How To Create Your Own Freaking Awesome Programming Language

TABLE OF CONTENT

1. Table of Content
2. Introduction
    1. Summary
    2. About The Author
    3. Before We Begin
3. Overview
    1. The Four Parts of a Language
    2. Meet Awesome: Our Toy Language
4. Lexer
    1. Lex (Flex)
    2. Ragel
    3. Python Style Indentation For Awesome
    4. Do It Yourself I
5. Parser
    1. Bison (Yacc)
    2. Lemon
    3. ANTLR
    4. PEGs
    5. Operator Precedence
    6. Connecting The Lexer and Parser in Awesome
    7. Do It Yourself II
6. Runtime Model
    1. Procedural
    2. Class-based
    3. Prototype-based
    4. Functional
    5. Our Awesome Runtime
    6. Do It Yourself III

7. Interpreter
    1. Evaluating The Nodes in Awesome
    2. Do It Yourself IV
8. Virtual Machine
    1. Byte-code
    2. The Stack
    3. Prototyping a VM in Ruby
9. Compilation
    1. Compiling to Byte-code
    2. Compiling to Machine Code
10. Mio, a minimalist homoiconic language
    1. Homoicowhat?
    2. Messages all the way down
    3. The Runtime
    4. Implementing Mio in Mio
    5. But it's ugly
11. Going Further
    1. Homoiconicity
    2. Self-Hosting
    3. What's Missing?
12. Resources
    1. Books & Papers
    2. Events
    3. Forums and Blogs
    4. Classes
    5. Interesting Languages
13. Farewell!
14. Solutions to Do It Yourself
    1. Solutions to Do It Yourself I
    2. Solutions to Do It Yourself II

3. Solutions to Do It Yourself III
4. Solutions to Do It Yourself IV

Revision #5, Published June 2013.

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*

Programming languages are very interesting artifacts because of their inherent nature of being text that's meant for both people to read, and for machines to execute. There's a fundamental, and often beautiful, balance to be struck between encoding clear instructions for the machine, the science part of it, and creating a piece of writing — and in this case, a language — that reads easily and naturally to the programmer, the art part of it. It is the perfect mix of art and science.

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 six 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

Along with this book, you should have received a `code` directory including all the code samples shown in this book. To run the examples you must have the following installed:

- Ruby 1.9 or 2.0
- Racc 1.4.6, install with: `gem install racc -v=1.4.6` (optional, to recompile the parser in the exercises).

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

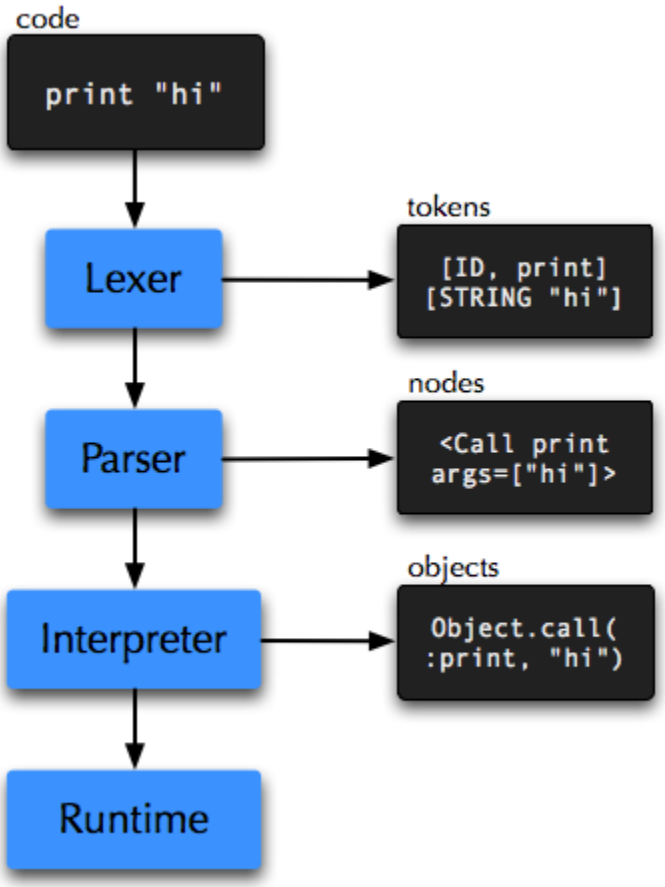
OVERVIEW

Since the early days of computing, the way we design a language has evolved, but most of the core principles haven't changed. 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
```

```
9  print(awesome.name)
10 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 has a receiver and no argument, parenthesis can be skipped, much like in Ruby. Eg.: `self.print` is the same as `self.print()`.
- The last value evaluated in a method is its return value.
- Everything is an object.

Lets get started with the first part of our language, the lexer.

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. If you think of your program as a series of sentences, tokens would be the words in those sentences. The job of the lexer is to extract those words (tokens) and tag them with a type (is it a string, number, operator?).

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"] [","]
                     [NUMBER 3] [","]
                     [IDENTIFIER pies]
                   [")"]
```

What the lexer does is split the code into atomic units (tokens) and tag each one with the type of token it contains. This job can be done by some parsers, as we'll see in the next chapter, but separating it into two distinct processes makes it simpler for us developers and easier to understand.

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

Each of these tools take a grammar that will be compiled into the actual lexer. The format of these grammars are all alike. Regular expressions on the left hand side are repeatedly matched, in order, against the next portion of the input code string. When a match is found, the action on the right is taken.

LEX (FLEX)

Flex is a modern version of Lex (that was coded by Eric Schmidt, Ex-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.

- Rexical for Ruby
- JFlex for Java

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

```
%%
// Whitespace
[ \t\n]+      /* 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. The `yy` prefix in the variable names is an heritage from Yacc, a parser compiler which we'll talk about in the next chapter.

More details in the Flex manual.

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

```
rule
  # Whitespace
  [\ \t]+        # 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 is very flexible, and can handle grammars of varying complexities and output lexers 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 (in Java)
- Potion's lexer (in C)

PYTHON STYLE INDENTATION FOR AWESOME

If you intend to build a fully-functioning language, you should use one of the previously mentioned 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:

```
[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:

In file code/lexer.rb

Our lexer will be used like so: `Lexer.new.tokenize("code")`, and will return an array of tokens (a token being a tuple of `[TOKEN_TYPE, TOKEN_VALUE]`).

```
class Lexer
```

First we define the special keywords of our language in a constant. It will be used later on in the tokenizing process to disambiguate an identifier (method name, local variable, etc.) from a keyword.

```
7   KEYWORDS = ["def", "class", "if", "true", "false", "nil"]
8
9   def tokenize(code)
10    code.chomp! # Remove extra line breaks
11    tokens = [] # This will hold the generated tokens
12
```

We need to know how deep we are in the indentation so we keep track of the current indentation level we are in, and previous ones in the stack so that when we dedent, we can check if we're on the correct level.

```
16    current_indent = 0 # number of spaces in the last indent
17    indent_stack = []
18
```

Here is how to implement a very simple scanner. Advance one character at the time until you find something to parse. We'll use regular expressions to scan from the current position (`i`) up to the end of the code.

```
23    i = 0 # Current character position
24    while i < code.size
25      chunk = code[i..-1]
26
```

Each of the following `if/elsif`s will test the current code chunk with a regular expression. The order is important as we want to match `if` as a keyword, and not a method name, we'll need to apply it first.

