How To Create Your Own Freaking Awesome Programming Language

TABLE OF CONTENT

Published November 2011.

Cover background image © Asja Boros

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.

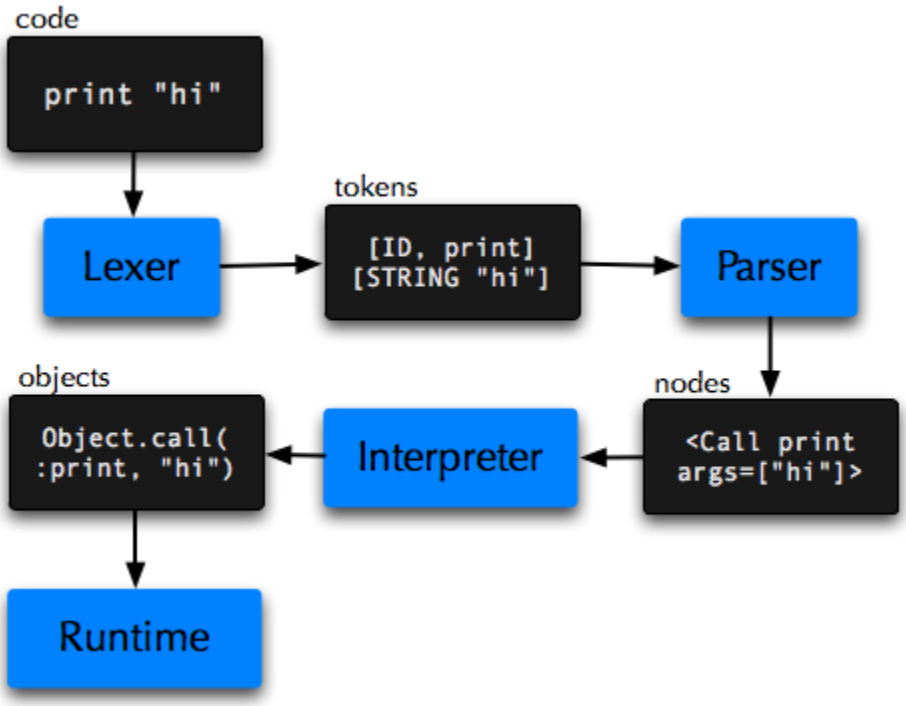
OVERVIEW

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



MEET AWESOME: OUR TOY LANGUAGE

The language we'll be coding in this book is called Awesome, because it is!

It's a mix of Ruby syntax and Python's indentation:

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

  def awesomeness:
    100

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

A couple of rules for our language:

- As in Python, blocks of code are delimited by their indentation.
- Classes are declared with the `class` keyword.
- Methods can be defined anywhere using the `def` keyword.
- Identifiers starting with a capital letter are constants which are globally accessible.
- Lower-case identifiers are local variables or method names.
- If a method takes no arguments, parenthesis can be skipped, much like in Ruby.
- The last value evaluated in a method is its return value.
- Everything is an object.

Some parts will be incomplete, but the goal with a toy language is to educate, experiment and set the foundation on which to build more.

LEXER

The lexer, or scanner, or tokenizer is the part of a language that converts the input, the code you want to execute, into tokens the parser can understand.

Let's say you have the following code:

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

Once this code goes through the lexer, it will look something like this:

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

What the lexer does is split the code and tag each part with the type of token it contains. This makes it easier for the parser to operate since it doesn't have to bother with details such as parsing a floating point number or parsing a complex string with escape sequences (`\n`, `\t`, etc.).

Lexers can be implemented using regular expressions, but more appropriate tools exists.

LEX (FLEX)

Flex is a modern version of Lex (that was coded by Eric Schmidt, CEO of Google, by the way) for generating C lexers. Along with Yacc, Lex is the most commonly used lexer for parsing.

It has been ported to several target languages.

- Rex for Ruby
- JFlex for Java

Lex and friends are not lexers per se. They are lexers compilers. You supply it a grammar and it will output a lexer. Here's what that grammar looks like:

```
BLANK        [ \t\n]+

%%

// Whitespace
{BLANK}       /* ignore */

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

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

On the left side a regular expression defines how the token is matched. On the right side, the action to take. The value of the token is stored in `yylval` and the type of token is returned.

More details in the Flex manual.

A Ruby equivalent, using the rexical gem (a port of Lex to Ruby), would be:

```
macro
  BLANK         [\ \t]+

rule
  # Whitespace
  {BLANK}        # ignore

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

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

Rexical follows a similar grammar as Lex. Regular expression on the left and action on the right. However, an array of two items is used to return the type and value of the matched token.

Mode details on the rexical project page.

RAGEL

A powerful tool for creating a scanner is Ragel. It's described as a *State Machine Compiler*: lexers, like regular expressions, are state machines. Being very flexible, they can handle grammars of varying complexities and output parser in several languages.

Here's what a Ragel grammar looks like:

```
%%{
machine lexer;

# Machine
number      = [0-9]+;
whitespace  = " ";
keyword     = "end" | "def" | "class" | "if" | "else" | "true" | "false" | "nil";

# Actions
main := |*
  whitespace;  # ignore
  number      => { tokens << [:NUMBER, data[ts..te].to_i] };
  keyword     => { tokens << [data[ts...te].upcase.to_sym, data[ts...te]] };
*|;

class Lexer
  def initialize
```

```
    %% write data;
  end

  def run(data)
    eof = data.size
    line = 1
    tokens = []
    %% write init;
    %% write exec;
    tokens
  end
end
}%%
```

More details in the Ragel manual (PDF).

Here are a few real-world examples of Ragel grammars used as language lexers:

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

PYTHON STYLE INDENTATION FOR AWESOME

If you intend to build a fully-functionning language, you should use one of the two previous tools. Since Awesome is a simplistic language and we want to illustrate the basic concepts of a scanner, we will build the lexer from scratch using regular expressions.

To make things more interesting, we'll use indentation to delimit blocks in our toy language, as in Python. All of indentation magic takes place within the lexer. Parsing blocks of code delimited with `{ ... }` is no different from parsing indentation when you know how to do it.

Tokenizing the following Python code:

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

will yield these tokens:

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

The block is wrapped in `INDENT` and `DEDENT` tokens instead of `{` and `}`.

The indentation-parsing algorithm is simple. You need to track two things: the current indentation level and the stack of indentation levels. When you encounter a line break followed by spaces, you update the indentation level. Here's our lexer for the Awesome language:

lexer.rb

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

  def tokenize(code)
    # Cleanup code by remove extra line breaks
    code.chomp!

