How To Create Your Own Freaking Awesome Programming Language

Tradução feita por João Oliveira

TABLE OF CONTENT

3. Prototype-based
4. Functional
5. Our Awesome Runtime
6. Do It Yourself III

Published November 2011.

INTRODUCTION

> When you don't create things, you become defined by your tastes rather than ability. Your tastes only narrow & exclude people. So create.
>
> *- Why the Lucky Stiff*

Creating a programming language is the perfect mix of art and science. You're creating a way to express yourself, but at the same time applying computer science principles to implement it. Since we aren't the first ones to create a programming language, some well established tools are around to ease most of the exercise. Nevertheless, it can still be hard to create a fully functional language because it's impossible to predict all the ways in which someone will use it. That's why making your own language is such a great experience. You never know what someone else might create with it!

I've written this book to help other developers discover the joy of creating a programming language. Coding my first language was one of the most amazing experiences in my programming career. I hope you'll enjoy reading this book, but mostly, I hope you'll write your own programming language.

If you find an error or have a comment or suggestion while reading the following pages, please send me an email at macournoyer@gmail.com.

SUMMARY

This book is divided into ten sections that will walk you through each step of language-building. Each section will introduce a new concept and then apply its principles to a language that we'll build together throughout the book. All technical chapters end with a *Do It Yourself* section that suggest some language-extending exercises. You'll find solutions to those at the end of this book.

Our language will be dynamic and very similar to Ruby and Python. All of the code will be in Ruby, but I've put lots of attention to keep the code as simple as possible so that you can understand what's happening even if you don't know Ruby.

The focus of this book is not on how to build a production-ready language. Instead, it should serve as an introduction in building your first toy language.

ABOUT THE AUTHOR

I'm Marc-André Cournoyer, a coder from Montréal, Québec passionate about programming languages and tequila, but usually not at the same time.

I coded tinyrb, the smallest Ruby Virtual Machine, Min, a toy language running on the JVM, Thin, the high performance Ruby web server, and a bunch of other stuff. You can find most of my projects on GitHub.

You can find me online, blogging or tweeting and offline, snowboarding and learning guitar.

BEFORE WE BEGIN

You should have received a `code.zip` file with this book including all the code samples. To run the examples you must have the following installed:

- Ruby 1.8.7 or 1.9.2
- Racc 1.4.6, install with: `gem install racc -v=1.4.6` (optional, to recompile the parser in the exercices).

Other versions might work, but the code was tested with those.

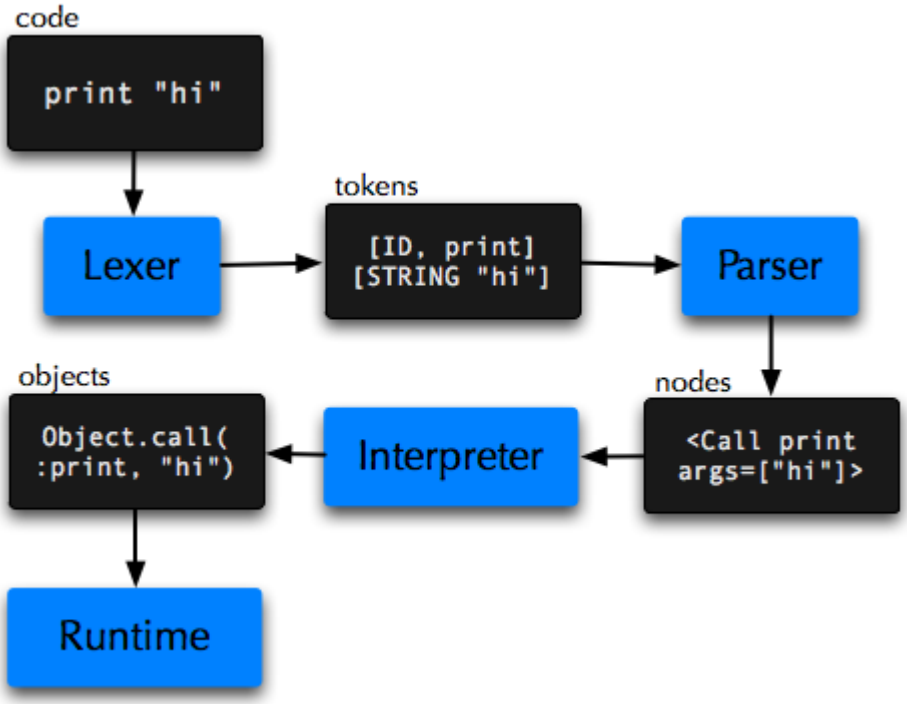
OVERVIEW

Although the way we design a language has evolved since its debuts, most of the core principles haven't changed. Contrary to Web Applications, where we've seen a growing number of frameworks, languages are still built from a lexer, a parser and a compiler. Some of the best books in this sphere have been written a long time ago: Principles of Compiler Design was published in 1977 and Smalltalk-80: The Language and its Implementation in 1983.

THE FOUR PARTS OF A LANGUAGE

Most dynamic languages are designed in four parts that work in sequence: the lexer, the parser, the interpreter and the runtime. Each one transforms the input of its predecessor until the code is run. Figure 1 shows an overview of this process. A chapter of this book is dedicated to each part.

Figure 1



CONHEÇA INCRÍVEL: NOSSA LINGUAGEM DE BRINQUEDO A linguagem que vamos codificar neste livro se chama Incrível, porque é! É uma mistura de sintaxe ruby e recuo de Python:

```
class Awesome:
  def name:
    "I'm Awesome"

  def awesomeness:
    100

awesome = Awesome.new
print(awesome.name)
print(awesome.awesomeness)
```

Algumas regras para a nossa língua:

Como em Python, blocos de código são delimitados por seu recuo.

As aulas são declaradas com a palavra-chave da classe.

▪ Métodos podem ser definidos em qualquer lugar usando a palavra-chave def.

Identificadores que começam com uma letra maiúscula são constantes que são globalmente acessíveis. Identificadores de casos inferiores são variáveis locais ou nomes de métodos.

Se um método não leva argumentos, parênteses podem ser ignorados, assim como em Ruby.

O último valor avaliado em um método é o seu valor de retorno. Tudo é um objeto.

Algumas partes estarão incompletas, mas o objetivo com uma linguagem de brinquedo é educar, experimentar e estabelecer a base sobre a qual construir mais.

LEXER

O lexer, ou scanner ou tokenizer é a parte de um idioma que converte a entrada, o código que você deseja executar, em tokens que o analisador pode entender.

Digamos que você tenha o seguinte código:

```
print "I ate",
      3,
      pies
```

Uma vez que este código passa pelo lexer, ele vai se parecer com algo assim:

```
[IDENTIFIER print] [STRING "I ate"] [COMMA]
                  [NUMBER 3] [COMMA]
                  [IDENTIFIER pies]
```

O que o lexer faz é dividir o código e marcar cada parte com o tipo de token que ele contém. Isso torna mais fácil para o analisador operar, já que ele não precisa se preocupar com detalhes como analisar um número de ponto flutuante ou analisar uma sequência complexa com sequências de fuga (\n, \t, etc.).

Lexers podem ser implementados usando expressões regulares, mas existem ferramentas mais apropriadas.

LEX (FLEX)

Flex é uma versão moderna da Lex (que foi codificada por Eric Schmidt, CEO do Google, por sinal) para gerar lexers C. Junto com Yacc, Lex é o lexer mais usado para analisar.

Foi portado para várias línguas-alvo.

- Rex for Ruby
- JFlex for Java
-

Lex e amigos não são lexers props. São compiladores lexers. Você fornece uma gramática e ela vai produzir um lexer. Eis como é essa gramática:

```
BLANK        [ \t\n]+

%%

// Whitespace
{BLANK}       /* ignore */

// Literals
[0-9]+        yylval = atoi(yytext); return T_NUMBER;

// Keywords
"end"         yylval = yytext; return T_END;
// ...
```

No lado esquerdo, uma expressão regular define como o token é combinado. Do lado direito, a ação a tomar. O valor do token é armazenado em yylval e o tipo de token é devolvido.

Mais detalhes no Flex manual.

Um equivalente Ruby, usando a joia rexical (uma porta de Lex para Ruby), seria:

```
macro
  BLANK         [\ \t]+

rule
  # Whitespace
  {BLANK}        # ignore

  # Literals
  [0-9]+         { [:NUMBER, text.to_i] }
```

```
10
11    # Keywords
12    end             { [:END, text] }
```

Rexical segue uma gramática semelhante à de Lex. Expressão regular à esquerda e ação à direita. No entanto, uma matriz de dois itens é usada para retornar o tipo e o valor do token combinado.

Detalhes do modo sobre o Detalhes do modo sobre o [rexical project page](#).

RAGEL

Uma ferramenta poderosa para criar um scanner é o Ragel. É descrito como um Compilador de Máquinas do Estado: lexers, como expressões regulares, são máquinas estatais. Sendo muito flexíveis, eles podem lidar com gramáticas de complexidades variadas e analisador de saída em várias línguas.

Aqui está como uma gramática Ragel se parece:

```
1  %%{
2  machine lexer;
3
4  # Machine
5  number      = [0-9]+;
6  whitespace  = " ";
7  keyword     = "end" | "def" | "class" | "if" | "else" | "true" | "false" | "nil";
8
9  # Actions
10 main := |*
11   whitespace;  # ignore
12   number      => { tokens << [:NUMBER, data[ts..te].to_i] };
13   keyword     => { tokens << [data[ts...te].upcase.to_sym, data[ts...te]] };
14 *|;
15
16 class Lexer
17   def initialize
```

```
18      %% write data;
19    end
20
21    def run(data)
22      eof = data.size
23      line = 1
24      tokens = []
25      %% write init;
26      %% write exec;
27      tokens
28    end
29  end
30  }%%
```

Aqui está como uma gramática Ragel se parece Ragel manual (PDF).

Aqui estão alguns exemplos reais de gramáticas Ragel usadas como lexers de linguagem:

- Min's lexer (Java)
- Potion's lexer (C)

PYTHON STYLE INDENTATION FOR AWESOME

Se você pretende construir uma linguagem totalmente funcional, você deve usar uma das duas ferramentas anteriores. Como o Awesome é uma linguagem simplista e queremos ilustrar os conceitos básicos de um scanner, vamos construir o lexer do zero usando expressões regulares.

Para tornar as coisas mais interessantes, usaremos o recuo para delimitar blocos em nossa linguagem de brinquedo, como em Python. Toda a magia do recuo acontece dentro do lexer. Analisar blocos de código delimitados com { ... } não é diferente de analisar o recuo quando você sabe como fazê-lo.

Tokenizando o seguinte código Python:

```
if tasty == True:
  print "Delicious!"
```

vai render esses tokens:

```
[IDENTIFIER if] [IDENTIFIER tasty] [EQUAL] [IDENTIFIER True]
[INDENT] [IDENTIFIER print] [STRING "Delicious!"]
[DEDENT]
```

O bloco é embrulhado em tokens INDENT e DEDENT em vez de { e }.