First, we'll scan for names: method names and variable names, which we'll call identifiers. Also scanning for special reserved keywords such as `if`, `def` and `true`.

```
34        if identifier = chunk[/\A([a-z]\w*)/, 1]
35          if KEYWORDS.include?(identifier) # keywords will generate [:IF, "if"]
36            tokens << [identifier.upcase.to_sym, identifier]
37          else
38            tokens << [:IDENTIFIER, identifier]
39          end
40          i += identifier.size # skip what we just parsed
41
```

Now scanning for constants, names starting with a capital letter. Which means, class names are constants in our language.

```
44        elsif constant = chunk[/\A([A-Z]\w*)/, 1]
45          tokens << [:CONSTANT, constant]
46          i += constant.size
47
```

Next, matching numbers. Our language will only support integers. But to add support for floats, you'd simply need to add a similar rule and adapt the regular expression accordingly.

```
50        elsif number = chunk[/\A([0-9]+)/, 1]
51          tokens << [:NUMBER, number.to_i]
52          i += number.size
53
```

Of course, matching strings too. Anything between `"..."`.

```
55        elsif string = chunk[/\A"([^"]*)"/, 1]
56          tokens << [:STRING, string]
57          i += string.size + 2 # skip two more to exclude the `"`.
58
```

And 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, at the same level.
continue  # 3) Dedent.
```

This `elsif` takes care of the first case. The number of spaces will determine the indent level.

```
68        elsif indent = chunk[/\A\:\n( +)/m, 1] # Matches ": <newline> <spaces>"
69          if indent.size <= current_indent # indent should go up when creating a block
70            raise "Bad indent level, got #{indent.size} indents, " +
71                  "expected > #{current_indent}"
72          end
73          current_indent = indent.size
74          indent_stack.push(current_indent)
75          tokens << [:INDENT, indent.size]
76          i += indent.size + 2
77
```

The next `elsif` takes care of the two last cases:

- Case 2: We stay in the same block if the indent level (number of spaces) is the same as `current_indent`.
- Case 3: Close the current block, if indent level is lower than `current_indent`.

```
83        elsif indent = chunk[/\A\n( *)/m, 1] # Matches "<newline> <spaces>"
84          if indent.size == current_indent # Case 2
85            tokens << [:NEWLINE, "\n"] # Nothing to do, we're still in the same block
86          elsif indent.size < current_indent # Case 3
87            while indent.size < current_indent
88              indent_stack.pop
89              current_indent = indent_stack.last || 0
90              tokens << [:DEDENT, indent.size]
```

```
91            end
92            tokens << [:NEWLINE, "\n"]
93          else # indent.size > current_indent, error!
94            raise "Missing ':'" # Cannot increase indent level without using ":"
95          end
96          i += indent.size + 1
97
```

Long operators such as `||`, `&&`, ==, etc. will be matched by the following block. One character long operators are matched by the catch all `else` at the bottom.

```
101       elsif operator = chunk[/\A(\|\||&&|==|!=|<=|>=)/, 1]
102         tokens << [operator, operator]
103         i += operator.size
104
```

We're ignoring spaces. Contrary to line breaks, spaces are meaningless in our language. That's why we don't create tokens for them. They are only used to separate other tokens.

```
107       elsif chunk.match(/\A /)
108         i += 1
109
```

Finally, catch all single characters, mainly operators. We treat all other single characters as a token. Eg.: `( ) , . ! + - <`.

```
112       else
113         value = chunk[0,1]
114         tokens << [value, value]
115         i += 1
116
117       end
118
119     end
120
```

Close all open blocks. If the code ends without dedenting, this will take care of balancing the `INDENT`...`DEDENT`s.

```
123       while indent = indent_stack.pop
124         tokens << [:DEDENT, indent_stack.first || 0]
125       end
126
127       tokens
128     end
129   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.

In file *code/test/lexer_test.rb*

```
1   code = <<-CODE
2   if 1:
3     if 2:
4       print("...")
5       if false:
6         pass
7       print("done!")
8     2
9
10  print "The End"
11  CODE
12  tokens = [
13    [:IF, "if"], [:NUMBER, 1],                          # if 1:
14      [:INDENT, 2],
15        [:IF, "if"], [:NUMBER, 2],                      #   if 2:
16        [:INDENT, 4],
17          [:IDENTIFIER, "print"], ["(", "("],           #     print("...")
18                                    [:STRING, "..."],
19                                  [")", ")"],
20                                  [:NEWLINE, "\n"],
21          [:IF, "if"], [:FALSE, "false"],               #     if false:
```

```
        [:INDENT, 6],
          [:IDENTIFIER, "pass"],                        #       pass
        [:DEDENT, 4], [:NEWLINE, "\n"],
        [:IDENTIFIER, "print"], ["(", "("],             #     print("done!")
                                  [:STRING, "done!"],
                                [")", ")"],
    [:DEDENT, 2], [:NEWLINE, "\n"],
    [:NUMBER, 2],                                       #   2
  [:DEDENT, 0], [:NEWLINE, "\n"],
  [:NEWLINE, "\n"],                                     #
  [:IDENTIFIER, "print"], [:STRING, "The End"]          # print "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 outputed by the lexer are just building blocks. The parser contextualizes them by organizing them in a tree structure. The lexer produces an array of tokens; the parser produces a tree of nodes. This tree of nodes will be how the language represents our program in memory. It will walk across it to run the program, store some parts of it in methods, some other parts in classes.

Lets take those tokens from the previous section:

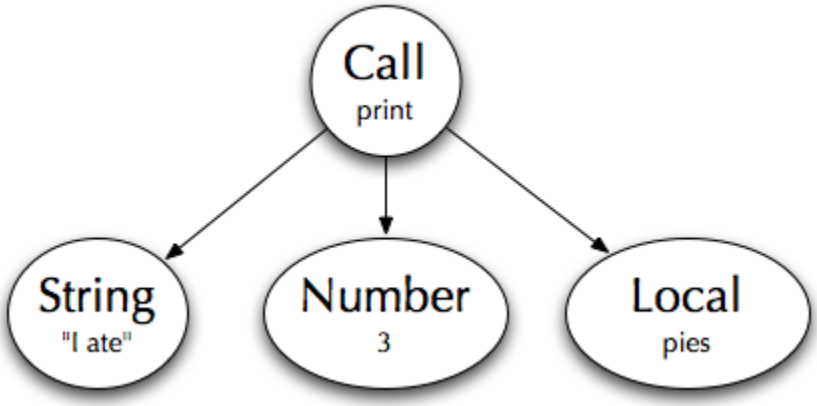
```
[IDENTIFIER print] ["("]
                     [STRING "I ate"] [","]
                     [NUMBER 3] [","]
                     [IDENTIFIER pies]
                   [")"]
```

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. 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.

Parser's grammars are similar to Lexer's. On the left you have things to match and on the right the action to take. But while the lexer matches against a sequence of characters, the parser will match particular patterns of tokens.

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 token types and other rules. On the right side, between brackets is the action to execute when the rule matches. In that block, we can reference token values being matched using `$1`, `$2`, etc. Finally, we store the result in `$$`.

LEMON

Lemon is quite similar to Yacc, with a few differences:

- Using a different grammar syntax which is less prone to programming errors.
- The generated parser is both re-entrant and thread-safe.
- Lemon makes it much easier to write a parser that don't leak memory.

Mainly it's a smaller, faster and more modern version of Yacc that was developed as part of the Sqlite database engine.

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.

It also offers powerful optimization techniques by allowing to easily rewrite the AST using a grammar.

The Java language bundled with this book uses ANTLR for generating it's parser.

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. PEG.js is also recent addition in the world of PEGs, this time for Javascript.

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, some languages

end up writing their own parser because the result is faster and provides better error reporting.

The Nodes

First, we need to define all types of node our parser will produce.

In file code/nodes.rb

The first type is responsible for holding a collection of nodes, each one representing an expression. You can think of it as the internal representation of a block of code.

Here we define nodes as Ruby classes that inherit from a `Struct`. This is a simple way, in Ruby, to create a class that holds some attributes (values). It is almost equivalent to:

```
class Nodes
  def initialize(nodes)
    @nodes = nodes
  end

  def nodes
    @nodes
  end
end

n = Nodes.new("this is stored @nodes")
n.nodes # => "this is stored @nodes"
```

But Ruby's `Struct` takes care of overriding the == operator for us and a bunch of other things that will make testing easier.

```
24 class Nodes < Struct.new(:nodes)
25   def <<(node) # Useful method for adding a node on the fly.
```

```
26      nodes << node
27      self
28    end
29  end
```

Literals are static values that have a Ruby representation. For example, a string, a number, `true`, `false`, `nil`, etc. We define a node for each one of those and store their Ruby representation inside their `value` attribute.

