

Linguagens de Programação

Fabio Mascarenhas - 2015.2

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

fun - uma mini-linguagem funcional

- Agora que vimos como se *usa* uma linguagem funcional como Scala, vamos estudar como se dá a *semântica* de uma linguagem funcional
- Vamos transformar nosso modelo informal de execução em um modelo *preciso*
- Para isso, vamos construir aos poucos um *interpretador* para uma linguagem funcional simples
- Um interpretador é uma função que vai levar um programa fun (uma árvore representando as expressões do programa) em um valor

fun - Aritmética

- Sintaxe concreta vs abstrata

```
exp : NUM
    | exp '+' exp
    | exp '*' exp
    | '(' exp ')'
```

$E \rightarrow T \{ + T \}$
 $T \rightarrow F \{ * F \}$
 $E \rightarrow \text{NUM} \mid (E)$

```
trait Exp
case class Num(v: Double) extends Exp
case class Soma(e1: Exp, e2: Exp) extends Exp
case class Mult(e1: Exp, e2: Exp) extends Exp
```

- Um *parser* converte, por ex, "2+2*3" em Soma(Num(2), Mult(Num(2), Num(3)))

Disgressão – Parser de Combinadores

- Não é difícil expressar o parser diretamente em uma linguagem funcional, através de *combinadores*
- Um combinador é apenas outro nome para uma função de alta ordem que recebe uma ou mais funções e retorna uma outra função, todas com o mesmo “formato”
- Podemos enxergar um parser como uma função da entrada para a saída do parser, e os combinadores são as diferentes formas de compor um parser (sequência, escolha, opcional, repetição, etc.)
- Nossos parsers vão ser um pouco mais ricos, para termos boa informação em caso de erro de sintaxe

O tipo Parser[A]

- Podemos modelar um parser como uma função, mas para ter acesso à expressão for temos uma *classe implícita* associada

```
type Parser[A] = (Vector[Char], Int, Int, Set[String]) =>  
  (Option[(A, Int)], Int, Set[String])
```

) alias de tipo

```
implicit class RichParser[A](val p: Parser[A]) extends AnyVal {  
  def flatMap[B](f: A => Parser[B]): Parser[B] = bind(p, f)  
  def map[B](f: A => B): Parser[B] = adapt(p, f)  
  def filter(f: A => Boolean): Parser[A] = constrain(p, f)  
}
```

) classe implícita associada

```
def empty[A](v: A): Parser[A] = ???  
def pred(pred: Char => Boolean): Parser[Char] = ???  
def choice[A](p1: Parser[A], p2: Parser[A]): Parser[A] = ???  
def bind[A,B](p: Parser[A], f: A => Parser[B]): Parser[B] = ???  
def adapt[A,B](p: Parser[A], f: A => B): Parser[B] = ???  
def constrain[A](p: Parser[A], f: A => Boolean): Parser[A] = ???  
def not[A,B](p: Parser[A], v: B): Parser[B] = ???  
def expect[A](p: Parser[A], name: String): Parser[A] = ???  
val pos: Parser[Int] = ???
```

) primitivos

Combinadores primitivos e derivados

- As funções do slide anterior são as primitivas de parsing
- Usando elas podemos definir vários outros combinadores úteis
- Por exemplo: reconhecer um caractere, transformar uma `List[Parser[A]]` em um `Parser[List[A]]`, reconhecer zero ou mais ocorrências de um `Parser[A]`, reconhecer uma ou nenhuma ocorrência de um `Parser[A]`, fazer um fold à esquerda de uma sequência de `Parser[A]` intercalados por um operador `Parser[(A,A) => A]`
- Podemos também construir parsers para os tokens de *fun*: espaço em branco, palavras-chave, identificadores, operadores e numerais

fun - Aritmética

- O interpretador de fun pode ser facilmente definido com uma função `eval` dentro de `Exp`, usando casamento de padrões
- O que são números em fun? Números de ponto flutuante de precisão dupla. Por quê? Porque podemos simplesmente usar `Double` em `Scala` e a aritmética de `Scala` para interpretar fun
- Outras representações para números (por ex., inteiros com precisão arbitrária) levariam a outros interpretadores
- A linguagem em que estamos definindo o interpretador influencia a linguagem interpretada, a não ser que tomemos bastante cuidado!