O algoritmo de análise de recuo é simples. Você precisa acompanhar duas coisas: o nível de recuo atual e a pilha de níveis de recuo. Quando você encontra uma quebra de linha seguida de espaços, você atualiza o nível de recuo. Aqui está nosso lexer para a linguagem Incrível:

lexer.rb

```
class Lexer
  KEYWORDS = ["def", "class", "if", "true", "false", "nil"]

  def tokenize(code)
    # Limpe o código removendo quebras de linha extras
    code.chomp!

    # Posição atual do personagem que estamos analisando
    i = 0

    # Coleção de todos os tokens de análise no formato [: TOKEN TYPE, value]
    tokens = []

    # O nível de indentação atual é o número de espaços no último indentação.
    current_indent = 0
    # Acompanhamos os níveis de indentação em que estamos para que, quando não o fizemos, possamos
    # verifique se estamos no nível correto.
    indent_stack = []

    # É assim que se implementa um scanner muito simples.
    # Analise um caractere por vez até encontrar algo para analisar.
```

```
22      while i < code.size
23        chunk = code[i..-1]
24
25        # Tokens padrão correspondentes.
26        #
27        # Correspondência de if, print, nomes de métodos, etc..
28        if identifier = chunk[/\A([a-z]\w*)/, 1]
29          # Palavras-chave são identificadores especiais marcados com seus próprios nomes,
            'se' resultará
30          # em um token [: IF, "if"]
31          if KEYWORDS.include?(identifier)
32            tokens << [identifier.upcase.to_sym, identifier]
33          # Os identificadores que não são de palavra-chave incluem nomes de métodos e
            variáveis.
34          else
35            tokens << [:IDENTIFIER, identifier]
36          end
37          # pule o que acabamos de analisar
38          i += identifier.size
39
40        # Combinar nomes de classes e constantes começando com uma letra maiúscula.
41        elsif constant = chunk[/\A([A-Z]\w*)/, 1]
42          tokens << [:CONSTANT, constant]
43          i += constant.size
44
45        elsif number = chunk[/\A([0-9]+)/, 1]
46          tokens << [:NUMBER, number.to_i]
47          i += number.size
48
49        elsif string = chunk[/\A"(.*?)"/, 1]
50          tokens << [:STRING, string]
51          i += string.size + 2
52
53        # Aqui está a magia do recuo!
54        #
55        # Temos que cuidar de 3 casos:
56        #
57        #   if true:  # 1) o bloco é criado
58        #     line 1
59        #     line 2  # 2) nova linha dentro de um bloco
60        #   continue  # 3) dedent
61        #
62        # Esse elseif cuida do primeiro caso. O número de espaços irá determinar
63        # o nível de indentação.
```

```
64        elsif indent = chunk[/\A\:\n( +)/m, 1] # Matches ": <newline> <spaces>"
65          # Quando criamos um novo bloco, esperamos que o nível de indentação aumente.
66          if indent.size <= current_indent
67            raise "Bad indent level, got #{indent.size} indents, " +
68                  "expected > #{current_indent}"
69          end
70          # Ajuste o nível de indentação atual.
71          current_indent = indent.size
72          indent_stack.push(current_indent)
73          tokens << [:INDENT, indent.size]
74          i += indent.size + 2
75
76        # Este elseif cuida dos dois últimos casos:
77        # Case 2: Ficamos no mesmo bloco se o nível de indentação (número de espaços) é o
78        #         igual a current_indent.
79        # Case 3: Feche o bloco atual, se o nível de indentação for inferior a
    current_indent.
80        elsif indent = chunk[/\A\n( *)/m, 1] # Matches "<newline> <spaces>"
81          if indent.size == current_indent # Case 2
82            # Nada a fazer, ainda estamos no mesmo quarteirão
83            tokens << [:NEWLINE, "\n"]
84          elsif indent.size < current_indent # Case 3
85            while indent.size < current_indent
86              indent_stack.pop
87              current_indent = indent_stack.first || 0
88              tokens << [:DEDENT, indent.size]
89            end
90            tokens << [:NEWLINE, "\n"]
91          else # indent.size > current_indent, error!
92            # Não é possível aumentar o nível de indentação sem usar ":", então isso é um
    erro.
93            raise "Missing ':'"
94          end
95          i += indent.size + 1
96
97        # Corresponde a operadores longos, como ||, &&, ==,! =, <= E> =.
98        # Operadores de um caractere são correspondidos pelo catch all `else` na parte
inferior.
99        elsif operator = chunk[/\A(\|\||&&|==|!=|<=|>=)/, 1]
100         tokens << [operator, operator]
101         i += operator.size
102
103       # Ignorar espaços em branco
104       elsif chunk.match(/\A /)
```

```
105             i += 1
```

```
106
107         # Pegue todos os personagens individuais
108         # Tratamos todos os outros caracteres únicos como um token. Por exemplo.: ( ) , . !
+ - <
109         else
110           value = chunk[0,1]
111           tokens << [value, value]
112           i += 1
113
114         end
115
116       end
117
118       # Close all open blocks
119       while indent = indent_stack.pop
120         tokens << [:DEDENT, indent_stack.first || 0]
121       end
122
123       tokens
124     end
125 end
```

Você mesmo pode testar o lexer executando o arquivo de teste incluído no livro. Execute ruby -Itest test / lexer_test.rb do diretório de código e deve produzir 0 falhas, 0 erros. Aqui está um trecho desse arquivo de teste.

test/lexer_test.rb

```
1  code = <<-CODE
2  if 1:
3    print "..."
4    if false:
5      pass
6    print "done!"
7  print "The End"
8  CODE
9  tokens = [
10   [:IF, "if"], [:NUMBER, 1],
11   [:INDENT, 2],
12     [:IDENTIFIER, "print"], [:STRING, "..."], [:NEWLINE, "\n"],
13     [:IF, "if"], [:FALSE, "false"],
14     [:INDENT, 4],
```

```
      [:IDENTIFIER, "pass"],
    [:DEDENT, 2], [:NEWLINE, "\n"],
    [:IDENTIFIER, "print"],
    [:STRING, "done!"],
 [:DEDENT, 0], [:NEWLINE, "\n"],
 [:IDENTIFIER, "print"], [:STRING, "The End"]
]
assert_equal tokens, Lexer.new.tokenize(code)
```

Alguns analisadores se encarregam da lexing e da análise gramatical. Veremos mais sobre eles na próxima seção.

DO IT YOURSELF I/ FAÇA VOCÊ MESMO I

a. Uma. Modifique o lexer para analisar: condição while: ... estruturas de controle.

b. Modifique o lexer para delimitar blocos com {...} em vez de indentação.

Solutions to Do It Yourself I.

PARSER

Por si só, a saída de tokens pelo lexer são apenas blocos de construção. O analisador contextualiza-os organizando-os em uma estrutura. O lexer produz uma série de tokens; o analisador produz uma árvore de nós.

Vamos pegar os tokens da seção anterior:

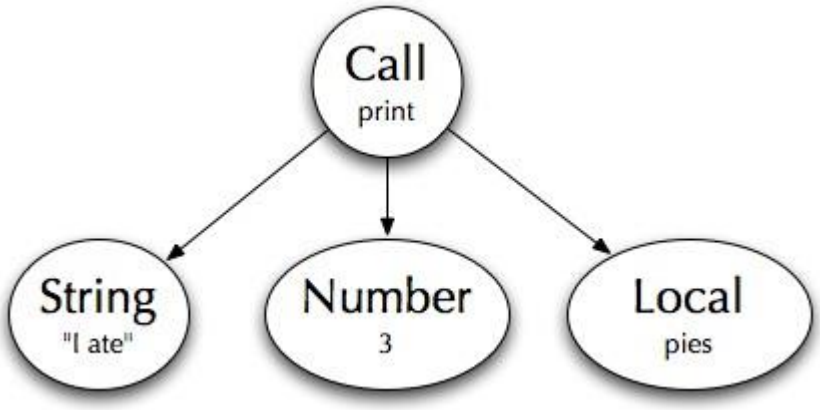
```
[IDENTIFIER print] [STRING "I ate"] [COMMA]
                  [NUMBER 3] [COMMA]
                  [IDENTIFIER pies]
```

A saída do analisador mais comum é uma árvore de sintaxe abstrata, ou AST. É uma árvore de nós que representa o que o código significa para a linguagem. Os tokens lexer anteriores produzirão o seguinte:

```
[<Call name=print,
       arguments=[<String value="I ate">,
                  <Number value=3>,
                  <Local name=pies>]
>]
```

Ou como uma árvore visual:

Figure 2



O analisador descobriu que print era uma chamada de método e os seguintes tokens são os argumentos.

Os geradores de analisador são comumente usados para realizar a tarefa tediosa de construir um analisador. Muito parecido com o idioma inglês, uma linguagem de programação precisa de uma gramática para definir suas regras. O gerador de analisador irá converter esta gramática em um analisador que irá compilar tokens lexer em nós AST.

BISON (YACC)

Bison é uma versão moderna do Yacc, o analisador mais amplamente usado. Yacc significa Yet Another Compiler Compiler, porque compila a gramática em um compilador de tokens. É usado em várias linguagens convencionais, como Ruby. Mais frequentemente usado com Lex, foi transferido para vários idiomas de destino.

- Racc for Ruby
- Ply for Python
- JavaCC for Java

Como Lex, do capítulo anterior, Yacc compila uma gramática em um analisador. Veja como uma regra gramatical do Yacc é definida:

```
Call: /* Nome da regra */
  Expression '.' IDENTIFIER                  { $$ = CallNode_new($1, $3, NULL); }
| Expression '.' IDENTIFIER '(' ArgList ')'  { $$ = CallNode_new($1, $3, $5); }
/*   $1      $2      $3     $4     $5   $6   <= os valores da regra são armazenados em
                                                 essas variáveis. */
;
```

À esquerda está definido como a regra pode ser combinada usando tokens e outras regras. No lado direito, entre colchetes está a ação a ser executada quando a regra corresponder.

Nesse bloco, podemos fazer referência aos tokens sendo correspondidos usando $ 1, $ 2, etc. Finalmente, armazenamos o resultado em `$$`.

LEMON

Lemon é bastante semelhante ao Yacc, com algumas diferenças. De seu site:

> - Usar uma sintaxe gramatical diferente que é menos propensa a erros de programação.
> - O analisador gerado pelo Lemon é tanto reentrante quanto thread-safe.
> - Lemon inclui o conceito de um destruidor não terminal, o que torna muito mais fácil escrever um analisador que não vaza memória.

Para obter mais informações, consulte o the manual ou verifique exemplos reais dentro Potion.

ANTLR

ANTLR ou verifique exemplos reais dentro several target languages.

PEGS

Parsing Expression Grammars, ou PEGs, são muito poderosos na análise de linguagens complexas. Eu usei um PEG gerado a partir de peg/leg em tinyrb para analisar a sintaxe infame de Ruby com resultados encorajadores (tinyrb's grammar).

Treetop é uma ferramenta Ruby interessante para a criação de PEG.

OPERATOR PRECEDENCE

Uma das armadilhas comuns da análise de linguagem é a precedência do operador.