    # Current character position we're parsing
    i = 0

    # Collection of all parsed tokens in the form [:TOKEN_TYPE, value]
    tokens = []

    # Current indent level is the number of spaces in the last indent.
    current_indent = 0
    # We keep track of the indentation levels we are in so that when we dedent, we can
    # check if we're on the correct level.
    indent_stack = []

    # This is how to implement a very simple scanner.
    # Scan one character at the time until you find something to parse.
```

```
    while i < code.size
      chunk = code[i..-1]

      # Matching standard tokens.
      #
      # Matching if, print, method names, etc.
      if identifier = chunk[/\A([a-z]\w*)/, 1]
        # Keywords are special identifiers tagged with their own name, 'if' will result
        # in an [:IF, "if"] token
        if KEYWORDS.include?(identifier)
          tokens << [identifier.upcase.to_sym, identifier]
        # Non-keyword identifiers include method and variable names.
        else
          tokens << [:IDENTIFIER, identifier]
        end
        # skip what we just parsed
        i += identifier.size

      # Matching class names and constants starting with a capital letter.
      elsif constant = chunk[/\A([A-Z]\w*)/, 1]
        tokens << [:CONSTANT, constant]
        i += constant.size

      elsif number = chunk[/\A([0-9]+)/, 1]
        tokens << [:NUMBER, number.to_i]
        i += number.size

      elsif string = chunk[/\A"(.*?)"/, 1]
        tokens << [:STRING, string]
        i += string.size + 2

      # Here's the indentation magic!
      #
      # We have to take care of 3 cases:
      #
      #   if true:  # 1) the block is created
      #     line 1
      #     line 2  # 2) new line inside a block
      #   continue  # 3) dedent
      #
      # This elsif takes care of the first case. The number of spaces will determine
      # the indent level.
```

```
64        elsif indent = chunk[/\A\:\n( +)/m, 1] # Matches ": <newline> <spaces>"
65          # When we create a new block we expect the indent level to go up.
66          if indent.size <= current_indent
67            raise "Bad indent level, got #{indent.size} indents, " +
68                  "expected > #{current_indent}"
69          end
70          # Adjust the current indentation level.
71          current_indent = indent.size
72          indent_stack.push(current_indent)
73          tokens << [:INDENT, indent.size]
74          i += indent.size + 2
75
76        # This elsif takes care of the two last cases:
77        # Case 2: We stay in the same block if the indent level (number of spaces) is the
78        #         same as current_indent.
79        # Case 3: Close the current block, if indent level is lower than current_indent.
80        elsif indent = chunk[/\A\n( *)/m, 1] # Matches "<newline> <spaces>"
81          if indent.size == current_indent # Case 2
82            # Nothing to do, we're still in the same block
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            # Cannot increase indent level without using ":", so this is an error.
93            raise "Missing ':'"
94          end
95          i += indent.size + 1
96
97        # Match long operators such as ||, &&, ==, !=, <= and >=.
98        # One character long operators are matched by the catch all `else` at the bottom.
99        elsif operator = chunk[/\A(\|\||&&|==|!=|<=|>=)/, 1]
100         tokens << [operator, operator]
101         i += operator.size
102
103       # Ignore whitespace
104       elsif chunk.match(/\A /)
105         i += 1
```

```
      # Catch all single characters
      # We treat all other single characters as a token. Eg.: ( ) , . ! + - <
      else
        value = chunk[0,1]
        tokens << [value, value]
        i += 1

      end

    end

    # Close all open blocks
    while indent = indent_stack.pop
      tokens << [:DEDENT, indent_stack.first || 0]
    end

    tokens
  end
end
```

You can test the lexer yourself by running the test file included with the book. Run `ruby -Itest test/lexer_test.rb` from the `code` directory and it should output `0 failures, 0 errors`. Here's an excerpt from that test file.

test/lexer_test.rb

```
code = <<-CODE
if 1:
  print "..."
  if false:
    pass
  print "done!"
print "The End"
CODE
tokens = [
  [:IF, "if"], [:NUMBER, 1],
  [:INDENT, 2],
    [:IDENTIFIER, "print"], [:STRING, "..."], [:NEWLINE, "\n"],
    [:IF, "if"], [:FALSE, "false"],
    [: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)
```

Some parsers take care of both lexing and parsing in their grammar. We'll see more about those in the next section.

DO IT YOURSELF I

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

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

Solutions to Do It Yourself I.

PARSER

By themselves, the tokens output by the lexer are just building blocks. The parser contextualizes them by organizing them in a structure. The lexer produces an array of tokens; the parser produces a tree of nodes.

Lets take those tokens from previous section:

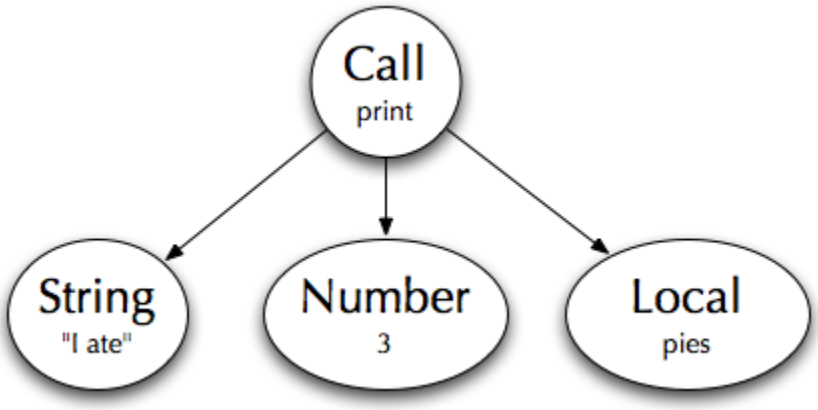
```
[IDENTIFIER print] [STRING "I ate"] [COMMA]
                  [NUMBER 3] [COMMA]
                  [IDENTIFIER pies]
```

The most common parser output is an Abstract Syntax Tree, or AST. It's a tree of nodes that represents what the code means to the language. The previous lexer tokens will produce the following:

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

Or as a visual tree:

Figure 2



The parser found that `print` was a method call and the following tokens are the arguments.

Parser generators are commonly used to accomplish the otherwise tedious task of building a parser. Much like the English language, a programming language needs a grammar to define its rules. The parser generator will convert this grammar into a parser that will compile lexer tokens into AST nodes.

BISON (YACC)

Bison is a modern version of Yacc, the most widely used parser. Yacc stands for Yet Another Compiler Compiler, because it compiles the grammar to a compiler of tokens. It's used in several mainstream languages, like Ruby. Most often used with Lex, it has been ported to several target languages.

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

Like Lex, from the previous chapter, Yacc compiles a grammar into a parser. Here's how a Yacc grammar rule is defined:

```
Call: /* Name of the rule */
  Expression '.' IDENTIFIER                  { $$ = CallNode_new($1, $3, NULL); }
| Expression '.' IDENTIFIER '(' ArgList ')'  { $$ = CallNode_new($1, $3, $5); }
/*   $1      $2      $3     $4     $5   $6   <= values from the rule are stored in
                                                these variables. */
;
```

On the left is defined how the rule can be matched using tokens and other rules. On the right side, between brackets is the action to execute when the rule matches.

In that block, we can reference tokens being matched using `$1`, `$2`, etc. Finally, we store the result in `$$`.

LEMON

Lemon is quite similar to Yacc, with a few differences. From its website:

> - Using a different grammar syntax which is less prone to programming errors.
> - The parser generated by Lemon is both re-entrant and thread-safe.
> - Lemon includes the concept of a non-terminal destructor, which makes it much easier to write a parser that does not leak memory.

For more information, refer to the the manual or check real examples inside Potion.

ANTLR

ANTLR is another parsing tool. This one let's you declare lexing and parsing rules in the same grammar. It has been ported to several target languages.

PEGS

Parsing Expression Grammars, or PEGs, are very powerful at parsing complex languages. I've used a PEG generated from peg/leg in tinyrb to parse Ruby's infamous syntax with encouraging results (tinyrb's grammar).

Treetop is an interesting Ruby tool for creating PEG.

OPERATOR PRECEDENCE

One of the common pitfalls of language parsing is operator precedence. Parsing `x + y * z` should not produce the same result as `(x + y) * z`, same for all other

operators. Each language has an operator precedence table, often based on mathematics order of operations. Several ways to handle this exist. Yacc-based parsers implement the Shunting Yard algorithm in which you give a precedence level to each kind of operator. Operators are declared in Bison and Yacc with `%left` and `%right` macros. Read more in Bison's manual.

Here's the operator precedence table for our language, based on the C language operator precedence:

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

The higher the precedence (top is higher), the sooner the operator will be parsed. If the line `a + b * c` is being parsed, the part `b * c` will be parsed first since `*` has higher precedence than +. Now, if several operators having the same precedence are competing to be parsed all the once, the conflict is resolved using associativity, declared with the `left` and `right` keyword before the token. For example, with the expression `a = b = c`. Since = has right-to-left associativity, it will start parsing from the right, `b = c`. Resulting in `a = (b = c)`.

For other types of parsers (ANTLR and PEG) a simpler but less efficient alternative can be used. Simply declaring the grammar rules in the right order will produce the desired result:

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

The parser will try to match rules recursively, starting from `expression` and finding its way to `primary`. Since `multiplicative` is the last rule called in the parsing process, it will have greater precedence.

CONNECTING THE LEXER AND PARSER IN AWESOME

For our Awesome parser we'll use Racc, the Ruby version of Yacc. It's much harder to build a parser from scratch than it is to create a lexer. However, most languages end up writing their own parser because the result is faster and provides better error reporting.

The input file you supply to Racc contains the grammar of your language and is very similar to a Yacc grammar.

grammar.y

```
class Parser

# Declare tokens produced by the 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

```

```
# Precedence table
# Based on http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B#Operator_precedence
prechigh
  left  '.'
  right '!'
  left  '*' '/'
  left  '+' '-'
  left  '>' '>=' '<' '<='
  left  '==' '!='
  left  '&&'
  left  '||'
  right '='
  left  ','
preclow

rule
  # All rules are declared in this format:
  #
  #   RuleName:
  #     OtherRule TOKEN AnotherRule    { code to run when this matches }
  #   | OtherRule                      { ... }
  #   ;
  #
  # In the code section (inside the {...} on the right):
  # - Assign to "result" the value returned by the rule.
  # - Use val[index of expression] to reference expressions on the left.


  # All parsing will end in this rule, being the trunk of the AST.
  Root:
    /* nothing */                     { result = Nodes.new([]) }
  | Expressions                       { result = val[0] }
  ;

  # Any list of expressions, class or method body, seperated by line breaks.
  Expressions:
    Expression                        { result = Nodes.new(val) }
  | Expressions Terminator Expression  { result = val[0] << val[2] }
    # To ignore trailing line breaks
  | Expressions Terminator            { result = val[0] }
  | Terminator                        { result = Nodes.new([]) }
  ;
```

```
57
58    # All types of expressions in our language
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    # All tokens that can terminate an expression
72    Terminator:
73      NEWLINE
74    | ";"
75    ;
76
77    # All hard-coded values
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    # A method call
87    Call:
88      # method
89      IDENTIFIER             { result = CallNode.new(nil, val[0], []) }
90      # method(arguments)
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 # Binary operators
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 # Assignment to a variable or constant
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 # Method definition
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   # Class definition
145   Class:
146     CLASS CONSTANT Block          { result = ClassNode.new(val[1], val[2]) }
147   ;
148
149   # if block
150   If:
151     IF Expression Block           { result = IfNode.new(val[1], val[2]) }
152   ;
153
154   # A block of indented code. You see here that all the hard work was done by the
155   # lexer.
156   Block:
157     INDENT Expressions DEDENT     { result = val[1] }
158   # If you don't like indentation you could replace the previous rule with the
159   # following one to separate blocks w/ curly brackets. You'll also need to remove the
160   # indentation magic section in the lexer.
161   # "{" Expressions "}"           { replace = val[1] }
162   ;
163 end
164
165 ---- header
166   require "lexer"
167   require "nodes"
168
169 ---- inner
170   # This code will be put as-is in the Parser class.
171   def parse(code, show_tokens=false)
172     @tokens = Lexer.new.tokenize(code) # Tokenize the code using our lexer
173     puts @tokens.inspect if show_tokens
174     do_parse # Kickoff the parsing process
175   end
176
177   def next_token
178     @tokens.shift
179   end
```

We then generate the parser with: `racc -o parser.rb grammar.y`. This will create a `Parser` class that we can use to parse our code. Run `ruby -Itest test/parser_test.rb` from the `code` directory to test the parser. Here's an excerpt from this file.

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)
```

Parsing code will return a tree of nodes. The root node will always be of type Nodes which contains children nodes.

DO IT YOURSELF II

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

b. Add a grammar rule to handle the `!` unary operators, eg.: `!x`. Making the following test pass (`test_unary_operator`):

Solutions to Do It Yourself II.

RUNTIME MODEL

The runtime model of a language is how we represent its objects, its methods, its types, its structure in memory. If the parser determines how you *talk* to the language, the runtime defines how the language *behaves*. Two languages could share the same parser but have different runtimes and be very different.

When designing your runtime, there are three factors you will want to consider:

- Speed: most of the speed will be due to the efficiency of the runtime.
- Flexibility: the more you allow the user to modify the language, the more powerful it is.
- Memory footprint: of course, all of this while using as little memory as possible.

