Linguagens de Programação

Fabio Mascarenhas - 2013.1

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 ')'
```

```
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)))`

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 Doubles 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!

Açúcar Sintático

- Podemos acrescentar expressões de subtração e negação a *fun* com modificações simples no parser e na sintaxe abstrata
- Mudar o interpretador (acrescentando casos novos) também não seria difícil, mas vamos implementar esses novos termos via *açúcar sintático*
- A transformação é bem simples: `e1 - e2 => e1 + -1 * e2` `e -e => -1 * e`
- Em nossa linguagem, tanto subtração quanto negação são *açúcar sintático*: uma transformação puramente local de expressões em uma linguagem extendida para uma linguagem mais simples
- Em geral açúcar sintático é implementado direto no parser!

Condicionais

- Para ter mais poder em nossa linguagem, vamos agora introduzir um operador relacional `<` e uma expressão condicional `if`

```
exp : ...
    | exp '<' exp
    | IF exp THEN exp ELSE exp END
```

```
case class Menor(e1: Exp, e2: Exp) extends Exp
case class If(cond: Exp, ethen: Exp, eelse: Exp) extends Exp
```

- Temos um problema: qual deve ser o resultado de `<`? Como o `if` avalia para uma expressão ou para outra?

Booleanos

- Poderíamos adotar a estratégia de C, e dizer que `e1 < e2` é 0 se o valor de e1 for menor que e2, e 1 se não for

- Mas vamos introduzir um novo tipo de dado em *fun*: booleanos

- O interpretador agora não pode mais produzir Doubles, precisamos de um tipo algébrico para os valores de *fun*

```
trait Valor
case class NumV(v: Double) extends Valor
case class Bool(v: Boolean) extends Valor
```