Analisando x + y * z não deve produzir o mesmo resultado que (x + y) * z, o mesmo para todos os outros

operadores. Cada idioma tem uma tabela de precedência de operadores, geralmente baseada na ordem matemática das operações. Existem várias maneiras de lidar com isso. Analisadores baseados em Yacc implementam o Shunting Yard algorithm em que você dá um nível de precedência para cada tipo de operador. Os operadores são declarados em Bison e Yacc com `%left` e `%right` macros. Leia mais em Bison's manual.

Esta é a tabela de precedência do operador para a nossa linguagem, com base no C languageoperator precedence:

```
left  '.'
right '!'
left  '*' '/'
left  '+' '-'
left  '>' '>=' '<' '<='
left  '==' '!='
left  '&&'
left  '||'
right '='
left  ','
```

Quanto mais alta a precedência (o topo é mais alto), mais cedo o operador será analisado. Se a linha a + b * c estiver sendo analisada, a parte b * c será analisada primeiro, pois * tem precedência superior a +. Agora, se vários operadores com a mesma precedência estão competindo para serem analisados todos de uma vez, o conflito é resolvido usando associatividade, declarada com as palavras-chave esquerda e direita antes do token. Por exemplo, com a expressão a = b = c. Como = tem associatividade da direita para a esquerda, ele começará a analisar da direita, b = c. Resultando em a = (b = c).

Para outros tipos de analisadores (ANTLR e PEG), uma alternativa mais simples, mas menos eficiente, pode ser usada. Simplesmente declarar as regras gramaticais na ordem certa irá produzir o resultado desejado:

```
expression:      equality
equality:        additive ( ( '==' | '!=' ) additive )*
additive:        multiplicative ( ( '+' | '-' ) multiplicative )*
multiplicative:  primary ( ( '*' | '/' ) primary )*
primary:         '(' expression ')' | NUMBER | VARIABLE | '-' primary
```

O analisador tentará combinar as regras recursivamente, começando pela expressão e encontrando seu caminho para o primário. Uma vez que multiplicativa é a última regra chamada no processo de análise, ela terá maior precedência.

CONNECTING THE LEXER AND PARSER IN AWESOME

Para nosso analisador Awesome, usaremos Racc, a versão Ruby do Yacc. É muito mais difícil construir um analisador do zero do que criar um lexer. No entanto, a maioria das linguagens acaba escrevendo seu próprio analisador porque o resultado é mais rápido e fornece um melhor relatório de erros.

O arquivo de entrada que você fornece ao Racc contém a gramática do seu idioma e é muito semelhante à gramática Yacc.

grammar.y

```
class Parser

# Declarar tokens produzidos pelo lexer
token IF ELSE
token DEF
token CLASS
token NEWLINE
token NUMBER
token STRING
token TRUE FALSE NIL
token IDENTIFIER
token CONSTANT
token INDENT DEDENT

```

```
15 # Tabela de precedência
16 # Baseado em http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B#Operator_precedence
17 prechigh
18   left  '.'
19   right '!'
20   left  '*' '/'
21   left  '+' '-'
22   left  '>' '>=' '<' '<='
23   left  '==' '!='
24   left  '&&'
25   left  '||'
26   right '='
27   left  ','
28 preclow
29
30 rule
31   # Todas as regras são declaradas neste formato:
32   #
33   #   RuleName:
34   #     OtherRule TOKEN AnotherRule    { código a ser executado quando
                                            este corresponder }
35   #   | OtherRule                      { ... }
36   #   ;
37   #
38   # Na seção de código (dentro do {...} à direita):
39   # - Atribuir ao "resultado" o valor retornado pela regra.
40   # - Use val [índice de expressão] para fazer referência a expressões à esquerda.
41
42
43   # Toda análise terminará nesta regra, sendo o tronco do AST.
44   Root:
45     /* nothing */                     { result = Nodes.new([]) }
46   | Expressions                       { result = val[0] }
47   ;
48
49  # Qualquer lista de expressões, classe ou corpo de método, separados por quebras de
    linha.
50   Expressions:
51     Expression                        { result = Nodes.new(val) }
52   | Expressions Terminator Expression  { result = val[0] << val[2] }
53     # Para
       ignorar
       quebras de
       linha final
54   | Expressions Terminator            { result = val[0] }
```

```
55    | Terminator                          { result = Nodes.new([]) }
56    ;
```

```
57
58    # Todos os tipos de expressões em nossa língua
59    Expression:
60      Literal
61    | Call
62    | Operator
63    | Constant
64    | Assign
65    | Def
66    | Class
67    | If
68    | '(' Expression ')'    { result = val[1] }
69    ;
70
71    # Todos os tokens que podem encerrar uma expressão
72    Terminator:
73      NEWLINE
74    | ";"
75    ;
76
77    # # Todos os
      valores
      embutidos em
      código
78    Literal:
79      NUMBER                      { result = NumberNode.new(val[0]) }
80    | STRING                      { result = StringNode.new(val[0]) }
81    | TRUE                        { result = TrueNode.new }
82    | FALSE                       { result = FalseNode.new }
83    | NIL                         { result = NilNode.new }
84    ;
85
86    # Um métodocall
87    Call:
88      # Método
89      IDENTIFIER                  { result = CallNode.new(nil, val[0], []) }
90      # Método(Argumentos)
91    | IDENTIFIER "(" ArgList ")"  { result = CallNode.new(nil, val[0], val[2]) }
92      # receiver.method
93    | Expression "." IDENTIFIER   { result = CallNode.new(val[0], val[2], []) }
94      # receiver.method(arguments)
95    | Expression "."
96        IDENTIFIER "(" ArgList ")"  { result = CallNode.new(val[0], val[2], val[4]) }
97    ;
98
```

```
99    ArgList:
100     /* nothing */                 { result = [] }
101   | Expression                    { result = val }
102   | ArgList "," Expression        { result = val[0] << val[2] }
103   ;
104
105   Operator:
106   # Operadores binários
107     Expression '||' Expression    { result = CallNode.new(val[0], val[1], [val[2]]) }
108   | Expression '&&' Expression    { result = CallNode.new(val[0], val[1], [val[2]]) }
109   | Expression '==' Expression    { result = CallNode.new(val[0], val[1], [val[2]]) }
110   | Expression '!=' Expression    { result = CallNode.new(val[0], val[1], [val[2]]) }
111   | Expression '>' Expression     { result = CallNode.new(val[0], val[1], [val[2]]) }
112   | Expression '>=' Expression    { result = CallNode.new(val[0], val[1], [val[2]]) }
113   | Expression '<' Expression     { result = CallNode.new(val[0], val[1], [val[2]]) }
114   | Expression '<=' Expression    { result = CallNode.new(val[0], val[1], [val[2]]) }
115   | Expression '+' Expression     { result = CallNode.new(val[0], val[1], [val[2]]) }
116   | Expression '-' Expression     { result = CallNode.new(val[0], val[1], [val[2]]) }
117   | Expression '*' Expression     { result = CallNode.new(val[0], val[1], [val[2]]) }
118   | Expression '/' Expression     { result = CallNode.new(val[0], val[1], [val[2]]) }
119   ;
120
121   Constant:
122     CONSTANT                      { result = GetConstantNode.new(val[0]) }
123   ;
124
125 # # Atribuição
    a uma variável
      ou constante
126   Assign:
127     IDENTIFIER "=" Expression     { result = SetLocalNode.new(val[0], val[2]) }
128   | CONSTANT "=" Expression       { result = SetConstantNode.new(val[0], val[2]) }
129   ;
130
131   # Definição de método
132   Def:
133     DEF IDENTIFIER Block          { result = DefNode.new(val[1], [], val[2]) }
134   | DEF IDENTIFIER
135       "(" ParamList ")" Block     { result = DefNode.new(val[1], val[3], val[5]) }
136   ;
137
138   ParamList:
139     /* nothing */                 { result = [] }
140   | IDENTIFIER                    { result = val }
```

```
141     | ParamList "," IDENTIFIER      { result = val[0] << val[2] }
142     ;
143
144     # Definição de classe
145     Class:
146       CLASS CONSTANT Block          { result = ClassNode.new(val[1], val[2]) }
147     ;
148
149     # Definição de classe
150     If:
151       IF Expression Block           { result = IfNode.new(val[1], val[2]) }
152     ;
153
154     # Um bloco de código recuado. Você vê aqui que todo o trabalho árduo foi feito pelo
155     # lexer.
156     Block:
157       INDENT Expressions DEDENT     { result = val[1] }
158     # Se você não gosta de recuo, você pode substituir a regra anterior pelo
159     # seguindo um para separar blocos com chaves. Você também precisará remover o
160     # seção mágica de recuo no lexer.
161     # "{" Expressions "}"           { replace = val[1] }
162     ;
163 end
164
165 ---------header
166     require "lexer"
167     require "nodes"
168
169 ---------inner
170     # Este código será colocado no estado em que se encontra na classe Parser.
171     def parse(code, show_tokens=false)
172       @tokens = Lexer.new.tokenize(code) # Tokenize o código usando nosso lexer
173       puts @tokens.inspect if show_tokens
174       do_parse # Kickoff o processo de análise
175     end
176
177     def next_token
178       @tokens.shift
179     end
```

Em seguida, geramos o analisador com: racc -o parser.rb grammar.y. Isso criará uma classe Parser que podemos usar para analisar nosso código. Execute ruby -Itest test / parser_test.rb do diretório de código para testar o analisador. Aqui está um trecho deste arquivo.

test/parser_test.rb

```
code = <<-CODE
def method(a, b):
  true
CODE

nodes = Nodes.new([
  DefNode.new("method", ["a", "b"],
    Nodes.new([TrueNode.new])
  )
])

assert_equal nodes, Parser.new.parse(code)
```

A análise do código retornará uma árvore de nós. O nó raiz será sempre do tipo Nós que contém nós filhos.

DO IT YOURSELF II / Tente você mesmo II

a. Adicione uma regra na gramática para analisar enquanto bloqueia.

b. Adicione uma regra gramatical para lidar com o! operadores unários, por exemplo:! x. Fazendo o seguinte teste passar (test_unary_operator):

Solutions to Do It Yourself II.

RUNTIME MODEL
MODELO DE TEMPO DE EXECUÇÃO

O modelo de tempo de execução de uma linguagem é como representamos seus objetos, seus métodos, seus tipos, sua estrutura na memória. Se o analisador determina como você se comunica com a linguagem, o tempo de execução define como a linguagem se comporta. Duas linguagens podem compartilhar o mesmo analisador, mas ter tempos de execução diferentes e ser muito diferentes.

Ao projetar seu tempo de execução, existem três fatores que você deve considerar:

- Velocidade: a maior parte da velocidade será devido à eficiência do tempo de execução.
- ▪ Flexibilidade: quanto mais você permite que o usuário modifique o idioma, mais poderoso ele é.
- ▪ Pegada de memória: claro, tudo isso usando o mínimo de memória possível.

Como você deve ter notado, essas três restrições são mutuamente conflitantes. Projetar uma linguagem é sempre um jogo de dar e receber.

Com essas considerações em mente, existem várias maneiras de modelar seu tempo de execução.