As you might have noticed, these three constraints are mutually conflictual. Designing a language is always a game of give-and-take.

With those considerations in mind, there are several ways you can model your runtime.

PROCEDURAL

One of the simplest runtime models, like C and PHP (before version 4). Everything is centered around methods (procedures). There aren't any objects and all methods often share the same namespace. It gets messy pretty quickly!

CLASS-BASED

The class-based model is the most popular at the moment. Think of Java, Python, Ruby, etc. It might be the easiest model to understand for the users of your language.

PROTOTYPE-BASED

Except for Javascript, no Prototype-based languages have reached widespread popularity yet. This model is the easiest one to implement and also the most flexible because everything is a clone of an object.

Ian Piumarta describes how to design an Open, Extensible Object Model that allows the language's users to modify its behavior at runtime.

Look at the appendix at the end of this book for a sample prototype-based language: Appendix: Mio, a minimalist homoiconic language.

FUNCTIONAL

The functional model, used by Lisp and other languages, treats computation as the evaluation of mathematical functions and avoids state and mutable data. This model has its roots in Lambda Calculus.

OUR AWESOME RUNTIME

Since most of us are familiar with Class-based runtimes, I decided to use that for our Awesome language. The following code defines how objects, methods and classes are stored and how they interact together.

The `AwesomeObject` class is the central object of our runtime. Since everything is an object in our language, everything we will put in the runtime needs to be an

object, thus an instance of this class. `AwesomeObjects` have a class and can hold a ruby value. This will allow us to store data such as a string or a number in an object to keep track of its Ruby representation.

runtime/object.rb

```
# Represents an Awesome object instance in the Ruby world.
class AwesomeObject
  attr_accessor :runtime_class, :ruby_value

  # Each object have a class (named runtime_class to prevent errors with Ruby's class
  # method). Optionaly an object can hold a Ruby value (eg.: numbers and strings).
  def initialize(runtime_class, ruby_value=self)
    @runtime_class = runtime_class
    @ruby_value = ruby_value
  end

  # Call a method on the object.
  def call(method, arguments=[])
    # Like a typical Class-based runtime model, we store methods in the class of the
    # object.
    @runtime_class.lookup(method).call(self, arguments)
  end
end
```

Remember that in Awesome, everything is an object. Even classes are instances of the `Class` class. `AwesomeClasses` hold the methods and can be instantiated via their `new` method.

runtime/class.rb

```
# Represents a Awesome class in the Ruby world. Classes are objects in Awesome so they
# inherit from AwesomeObject.
class AwesomeClass < AwesomeObject
  attr_reader :runtime_methods

  # Creates a new class. Number is an instance of Class for example.
  def initialize
    @runtime_methods = {}

    # Check if we're bootstrapping (launching the runtime). During this process the
```

```
    # runtime is not fully initialized and core classes do not yet exists, so we defer
    # using those once the language is bootstrapped.
    # This solves the chicken-or-the-egg problem with the Class class. We can
    # initialize Class then set Class.class = Class.
    if defined?(Runtime)
      runtime_class = Runtime["Class"]
    else
      runtime_class = nil
    end

    super(runtime_class)
  end

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

  # Create a new instance of this class
  def new
    AwesomeObject.new(self)
  end

  # Create an instance of this Awesome class that holds a Ruby value. Like a String,
  # Number or true.
  def new_with_value(value)
    AwesomeObject.new(self, value)
  end
end
```

And here's the method object which will store methods defined from within our runtime.

runtime/method.rb

```
# Represents a method defined in the runtime.
class AwesomeMethod
  def initialize(params, body)
```

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

  def call(receiver, arguments)
    # Create a context of evaluation in which the method will execute.
    context = Context.new(receiver)

    # Assign arguments to local variables
    @params.each_with_index do |param, index|
      context.locals[param] = arguments[index]
    end

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

Notice that we use the `call` method for evaluating a method. That will allow us to define runtime methods from Ruby using `Procs`. Here's why:

```
p = proc do |arg1, arg2|
  # ...
end
p.call(1, 2) # execute the block of code passed to proc (everything between do ... end)
```

`Procs` can be executed via their `call` method. We'll see how this is used to defined runtime methods from Ruby in the bootstrapping section of the runtime.

Before we bootstrap our runtime, there is one missing object we need to define and that is the context of evaluation. The `Context` object encapsulates the environment of evaluation of a specific block of code. It will keep track of the following:

- Local variables.
- The current value of `self`, the object on which methods with no receivers are called, eg.: `print` is like `self.print`.

- The current class, the class on which methods are defined with the `def` keyword.

This is also where our constants (ie. classes) will be stored.

runtime/context.rb

```
# The evaluation context.
class Context
  attr_reader :locals, :current_self, :current_class

  # We store constants as class variable (class variables start with @@ and instance
  # variables start with @ in Ruby) since they are globally accessible. If you want to
  # implement namespacing of constants, you could store it in the instance of this
  # class.
  @@constants = {}

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

  # Shortcuts to access constants, Runtime[...] instead of Runtime.constants[...]
  def [](name)
    @@constants[name]
  end
  def []=(name, value)
    @@constants[name] = value
  end
end
```

Finally, we bootstrap the runtime. At first, no objects exist in the runtime. Before we can execute our first expression, we need to populate that runtime with a few objects: `Class`, `Object`, `true`, `false`, `nil` and a few core methods.

runtime/bootstrap.rb

```
# Bootstrap the runtime. This is where we assemble all the classes and objects together
# to form the runtime.
                                        # 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

# Create the Runtime object (the root context) on which all code will start its
# evaluation.
Runtime = Context.new(object_class.new)

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

# Everything is an object in our language, even true, false and nil. So they need
# to have a class too.
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)

# Add a few core methods to the runtime.

# Add the `new` method to classes, used to instantiate a class:
#  eg.: Object.new, Number.new, String.new, etc.
Runtime["Class"].runtime_methods["new"] = proc do |receiver, arguments|
  receiver.new
end

# Print an object to the console.
# eg.: print("hi there!")
Runtime["Object"].runtime_methods["print"] = proc do |receiver, arguments|
  puts arguments.first.ruby_value
  Runtime["nil"]
end
```

Now that we got all the pieces together we can call methods and create objects inside our runtime.

test/runtime_test.rb

```ruby
# Mimic Object.new in the language
object = Runtime["Object"].call("new")

assert_equal Runtime["Object"], object.runtime_class # assert object is an Object
```

Can you feel the language coming alive? We'll learn how to map that runtime to the nodes we created from our parser in the next section.