```
34  class LiteralNode < Struct.new(:value); end
35
36  class NumberNode < LiteralNode; end
37
38  class StringNode < LiteralNode; end
39
40  class TrueNode < LiteralNode
41    def initialize
42      super(true)
43    end
44  end
45
46  class FalseNode < LiteralNode
47    def initialize
48      super(false)
49    end
50  end
51
52  class NilNode < LiteralNode
53    def initialize
54      super(nil)
55    end
56  end
```

The node for a method call holds the `receiver`, the object on which the method is called, the `method` name and its arguments, which are other nodes.

```
61  class CallNode < Struct.new(:receiver, :method, :arguments); end
```

Retrieving the value of a constant by its `name` is done by the following node.

```
64 class GetConstantNode < Struct.new(:name); end
```

And setting its value is done by this one. The `value` will be a node. If we're storing a number inside a constant, for example, `value` would contain an instance of `NumberNode`.

```
69 class SetConstantNode < Struct.new(:name, :value); end
```

Similar to the previous nodes, the next ones are for dealing with local variables.

```
72 class GetLocalNode < Struct.new(:name); end
73
74 class SetLocalNode < Struct.new(:name, :value); end
```

Each method definition will be stored into the following node. It holds the `name` of the method, the name of its parameters (`params`) and the `body` to evaluate when the method is called, which is a tree of node, the root one being a `Nodes` instance.

```
79 class DefNode < Struct.new(:name, :params, :body); end
```

Class definitions are stored into the following node. Once again, the `name` of the class and its `body`, a tree of nodes.

```
83 class ClassNode < Struct.new(:name, :body); end
```

`if` control structures are stored in a node of their own. The `condition` and `body` will also be nodes that need to be evaluated at some point. Look at this node if you want to implement other control structures like `while`, `for`, `loop`, etc.

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

Once we have our nodes setup, we're ready to define the rules for parsing our language.

The Grammar

As mention earlier in this chapter, parsing rules are defined inside a grammar.

In file code/grammar.y

```
1 class Parser
```

We need to tell the parser what tokens to expect. So each type of token produced by our lexer needs to be declared here.

```
5  token IF
6  token DEF
7  token CLASS
8  token NEWLINE
9  token NUMBER
10 token STRING
11 token TRUE FALSE NIL
12 token IDENTIFIER
13 token CONSTANT
14 token INDENT DEDENT
```

Here is the Operator Precedence Table. As presented before, it tells the parser in which order to parse expressions containing operators. This table is based on the C and C++ Operator Precedence Table.

```
19 prechigh
20   left  '.'
21   right '!'
22   left  '*' '/'
```

```
    left  '+' '-'
    left  '>' '>=' '<' '<='
    left  '==' '!='
    left  '&&'
    left  '||'
    right '='
    left  ','
  preclow
```

In the following `rule` section, we define the parsing rules. All rules are declared using the following format:

```
RuleName:
  OtherRule TOKEN AnotherRule    { result = Node.new }
| OtherRule                      { ... }
;
```

In the action section (inside the `{...}` on the right), you can do the following:

- Assign to `result` the value returned by the rule, usually a node for the AST.
- Use `val[index of expression]` to get the `result` of a matched expressions on the left.

```
  rule
```

First, parsers are dumb, we need to explicitly tell it how to handle empty programs. This is what the first rule does. Note that everything between `/* ... */` is a comment.

```
    Program:
      /* nothing */                  { result = Nodes.new([]) }
    | Expressions                    { result = val[0] }
    ;

```

Next, we define what a list of expressions is. Simply put, it's series of expressions separated by a terminator (a new line or `;` as defined later). But once again, we need to explicitly define how to handle trailing and orphans line breaks (the last two lines).

One very powerful trick we'll use to define variable rules like this one (rules which can match any number of tokens) is *left-recursion*. Which means we reference the rule itself, directly or indirectly, on the left side **only**. This is true for the current type of parser we're using (LR). For other types of parsers like ANTLR (LL), it's the opposite, you can only use right-recursion.

As you'll see bellow, the `Expressions` rule references `Expressions` itself. In other words, a list of expressions can be another list of expressions followed by another expression.

```
67  Expressions:
68    Expression                          { result = Nodes.new(val) }
69  | Expressions Terminator Expression   { result = val[0] << val[2] }
70  | Expressions Terminator              { result = val[0] }
71  | Terminator                          { result = Nodes.new([]) }
72  ;
```

Every type of expression supported by our language is defined here.

```
75  Expression:
76    Literal
77  | Call
78  | Operator
79  | GetConstant
80  | SetConstant
81  | GetLocal
82  | SetLocal
83  | Def
84  | Class
85  | If
```

```
86    | '(' Expression ')'    { result = val[1] }
87    ;
```

Notice how we implement support for parentheses using the previous rule. `'('` `Expression` `')'` will force the parsing of `Expression` in its entirety first. Parentheses will then be discarded leaving only the fully parsed expression.

Terminators are tokens that can terminate an expression. When using tokens to define rules, we simply reference them by their type which we defined in the lexer.

```
96    Terminator:
97      NEWLINE
98    | ";"
99    ;
100
```

Literals are the hard-coded values inside the program. If you want to add support for other literal types, such as arrays or hashes, this it where you'd do it.

```
103   Literal:
104     NUMBER                  { result = NumberNode.new(val[0]) }
105   | STRING                  { result = StringNode.new(val[0]) }
106   | TRUE                    { result = TrueNode.new }
107   | FALSE                   { result = FalseNode.new }
108   | NIL                     { result = NilNode.new }
109   ;
110
```

Method calls can take three forms:

- Without a receiver (`self` is assumed): `method(arguments)`.
- With a receiver: `receiver.method(arguments)`.
- And a hint of syntactic sugar so that we can drop the `()` if no arguments are given: `receiver.method`.

Each one of those is handled by the following rule.

```
119   Call:
120     IDENTIFIER Arguments          { result = CallNode.new(nil, val[0], val[1]) }
121   | Expression "." IDENTIFIER
122       Arguments                   { result = CallNode.new(val[0], val[2], val[3]) }
123   | Expression "." IDENTIFIER     { result = CallNode.new(val[0], val[2], []) }
124   ;
125
126   Arguments:
127     "(" ")"                       { result = [] }
128   | "(" ArgList ")"               { result = val[1] }
129   ;
130
131   ArgList:
132     Expression                    { result = val }
133   | ArgList "," Expression        { result = val[0] << val[2] }
134   ;
135
```

In our language, like in Ruby, operators are converted to method calls. So `1 + 2` will be converted to `1.+(2)`. `1` is the receiver of the + method call, passing `2` as an argument. Operators need to be defined individually for the Operator Precedence Table to take action.

```
143   Operator:
144     Expression '||' Expression  { result = CallNode.new(val[0], val[1], [val[2]]) }
145   | Expression '&&' Expression  { result = CallNode.new(val[0], val[1], [val[2]]) }
146   | Expression '==' Expression  { result = CallNode.new(val[0], val[1], [val[2]]) }
147   | Expression '!=' Expression  { result = CallNode.new(val[0], val[1], [val[2]]) }
148   | Expression '>'  Expression  { result = CallNode.new(val[0], val[1], [val[2]]) }
149   | Expression '>=' Expression  { result = CallNode.new(val[0], val[1], [val[2]]) }
150   | Expression '<'  Expression  { result = CallNode.new(val[0], val[1], [val[2]]) }
151   | Expression '<=' Expression  { result = CallNode.new(val[0], val[1], [val[2]]) }
152   | Expression '+'  Expression  { result = CallNode.new(val[0], val[1], [val[2]]) }
153   | Expression '-'  Expression  { result = CallNode.new(val[0], val[1], [val[2]]) }
154   | Expression '*'  Expression  { result = CallNode.new(val[0], val[1], [val[2]]) }
155   | Expression '/'  Expression  { result = CallNode.new(val[0], val[1], [val[2]]) }
```

```
156     ;
157
```

Then we have rules for getting and setting values of constants and local variables.

