Compiladores 2018.1

Análise da linguagem Ruby e seu suporte a PEG

Júlia Falcão, Raffael Paranhos, Wallace Baleroni

21 de Março de 2018

Universidade Federal Fluminense

Ruby

Ruby é uma linguagem de programação desenvolvida em 1995 no Japão por Yukihiro Matsumoto, e está atualmente entre as 10 linguagens de programação mais populares. Algumas de suas características são:

- Ruby é multiparadigma: suporta programação orientada a objetos (*class-based*), funcional, imperativa e reflexiva.
- Tem tipagem dinâmica e forte: todos os "tipos primitivos" são na verdade classes, e toda variável deve ter um, mas ele pode ser alterado dinamicamente.
- Sua implementação padrão é escrita em C.
- É interpretada e tem gerenciamento de memória automático.
- Seu criador queria uma linguagem de *script* mais poderosa que Perl, e mais orientada a objetos do que Python.

Ruby

A instalação de Ruby é bem simples, com várias maneiras diferentes disponíveis no site oficial:

- **Instaladores**: ruby-build e ruby-install para sistemas UNIX; RubyInstaller para Windows.
- **Gerenciadores de pacotes** (em sistemas UNIX):
  - Homebrew (macOS): `$ brew install ruby`
  - apt (Ubuntu, Debian):
    `$ sudo apt-get install ruby-full`
  - pacman (Arch Linux): `$ sudo pacman -S ruby`
- **Compilando a partir do código fonte**:

```
$ ./configure
$ make
$ sudo make install
```

Ruby

Além da programação **imperativa**, onde o código diz como algo será feito, Ruby também suporta o paradigma **declarativo**, onde é dito apenas o que se quer fazer, sem especificar como.

Exemplo: retornar subconjunto de elementos ímpares de uma lista

- **Imperativa:** manualmente adicionar elementos ímpares ao resultado

```
odds = []
array.each do | element |
    odds << element if element.odd?
end
```

- **Declarativa:** usar *select*

```
array.select { | element | element.odd? }
```

Ruby

Ruby é uma linguagem **interpretada**, e o processo acontece em 4 fases:

1. **Tokenizing:** O programa é quebrado em pequenos pedaços chamados *tokens*.
2. **Lexing:** Dados adicionais são acrescentados aos *tokens*.
3. **Parsing:** O texto é transformado em uma árvore sintática abstrata (AST), uma estrutura de dados que representa o programa em memória.
   Até a versão 1.9, o código podia ser executado diretamente pela análise da AST, mas atualmente há um passo a mais.
4. **Compilação:** Finalmente, a AST é compilada para bytecode, que é então executado pela máquina virtual de Ruby.

JRuby

É uma implementação de Ruby que roda sobre a Java Virtual Machine (JVM). Isso permite o uso da linguagem Ruby normalmente mas com todas as vantagens da JVM, incluindo:

- **Portabilidade:** a JVM pode ser emulada em qualquer sistema que suporte C++, trazendo muito mais portabilidade para Ruby do que a Ruby VM.
- **Performance:** JRuby tem maior *overhead* na inicialização, mas em retorno o *throughput* é bem maior, o que faz valer a pena usá-la para aplicações maiores como aplicações Rails.
- **Concorrência:** Ao contrário da implementação original, JRuby roda *threads* simultaneamente, o que torna o programa mais rápido se ele tiver sido escrito corretamente para isso.

Exemplos

- Hello World:

```
puts 'Hello World!'
```

- Input:

```
print 'Please type name >'
name = gets.chomp
puts "Hello #{name}."
```

Insertion Sort

```
def insertion_sort(array)
    final = [array[0]]
    array.delete_at(0)
    for i in array
        final_index = 0
        while final_index < final.length
            if i <= final[final_index]
                final.insert(final_index,i)
                break
            elsif final_index == final.length-1
                final.insert(final_index+1,i)
                break
            end
            final_index+=1
        end
    end
    final
end
```

Definição de Classe

```
class Customer
   @@no_of_customers = 0
   def initialize(id, name, addr)
      @cust_id = id
      @cust_name = name
      @cust_addr = addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
   end
   def total_no_of_customers()
      @@no_of_customers += 1
      puts "Total number of customers: #@@no_of_customers"
   end
end
```

Suporte à construção de compiladores

Apesar do desempenho relativamente fraco de Ruby torná-la muito mais adequada para aplicações *web* do que a construção de compiladores, o único fator realmente decisivo é se a linguagem possui ou não as estruturas necessárias para construir o *parser*, e ela possui.

Com o auxílio de bibliotecas externas como Treetop e Parslet, é perfeitamente possível usar Ruby para construir um compilador, tanto que podemos encontrar na Internet projetos de compiladores escritos em Ruby para linguagens como x86 assembly e até C. O primeiro compilador de CoffeeScript, uma linguagem que transcompila para JavaScript, era escrito em Ruby.