DO IT YOURSELF III

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

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

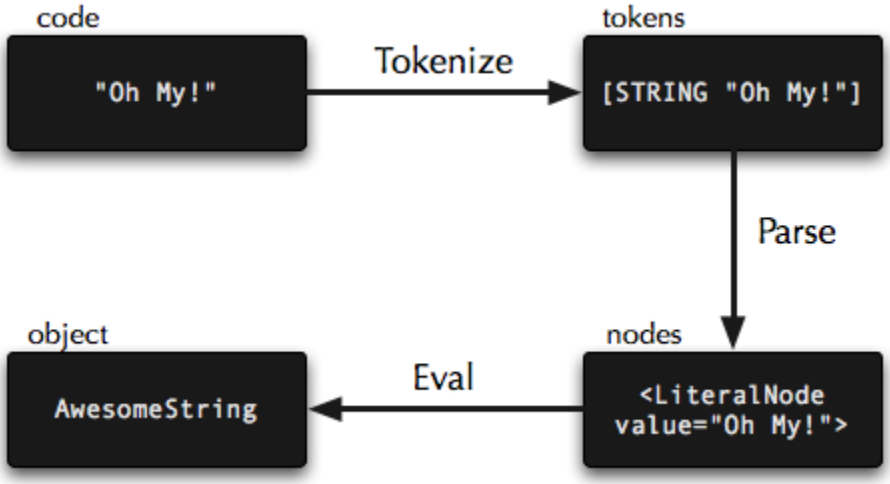
Solutions to Do It Yourself III.

INTERPRETER

The interpreter is the module that evaluates the code. It reads the AST produced by the parser and executes each action associated with the nodes, modifying the runtime.

Figure 3 recapitulates the path of a string in our language.

Figure 3



The lexer creates the token, the parser takes those tokens and converts it into nodes. Finally, the interpreter evaluates the nodes.

A common approach to execute an AST is to implement a Visitor class that visits all the nodes one by one, running the appropriate code. This make things even more modular and eases the optimization efforts on the AST. But for the purpose of this book, we'll keep things simple and let each node handle its evaluation.

Remember the nodes we created in the parser: `StringNode` for a string, `ClassNode` for a class definition? Here we're reopening those classes and adding a new method to each one: `eval`. This method will be responsible for interpreting that particular node.

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
  # This method is the "interpreter" part of our language. All nodes know how to eval
  # itself and returns the result of its evaluation by implementing the "eval" method.
  # The "context" variable is the environment in which the node is evaluated (local
  # variables, current class, etc.).
  def eval(context)
    return_value = nil
    nodes.each do |node|
      return_value = node.eval(context)
    end
    # The last value evaluated in a method is the return value. Or nil if none.
    return_value || Runtime["nil"]
  end
end

class NumberNode
  def eval(context)
    # Here we access the Runtime, which we'll see in the next section, to create a new
    # instance of the Number class.
    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)
    # If there's no receiver and the method name is the name of a local variable, then
    # it's a local variable access. This trick allows us to skip the () when calling a
    # method.
    if receiver.nil? && context.locals[method] && arguments.empty?
      context.locals[method]

    # Method call
    else
      if receiver
        value = receiver.eval(context)
      else
        # In case there's no receiver we default to self, calling "print" is like
        # "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)
    # Defining a method is adding a method to the current class.
    method = AwesomeMethod.new(params, body)
    context.current_class.runtime_methods[name] = method
  end
end

class ClassNode
  def eval(context)
    # Try to locate the class. Allows reopening classes to add methods.
    awesome_class = context[name]

    unless awesome_class # Class doesn't exist yet
      awesome_class = AwesomeClass.new
      # Register the class as a constant in the runtime.
      context[name] = awesome_class
    end

    # Evaluate the body of the class in its context. Providing a custom context allows
    # to control where methods are added when defined with the def keyword. In this
    # case, we add them to the newly created class.
    class_context = Context.new(awesome_class, awesome_class)

```

```
    body.eval(class_context)

    awesome_class
  end
end

class IfNode
  def eval(context)
    # We turn the condition node into a Ruby value to use Ruby's "if" control
    # structure.
    if condition.eval(context).ruby_value
      body.eval(context)
    end
  end
end
```

The interpreter part (the `eval` method) is the connector between the parser and the runtime of our language. Once we call `eval` on the root node, all children nodes are evaluated recursively. This is why we call the output of the parser an AST, for Abstract Syntax Tree. It is a tree of nodes. And evaluating the top level node of that tree will have the cascading effect of evaluating each of its children.

Let's run our first full program!

test/interpreter_test.rb

```
code = <<-CODE
class Awesome:
  def does_it_work:
    "yeah!"

awesome_object = Awesome.new
if awesome_object:
  print(awesome_object.does_it_work)
CODE

assert_prints("yeah!\n") { Interpreter.new.eval(code) }
```

To complete our language we can create a script to run a file or an REPL (for read-eval-print-loop), or interactive interpreter.

awesome

```
#!/usr/bin/env ruby
# The Awesome language!
#
# usage:
#   ./awesome example.awm # to eval a file
#   ./awesome             # to start the REPL
#
# on Windows run with: ruby awesome [options]

$:.unshift "." # Fix for Ruby 1.9
require "interpreter"
require "readline"

interpreter = Interpreter.new

# If a file is given we eval it.
if file = ARGV.first
  interpreter.eval File.read(file)

# Start the REPL, read-eval-print-loop, or interactive interpreter
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
```

Run the interactive interpreter by running `./awesome` and type a line of Awesome code, eg.: `print("It works!")`. Here's a sample Awesome session:

```
Awesome REPL, CTRL+C to quit
>> m = "This is Awesome!"
```

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

Also, try running a file: `./awesome example.awm`.

DO IT YOURSELF IV

a. Implement the `WhileNode` `eval` method.

Solutions to Do It Yourself IV.

COMPILATION

Dynamic languages like Ruby and Python have many advantages, like development speed and flexibility. But sometimes, compiling a language to a more efficient format might be more appropriate. When you write C code, you compile it to machine code. Java is compiled to a specific byte-code that you can run on the Java Virtual Machine (JVM).

These days, most dynamic languages also compile source code to machine code or byte-code on the fly, this is called Just In Time (JIT) compilation. It yields faster execution because executing code by walking an AST of nodes (like in Awesome) is less efficient then running machine code or byte-code. The reason why byte-code execution is faster is because it's closer to machine code then an AST is. Bringing your execution model closer to the machine will always produce faster results.

If you want to compile your language you have multiple choices, you can:

- compile to your own byte-code format and create a Virtual Machine to run it,
- compile it to JVM byte-code using tools like ASM,
- write your own machine code compiler,
- use an existing compiler framework like LLVM.

Compiling to a custom byte-code format is the most popular option for dynamic languages. Python and new Ruby implementations are going this way. But, this only makes sense if you're coding in a low level language like C. Because Virtual Machines are very simple (it's basically a loop and switch-case) you need total control over the structure of your program in memory to make it as fast a possible.

If you want more details on Virtual Machines, jump to the next chapter.

USING LLVM FROM RUBY

We'll be using LLVM Ruby bindings to compile a subset of Awesome to machine code on the fly. To compile a full language, most parts of the runtime have to be rewritten to be accessible from inside LLVM.

First, you'll need to install LLVM and the Ruby bindings. You can find instructions on ruby-llvm project page. Installing LLVM can take quite some time, make sure you got yourself a tasty beverage before launching compilation.

Here's how to use LLVM from Ruby.