```
159     GetConstant:
160       CONSTANT                      { result = GetConstantNode.new(val[0]) }
161     ;
162
163     SetConstant:
164       CONSTANT "=" Expression       { result = SetConstantNode.new(val[0], val[2]) }
165     ;
166
167     GetLocal:
168       IDENTIFIER                    { result = GetLocalNode.new(val[0]) }
169     ;
170
171     SetLocal:
172       IDENTIFIER "=" Expression     { result = SetLocalNode.new(val[0], val[2]) }
173     ;
```

Our language uses indentation to separate blocks of code. But the lexer took care of all that complexity for us and wrapped all blocks in `INDENT ... DEDENT`. A block is simply an increment in indentation followed by some code and closing with an equivalent decrement in indentation.

If you'd like to use curly brackets or `end` to delimit blocks instead, you'd simply need to modify this one rule. You'll also need to remove the indentation logic from the lexer.

```
183     Block:
184       INDENT Expressions DEDENT     { result = val[1] }
185     ;
186
```

The `def` keyword is used for defining methods. Once again, we're introducing a bit of syntactic sugar here to allow skipping the parentheses when there are no parameters.

```
189     Def:
190       DEF IDENTIFIER Block          { result = DefNode.new(val[1], [], val[2]) }
191     | DEF IDENTIFIER
192         "(" ParamList ")" Block     { result = DefNode.new(val[1], val[3], val[5]) }
193     ;
194
195     ParamList:
196       /* nothing */                 { result = [] }
197     | IDENTIFIER                    { result = val }
198     | ParamList "," IDENTIFIER      { result = val[0] << val[2] }
199     ;
200
```

Class definition is similar to method definition. Class names are also constants because they start with a capital letter.

```
203     Class:
204       CLASS CONSTANT Block          { result = ClassNode.new(val[1], val[2]) }
205     ;
206
```

Finally, `if` is similar to `class` but receives a *condition*.

```
208     If:
209       IF Expression Block           { result = IfNode.new(val[1], val[2]) }
210     ;
211   end
```

The final code at the bottom of this Racc file will be put as-is in the generated `Parser` class. You can put some code at the top (`header`) and some inside the class (`inner`).

```
---- header
  require "lexer"
  require "nodes"

---- inner
  def parse(code, show_tokens=false)
    @tokens = Lexer.new.tokenize(code) # Tokenize the code using our lexer
    puts @tokens.inspect if show_tokens
    do_parse # Kickoff the parsing process
  end

  def next_token
    @tokens.shift
  end
```

Compiling The Grammar To Form The Parser

We're now ready to generate the parser with the command: `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.

In file *code/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 the children nodes.

Nodes are the *dead* representation of your program. For example, when parsing a number, a `NumberNode` will be created, not an actual number inside our language. The next step is to bring that node to life in our language running environment. That running environment is called the runtime. We'll take a look at this in the next chapter.

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 conflicting. 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. Methods are stored into classes and objects are instances of classes. It

might be the easiest model to understand for the users of your language. This is the model we'll use in our Awesome 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. In practice, this model is very close to the Class-based one, with the exception that there are no classes. Instead, methods are stored directly into objects. Additionally, instead of having classes, objects have one or several parent objects from which they inherit their methods or attributes.

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 chapter Mio, a minimalist homoiconic language towards the end of this book for a sample prototype-based language.

FUNCTIONAL

The functional model, used by Haskell 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 and is close to the Prototype-based model. However, functions are made into first-class values and mutation of the data is discouraged.

OUR AWESOME RUNTIME

Since most of us are familiar with Class-based runtimes, we'll use that for our Awesome language. The following code defines how objects, methods and classes

are stored and how they interact together. It is strongly inspired by the Ruby runtime model.

Objects

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.

In file *code/runtime/object.rb*

```
1  class AwesomeObject
```

Each object has a class (named `runtime_class` to prevent conflicts with Ruby's `class` keyword). Optionally an object can hold a Ruby value. Eg.: numbers and strings will store their number or string Ruby equivalent in that variable.

```
6    attr_accessor :runtime_class, :ruby_value
7
8    def initialize(runtime_class, ruby_value=self)
9      @runtime_class = runtime_class
10     @ruby_value = ruby_value
11   end
```

Like a typical Class-based runtime model, we store methods in the class of the object. When calling a method on an object, we need to first lookup that method in the class, and then call it.

```
16   def call(method, arguments=[])
17     @runtime_class.lookup(method).call(self, arguments)
```

```
18    end
19  end
```

Classes And Methods

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

In file code/runtime/class.rb

```
1  class AwesomeClass < AwesomeObject
2    # Classes are objects in Awesome so they inherit from AwesomeObject.
3
4    attr_reader :runtime_methods
5
6    def initialize
7      @runtime_methods = {}
8      @runtime_class = Constants["Class"]
9    end
10
11   # Lookup a method
12   def lookup(method_name)
13     method = @runtime_methods[method_name]
14     raise "Method not found: #{method_name}" if method.nil?
15     method
16   end
17
18   # Helper method to define a method on this class from Ruby.
19   def def(name, &block)
20     @runtime_methods[name.to_s] = block
21   end
22
23   # Create a new instance of this class
24   def new
25     AwesomeObject.new(self)
26   end
27
28   # Create an instance of this Awesome class that holds a Ruby value. Like a String,
```

```
29    # Number or true.
30    def new_with_value(value)
31      AwesomeObject.new(self, value)
32    end
33  end
```

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

In file code/runtime/method.rb

```
1   class AwesomeMethod
2     def initialize(params, body)
3       @params = params
4       @body = body
5     end
6
7     def call(receiver, arguments)
8       # Create a context of evaluation in which the method will execute.
9       context = Context.new(receiver)
10
11      # Assign passed arguments to local variables.
12      @params.each_with_index do |param, index|
13        context.locals[param] = arguments[index]
14      end
15
16      # The body is a node (created in the parser).
17      # We'll talk in details about the `eval` method in the interpreter chapter.
18      @body.eval(context)
19    end
20  end
```

Notice that we use the `call` method for evaluating a method. That will allow us to define runtime methods from Ruby too using the `AwesomeClass`'s `def` method.

Context Of Evaluation

There is one missing piece we need to introduce in our runtime. It's 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.

In file code/runtime/context.rb

```
1 class Context
2   attr_reader :locals, :current_self, :current_class
3
4   def initialize(current_self, current_class=current_self.runtime_class)
5     @locals = {}
6     @current_self = current_self
7     @current_class = current_class
8   end
9 end
```

When a piece of code executes in our runtime, it will always execute within a given context.

Bootstrapping The Runtime

The last step is to bring the runtime to life by initializing the core classes and objects. This process is called bootstrapping. 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.

In file code/runtime/bootstrap.rb

First, we create a Ruby Hash in which we'll store all constants accessible from inside our runtime. Then, we populate this Hash with the core classes of our language.

```
4  Constants = {}
5
6  Constants["Class"] = AwesomeClass.new                # Defining the `Class` class.
7  Constants["Class"].runtime_class = Constants["Class"] # Setting `Class.class = Class`.
8  Constants["Object"] = AwesomeClass.new               # Defining the `Object` class
9  Constants["Number"] = AwesomeClass.new               # Defining the `Number` class
10 Constants["String"] = AwesomeClass.new
```

The root context will be the starting point where all our programs will start their evaluation. This will also set the value of `self` at the root of our programs.

```
15 root_self = Constants["Object"].new
16 RootContext = Context.new(root_self)
```

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

```
20 Constants["TrueClass"] = AwesomeClass.new
21 Constants["FalseClass"] = AwesomeClass.new
22 Constants["NilClass"] = AwesomeClass.new
23
24 Constants["true"] = Constants["TrueClass"].new_with_value(true)
25 Constants["false"] = Constants["FalseClass"].new_with_value(false)
26 Constants["nil"] = Constants["NilClass"].new_with_value(nil)
```

Now that we have injected all the core classes into the runtime, we can define methods on those classes.

The first method we'll define will allow us to do `Object.new` or `Number.new`. Keep in mind, `Object` or `Number` are instances of the `Class` class. By defining the `new` method on `Class`, it will be accessible on all its instances.