Rubinius

Rubinius é uma implementação alternativa criada com o objetivo de implementar nativamente o máximo possível da linguagem Ruby usando Ruby. No início do projeto, a biblioteca principal era escrita quase inteiramente em Ruby, assim como o compilador de *bytecode* e o *debugger*. No entanto, Rubinius acabou sendo a primeira implementação de Ruby baseada em C++, utilizando mais C++ do que Ruby no código. A ideia é interessante e é um projeto ambicioso, mas na prática, Rubinius ainda não cumpre a promessa de alto desempenho.

Parsing Expression Grammars (PEGs)

- Criado por Bryan Ford, é um formalismo que descreve um conjunto de regras para reconhecer *strings* em uma linguagem.
- É semelhante à Gramática Livre de Contexto, porém o operador de escolha na PEG seleciona a primeira correspondência enquanto na GLC é ambíguo.
- Não existe ambiguidade, apenas uma árvore de derivação existe para cada *string*
- A biblioteca Parslet permite a construção de *parsers* da forma PEG no Ruby.

Parslet

O processo de criação de um compilador ou interpretador para uma linguagem pode ser dividido em quatro estágios:

1. *Parsing*
2. Construção da Árvore Sintática Abstrata
3. Otimização da Árvore
4. Geração do código ou execução

O Parslet nos auxiliará nos dois primeiros passos através das classes:

```
Parslet::Parser
Parslet::Transform
```

Instalação e Exemplos Parslet

A biblioteca Parslet está disponível na forma de Pacote *Gem* e pode ser instalado com o comando:

```
gem install parslet
```

Criar um *parser* que reconhece numeros é bem simples:

```
class Mini < Parslet::Parser
   rule(:integer) { match('[0-9]').repeat(1) }
   root(:integer)
end
```

Mais exemplos

Extendendo um pouco o *parser* anterior, é possível criar um que reconhece algumas expressões, por exemplo "2 + 2":

```
class Mini < Parslet::Parser
    rule(:integer)    { match('[0-9]').repeat(1)>>space? }
    rule(:space)      { match('\s').repeat(1) }
    rule(:space?)     { space.maybe }
    rule(:operator)   { match('[+]')>>space? }
    rule(:sum)        { integer>>operator>>expression }
    rule(:expression) { sum | integer }
    root :expression
end
```

Erros

O Parslet auxilia também proporcionando o melhor relatório de erros possível. A expressão "a + 2" no *parser* anterior gera o seguinte erro:

```
Expected one of [SUM, INTEGER] at line 1 char 1.
|- Failed to match sequence (INTEGER OPERATOR EXPRESSION) a
|  `- Failed to match sequence ([0-9]{1, } SPACE?) at line
|     `- Expected at least 1 of [0-9] at line 1 char 1.
|        `- Failed to match [0-9] at line 1 char 1.
`- Failed to match sequence ([0-9]{1, } SPACE?) at line 1 c
   `- Expected at least 1 of [0-9] at line 1 char 1.
      `- Failed to match [0-9] at line 1 char 1.
```

Transform

O *parser* tem como saída uma estrutura difícil de ser trabalhada, as *deep nested hashes*. Por isso, o Parslet possui uma classe que transforma essa saída.

Um transformador que coloca "b" no lugar de "a" pode ser escrito da seguinte maneira:

```
class MyTransform < Parslet::Transform
   rule('a') { 'b' }
 end
```

A regra de transformaçao possui duas partes, um padrão, o qual deverá ser reconhecido e substituido (a) e um bloco que possui o que deverá ser colocado no lugar do padrão (b).

Prós e contras

Desvantagem: **desempenho**

Ruby é considerada uma linguagem lenta em comparação com outras linguagens de alto nível, e mais ainda em comparação com C, a linguagem mais popularmente usada na construção de compiladores. Isso se dá por algumas razões:

- É uma linguagem interpretada, e essas tendem a ser mais lentas que linguagens compiladas.
- O *garbage collector* de tempos em tempos para a execução do programa para limpar a memória alocada não utilizada.
- Chamadas de métodos são lentas pois o código não possui um ponteiro para os métodos e sim o nome para buscá-lo na classe a qual ele pertence, ou nos ancestrais dessa classe.
- Não usa *multithreading*; o programa inteiro é executado em uma única *thread*.

Prós e contras

Vantagens:

- **Legibilidade:** Por ser uma linguagem de alto nível, a sintaxe de Ruby é simples e inteligível, e atividades complexas podem ser executadas em poucos comandos, característica a qual muitos se referem como a "mágica"do Ruby.
- **Velocidade de escrita:** Ruby é particularmente útil em situações onde a maior produtividade do programador é mais importante que a velocidade da execução. Muitos desenvolvedores *web* consideram justo sacrificar o desempenho pela rapidez na qual escrevem código em Ruby.