```
# Creates a new module to hold the code
mod = LLVM::Module.create("awesome")
# Creates the main function that will be called
main = mod.functions.add("main", [INT, LLVM::Type.pointer(PCHAR)], INT)
# Create a block of code to build machine code within
builder = LLVM::Builder.create
builder.position_at_end(main.basic_blocks.append)

# Find the function we're calling in the module
func = mod.functions.named("puts")
# Call the function
builder.call(func, builder.global_string_pointer("hello"))
# Return
builder.ret(LLVM::Int(0))
```

This is the equivalent of the following C code. In fact, it will generate similar machine code.

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

The difference is that with LLVM we can dynamically generate the machine code. Because of that, we can create machine code from Awesome code.

COMPILING AWESOME TO MACHINE CODE

To compile Awesome to machine code, we'll create a `Compiler` class that will encapsulate the logic of calling LLVM to generate the byte-code. Then, we'll extend the nodes created by the parser to make them use the compiler.

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

# Compiler is used in a similar way as the runtime. But, instead of executing code, it
# will generate LLVM byte-code for later execution.
class Compiler

  # Initialize LLVM types
  PCHAR = LLVM::Type.pointer(LLVM::Int8) # equivalent to *char in C
  INT   = LLVM::Int # equivalent to int in C

  attr_reader :locals

  def initialize(mod=nil, function=nil)
    # Create the LLVM module in which to store the code
    @module = mod || LLVM::Module.create("awesome")

    # To track local names during compilation
    @locals = {}

    # Function in which the code will be put
    @function = function ||
```

```
31                # By default we create a main function as it's the standard entry point
32                @module.functions.named("main") ||
33                @module.functions.add("main", [INT, LLVM::Type.pointer(PCHAR)], INT)
34
35    # Create an LLVM byte-code builder
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  # Initial header to initialize the module.
43  def preamble
44    define_external_functions
45  end
46
47  def finish
48    @builder.ret(LLVM::Int(0))
49  end
50
51  # Create a new string.
52  def new_string(value)
53    @builder.global_string_pointer(value)
54  end
55
56  # Create a new number.
57  def new_number(value)
58    LLVM::Int(value)
59  end
60
61  # Call a function.
62  def call(func, args=[])
63    f = @module.functions.named(func)
64    @builder.call(f, *args)
65  end
66
67  # Assign a local variable
68  def assign(name, value)
69    # Allocate the memory and returns a pointer to it
70    ptr = @builder.alloca(value.type)
71    # Store the value insite the pointer
72    @builder.store(value, ptr)
```

```
73      # Keep track of the pointer so the compiler can find it back name later.
74      @locals[name] = ptr
75    end
76
77    # Load the value of a local variable.
78    def load(name)
79      @builder.load(@locals[name])
80    end
81
82    # Defines a function.
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    # Optimize the generated LLVM byte-code.
91    def optimize
92      @module.verify!
93      pass_manager = LLVM::PassManager.new(@engine)
94      pass_manager.simplifycfg! # Simplify the CFG
95      pass_manager.mem2reg!     # Promote Memory to Register
96      pass_manager.gdce!        # Dead Global Elimination
97    end
98
99    # JIT compile and run the LLVM byte-code.
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 # Reopen class supported by the compiler to implement how each node is compiled
125 # (compile method).
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     # Local variable access
150     if receiver.nil? && arguments.empty? && compiler.locals[method]
151       compiler.load(method)
152
153     # Method call
154     else
155       compiled_arguments = arguments.map { |arg| arg.compile(compiler) }
156       compiler.call(method, compiled_arguments)
```

```
    end
  end
end

class SetLocalNode
  def compile(compiler)
    compiler.assign(name, value.compile(compiler))
  end
end

class DefNode
  def compile(compiler)
    raise "Parameters not supported for compilation" if !params.empty?
    compiler.function(name) do |function|
      body.compile(function)
    end
  end
end
```

With the compiler integrated with the nodes, we can now compile a simple program.

test/compiler_test.rb

```
code = <<-CODE
def say_it:
  x = "This is compiled!"
  puts(x)
say_it
CODE

# Parse the code
node = Parser.new.parse(code)

# Compile it
compiler = Compiler.new
compiler.preamble
node.compile(compiler)
compiler.finish

# Uncomment to output LLVM byte-code
# compiler.dump
```

```
19
20 # Optimize the LLVM byte-code
21 compiler.optimize
22
23 # JIT compile & execute
24 compiler.run
```

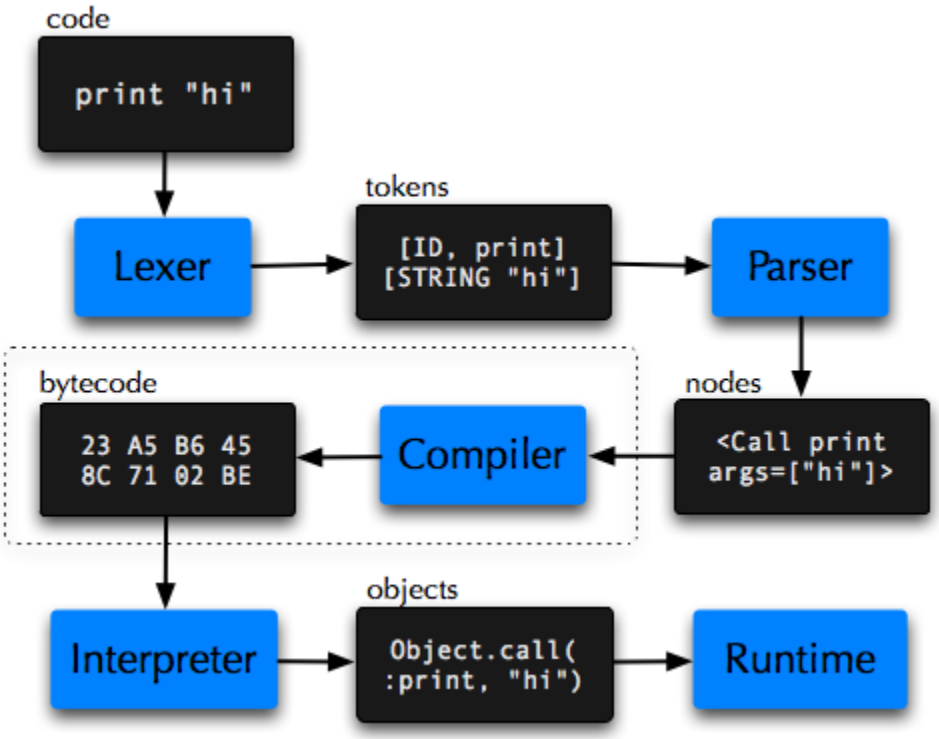
Remember that the compiler only supports a subset of our Awesome language. For example, object-oriented programming is not supported. To implemented this, the runtime and structures used to store the classes and objects have to be loaded from inside the LLVM module. You can do this by compiling your runtime to LLVM byte-code, either writing it in C and using the C-to-LLVM compiler shipped with LLVM or by writing your runtime in a subset of your language that can be compiled to LLVM byte-code.