PROCEDURAL

Um dos modelos de tempo de execução mais simples, como C e PHP (antes da versão 4). Tudo está centrado em métodos (procedimentos). Não há objetos e todos os métodos geralmente compartilham o mesmo namespace. Fica bagunçado muito rapidamente!

CLASS-BASED

O modelo baseado em classes é o mais popular no momento. Pense em Java, Python, Ruby, etc. Pode ser o modelo mais fácil de entender para os usuários de sua linguagem.

PROTOTYPE-BASED

Exceto para Javascript, nenhuma linguagem baseada em protótipo alcançou grande popularidade ainda. Este modelo é o mais fácil de implementar e também o mais flexível porque tudo é um clone de um objeto.

Ian Piumarta descreve como projetar um modelo de objeto aberto e extensível que permite aos usuários da linguagem modificar seu comportamento em tempo de execução.

Veja o apêndice no final deste livro para um exemplo de linguagem baseada em protótipo: Appendix: Mio, a minimalist homoiconic language.

FUNCTIONAL

O modelo funcional, usado pelo Lisp e outras linguagens, trata a computação como a avaliação de funções matemáticas e evita estado e dados mutáveis. Este modelo tem suas raízes no cálculo Lambda.

OUR AWESOME RUNTIME / NOSSO TEMPO DE EXECUÇÃO INCRÍVEL

Como a maioria de nós está familiarizada com tempos de execução baseados em classes, decidi usá-los em nossa linguagem Awesome. O código a seguir define como objetos, métodos e classes são armazenados e como eles interagem juntos.

A classe AwesomeObject é o objeto central de nosso tempo de execução. Uma vez que tudo é um objeto em nossa linguagem, tudo o que colocaremos no tempo de execução precisa ser um

objeto, portanto, uma instância desta classe. AwesomeObjects tem uma classe e pode conter um valor ruby. Isso nos permitirá armazenar dados como uma string ou um número em um objeto para manter o controle de sua representação Ruby.

runtime/object.rb

```
# Representa uma instância de objeto Awesome no mundo Ruby.
class AwesomeObject
  attr_accessor :runtime_class, :ruby_value

  # Cada objeto tem uma classe (chamada runtime_class para evitar erros com a classe de Ruby
  # método). Opcionalmente, um objeto pode conter um valor Ruby (por exemplo: números e strings).
  def initialize(runtime_class, ruby_value=self)
    @runtime_class = runtime_class
    @ruby_value = ruby_value
  end

  # Chame um método no objeto.
  def call(method, arguments=[])
    # Como um modelo de tempo de execução baseado em classe típico, armazenamos métodos na classe do
    # object.
    @runtime_class.lookup(method).call(self, arguments)
  end
end
```

Lembre-se de que no Awesome tudo é um objeto. Mesmo as classes são instâncias da classe Class. AwesomeClasses contém os métodos e podem ser instanciados por meio de seu novo método.

runtime/class.rb

```
# Representa uma classe incrível no mundo Ruby. Classes são objetos em Awesome, então
# herdar de AwesomeObject.
class AwesomeClass < AwesomeObject
  attr_reader :runtime_methods

  # Cria uma nova classe. Number é uma instância de Class, por exemplo.
  def initialize
    @runtime_methods = {}

    # Verifique se estamos inicializando (iniciando o tempo de execução). Durante este
```

processo, o

```
    # o tempo de execução não foi totalmente inicializado e as classes principais ainda não existem, então adiamos
    # usando-os assim que a linguagem é inicializada.
    # Isso resolve o problema da galinha ou do ovo com a classe Class. Nós podemos
    # inicialize a classe e, em seguida, defina Class.class = Class.
    if defined?(Runtime)
      runtime_class = Runtime["Class"]
    else
      runtime_class = nil
    end

    super(runtime_class)
  end

  # Procure um método
  def lookup(method_name)
    method = @runtime_methods[method_name]
    unless method
      raise "Method not found: #{method_name}"
    end
    method
  end

  # Crie uma nova instância desta classe
  def new
    AwesomeObject.new(self)
  end

  # Crie uma instância dessa classe Awesome que contém um valor Ruby. Como uma corda,
  # Número ou verdadeiro.
  def new_with_value(value)
    AwesomeObject.new(self, value)
  end
end
```

E aqui está o objeto de método que armazenará métodos definidos em nosso tempo de execução.

runtime/method.rb

```
# Representa um método definido no tempo de execução.
class AwesomeMethod
  def initialize(params, body)
```

```
    @params = params
    @body = body
  end

  def call(receiver, arguments)
    # Crie um contexto de avaliação no qual o método será executado.
    context = Context.new(receiver)

    # Atribuir argumentos a variáveis locais
    @params.each_with_index do |param, index|
      context.locals[param] = arguments[index]
    end

    @body.eval(context)
  end
end
```

Observe que usamos o método de chamada para avaliar um método. Isso nos permitirá definir métodos de tempo de execução do Ruby usando Procs. Aqui está o porquê:

```
p = proc do |arg1, arg2|
  # ...
end
p.call(1, 2) # executa o bloco de código passado para proc (tudo entre do ... end)
```

Procs podem ser executados por meio de seu método de chamada. Veremos como isso é usado para definir métodos de tempo de execução do Ruby na seção de inicialização do tempo de execução.

Antes de inicializarmos nosso tempo de execução, há um objeto ausente que precisamos definir e esse é o contexto de avaliação. O objeto Context encapsula o ambiente de avaliação de um bloco específico de código. Ele acompanhará o seguinte:

- Variáveis locais.
- O valor atual de self, o objeto no qual métodos sem receptores are

chamado, por exemplo: print é como self.print.

▪ A classe atual, a classe na qual os métodos são definidos com o def palavra-chave.

É aqui também que nossas constantes (ou seja, classes) serão armazenadas.

runtime/context.rb

```ruby
# O contexto da avaliação.
class Context
  attr_reader :locals, :current_self, :current_class

  # Nós armazenamos constantes como variáveis de classe (variáveis de classe começam com @@ e instância
  # variáveis começam com @ em Ruby), uma vez que são globalmente acessíveis. se você quiser
  # implementar namespacing de constantes, você pode armazená-lo na instância desta
  # class.
  @@constants = {}

  def initialize(current_self, current_class=current_self.runtime_class)
    @locals = {}
    @current_self = current_self
    @current_class = current_class
  end

  # Atalhos para acessar constantes, Runtime [...] em vez de Runtime.constants [...]
  def [](name)
    @@constants[name]
  end
  def []=(name, value)
    @@constants[name] = value
  end
end
```

Finalmente, inicializamos o tempo de execução. A princípio, nenhum objeto existe no tempo de execução. Antes de podermos executar nossa primeira expressão, precisamos preencher esse tempo de execução com alguns objetos: Class, Object, true, false, nil e alguns métodos principais.

runtime/bootstrap.rb

```ruby
# Inicialize o tempo de execução. É aqui que reunimos todas as classes e objetos
# para formar o tempo de execução.
                                        # What's happening in the runtime:
```

```
awesome_class = AwesomeClass.new             #  Class
awesome_class.runtime_class = awesome_class #  Class.class = Class
object_class = AwesomeClass.new              #  Object = Class.new
object_class.runtime_class = awesome_class  #  Object.class = Class

# Crie o objeto Runtime (o contexto raiz) no qual todo o código iniciará sua avaliação.
Runtime = Context.new(object_class.new)

Runtime["Class"] = awesome_class
Runtime["Object"] = object_class
Runtime["Number"] = AwesomeClass.new
Runtime["String"] = AwesomeClass.new

# Tudo é um objeto em nossa linguagem, mesmo verdadeiro, falso e nulo. Então eles precisam
# ter uma aula também.
Runtime["TrueClass"] = AwesomeClass.new
Runtime["FalseClass"] = AwesomeClass.new
Runtime["NilClass"] = AwesomeClass.new

Runtime["true"] = Runtime["TrueClass"].new_with_value(true)
Runtime["false"] = Runtime["FalseClass"].new_with_value(false)
Runtime["nil"] = Runtime["NilClass"].new_with_value(nil)

# Adicione alguns métodos básicos ao tempo de execução.

# Adicione o método `new` às classes, usado para instanciar uma classe:
# Ex .: Object.new, Number.new, String.new, etc.
Runtime["Class"].runtime_methods["new"] = proc do |receiver, arguments|
  receiver.new
end

# Imprima um objeto no console.
# por exemplo: imprimir ("olá!")
Runtime["Object"].runtime_methods["print"] = proc do |receiver, arguments|
  puts arguments.first.ruby_value
  Runtime["nil"]
end
```

Agora que reunimos todas as peças, podemos chamar métodos e criar objetos dentro de nosso tempo de execução.

test/runtime_test.rb

```ruby
# Mimic Object.new na linguagem
object = Runtime["Object"].call("new")

assert_equal Runtime["Object"], object.runtime_class # afirmar que o objeto é um objeto
```

Você pode sentir a linguagem ganhando vida? Aprenderemos como mapear esse tempo de execução para os nós que criamos de nosso analisador na próxima seção.

DO IT YOURSELF III

a. Implemente a herança adicionando uma superclasse a cada classe Awesome.

b. Adicione o método para lidar com x + 2.

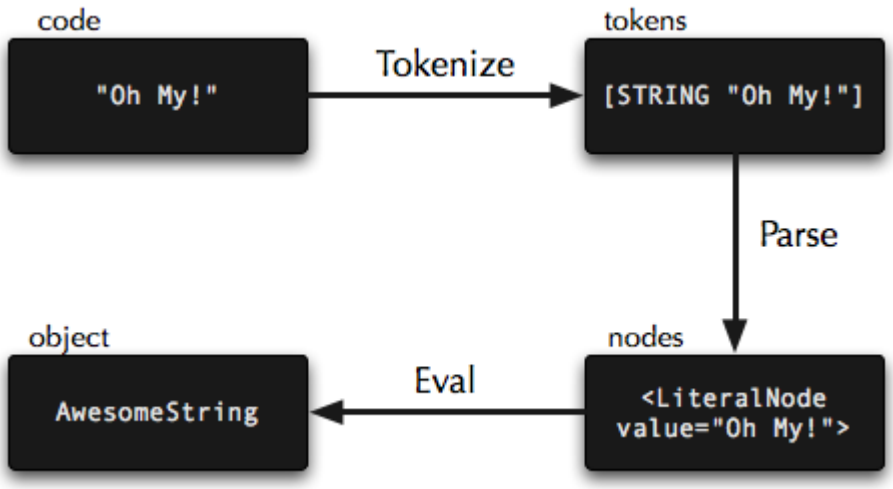
Solutions to Do It Yourself III.

INTERPRETER

O interpretador é o módulo que avalia o código. Ele lê o AST produzido pelo analisador e executa cada ação associada aos nós, modificando o tempo de execução.

A Figura 3 recapitula o caminho de uma string em nossa língua.

Figura 3



O lexer cria o token, o analisador pega esses tokens e os converte em nós. Finalmente, o interpretador avalia os nós.

Uma abordagem comum para executar um AST é implementar uma classe Visitor que visita todos os nós um por um, executando o código apropriado. Isso torna as coisas ainda mais modulares e facilita os esforços de otimização no AST. Mas para o propósito deste livro, vamos manter as coisas simples e deixar cada nó lidar com sua avaliação.