```
35 Constants["Class"].def :new do |receiver, arguments|
36   receiver.new
37 end
```

Next, we'll define the `print` method. Since we want to be able to call it from everywhere, we'll define it on `Object`. Remember from the parser's `Call` rule, methods without any receiver will be sent to `self`. So `print()` is the same as `self.print()`, and `self` will always be an instance of `Object`.

```
44 Constants["Object"].def :print do |receiver, arguments|
45   puts arguments.first.ruby_value
46   Constants["nil"] # We always want to return objects from our runtime
47 end
```

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

In file *code/test/runtime_test.rb*

```
1 # Mimic Object.new in the language
2 object = Constants["Object"].call("new")
3
4 assert_equal Constants["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`.

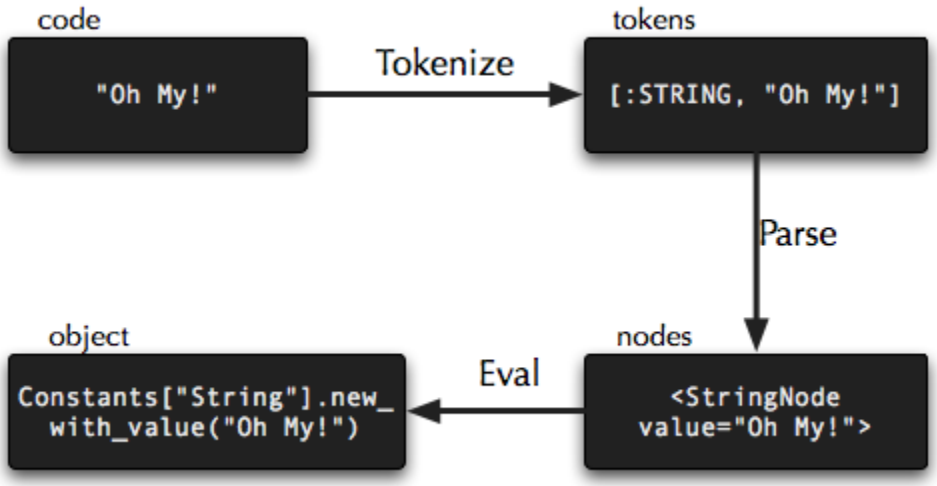
Solutions to Do It Yourself III.

INTERPRETER

The interpreter is the module that evaluates (interprets) the nodes produced by the parser and modify the runtime. Think of the nodes are a lifeless representation of your program, the runtime as the world your program lives in and the interpreter as the code that controls it.

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.

EVALUATING THE NODES IN AWESOME

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.

The Interpreter

Remember the nodes we created in the parser: `StringNode` for a string, `ClassNode` for a class definition? Here we're reopening those classes and add a new method to each one: `eval`. This method will be responsible for performing the action that the node represents, doing the addition for a + node, calling the method for a `CallNode`, and so on.

In file *code/interpreter.rb*

```
1  require "parser"
2  require "runtime"
```

First, we create an simple wrapper class to encapsulate the interpretation process. All this does is parse the code and call `eval` on the node at the top of the AST.

```
6  class Interpreter
7    def initialize
8      @parser = Parser.new
9    end
10
11   def eval(code)
12     @parser.parse(code).eval(RootContext)
13   end
14 end
```

The `Nodes` class will always be at the top of the AST. Its only purpose it to contain other nodes. It correspond to a block of code or a series of expressions.

The `eval` method of every node is the "interpreter" part of our language. All nodes know how to evalualte themselves and return the result of their evaluation. The `context` variable is the `Context` in which the node is evaluated (local variables, current self and current class).

```
class Nodes
  def eval(context)
    return_value = nil
    nodes.each do |node|
      return_value = node.eval(context)
    end
    return_value || Constants["nil"] # Last result is return value (or nil if none).
  end
end
```

We're using `Constants` that we created before when bootstrapping the runtime to access the objects and classes from inside the runtime.

Next, we implement `eval` on other node types. Think of that `eval` method as how the node bring itself to life inside the runtime.

```
class NumberNode
  def eval(context)
    Constants["Number"].new_with_value(value)
  end
end

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

class TrueNode
  def eval(context)
    Constants["true"]
  end
end

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

```
61
62 class NilNode
63   def eval(context)
64     Constants["nil"]
65   end
66 end
67
68 class GetConstantNode
69   def eval(context)
70     Constants[name]
71   end
72 end
73
74 class GetLocalNode
75   def eval(context)
76     context.locals[name]
77   end
78 end
```

When setting the value of a constant or a local variable, the `value` attribute is a node, created by the parser. We need to evaluate the node first, to convert it to an object, before storing it into a variable or constant.

```
83 class SetConstantNode
84   def eval(context)
85     Constants[name] = value.eval(context)
86   end
87 end
88
89 class SetLocalNode
90   def eval(context)
91     context.locals[name] = value.eval(context)
92   end
93 end
```

The `CallNode` for calling a method is a little more complex. It needs to set the receiver first and then evaluate the arguments before calling the method.

```
class CallNode
  def eval(context)
    if receiver
      value = receiver.eval(context)
    else
      value = context.current_self # Default to `self` if no receiver.
    end

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

Defining a method, using the `def` keyword, is done by adding a method to the current class.

```
class DefNode
  def eval(context)
    method = AwesomeMethod.new(params, body)
    context.current_class.runtime_methods[name] = method
  end
end
```

Defining a class is done in three steps:

1. Reopen or define the class.
2. Create a special context of evaluation (set `current_self` and `current_class` to the new class).
3. Evaluate the body of the class inside that context.

Check back how `DefNode` was implemented, adding methods to `context.current_class`. Here is where we set the value of `current_class`.

```
class ClassNode
  def eval(context)
    awesome_class = Constants[name] # Check if class is already defined
```

```
129
130       unless awesome_class # Class doesn't exist yet
131         awesome_class = AwesomeClass.new
132         Constants[name] = awesome_class # Define the class in the runtime
133       end
134
135       class_context = Context.new(awesome_class, awesome_class)
136       body.eval(class_context)
137
138       awesome_class
139     end
140   end
```

Finally, to implement `if` in our language, we turn the condition node into a Ruby value to use Ruby's `if`.

```
144 class IfNode
145   def eval(context)
146     if condition.eval(context).ruby_value
147       body.eval(context)
148     else # If no body is evaluated, we return nil.
149       Constants["nil"]
150     end
151   end
152 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.

Running Our First Program

Are you ready? Let's do this! Here we're running our first program!

In file code/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) }
```

The REPL

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

In file code/awesome

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

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(">> ")       # 1. Read
    Readline::HISTORY.push(line)
    value = interpreter.eval(line)         # 2. Eval
    puts "=> #{value.ruby_value.inspect}"  # 3. Print
  end                                      # 4. Loop

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.