VIRTUAL MACHINE

If you're serious about speed and are ready to implement your language in C/C++, you need to introduce a VM (short for Virtual Machine) in your design.

When running your language on a VM you have to compile your AST nodes to byte-code. Despite adding an extra step, execution of the code is faster because byte-code format is closer to machine code than an AST. Figure 4 shows how you can introduce a byte-code compiler into your language design.

Figure 4



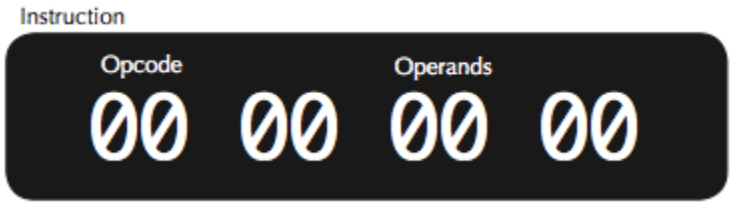
Look at tinypy and tinyrb in the Interesting Languages section for complete examples of small VM-based languages.

BYTE-CODE

The purpose of a VM is to bring the code as close to machine code as possible but without being too far from the original language making it hard (slow) to compile. An ideal byte-code is fast to compile from the source language and fast to execute while being very compact in memory.

The byte-code of a program consists of instructions. Each instruction starts with an opcode, specifying what this instruction does, followed by operands, which are the arguments of the instructions.

Figure 5



Most VMs share a similar set of instructions. Common ones include, `getlocal` to push a local variable value to the stack, `putstring` to push a string to the stack, `pop` to remove an item from the stack, `dup` to duplicate it, etc. You can see Ruby 1.9 (YARV) instruction set at lifegoo.pluskid.org. And the JVM instruction set at xs4all.nl.

TYPES OF VM

Stack-based virtual machines are most common and simpler to implement. Python, Ruby 1.9 and the JVM are all stack-based VMs. The execution of those is based around a stack in which values are stored to be passed around to instructions. Instructions will often pop values from the stack, modify it and push the result back to the stack.

Register-based VMs are becoming increasingly popular. Lua is register-based. They are closer to how a real machine work but produce bigger instructions with lots of operands but often fewer total instructions then a stack-based VM.

PROTOTYPING A VM IN RUBY

For the sake of understanding the inner working of a VM, we'll look at a prototype written in Ruby. It should be noted that, never in any case, a real VM should be written in a high level languages like Ruby. It defeats the purpose of bringing the code closer to the machine. Each line of code in a Virtual Machine is optimized to require as little machine instructions as possible to execute, high level language do not provide that control but C and C++ do.

vm/vm.rb

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

class VM
  def run(bytecode)
    # Stack to pass value between instructions.
    stack = []
    # Instruction Pointer, index of current instruction being executed in 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
      end
```

```
25
26        # Continue to next intruction
27        ip += 1
28      end
29    end
30  end
31
32  VM.new.run [
33    # Here is the bytecode of our program, the equivalent of: print 1 + 2.
34    # Opcode, Operand     # Status of the stack after execution of the instruction.
35    PUSH,     1,          # stack = [1]
36    PUSH,     2,          # stack = [2, 1]
37    ADD,                  # stack = [3]
38    PRINT,                # stack = []
39    RETURN
40  ]
```

As you can see, a Virtual Machine is simply a loop and a `switch-case`.

The byte-code that we execute is the equivalent of print 1 + 2. To achieve this, we push 1 and 2 to the stack and execute the ADD instruction which push the result of summing everything on the stack.

Look in the `code/vm` directory for the full language using this VM.

GOING FURTHER

Building your first language is fun, but it's only the tip of the iceberg. There's so much to discover in that field. Here are a few things I've been playing with in the last years.

HOMOICONICITY

That's the word you want to ostentatiously throw in a geek conversation. While it sounds obscure and complex, it means that the primary representation of your program (the AST) is accessible as a data structure inside the runtime of the language. You can inspect and modify the program as it's running. This gives you godlike powers.

Look in the Interesting Languages section of the References chapter for the *Io* language and in the Appendix: Mio, a minimalist homoiconic language of this book for a sample homoiconic language implementing `if` and boolean logic in itself.

SELF-HOSTING

A self-hosting, or metacircular interpreter aims to implement the interpreter in the target language. This is very tedious since you need to implement an interpreter first to run the language, which causes a circular dependency between the two.

CoffeeScript is a little language that compiles into JavaScript. The CoffeeScript compiler is itself written in CoffeeScript.

Rubinius is a Ruby implementation that aims to be self-hosted in the future. At the moment, some parts of the runtime are still not written in Ruby.

PyPy is trying to achieve this in a much simpler way: by using a restrictive subset of the Python language to implement Python itself.

WHAT'S MISSING?

If you're serious about building a real language (real as in production-ready), then you should consider implementing it in a faster and more robust environment. Ruby is nice for quick prototyping but horrible for language implementation.

The two obvious choices are Java on the JVM, which gives you a garbage collector and a nice collection of portable libraries, or C/C++, which gives you total control over what you're doing.

Now get out there and make your own awesome language!

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?

Homoiconicity is a hard concept to grasp. The best way to understand it fully is to implement it. That is the purpose of this section. It should also give you glimpse at an unconventional language.

We'll build a tiny language called Mio (for mini-Io). It is derived from the Io language. The central component of our language will be messages. Messages are a data type in Mio and also how programs are represented and parsed, thus its homoiconicity. We'll again implement the core of our language in Ruby, but this one will take less than 200 lines of code.

MESSAGES ALL THE WAY DOWN

Like in Awesome, everything is an object in Mio. Additionally, a program being method calls and literals, is simply a series of messages. And messages are separated by spaces not dots, which makes our language looks a lot like plain english.

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

Is the semantic equivalent of the following Ruby code:

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

THE RUNTIME

Unlike Awesome but like Javascript, Mio is prototype-based. Thus, it doesn't have any classes or instances. We create new objects by cloning existing ones. Objects don't have classes, but prototypes (`protos`), their parent objects.

Mio objects are like dictionaries or hashes (again, much like Javascript). They contain slots in which we can store methods and values such as strings, numbers and other objects.

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

    # Lookup a slot in the current object and protos.
    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

    # Set a slot
    def []=(name, message)
      @slots[name] = message
    end

    # The call method is used to eval an object.
    # By default objects eval to themselves.
    def call(*)
      self
    end
```

```
29
30      def clone(val=nil)
31        val ||= @value && @value.dup rescue TypeError
32        Object.new(self, val)
33      end
34    end
35  end
```