Lembre-se dos nós que criamos no analisador: StringNode para uma string, ClassNode para uma definição de classe? Aqui, estamos reabrindo essas classes e adicionando um novo método a cada uma: eval. Este método será responsável por interpretar esse nó particular.

interpreter.rb

```ruby
require "parser"
require "runtime"

class Interpreter
  def initialize
    @parser = Parser.new
  end

  def eval(code)
    @parser.parse(code).eval(Runtime)
  end
end

class Nodes
  # Este método é a parte "intérprete" da nossa linguagem. Todos os nós sabem como avaliar
  # em si e retorna o resultado de sua avaliação implementando o método "eval".
  # A variável "contexto" é o ambiente em que o nó é avaliado (local
# variáveis, classe atual, etc.).
  def eval(context)
    return_value = nil
    nodes.each do |node|
      return_value = node.eval(context)
    end
    # O último valor avaliado em um método é o valor de retorno. Ou nulo se nenhum.
    return_value || Runtime["nil"]
  end
end

class NumberNode
  def eval(context)
    # Aqui, acessamos o Runtime, que veremos na próxima seção, para criar um novo
# instância da classe Number.
    Runtime["Number"].new_with_value(value)
  end
end

class StringNode
  def eval(context)
    Runtime["String"].new_with_value(value)
  end
end

```

```ruby
class TrueNode
  def eval(context)
    Runtime["true"]
  end
end

class FalseNode
  def eval(context)
    Runtime["false"]
  end
end

class NilNode
  def eval(context)
    Runtime["nil"]
  end
end

class CallNode
  def eval(context)
    # Se não houver receptor e o nome do método for o nome de uma variável local, então
# é um acesso variável local. Este truque nos permite pular o () ao chamar um
# método.
    if receiver.nil? && context.locals[method] && arguments.empty?
      context.locals[method]

    # Chamada de método
    else
      if receiver
        value = receiver.eval(context)
      else
        # No caso de não haver receptor, o padrão é self, chamar "print" é como
        # "self.print".
        value = context.current_self
      end

      eval_arguments = arguments.map { |arg| arg.eval(context) }
      value.call(method, eval_arguments)
    end
  end
end

```

```ruby
class GetConstantNode
  def eval(context)
    context[name]
  end
end

class SetConstantNode
  def eval(context)
    context[name] = value.eval(context)
  end
end

class SetLocalNode
  def eval(context)
    context.locals[name] = value.eval(context)
  end
end

class DefNode
  def eval(context)
    # Definir um método é adicionar um método à classe atual.
    method = AwesomeMethod.new(params, body)
    context.current_class.runtime_methods[name] = method
  end
end

class ClassNode
  def eval(context)
    # Tente localizar a classe. Permite a reabertura de classes para adicionar métodos.
    awesome_class = context[name]

    unless awesome_class # Classe ainda não existe
    awesome_class = AwesomeClass.new
      # Registre a classe como uma constante no tempo de execução.
      context[name] = awesome_class
    end

    # Avalie o corpo da classe em seu contexto. Fornecer um contexto personalizado permite
    # para controlar onde os métodos são adicionados quando definidos com a palavra-chave def. Nisso
    # caso, nós os adicionamos à classe recém-criada.
    class_context = Context.new(awesome_class, awesome_class)

```

```
127       body.eval(class_context)
128
129       awesome_class
130     end
131   end
132
133   class IfNode
134     def eval(context)
135       # Transformamos o nó de condição em um valor Ruby para usar o controle "if" do Ruby
136       # estrutura.
137       if condition.eval(context).ruby_value
138         body.eval(context)
139       end
140     end
141   end
```

A parte do interpretador (o método eval) é o conector entre o analisador e o tempo de execução de nossa linguagem. Assim que chamarmos eval no nó raiz, todos os nós filhos serão avaliados recursivamente. É por isso que chamamos a saída do analisador de AST, para Abstract Syntax Tree. É uma árvore de nós. E avaliar o nó de nível superior dessa árvore terá o efeito em cascata de avaliar cada um de seus filhos.

Vamos executar nosso primeiro programa completo!

test/interpreter_test.rb

```
1  code = <<-CODE
2  class Awesome:
3    def does_it_work:
4      "yeah!"
5
6  awesome_object = Awesome.new
7  if awesome_object:
8    print(awesome_object.does_it_work)
9  CODE
10
11 assert_prints("yeah!\n") { Interpreter.new.eval(code) }
```

Para completar nossa linguagem, podemos criar um script para executar um arquivo ou um REPL (para leitura-eval-impressão-loop), ou interpretador interativo.

awesome

```
#!/usr/bin/env ruby
# A linguagem incrível!
#
# usage:
#   ./awesome example.awm # avaliar um arquivo
#   ./awesome             # para iniciar o REPL
#
# no Windows, execute com: ruby awesome [opções]

$:.unshift "." # Correção para Ruby 1.9
require "interpreter"
require "readline"

interpreter = Interpreter.new

# Se um arquivo for fornecido, nós o avaliamos.
if file = ARGV.first
  interpreter.eval File.read(file)

# Inicie o REPL, read-eval-print-loop ou interpretador interativo
else
  puts "Awesome REPL, CTRL+C to quit"
  loop do
    line = Readline::readline(">> ")
    Readline::HISTORY.push(line)
    value = interpreter.eval(line)
    puts "=> #{value.ruby_value.inspect}"
  end

end
```

Execute o interpretador interativo executando ./awesome e digite uma linha de código Awesome, por exemplo: print ("Funciona!"). Aqui está um exemplo de sessão incrível:

```
Awesome REPL, CTRL+C to quit
```

```
>> m = "This is Awesome!"
```

```
=> "This is Awesome!"
>> print(m)
This is Awesome!
=> nil
```

Além disso, tente executar um arquivo: ./awesome example.awm.

DO IT YOURSELF IV / Tente voce mesmo!

a. Implemente o método de avaliação WhileNode.

[Solutions to Do It Yourself IV](#).

COMPILAÇÃO

Linguagens dinâmicas como Ruby e Python têm muitas vantagens, como velocidade de desenvolvimento e flexibilidade. Mas às vezes, compilar uma linguagem para um formato mais eficiente pode ser mais apropriado. Ao escrever código C, você o compila em código de máquina. Java é compilado para um código de byte específico que pode ser executado na Java Virtual Machine (JVM).

Hoje em dia, a maioria das linguagens dinâmicas também compila código-fonte para código de máquina ou código de byte em tempo real, isso é chamado de compilação Just In Time (JIT). Ele produz uma execução mais rápida porque executar o código percorrendo um AST de nós (como no Awesome) é menos eficiente do que executar o código de máquina ou código de bytes. A razão pela qual a execução do byte-code é mais rápida é porque está mais próxima do código de máquina do que um AST.
Trazer seu modelo de execução para mais perto da máquina sempre produzirá resultados mais rápidos.

Se você deseja compilar sua linguagem, você tem várias opções, você pode:

- compilar em seu próprio formato de código de byte e criar uma máquina virtual para executá-lo,
- compilá-lo para byte-code JVM usando ferramentas como ASM,
- escrever seu próprio compilador de código de máquina,
- usar uma estrutura de compilador existente como LLVM.

Compilar para um formato de código de byte personalizado é a opção mais popular para linguagens dinâmicas. Python e novas implementações de Ruby estão indo nessa direção. Mas, isso só faz sentido se você estiver codificando em uma linguagem de baixo nível como C. Como as máquinas virtuais são muito simples (é basicamente um loop e um caso de switch), você precisa de controle total sobre a estrutura do seu programa na memória para fazê-lo o mais rápido possível.

Se você quiser mais detalhes sobre as máquinas virtuais, pule para o próximo capítulo.

USANDO LLVM DE RUBY

Estaremos usando ligações LLVM Ruby para compilar um subconjunto do Awesome para código de máquina em tempo real. Para compilar uma linguagem completa, a maioria das partes do tempo de execução deve ser reescrita para ser acessível de dentro do LLVM.

Primeiro, você precisará instalar o LLVM e as ligações Ruby. Você pode encontrar instruções sobre ruby-llvm project page. A instalação do LLVM pode levar algum tempo, certifique-se de ter uma bebida saborosa antes de iniciar a compilação.

Veja como usar o LLVM do Ruby.

```
# Cria um novo módulo para conter o código
mod = LLVM::Module.create("awesome")
# Cria a função principal que será chamada
main = mod.functions.add("main", [INT, LLVM::Type.pointer(PCHAR)], INT)
# Crie um bloco de código para construir o código de máquina dentro
builder = LLVM::Builder.create
builder.position_at_end(main.basic_blocks.append)

# Encontre a função que estamos chamando no módulo
func = mod.functions.named("puts")
# Call the function
builder.call(func, builder.global_string_pointer("hello"))
# Retornar
builder.ret(LLVM::Int(0))
```

Isso é equivalente ao código C a seguir. Na verdade, ele irá gerar um código de máquina semelhante.

```
int main (int argc, char const *argv[]) {
  puts("hello");
  return 0;
}
```

A diferença é que com o LLVM podemos gerar dinamicamente o código de máquina. Por causa disso, podemos criar código de máquina a partir do código Awesome.

COMPILANDO INCRÍVEL PARA O CÓDIGO DA MÁQUINA

Para compilar Awesome para código de máquina, criaremos uma classe Compiler que encapsulará a lógica de chamar LLVM para gerar o código de byte. Então, vamos estender os nós criados pelo analisador para fazê-los usar o compilador.