VIRTUAL MACHINE

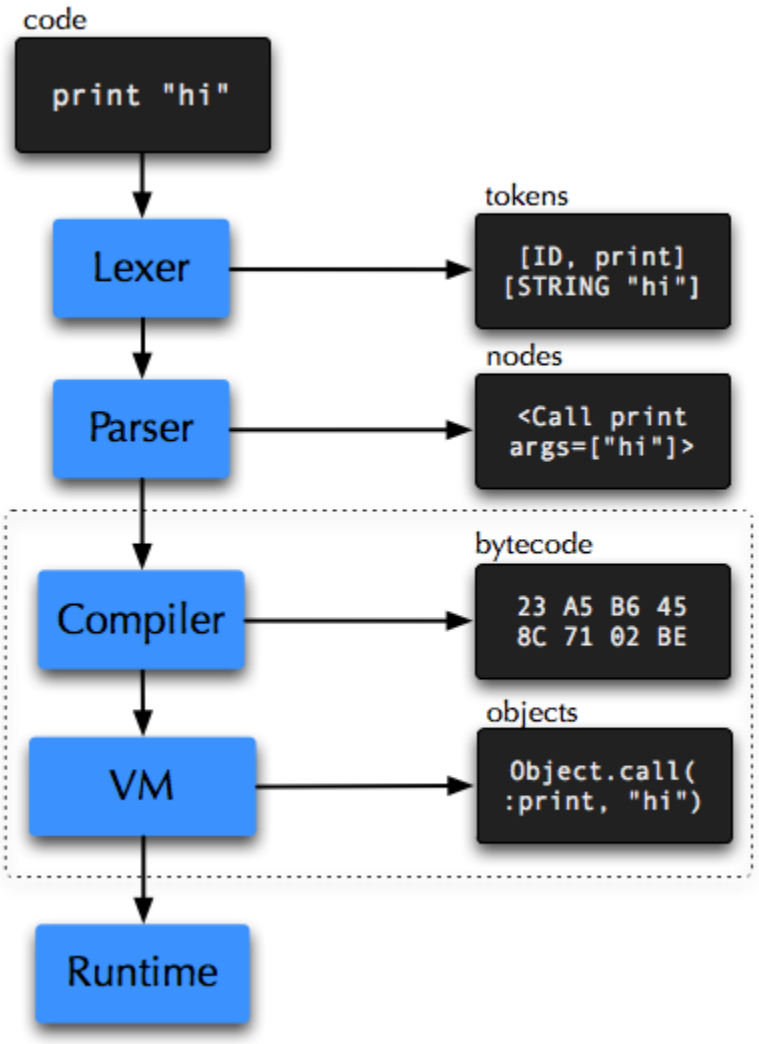
In the previous section, we've built what we call a walking-tree interpreter. The interpreter has to walk across the tree of nodes each time we execute a piece of code. This is fine if performance is not your greatest concern and you don't care about your AST using all the RAM. But most mainstream languages such as Ruby, Python, PHP and Java compile to a much more efficient format called byte-code.

Byte-code is similar to the machine code running on your processor, but is designed specifically for your language. The reason byte-code execution is faster, despite adding a few extra layers in your language, is because it is much closer to machine code than an AST is. Bringing your execution model closer to the machine will always yield faster results.

However, note that implementing a VM only make sense when you're coding your language in a low level language such as C. Because Virtual Machines are very simple, you need total control over the structure of your program in memory to make it as fast a possible. If you do not plan on coding your language in a low level language, jump to the next section about compiling to machine code.

Figure 4 shows how we'll introduce a VM into our language design.

Figure 4

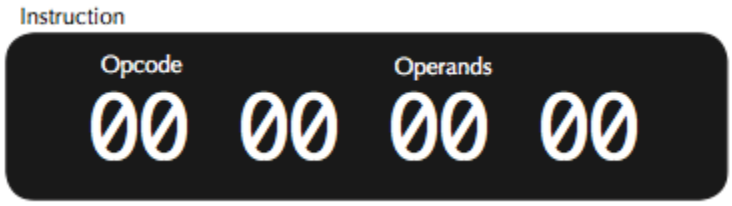


BYTE-CODE

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. Also, byte-code is portable as opposed to machine code.

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 get the value a local variable, `putstring` to load a string in memory, `call` or `send` to call a method, etc. You can see Ruby 1.9 (YARV) instruction set at lifegoo.pluskid.org. And the JVM instruction set at xs4all.nl.

THE STACK

The VM execution revolves around a stack that is used to pass values between instructions. Instructions will often pop values from the stack, modify it or consume it, and push the result back to the stack so that it can be used by the next instruction.

However, one important clarification, this is not the same stack that is responsible for the `StackOverflow` type of errors. There are two stacks in action here. One for passing values around between instructions and another one, the call stack, responsible for keeping track of which method we're currently in and also responsible for the famous `StackOverflow` error raised when the call stack is full. Each time you call a method, a new entry is pushed to the call stack.

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. However, please note that never in any case should a real VM should be written in a high level languages such as 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 languages do not provide that control, but C and C++ do.

The VM we're building will use a stack to pass values between instructions. The format of these instructions and what should be passed on the stack is defined in the byte-code format of the VM. Here is the one we will use.

In file code/*bytecode.rb*

```
#
#                              Bytecode format
#
# Opcode           Operands                        Stack before  /  after
# ------------------------------------------------------------------------
PUSH_NUMBER = 0  # Number to push on the stack   []               [number]
PUSH_SELF   = 1  #                               []               [self]
CALL        = 2  # Method, Number of arguments   [receiver, args] [result]
RETURN      = 3
```

Don't worry if this doesn't make total sense for now. We'll get to the details of each instruction when we implement the VM. At the very least, you can see that each instruction is very specialized, does only one thing and is expected to do it fast.

Other than having a stack, the VM will use an instruction pointer (`ip`) to keep track of the current instruction being executed. And finally, everything will be wrapped in a loop and a `case-when` (`switch-case` in other languages), one case for each type of instruction. All VMs share that same exact structure.

In file code/vm.rb

```
require "bytecode"
require "runtime"

class VM
  def run(bytecode)
```

First, we create the stack to pass values between instructions. And initialize the Instruction Pointer (`ip`), the index of current instruction being executed in `bytecode`.

```
    stack = []
    ip = 0

```

Next, we enter into the VM loop. Inside, we will advance one byte at the time in the `bytecode`. The first byte will be an opcode.

```
    while true
      case bytecode[ip] # Inspect the current byte, this will be the opcode.
```

Each of the following `when` block handles one type of instruction. They are all structured in the same way.

The first instruction will push a number on the stack.

```
      when PUSH_NUMBER
        ip += 1 # Advance to next byte, the operand.
        value = bytecode[ip] # Read the operand.

        stack.push Constants["Number"].new_with_value(value)
```

Since calling methods on `self` is something we do often we have a special instruction for pushing the value of `self` on the stack.

```
29        when PUSH_SELF
30          stack.push RootContext.current_self
```

The most complex instruction of our VM is `CALL`, for calling a method. It has two operands and expects several things to be on the stack.

```
34        when CALL
35          ip += 1 # Next byte contains the method name to call.
36          method = bytecode[ip]
37
38          ip += 1 # Next byte, the number of arguments on the stack.
39          argc = bytecode[ip]
```

At this point we assume arguments and the receiver of the method call have been pushed to the stack by other instructions. For example, if we were to call a method on `self` passing a number as an argument, we would find those two on the stack. Now pop all of those.

```
45          args = []
46          argc.times do
47            args << stack.pop
48          end
49          receiver = stack.pop
```

Using those values, we make the call exactly like we did in the interpreter (`CallNode`'s `eval`).

```
53          stack.push receiver.call(method, args)
```

Here is how we exit the VM loop. Each program must end with this instruction.

```
56        when RETURN
57          return stack.pop
58
59        end
60
```

Finally, we move forward one more byte to the next operand, to prepare for the next turn in the loop.

```
63        ip += 1
64      end
65    end
66  end
```

We can now execute a subset of the language we implemented so far by passing the proper byte-code to the VM. Lets do this with a very simple piece of code: `1 + 2` (equivalent to `1.+(2)`). To achieve this, we will call the + method on the number `1` passing the argument `2`. This can be done in the VM using a `CALL` instruction.

The `CALL` instruction takes two operands, the method to call (+) and the number of arguments we're passing (1). It expects two things to be on the stack before being called: the receiver of the method call (1), followed by all the arguments (2). In summary, to execute `1 + 2`, we'll push 1 and 2 on the stack and execute `CALL "+", 1`.

In file *code/test/vm_test.rb*

```
1  bytecode = [
2    # opcode     operands      stack after     description
3    # ------------------------------------------------------------------------
4    PUSH_NUMBER, 1,          # stack = [1]     push 1, the receiver of "+"
5    PUSH_NUMBER, 2,          # stack = [1, 2]  push 2, the argument for "+"
6    CALL,        "+", 1,     # stack = [3]     call 1.+(2) and push the result
```

```
  RETURN                  # stack = []
]

result = VM.new.run(bytecode)

assert_equal 3, result.ruby_value
```

Now that we have our VM, we need to translate our tree of nodes to byte-code. Also known as compiling. Thus, the next step is to create a byte-code compiler.

