yield <exp>
```

- Um gerador é um termo `<padrão> <- <exp>`, onde `<exp>` é uma expressão de tipo `List[T]` e `<padrão>` é um padrão que casa com um valor de tipo `T`
- Um filtro é um termo `if <exp>`, onde `<exp>` é uma expressão de tipo `Boolean`
- Qualquer variável introduzida pelo lado esquerdo de um gerador é visível em geradores e filtros subsequentes, e na expressão que fecha o *for*

# Exemplo

---

- Uma expressão for para o problema dos primos entre si:

```
for {  
  x <- l1  
  y <- l2  
  if mdc(x, y) == 1  
} yield (x, y)
```

- Uma expressão for com uma expressão final de tipo T avalia para uma List[T]
- Ela é uma expressão como qualquer outra, então podemos continuar processando ela:

```
(for {  
  (x, y) <- l1.zip(l2)  
} yield x * y).foldLeft(0)((a, b) => a + b)
```

# Compilando *for*

---

- Uma expressão *for* é apenas açúcar sintático!

```
for {  
  p <- e1  
  if e2  
  ..  
} yield e3  
→  
for {  
  p <- e1.filter({ case p => e2 })  
  ..  
} yield e3
```

```
for {  
  p <- e1  
  ..  
} yield e2  
→  
e1.flatMap({ case p => for {  
  ..  
} yield e2 })
```

```
for {  
  p <- e1  
} yield e2  
→  
e1.map({ case p => e2 })
```



# Tipos algébricos

---

- Recapitulando, uma lista é definida como a lista vazia ou um par composto de um elemento e outra lista
- Esse tipo de definição de uma estrutura de dados é uma instância de um padrão mais geral: os *tipos algébricos*
- A definição de um tipo algébrico é dada por um ou mais *construtores*, onde os construtores que têm parâmetros do próprio tipo algébrico são os *casos indutivos*, e os construtores que não têm parâmetros do próprio tipo algébrico são os *casos base*
- Para listas, o construtor Nil é o único caso base, e o construtor :: é o único caso indutivo

# Case classes

---

- Tipos algébricos se prestam naturalmente à desconstrução por casamento de padrões, e por isso linguagens funcionais costumam oferecer sintaxe para sua definição
- Em Scala, definimos tipos algébricos usando *case classes*

```
trait Lista[T]
case class Vazia[T]() extends Lista[T]
case class Cons[T](hd: T, tl: Lista[T]) extends Lista[T]

def tamanho[T](l: Lista[T]): Int = l match {
  case Vazia() => 0
  case Cons(hd, tl) => 1 + tamanho(tl)
}
```

# Case classes

---

- A declaração do tipo algébrico é dada por um *trait*
- Cada construtor é uma *case class* que estende o tipo algébrico, e lista seus parâmetros
- O construtor define tanto uma função para construir elementos do tipo algébrico, quanto um padrão para ser usado no *match*
- Podemos definir as funções que operam em um tipo algébrico dentro do seu *trait* também, e usá-las com a sintaxe OO de Scala

# Case classes estilo OO

---

- Definindo uma função *map* em `Lista[T]` com a sintaxe OO:

```
trait Lista[T] {  
  def map[U](f: T => U): Lista[U] = this match {  
    case Vazia() => Vazia[U]  
    case Cons(hd, tl) => Cons(f(hd), tl.map(f))  
  }  
}  
case class Vazia[T]() extends Lista[T]  
case class Cons[T](hd: T, tl: Lista[T]) extends Lista[T]
```

# Option[T]

---

- O tipo `Option[T]` é outro tipo algébrico pré-definido em Scala, para representar valores opcionais de maneira mais segura que o uso de *null*
- Um `Option[T]` pode ser ou `None`, que quer dizer que não há nenhum valor, ou `Some(x: T)`
- Podemos usar `Option[T]` para criar uma versão segura da função que obtém o primeiro elemento de uma lista:

```
def primeiro[T](l: List[T]): Option[T] = l match {  
  case Nil => None  
  case h :: t => Some(h)  
}
```

# Options e *for*

---

- Uma maneira de enxergar um `Option[T]` é como uma lista contendo no máximo um elemento do tipo `T`
- Em Scala, o tipo `Option[T]` também implementa algumas funções que vimos para listas, em especial `flatMap` e `filter`
- Isso quer dizer que podemos usar a sintaxe do `for` com geradores que retornam `Option[T]` também:

```
def multPrimeiro(l1: List[Int], l2: List[Int]): Option[Int] =  
  for {  
    x <- primeiro(l1)  
    y <- primeiro(l2)  
  } yield x * y
```

# Árvores

---

- Uma aplicação comum para tipos algébricos são estruturas em árvore
- Por exemplo, uma árvore binária rotulada é simples de definir:

```
trait ArvoreBin[T]  
case class Folha[T](rot: T) extends ArvoreBin[T]  
case class Ramo[T](rot: T,  
                  esq: ArvoreBin[T],  
                  dir: ArvoreBin[T]) extends ArvoreBin[T]
```

- Claro que variantes são possíveis, como árvores sem rótulos nas folhas, e árvores com rótulos apenas nas folhas

# map e fold em árvores

---

- É bem fácil definir um equivalente de *map* para nossas árvores binárias:

```
trait ArvoreBin[T] {  
  def map[U](f: T => U): ArvoreBin[U] = this match {  
    case Folha(x) => Folha(f(x))  
    case Ramo(x, e, d) => Ramo(f(x), e.map(f), d.map(f))  
  }  
}
```

- Um *fold* é mais complicado; uma maneira seria definir folds equivalentes aos folds resultantes da lista que temos quando caminhamos a árvore em pré-ordem, ordem ou pós-ordem



# Catamorfismos

---

- Catamorfismos são generalizações da operação fold para outros tipos algébricos
- Um catamorfismo é a substituição dos construtores de um tipo algébrico por outras funções
- No caso do fold de listas, o construtor Nil é substituído por uma [função] constante  $z$  e o construtor  $::$  por uma função binária  $f$
- Logo, para árvores binárias, vamos substituir o construtor Folha por uma função de um parâmetro  $f$  e o construtor Ramo por uma função de três parâmetros  $g$ !