compiler.rb

```
require "rubygems"
require "parser"
require "nodes"

require 'llvm/core'
require 'llvm/execution_engine'
require 'llvm/transforms/scalar'
require 'llvm/transforms/ipo'

LLVM.init_x86

# O compilador é usado de maneira semelhante ao tempo de execução. Mas, em vez de executar código,
# irá gerar byte-code LLVM para execução posterior.
class Compiler

  # Inicializar tipos LLVM
  PCHAR = LLVM::Type.pointer(LLVM::Int8) # equivalente a * char em C
  INT   = LLVM::Int # equivalente a int em C

  attr_reader :locals

  def initialize(mod=nil, function=nil)
    # Crie o módulo LLVM no qual armazenar o código
    @module = mod || LLVM::Module.create("awesome")

    # Para rastrear nomes locais durante a compilação
    @locals = {}

    # Função em que o código será colocado
    @function = function ||
```

```
31  # Por padrão, criamos uma função principal, pois é o ponto de entrada padrão
32                  @module.functions.named("main") ||
33                  @module.functions.add("main", [INT, LLVM::Type.pointer(PCHAR)], INT)
34
35      # Crie um construtor de código de bytes LLVM
36      @builder = LLVM::Builder.create
37      @builder.position_at_end(@function.basic_blocks.append)
38
39      @engine = LLVM::ExecutionEngine.create_jit_compiler(@module)
40    end
41
42    # Cabeçalho inicial para inicializar o módulo.
43    def preamble
44      define_external_functions
45    end
46
47    def finish
48      @builder.ret(LLVM::Int(0))
49    end
50
51    # Crie uma nova string.
52    def new_string(value)
53      @builder.global_string_pointer(value)
54    end
55
56    # Crie um novo número.
57    def new_number(value)
58      LLVM::Int(value)
59    end
60
61    # Chame uma função.
62    def call(func, args=[])
63      f = @module.functions.named(func)
64      @builder.call(f, *args)
65    end
66
67    # Atribuir uma variável local
68    def assign(name, value)
69      # Aloca a memória e retorna um ponteiro
      para ela
70      ptr = @builder.alloca(value.type)
71      # Armazene o valor dentro do ponteiro
72      @builder.store(value, ptr)
```

```
73      # Acompanhe o ponteiro para que o compilador possa encontrar o nome de volta mais
tarde.
74      @locals[name] = ptr
75    end
76
77    # Carregue o valor de uma variável local.
78    def load(name)
79      @builder.load(@locals[name])
80    end
81
82    # Define uma função.
83    def function(name)
84      func = @module.functions.add(name, [], INT)
85      generator = Compiler.new(@module, func)
86      yield generator
87      generator.finish
88    end
89
90    # Otimize o código de bytes LLVM gerado.
91    def optimize
92      @module.verify!
93      pass_manager = LLVM::PassManager.new(@engine)
94      pass_manager.simplifycfg! # Simplifique o CFG
95      pass_manager.mem2reg!     # Promova a memória para registrar
96      pass_manager.gdce!        # Eliminação global morta
97    end
98
99    # JIT compila e executa o byte-code LLVM.
100   def run
101     @engine.run_function(@function, 0, 0)
102   end
103
104   def dump
105     @module.dump
106   end
107
108   private
109     def define_external_functions
110       fun = @module.functions.add("printf", [LLVM::Type.pointer(PCHAR)], INT, { :varargs =
111       fun.linkage = :external
112
113       fun = @module.functions.add("puts", [PCHAR], INT)
114       fun.linkage = :external
```

```
115
116         fun = @module.functions.add("read", [INT, PCHAR, INT], INT)
117         fun.linkage = :external
118
119         fun = @module.functions.add("exit", [INT], INT)
120         fun.linkage = :external
121       end
122   end
123
124         # Classe de reabertura com suporte do compilador para implementar como cada nó é
          compilado
125         # (método de compilação).126
127   class Nodes
128     def compile(compiler)
129       nodes.map { |node| node.compile(compiler) }.last
130     end
131   end
132
133   class NumberNode
134     def compile(compiler)
135       compiler.new_number(value)
136     end
137   end
138
139   class StringNode
140     def compile(compiler)
141       compiler.new_string(value)
142     end
143   end
144
145   class CallNode
146     def compile(compiler)
147       raise "Receiver not supported for compilation" if receiver
148
149       # Acesso variável local
150       if receiver.nil? && arguments.empty? && compiler.locals[method]
151         compiler.loa d(method)
152
153       # Method call
154       else
155         compiled_arguments = arguments.map { |arg| arg.compile(compiler) }
156         compiler.call(method, compiled_arguments)
```

```
157       end
158     end
159   end
160
161   class SetLocalNode
162     def compile(compiler)
163       compiler.assign(name, value.compile(compiler))
164     end
165   end
166
167   class DefNode
168     def compile(compiler)
169       raise "Parameters not supported for compilation" if !params.empty?
170       compiler.function(name) do |function|
171         body.compile(function)
172       end
173     end
174   end
```

Com o compilador integrado aos nós, podemos agora compilar um programa simples.

test/compiler_test.rb

```
1  code = <<-CODE
2  def say_it:
3    x = "This is compiled!"
4    puts(x)
5  say_it
6  CODE
7
8  # Analise o código
9  node = Parser.new.parse(code)
10
11 # Compile-o
12 compiler = Compiler.new
13 compiler.preamble
14 node.compile(compiler)
15 compiler.finish
16
17 # Remova o comentário para gerar byte-code LLVM
18 # compiler.dump
```

```
19
20 # Otimize o byte-code LLVM
21 compiler.optimize
22
23 # JIT compilar e executar
24 compiler.run
```

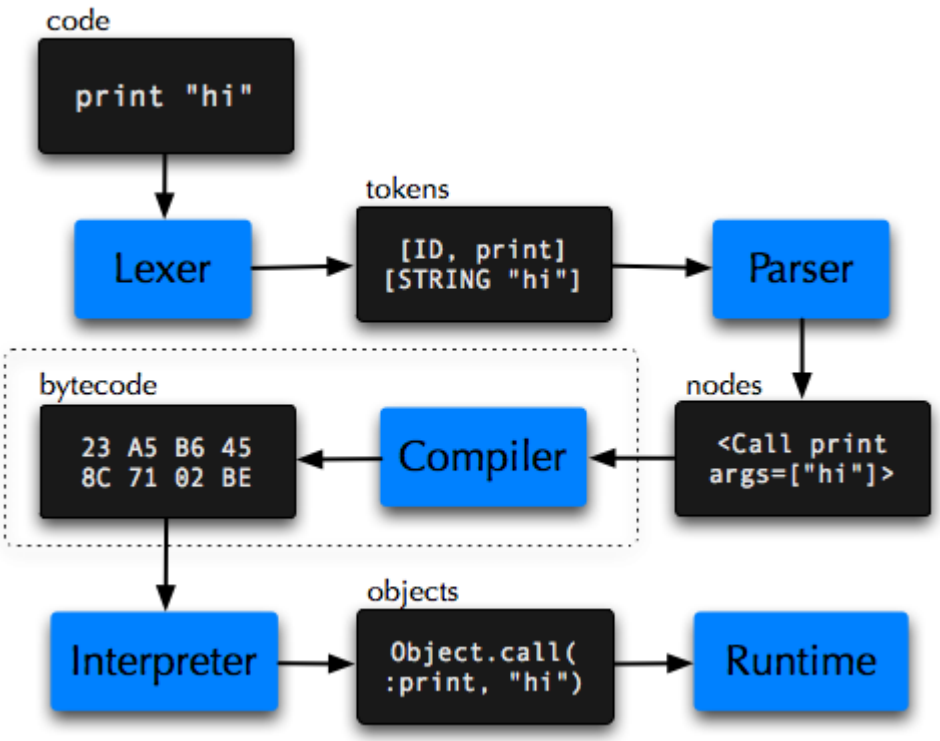
Lembre-se de que o compilador suporta apenas um subconjunto de nossa linguagem Awesome. Por exemplo, a programação orientada a objetos não é suportada. Para implementar isso, o tempo de execução e as estruturas usadas para armazenar as classes e objetos devem ser carregados de dentro do módulo LLVM. Você pode fazer isso compilando seu tempo de execução para o byte-code LLVM, seja escrevendo em C e usando o compilador C para LLVM enviado com
LLVM ou escrevendo seu tempo de execução em um subconjunto de sua linguagem que pode ser compilado para código de bytes LLVM.

VIRTUAL MACHINE

Se você leva a velocidade a sério e está pronto para implementar sua linguagem em C / C ++, é necessário introduzir uma VM (abreviação de Virtual Machine) em seu design.

Ao executar sua linguagem em uma VM, você deve compilar seus nós AST para o código de bytes. Apesar de adicionar uma etapa extra, a execução do código é mais rápida porque o formato do código de bytes está mais próximo do código de máquina do que um AST. A Figura 4 mostra como você pode introduzir um compilador de código de bytes em seu design de linguagem.

Figura 4



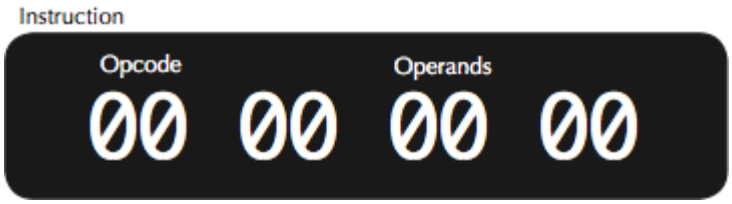
Olhe para tinypy e tinyrb na seção de linguagens interessantes para exemplos completos de pequenas linguagens baseadas em VM.

BYTE-CODE

O objetivo de uma VM é trazer o código o mais próximo possível do código de máquina, mas sem estar muito longe da linguagem original, tornando-a difícil (lenta) de compilar. Um byte-code ideal é rápido para compilar a partir do idioma de origem e rápido para executar, embora seja muito compacto na memória.

O código de byte de um programa consiste em instruções. Cada instrução começa com um opcode, especificando o que essa instrução faz, seguido por operandos, que são os argumentos das instruções.

Figura 5



A maioria das VMs compartilha um conjunto semelhante de instruções. Os mais comuns incluem getlocal para enviar um valor de variável local para a pilha, putstring para enviar uma string para a pilha, pop para remover um item da pilha, duplicar para duplicá-lo, etc. Você pode ver Ruby
1.9 (YARV) conjunto de instruções em lifegoo.pluskid.org. E a instrução JVM definida em xs4all.nl.

TYPES OF VM

As máquinas virtuais baseadas em pilha são mais comuns e mais simples de implementar. Python, Ruby 1.9 e o JVM são todos VMs baseados em pilha. A execução deles é baseada em uma pilha na qual os valores são armazenados para serem passados às instruções.

As instruções freqüentemente irão retirar valores da pilha, modificá-los e empurrar o resultado de volta para a pilha.

VMs baseadas em registro estão se tornando cada vez mais populares. Lua é baseada em registros. Eles estão mais próximos de como uma máquina real funciona, mas produzem instruções maiores com muitos operandos, mas geralmente menos instruções totais do que uma VM baseada em pilha.

PROTOTYPING A VM IN RUBY

Para entender o funcionamento interno de uma VM, veremos um protótipo escrito em Ruby. Deve-se notar que, em nenhum caso, uma VM real deve ser escrita em linguagens de alto nível como Ruby. Isso anula o propósito de trazer o código para mais perto da máquina. Cada linha de código em uma máquina virtual é otimizada para exigir o mínimo de instruções de máquina possível para executar, a linguagem de alto nível não fornece esse controle, mas C e C ++ fornecem.

vm/vm.rb

```ruby
# Bytecode
PUSH   = 0
ADD    = 1
PRINT  = 2
RETURN = 3

class VM
  def run(bytecode)
    # Pilha para passar o valor entre as instruções.
    stack = []
    # Ponteiro de instrução, índice da instrução atual sendo executada em bytecode.
    ip = 0

    while true
      case bytecode[ip]
      when PUSH
        stack.unshift bytecode[ip+=1]
      when ADD
        stack.unshift stack.pop + stack.pop
      when PRINT
        puts stack.pop
      when RETURN
        return
```

```
24          end
```

```
25
26          # Continue para a próxima
            introdução
27          ip += 1
28        end
29      end
30    end
31
32    VM.new.run [
33      # Aqui está o bytecode do nosso programa, o equivalente a: print 1 + 2.
34      # Opcode, Operand # Status da pilha após há execução da instrução.
35      PUSH,     1,          # stack = [1]
36      PUSH,     2,          # stack = [2, 1]
37      ADD,                  # stack = [3]
38      PRINT,                # stack = []
39      RETURN
40    ]
```

Como você pode ver, uma máquina virtual é simplesmente um loop e uma caixa de switch.