COMPILATION

The act of translating your code from one format to another is called compilation. Compiling a tree of nodes to byte-code, byte-code to machine code, rewriting JavaScript or even template engines like HAML, SASS or LESS are all done by a compiler.

If you want to compile your language you have two options.

1. Compiling to your own intermediate format and build an interpreter for it.
2. Compiling to an existing format:
    - Another language (or markup): Javascript, HTML, CSS
    - JVM byte-code (see ASM)
    - Directly to machine code.

First, we'll look at the first option. The intermediate format will be our own byte-code format. The interpreter of that byte-code will be the VM we built in the previous chapter.

We'll then create a machine code compiler for a subset of our language using LLVM.

COMPILING TO BYTE-CODE

In file *code/bytecode_compiler.rb*

```
1 require "parser"
```

The following code is structured almost exactly like `interpreter.rb`. The difference being that we won't evaluate the code on the spot, but generate byte-code that will achieve the same results when run inside the virtual machine (which is in fact a byte-code interpreter).

`BytecodeCompiler` here is the same as `Interpreter`, a simple wrapper around the parser and the nodes `compile` method, with the addition of an `emit` method to help generate the bytecode.

```
class BytecodeCompiler
  def initialize
    @parser = Parser.new
    @bytecode = []
  end

  def compile(code)
    @parser.parse(code).compile(self)
    emit RETURN
    @bytecode
  end

  def emit(opcode, *operands) # Usage: emit OPCODE, operand1, operand2, ..., operandX
    @bytecode << opcode
    @bytecode.concat operands
  end
end
```

Like in the interpreter, we reopen each node class supported by our compiler and add a `compile` method. Instead of passing an evaluation context, like in the interpreter, we pass the instance of `BytecodeCompiler` so that we can call its `emit` method to generate the byte-code.

```
class Nodes
  def compile(compiler)
    nodes.each do |node|
      node.compile(compiler)
    end
  end
end

class NumberNode
  def compile(compiler)
```

```
43      compiler.emit PUSH_NUMBER, value
44    end
45  end
```

Remember how we implemented the `CALL` instruction in the VM? It expects two things on the stack when called: the receiver and the arguments. Compiling those will emit the proper bytecode, which will in turn push the proper values on the stack.

One important thing to note about our compiler. Although it is very close to how a real compiler work, some parts have been simplified. Normally, we would not store the method name in the byte-code as is, but instead in a literal table. Then we'd refer to that method using its index in the literal table.

Eg.:
Literal table: `{ [0] 'print' }`
Instructions: `CALL 0, 1` (first operand being the index of 'print')

```
61  class CallNode
62    def compile(compiler)
63      if receiver
64        receiver.compile(compiler)
65      else
66        compiler.emit PUSH_SELF # Default to self if no receiver
67      end
68
69      arguments.each do |argument| # Compile the arguments
70        argument.compile(compiler)
71      end
72
73      compiler.emit CALL, method, arguments.size
74    end
75  end
```

We can now complete our new VM-based language by compiling to byte-code and executing that byte-code inside our VM instead of using the interpreter.

In file *code/test/bytecode_compiler_test.rb*

```
bytecode = BytecodeCompiler.new.compile("print(1+2)")

expected_bytecode = [
#                          Generated bytecode
#
# opcode       operands      stack after           description
# ------------------------------------------------------------------------------
  PUSH_SELF,               # stack = [self]        push the receiver of "print"
  PUSH_NUMBER, 1,          # stack = [1]
  PUSH_NUMBER, 2,          # stack = [self, 1, 2]  push the argument for "+"
  CALL,        "+", 1,     # stack = [self, 3]     call 1.+(2) and push the result
  CALL,        "print", 1, # stack = []            call self.print(3)
  RETURN
]

# Make sure the compiler generates the previous bytecode.
assert_equal expected_bytecode, bytecode

# Make sure the VM can run that bytecode.
assert_prints("3\n") { VM.new.run(bytecode) }
```

A Word About Static Typing

Our language uses Dynamic Typing, which means the type of a variable is determined dynamically at runtime and can change. With Static Typing, the type of a variable is determined at compile time (you declare it in your code) and can't change.

For example, when declaring a variable in Java, you need to define its type:

```
int a = 1;
```

The `int` part is telling the compiler that this variable can only hold integer values. This has two advantages:

- It allows the compiler to check that the values you assign to that variable are indeed integers (Type Checking).
- It allows the compiler to optimize your code a lot more.

For example, the compiler could generate a highly specialized `ADD_INT` instruction when you add two values that are known to be `int`s. That instruction can be optimized for this one very specific case and thus provide great performance.

Note that this approach is also possible with Dynamic Typing by gathering usage statistics from our runtime and optimizing for the most common type.

COMPILING TO MACHINE CODE

A VM is right in the middle between ease of development and performance. But if you need to go the extra mile in terms of performance, compiling to machine code is the ultimate step.

The challenge with machine code is that its format varies depending on which type of processor your machine is running. But thankfully, other people have tackled this problem for us and created libraries to generate machine code across all types of processors with one single API.

One of the most advanced library for doing this is LLVM. We'll use it via its Ruby bindings.

Using LLVM from Ruby

We'll be using LLVM Ruby bindings to compile a subset of Awesome to machine code on the fly. However, compiling a full language is much more challenging as 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 the compilation.

Here's how to use LLVM from Ruby.

```
# Creates a new module to hold the code
mod = LLVM::Module.new("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.new
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 while our program is running. 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 an `LLVMCompiler` 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. The structure is very similar to the one we used for our byte-code compiler.

In file *code/llvm_compiler.rb*

```
1  require "rubygems"
2  require "parser"
3  require "nodes"
4
5  require "llvm/core"
6  require "llvm/execution_engine"
7  require "llvm/transforms/scalar"
8
9  LLVM.init_x86
10
11 class LLVMCompiler
```

First we initialize some data types we'll use during compilation. Both correspond to common C types.

```
14   PCHAR = LLVM.Pointer(LLVM::Int8) # equivalent to *char in C
15   INT   = LLVM::Int # equivalent to int in C
16
```

When storing a value in a local variable, LLVM will return back a pointer. We need to keep track of the mapping local variable name => pointer. This is what the following Hash does.

```
20    attr_reader :locals
21
```

An instance of `LLVMCompiler` is responsible for compiling a given function. We pass an LLVM module (`mod`), which is a container in which to store the code, and a function to compile the code into.

By default the function will be the standard C entry point: `void main()`.

```
27    def initialize(mod=nil, function=nil)
28      @module = mod || LLVM::Module.new("awesome")
29
30      @locals = {} # To track local names during compilation
31
32      @function = function ||
33                  @module.functions.named("main") || # Default the function to `main`
34                  @module.functions.add("main", [], LLVM.Void)
35
36      @builder = LLVM::Builder.new # Prepare a builder to build code.
37      @builder.position_at_end(@function.basic_blocks.append)
38
39      @engine = LLVM::JITCompiler.new(@module) # The machine code compiler.
40    end
41
```

Before compiling our code, we'll declare external C functions we'll call from within our program. Here is where our compiler will cheat quite a bit. Instead of reimplementing our runtime inside the LLVM module, we won't support any of the OOP features and only allow calling basic C functions we declare here, namely `int puts(char*)`.

```
46    def preamble
47      fun = @module.functions.add("puts", [PCHAR], INT)
48      fun.linkage = :external
49    end
50
```

Always finish the function with a `return`.

```
52    def finish
53      @builder.ret_void
54    end
55
```

We'll also need to load literal values and be able to call functions. LLVM got us covered there.