Mio programs are a chain of messages. Each message being a token. The following piece of code:

```
1  "hello" print
2  1 to_s print
```

is parsed as the following chain of messages:

```
1  Message.new('"hello"',
2    Message.new("print",
3      Message.new("\n",
4        Message.new("1",
5          Message.new("to_s",
6            Message.new("print"))))))
```

Notice line breaks (and dots) are also messages. When executed, they simply reset the receiver of the message.

```
1  self print # <= Line break resets the receiver to self
2  self print # So now it looks as if we're starting a new expression
3             # with the same receiver as before.
```

This results in the same behaviour as in languages such as Awesome, where each line is an expression.

The unification of all types of expression into one data type makes our language extremely easy to parse (see `parse_all` method in the code bellow). Messages

are much like tokens, thus our parsing code will be similar to the one of our lexer in Awesome. We don't even need a grammar with parsing rules!

mio/message.rb

```
module Mio
  # Message is a chain of tokens produced when parsing.
  #   1 print.
  # is parsed to:
  #   Message.new("1",
  #               Message.new("print"))
  # You can then +call+ the top level Message to eval it.
  class Message < Object
    attr_accessor :next, :name, :args, :line, :cached_value

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

      # Literals are static values, we can eval them right
      # away and cache the value.
      @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

    # Call (eval) the message on the +receiver+.
    def call(receiver, context=receiver, *args)
      if @terminator
        # reset receiver to object at begining of the chain.
        # eg.:
        #   hello there. yo
        #  ^           ^__ "." resets back to the receiver here
        #  \_____________________________________________/
        value = context
```

```
      elsif @cached_value
        # We already got the value
        value = @cached_value
      else
        # Lookup the slot on the receiver
        slot = receiver[name]

        # Eval the object in the slot
        value = slot.call(receiver, context, *@args)
      end

      # Pass to next message if some
      if @next
        @next.call(value, context)
      else
        value
      end
    rescue Mio::Error => e
      # Keep track of the message that caused the error to output
      # line number and such.
      e.current_message ||= self
      raise
    end

    def to_s(level=0)
      s = "  " * level
      s << "<Message @name=#{@name}"
      s << ", @args=" + @args.inspect unless @args.empty?
      s << ", @next=\n" + @next.to_s(level + 1) if @next
      s + ">"
    end

    # Parse a string into a chain of messages
    def self.parse(code)
      parse_all(code, 1).last
    end

    private
      def self.parse_all(code, line)
        code = code.strip
        i = 0
        message = nil
```

```
81          messages = []
82
83          # Marrrvelous parsing code!
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
```

The only missing part of our language at this point is a method. This will allow us to store a block of code and execute it later in its original context and on the receiver.

But, there will be one special thing about our method arguments. They won't be implicitly evaluated. For example, calling `method(x)` won't evaluate `x` when calling the method, it will pass it as a message. This is called lazy evaluation. It will allow us to implement control structure right from inside our language. When an argument needs to be evaluated, we do so explicitly by calling the method `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... lots of contexts here... lets clear that up:
      #   @definition_context: where the method was defined
      #       calling_context: where the method was called
      #        method_context: where the method body (message) is executing
      method_context = @definition_context.clone
      method_context["self"] = receiver
      method_context["arguments"] = Lobby["List"].clone(args)
      # Note: no argument is evaluated here. Our little language only has lazy argument
      # evaluation. If you pass args to a method, you have to eval them explicitly,
      # using the following method.
```

```
      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
```

Now that we have all the objects in place we're ready to bootstrap our runtime.

Our Awesome language had a `Context` object, which served as the environment of execution. In Mio, we'll simply use an object as the context of evaluation. Local variables will be stored in the slots of that object. The root object is called the `Lobby`. Because … it's where all the objects meet, in the lobby. (Actually, the term is taken from 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

  # Introducing the Lobby! Where all the fantastic objects live and also the root context
  # of evaluation.
  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

  # The method we'll use to define methods.
  Lobby["method"] = proc { |receiver, context, message| Method.new(context, message) }
end
```

IMPLEMENTING MIO IN MIO

This is all we need to start implementing our language in itself.

First, here's what we're already able to do: cloning objects, setting and getting slot values.

test/mio/oop.mio

```
# Create a new object, by cloning the master Object
set_slot("dude", Object clone)
# Set a slot on it
dude set_slot("name", "Bob")
# Call the slot to retrieve it's value
dude name print
# => Bob

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

  # Eval the first argument
  eval_arg(0) print
  # => hello...

  # Access the receiver via `self`
  self name print
  # => Bob
))
```

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

Here's where the lazy argument evaluation comes in. We're able to implement the `and` and `or` operators from inside our language.

mio/boolean.mio

```
1  # An object is always truish
2
3  Object set_slot("and", method(
4    eval_arg(0)
5  ))
6  Object set_slot("or", method(
7    self
8  ))
9
10 # ... except nil and false which are false
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
```

Using those two operators, we can implement `if`.

mio/if.mio

```
# Implement if using boolean logic

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

And now… holy magical pony!

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
```

`if` defined from inside our language!

BUT IT'S UGLY

All right… it's working, but the syntax is not as nice as Awesome. An addition would be written as: `1 +(2)` for example, and we need to use `set_slot` for assignment, nothing to impress your friends and foes.

To solve this problem, we can again borrow from Io and implement operator shuffling. This simply means reordering operators. During the parsing phase, we would turn `1 + 2` into `1 +(2)`. Same goes for ternary operators such as assignment. `x = 1` would be rewritten as `=(x, 1)`. This introduces syntactic sugar into our language without impacting its homoiconicity and awesomness.

You can find all the source code for Mio under the `code/mio` directory and run its unit tests with the command: `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*