O byte-code que executamos é equivalente a print 1 + 2. Para conseguir isso, empurramos 1 e 2 para a pilha e executamos a instrução ADD que empurra o resultado da soma de tudo na pilha.

Procure no diretório code / vm o idioma completo usando esta VM.

GOING FURTHER
INDO ALÉM

Construir sua primeira língua é divertido, mas é apenas a ponta do iceberg. Há muito a descobrir nessa área. Aqui estão algumas coisas com as quais tenho brincado nos últimos anos.

HOMOICONICITY

Essa é a palavra que você deseja lançar ostensivamente em uma conversa geek. Embora pareça obscuro e complexo, significa que a representação primária do seu programa (o AST) pode ser acessada como uma estrutura de dados dentro do tempo de execução da linguagem. Você pode inspecionar e modificar o programa durante sua execução. Isso lhe dá poderes divinos.

Olhe no Interesting Languages seção do capítulo Referências para a linguagem Io e no Appendix: Mio, a minimalist homoiconic language deste livro para um exemplo de linguagem homoicônica que implementa `if` e boolean lógica em si.

SELF-HOSTING

Um intérprete de auto-hospedagem ou metacircular visa implementar o intérprete no idioma de destino. Isso é muito tedioso, pois você precisa implementar um interpretador primeiro para executar a linguagem, o que causa uma dependência circular entre os dois.

CoffeeScript é uma pequena linguagem que compila em JavaScript. O compilador CoffeeScript é ele mesmo written in CoffeeScript.

Rubinius é uma implementação Ruby que pretende ser auto-hospedada no futuro. No momento, algumas partes do tempo de execução ainda não foram escritas em

Ruby.

PyPy está tentando fazer isso de uma maneira muito mais simples: usando um subconjunto restritivo da linguagem Python para implementar o próprio Python.

WHAT'S MISSING?

Se você leva a sério a construção de uma linguagem real (real como em produção), então você deve considerar implementá-la em um ambiente mais rápido e robusto.
Ruby é bom para prototipagem rápida, mas horrível para implementação de linguagem.

As duas escolhas óbvias são Java na JVM, que oferece um coletor de lixo e uma boa coleção de bibliotecas portáteis, ou C / C ++, que oferece controle total sobre o que você está fazendo.

Agora vá lá e faça sua própria linguagem incrível!

RESOURCES

BOOKS & PAPERS

Language Implementation Patterns, by Terence Parr, from The Programmatic Programmers.

Smalltalk-80: The Language and its Implementation by Adele Goldberg and al., published by Addison-Wesley, May 1983.

A No-Frills Introduction to Lua 5.1 VM Instructions, by Kein-Hong Man.

The Implementation of Lua 5.0, by Roberto Ierusalimschy et al.

EVENTS

OOPSLA, The International Conference on Object Oriented Programming, Systems, Languages and Applications, is a gathering of many programming language authors.

The JVM Language Summit is an open technical collaboration among language designers, compiler writers, tool builders, runtime engineers, and VM architects for sharing experiences as creators of programming languages for the JVM.

FORUMS AND BLOGS

Lambda the Ultimate, The Programming Languages Weblog, discuss about new trends, research papers and various programming language topics.

INTERESTING LANGUAGES

Io:

> Io is a small, prototype-based programming language. The ideas in Io are mostly inspired by Smalltalk (all values are objects, all messages are dynamic), Self (prototype-based), NewtonScript (differential inheritance), Act1 (actors and futures for concurrency), LISP (code is a runtime inspectable/modifiable tree) and Lua (small, embeddable).

A few things to note about Io. It doesn't have any parser, only a lexer that converts the code to Message objects. This language is Homoiconic.

Factor is a concatenative programming language where references to dynamically-typed values are passed between words (functions) on a stack.

Lua:

> Lua is a powerful, fast, lightweight, embeddable scripting language.
>
> Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

tinypy and tinyrb are both subsets of more complete languages (Python and Ruby respectively) running on Virtual Machines inspired by Lua. Their code is only a few thousand lines long. If you want to introduce a VM in your language design, those are good starting points.

Need inspiration for your awesome language? Check out Wikipedia's programming lists: (List of programming languages)[http://en.wikipedia.org/wiki/List_of_programming_languages], (Esoteric programming languages)[http://en.wikipedia.org/wiki/Esoteric_programming_languagee]

In addition to the languages we know and use every day (C, C++, Perl, Java, etc.), you'll find many lesser-known languages, many of which are very interesting. You'll even find some esoteric languages such as Piet, a language that is programmed using images that look like abstract art. You'll also find some languages that are voluntarily impossible to use like Malbolge and BrainFuck and some amusing languages like LOLCODE, whose sole purpose is to be funny.

While some of these languages aren't practical, they can widen your horizons, and alter your conception of what constitutes a computer language. If you're going to design your own language, that can only be a good thing.

SOLUTIONS TO *DO IT YOURSELF*

SOLUTIONS TO *DO IT YOURSELF I*

a. Modify the lexer to parse: `while condition: ...` control structures.

Simply add `while` to the `KEYWORD` array on line 2.

```
1 KEYWORDS = ["def", "class", "if", "else", "true", "false", "nil", "while"]
```

b. Modify the lexer to delimit blocks with `{ ... }` instead of indentation.

Remove all indentation logic and add an `elsif` to parse line breaks.

bracket_lexer.rb

```
1  class BracketLexer
2    KEYWORDS = ["def", "class", "if", "else", "true", "false", "nil"]
3
4    def tokenize(code)
5      code.chomp!
6      i = 0
7      tokens = []
8
9      while i < code.size
10       chunk = code[i..-1]
11
12       if identifier = chunk[/\A([a-z]\w*)/, 1]
13         if KEYWORDS.include?(identifier)
14           tokens << [identifier.upcase.to_sym, identifier]
15         else
16           tokens << [:IDENTIFIER, identifier]
17         end
18         i += identifier.size
19
20       elsif constant = chunk[/\A([A-Z]\w*)/, 1]
21         tokens << [:CONSTANT, constant]
22         i += constant.size
```

```
23
24        elsif number = chunk[/\A([0-9]+)/, 1]
25          tokens << [:NUMBER, number.to_i]
26          i += number.size
27
28        elsif string = chunk[/\A"(.*?)"/, 1]
29          tokens << [:STRING, string]
30          i += string.size + 2
31
32        ######
33        # All indentation magic code was removed and only this elsif was added.
34        elsif chunk.match(/\A\n+/)
35          tokens << [:NEWLINE, "\n"]
36          i += 1
37        ######
38
39        elsif chunk.match(/\A /)
40          i += 1
41
42        else
43          value = chunk[0,1]
44          tokens << [value, value]
45          i += 1
46
47        end
48
49      end
50
51      tokens
52    end
53  end
```

SOLUTIONS TO *DO IT YOURSELF II*

a. Add a rule in the grammar to parse `while` blocks.

This rule is very similar to If.

```
# At the top add:
token WHILE

# ...

Expression:
# ...
| While
;

# ...

While:
  WHILE Expression Block  { result = WhileNode.new(val[1], val[2]) }
;
```

And in the `nodes.rb` file, you will need to create the class:

```
class WhileNode < Struct.new(:condition, :body); end
```

b. Add a grammar rule to handle the `!` unary operators.

Similar to the binary operator. Calling `!x` is like calling `x.!`.

```
Operator:
# ...
| '!' Expression    { result = CallNode.new(val[1], val[0], []) }
;
```

SOLUTIONS TO *DO IT YOURSELF III*

a. Implement inheritance by adding a superclass to each Awesome class.

```
class AwesomeClass < AwesomeObject
  # ...

  def initialize(superclass=nil)
```

```
    @runtime_methods = {}
    @runtime_superclass = superclass
    # ...
  end

  def lookup(method_name)
    method = @runtime_methods[method_name]
    unless method
      if @runtime_superclass
        return @runtime_superclass.lookup(method_name)
      else
        raise "Method not found: #{method_name}"
      end
    end
    method
  end
end

# ...

Runtime["Number"] = AwesomeClass.new(Runtime["Object"])
```

b. Add the method to handle `x + 2`.

```
Runtime["Number"].runtime_methods["+"] = proc do |receiver, arguments|
  result = receiver.ruby_value + arguments.first.ruby_value
  Runtime["Number"].new_with_value(result)
end
```

SOLUTIONS TO *DO IT YOURSELF IV*

a. Implement the `WhileNode`.

`while` is very similar to `if`.

```
class WhileNode
  def eval(context)
    while @condition.eval(context).ruby_value
```

```
        @body.eval(context)
      end
    end
end
```

APPENDIX: MIO, A MINIMALIST HOMOICONIC LANGUAGE

HOMOICOWHAT?

Homoiconicidade é um conceito difícil de entender. A melhor maneira de entendê-lo totalmente é implementá-lo. Esse é o objetivo desta seção. Também deve dar a você um vislumbre de uma linguagem não convencional.

Vamos construir uma pequena linguagem chamada Mio (para mini-Io). É derivado do Io language. O componente central de nossa linguagem serão as mensagens. As mensagens são um tipo de dados no Mio e também como os programas são representados e analisados, daí sua homoiconicidade. Implementaremos novamente o núcleo da nossa linguagem em Ruby, mas esta ocupará menos de 200 linhas de código.

MESSAGES ALL THE WAY DOWN

Como no Awesome, tudo é um objeto no Mio. Além disso, um programa, sendo chamadas de método e literais, é simplesmente uma série de mensagens. E as mensagens são separadas por espaços, não pontos, o que faz com que nossa linguagem se pareça muito com o inglês normal.

```
object method1 method2(argument)
```

É o equivalente semântico do seguinte código Ruby:

```
object.method1.method2(argument)
```

THE RUNTIME

Ao contrário de Awesome, mas como Javascript, Mio é baseado em protótipo. Portanto, ele não possui classes ou instâncias. Criamos novos objetos clonando os existentes. Objetos não têm classes, mas protótipos (protos), seus objetos pais.

Mio objetos são como dicionários ou hashes (novamente, muito parecido com Javascript). Eles contêm slots nos quais podemos armazenar métodos e valores, como strings, números e outros objetos.

mio/object.rb

```
module Mio
  class Object
    attr_accessor :slots, :protos, :value

    def initialize(proto=nil, value=nil)
      @protos = [proto].compact
      @value = value
      @slots = {}
    end

    # Procure um slot no objeto e protos atuais.
    def [](name)
      return @slots[name] if @slots.key?(name)
      message = nil
      @protos.each { |proto| return message if message = proto[name] }
      raise Mio::Error, "Missing slot: #{name.inspect}"
    end

    # Defina um slot
    def []=(name, message)
      @slots[name] = message
    end

    # O método de chamada é usado para avaliar um objeto.
    # Por padrão, os objetos avaliam a si mesmos.
    def call(*)
      self
    end
```

```
    def clone(val=nil)
      val ||= @value && @value.dup rescue TypeError
      Object.new(self, val)
    end
  end
end
```

Mio programas são uma cadeia de mensagens. Cada mensagem é um token. O seguinte trecho de código:

```
"hello" print
1 to_s print
```

é analisado como a seguinte cadeia de mensagens:

```
Message.new('"hello"',
  Message.new("print",
    Message.new("\n",
      Message.new("1",
        Message.new("to_s",
          Message.new("print"))))))
```

Observe que as quebras de linha (e pontos) também são mensagens. Quando executados, eles simplesmente zeram o receptor da mensagem.

```
self print # <= Quebra de linha redefine o receptor para self
self print # Agora parece que estamos iniciando uma nova expressão
           # com o mesmo receptor de antes.
```

Isso resulta no mesmo comportamento que em linguagens como Awesome, onde cada linha é uma expressão.

A unificação de todos os tipos de expressão em um tipo de dados torna nossa linguagem extremamente fácil de analisar (veja o método parse_all no código abaixo). Mensagens

são muito parecidos com tokens, portanto, nosso código de análise será semelhante ao de nosso lexer no Awesome. Nem precisamos de uma gramática com regras de análise!

mio/message.rb

```
module Mio
  # A mensagem é uma cadeia de tokens produzidos durante a análise.
  #   1 print.
  # é analisado para:
  #   Message.new("1",
  #               Message.new("print"))
  # Você pode então + chamar + a Mensagem de nível superior para avaliá-la.
  class Message < Object
    attr_accessor :next, :name, :args, :line, :cached_value

    def initialize(name, line)
      @name = name
      @args = []
      @line = line

      # Literais são valores estáticos, podemos avaliá-los corretamente
      # longe e armazene o valor em cache.
      @cached_value = case @name
      when /^\d+/
        Lobby["Number"].clone(@name.to_i)
      when /^"(.*)"$/
        Lobby["String"].clone($1)
      end

      @terminator = [".", "\n"].include?(@name)

      super(Lobby["Message"])
    end

    # Chame (aval) a mensagem no + receptor +.
    def call(receiver, context=receiver, *args)
      if @terminator
        # redefinir o receptor para o objeto no início da cadeia.
        # eg.:
        #   hello there. yo
        #  ^           ^___"." redefine de volta para o receptor aqui
        #  \_____________________________________________/
```

```
38          value = context
```

```
39        elsif @cached_value
40          # Já temos o valor
41          value = @cached_value
42        else
43            # Procure o slot no receptor
44          slot = receiver[name]
45
46          # Avalie o objeto no slot
47          value = slot.call(receiver, context, *@args)
48        end
49
50        # Passe para a próxima mensagem se
          houver
51        if @next
52          @next.call(value, context)
53        else
54          value
55        end
56      rescue Mio::Error => e
57        # Acompanhe a mensagem que causou o erro na saída
58        # número da linha e tal.
59        e.current_message ||= self
60        raise
61      end
62
63      def to_s(level=0)
64        s = "  " * level
65        s << "<Message @name=#{@name}"
66        s << ", @args=" + @args.inspect unless @args.empty?
67        s << ", @next=\n" + @next.to_s(level + 1) if @next
68        s + ">"
69      end
70
71      # Analisa uma string em uma cadeia de mensagens
72      def self.parse(code)
73        parse_all(code, 1).last
74      end
75
76      private
77        def self.parse_all(code, line)
78          code = code.strip
79          i = 0
80          message = nil
```

```
81          messages = []
82
83          # Código de análise Marrrvelous!
84          while i < code.size
85            case code[i..-1]
86            when /\A("[^"]*")/, # string
87                 /\A(\d+)/,     # number
88                 /\A(\.)+/,     # dot
89                 /\A(\n)+/,     # line break
90                 /\A(\w+)/      # name
91              m = Message.new($1, line)
92              if messages.empty?
93                messages << m
94              else
95                message.next = m
96              end
97              line += $1.count("\n")
98              message = m
99              i += $1.size - 1
100           when /\A(\(\s*)/ # arguments
101             start = i + $1.size
102             level = 1
103             while level > 0 && i < code.size
104               i += 1
105               level += 1 if code[i] == ?\(
106               level -= 1 if code[i] == ?\)
107             end
108             line += $1.count("\n")
109             code_chunk = code[start..i-1]
110             message.args = parse_all(code_chunk, line)
111             line += code_chunk.count("\n")
112           when /\A,(\s*)/
113             line += $1.count("\n")
114             messages.concat parse_all(code[i+1..-1], line)
115             break
116           when /\A(\s+)/, # ignore whitespace
117                /\A(#.*$)/ # ignore comments
118             line += $1.count("\n")
119             i += $1.size - 1
120           else
121             raise "Unknown char #{code[i].inspect} at line #{line}"
122           end
```

```
            i += 1
          end
          messages
        end
    end
end
```

A única parte que falta em nossa linguagem neste ponto é um método. Isso nos permitirá armazenar um bloco de código e executá-lo posteriormente em seu contexto original e no receptor.

Mas, haverá uma coisa especial sobre os argumentos de nosso método. Eles não serão avaliados implicitamente. Por exemplo, chamar o método (x) não avaliará x ao chamar o método, ele o passará como uma mensagem. Isso é chamado de avaliação preguiçosa. Isso nos permitirá implementar a estrutura de controle diretamente de dentro da nossa linguagem. Quando um argumento precisa ser avaliado, fazemos isso explicitamente chamando o método eval_arg (arg_index).

mio/method.rb

```
module Mio
  class Method < Object
    def initialize(context, message)
      @definition_context = context
      @message = message
      super(Lobby["Method"])
    end

    def call(receiver, calling_context, *args)
      # Woo ... muitos contextos aqui ... vamos esclarecer isso:
      #   @definition_context: onde o método foi definido
      #       calling_context: onde o método foi chamado
      #        method_context: onde o corpo do método (mensagem) está executando
      method_context = @definition_context.clone
      method_context["self"] = receiver
      method_context["arguments"] = Lobby["List"].clone(args)
      # Note: nenhum argumento é avaliado aqui. Nossa pequena linguagem só tem argumentos preguiçosos
      # evaluation. Se você passar args para um método, você deve avaliá-los explicitamente,
      # usando o seguinte método.
```

```
      method_context["eval_arg"] = proc do |receiver, context, at|
        (args[at.call(context).value] || Lobby["nil"]).call(calling_context)
      end
      @message.call(method_context)
    end
  end
end
```

Agora que temos todos os objetos no lugar, estamos prontos para inicializar nosso tempo de execução.

Nossa linguagem Awesome tinha um objeto Context, que servia como ambiente de execução. No Mio, vamos simplesmente usar um objeto como contexto de avaliação. Variáveis locais serão armazenadas nos slots desse objeto. O objeto raiz é chamado de Lobby. Porque ... é onde todos os objetos se encontram, no saguão. (Na verdade, o termo é retirado de Io.)

mio/bootstrap.rb

```
module Mio
  # Bootstrap
  object = Object.new

  object["clone"] = proc { |receiver, context| receiver.clone }
  object["set_slot"] = proc do |receiver, context, name, value|
    receiver[name.call(context).value] = value.call(context)
  end
  object["print"] = proc do |receiver, context|
    puts receiver.value
    Lobby["nil"]
  end

  # Apresentando o Lobby! Onde vivem todos os objetos fantásticos e também o contexto raiz
  # de avaliação.
  Lobby = object.clone

  Lobby["Lobby"]   = Lobby
  Lobby["Object"]  = object
  Lobby["nil"]     = object.clone(nil)
  Lobby["true"]    = object.clone(true)
  Lobby["false"]   = object.clone(false)
```

```
  Lobby["Number"]  = object.clone(0)
  Lobby["String"]  = object.clone("")
  Lobby["List"]    = object.clone([])
  Lobby["Message"] = object.clone
  Lobby["Method"]  = object.clone

  # O método que usaremos para definir métodos.
  Lobby["method"] = proc { |receiver, context, message| Method.new(context, message) }
end
```

IMPLEMENTING MIO IN MIO

Isso é tudo de que precisamos para começar a implementar nossa própria linguagem.

Primeiro, aqui está o que já podemos fazer: clonar objetos, definir e obter valores de slot.

test/mio/oop.mio

```
# Crie um novo objeto, clonando o objeto mestre
set_slot("dude", Object clone)
# Defina um slot nele
dude set_slot("name", "Bob")
# Chame o slot para recuperar seu valor
dude name print
# => Bob

# Define a method
dude set_slot("say_name", method(
  # Print unevaluated arguments (messages)
  arguments print
  # => <Message @name="hello..."">

  # Avalie o primeiro argumento
  eval_arg(0) print
  # => hello...

  # Acesse o receptor via `self`
  self name print
  # => Bob
))
```

```
23
24 # Call that method
25 dude say_name("hello...")
```

É aqui que entra a avaliação do argumento preguiçoso. Somos capazes de implementar o e e ou operadores de dentro do nosso idioma.

mio/boolean.mio

```
1  # Um objeto é sempre verdadeiro
2
3  Object set_slot("and", method(
4    eval_arg(0)
5  ))
6  Object set_slot("or", method(
7    self
8  ))
9
10 # ... exceto nulo e falso, que são falsos
11
12 nil set_slot("and", nil)
13 nil set_slot("or", method(
14   eval_arg(0)
15 ))
16
17 false set_slot("and", false)
18 false set_slot("or", method(
19   eval_arg(0)
20 ))
```

test/mio/boolean.mio

```
1  "yo" or("hi") print
2  # => yo
3
4  nil or("hi") print
5  # => hi
6
7  "yo" and("hi") print
8  # => hi
9
10 1 and(2 or(3)) print
11 # => 2
```

Usando esses dois operadores, podemos implementar if.

mio/if.mio

```
# Implementar se estiver usando lógica booleana

set_slot("if", method(
  # condição de avaliação
  set_slot("condition", eval_arg(0))
  condition and( # if true
    eval_arg(1)
  )
  condition or( # if false (else)
    eval_arg(2)
  )
))
```

E agora ... pônei mágico sagrado!

test/mio/if.mio

```
if(true,
  "condition is true" print,
  # else
  "nope" print
)
# => condition is true

if(false,
  "nope" print,
  # else
  "condition is false" print
)
# => condition is false
```

se definido de dentro de nossa linguagem!

BUT IT'S UGLY/

Tudo bem ... está funcionando, mas a sintaxe não é tão boa quanto impressionante. Uma adição seria escrita como: 1 + (2) por exemplo, e precisamos usar set_slot para atribuição, nada para impressionar seus amigos e inimigos.

Para resolver esse problema, podemos novamente pegar emprestado de Io e implementar o embaralhamento de operadores. Isso significa simplesmente reordenar operadores. Durante a fase de análise, transformaríamos 1 + 2 em 1 + (2). O mesmo vale para operadores ternários, como atribuição. x = 1 seria reescrito como = (x, 1). Isso introduz o açúcar sintático em nossa linguagem sem afetar sua homoiconicidade e admiração.

Você pode encontrar todo o código-fonte do Mio no diretório code / mio e executar seus testes de unidade com o comando: ruby -Itest test / mio_test.rb.

FAREWELL!

That is all for now. I hope you enjoyed my book!

If you find an error or have a comment or suggestion, please send me an email at macournoyer@gmail.com.

If you end up creating a programming language let me know, I'd love to see it!

Thanks for reading.

- Marc

Tradução feita por João Oliveira