```
57    def new_string(value)
58      @builder.global_string_pointer(value)
59    end
60
61    def new_number(value)
62      LLVM::Int(value)
63    end
64
65    def call(name, args=[])
66      function = @module.functions.named(name)
67      @builder.call(function, *args)
68    end
69
```

Keep in mind we're compiling to machine code that will run right inside the processor. There is no extra layer of abstraction here. When assigning local variables, we first need to allocate memory for it. This is what we do here using `alloca` and then store the value at that address in memory.

```
74    def assign(name, value)
75      ptr = @builder.alloca(value.type) # Allocate memory.
76      @builder.store(value, ptr) # Store the value in the allocated memory.
77      @locals[name] = ptr # Keep track of where we stored the local.
78    end
79
80    def load(name)
81      ptr = @locals[name]
```

```
82      @builder.load(ptr, name) # Load back the value stored for that local.
83    end
84
```

Methods defined inside our runtime are compiled to functions (like C functions). Functions are compiled using their own `LLVMCompiler` instance to scope their local variables and code blocks.

```
88    def function(name)
89      func = @module.functions.add(name, [], LLVM.Void)
90      compiler = LLVMCompiler.new(@module, func)
91      yield compiler
92      compiler.finish
93    end
94
```

One of the biggest advantage of using LLVM and not rolling our owning machine code compiler is that we're able to take advantage of all the optimizations. Compiling to machine code is the "easy" (super giant quotes here) part. But by default your code will not be that fast, you need to optimize it. This is what the `-O2` option of your C compiler does. Here we'll only use one optimization as an example, but LLVM has a lot of them.

```
101   def optimize
102     @module.verify! # Verify the code is valid.
103     pass_manager = LLVM::PassManager.new(@engine)
104     pass_manager.mem2reg! # Promote memory to machine registers.
105   end
106
```

Here is where the magic happens! We JIT compile and run the LLVM code. JIT, for just-in-time, because we compile it right before we execute it as opposed to AOT, for ahead-of-time where we compile it upfront, like C.

```
110     def run
111       @engine.run_function(@function)
112     end
113
```

LLVM doesn't compile directly to machine code but to an intermediate format called IR, which is similar to assembly. If you want to inspect the generated IR for this module, call the following method.

```
117     def dump
118       @module.dump
119     end
120   end
```

Now that we have our compiler ready we use the same approach as before and reopen all the supported nodes and implement how each one is compiled.

```
124   class Nodes
125     def llvm_compile(compiler)
126       nodes.map { |node| node.llvm_compile(compiler) }.last
127     end
128   end
129
130   class NumberNode
131     def llvm_compile(compiler)
132       compiler.new_number(value)
133     end
134   end
135
136   class StringNode
137     def llvm_compile(compiler)
138       compiler.new_string(value)
139     end
140   end
```

To keep things simple, our compiler only supports a subset of our Awesome language. For example, we only support calling methods on `self` and not on

given receivers. We also don't support method parameters when defining methods. This is why we raise an error in the following nodes. See at the end of this chapter for more details on what is not supported and where to go from here.

```
148 class CallNode
149   def llvm_compile(compiler)
150     raise "Receiver not supported for compilation" if receiver
151
152     compiled_arguments = arguments.map { |arg| arg.llvm_compile(compiler) }
153     compiler.call(method, compiled_arguments)
154   end
155 end
156
157 class GetLocalNode
158   def llvm_compile(compiler)
159     compiler.load(name)
160   end
161 end
162
163 class SetLocalNode
164   def llvm_compile(compiler)
165     compiler.assign(name, value.llvm_compile(compiler))
166   end
167 end
168
169 class DefNode
170   def llvm_compile(compiler)
171     raise "Parameters not supported for compilation" if !params.empty?
172     compiler.function(name) do |function|
173       body.llvm_compile(function)
174     end
175   end
176 end
```

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

In file *code/test/llvm_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 = LLVMCompiler.new
compiler.preamble
node.llvm_compile(compiler)
compiler.finish

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

# Optimize the LLVM byte-code
compiler.optimize

# JIT compile & execute
compiler.run
```

Remember that the compiler only supports a subset of our Awesome language. For example, object-oriented programming is not supported. To implement 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.

MIO, A MINIMALIST HOMOICONIC LANGUAGE

In the hope of expanding your view on programming languages and perhaps inspire you, we'll now take a look at another language that is drastically different from the one we've just built and from most mainstream languages.

Note that this chapter is a little more advanced and requires a deeper understanding of Ruby. You might need to read it and play with the code a few times before fully understanding how this language works. You can safely skip this chapter if you're only interested in creating a "standard" language like Awesome.

However if you'd like to enter the fascinating world of homoiconicity, prepare your brain to be melted!

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 (`proto`), their parent object.

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.

In file *code/mio/object.rb*

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

    def initialize(proto=nil, value=nil)
      @proto = proto # Prototype: parent object. Like JavaScript's __proto__.
      @value = value # The Ruby equivalent value.
      @slots = {} # Slots are where we store methods and attributes of an object.
    end

    # Lookup a slot in the current object and proto.
    def [](name)
      return @slots[name] if @slots.key?(name)
      return @proto[name] if @proto # Check if parent prototypes
```

```
      raise Mio::Error, "Missing slot: #{name.inspect}"
    end

    # Store a value in a slot
    def []=(name, value)
      @slots[name] = value
    end

    # Store a method into a slot
    def def(name, &block)
      @slots[name] = block
    end

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

    # The only way to create a new object in Mio is to clone an existing one.
    def clone(ruby_value=nil)
      Object.new(self, ruby_value)
    end
  end
end
```

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

```
"hello" print
1 +(2) print
```

is parsed as the following tree of messages:

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

```
        Message.new("+", [
                          Message.new("2")
                         ],
          Message.new("print"))))))
```

This is our AST much like the tree of nodes our Awesome parser produced. Except in this case we translate one-to-one tokens to messages. Notice line breaks (and dots) are also messages. When executed, they simply reset the receiver of the message.

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

This results in the same behavior 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!

In file *code/mio/message.rb*

```
module Mio
  # Message is a tree 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)
```

```
12        @name = name
13        @args = []
14        @line = line
15
16        # Literals are static values, we can eval them right
17        # away and cache the value.
18        @cached_value = case @name
19        when /^\d+/
20          Lobby["Number"].clone(@name.to_i)
21        when /^"(.*)"$/
22          Lobby["String"].clone($1)
23        end
24
25        @terminator = [".", "\n"].include?(@name)
26
27        super(Lobby["Message"])
28      end
29
30      # Call (eval) the message on the +receiver+.
31      def call(receiver, context=receiver, *args)
32        if @terminator
33          # reset receiver to object at beginning of the chain.
34          # eg.:
35          #   hello there. yo
36          #  ^           ^__ "." resets back to the receiver here
37          #  \_____________________________________________/
38          value = context
39        elsif @cached_value
40          # We already got the value
41          value = @cached_value
42        else
43          # Lookup the slot on the receiver
44          slot = receiver[name]
45
46          # Eval the object in the slot
47          value = slot.call(receiver, context, *@args)
48        end
49
50        # Pass to next message if some
51        if @next
52          @next.call(value, context)
53        else
```

```
54          value
55        end
56      rescue Mio::Error => e
57        # Keep track of the message that caused the error to output
58        # line number and such.
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      alias inspect to_s
71
72      # Parse a string into a tree of messages
73      def self.parse(code)
74        parse_all(code, 1).last
75      end
76
77      private
78        def self.parse_all(code, line)
79          code = code.strip
80          i = 0
81          message = nil
82          messages = []
83
84          # Parsing code. Very similar to the Lexer we built for Awesome.
85          while i < code.size
86            case code[i..-1]
87            when /\A("[^"]*")/, # string
88                 /\A(\d+)/,     # number
89                 /\A(\.)+/,     # dot
90                 /\A(\n)+/,     # line break
91                 /\A(\w+)/      # name
92              m = Message.new($1, line)
93              if messages.empty?
94                messages << m
95              else
```

```
96                message.next = m
97              end
98              line += $1.count("\n")
99              message = m
100             i += $1.size - 1
101           when /\A(\(\s*)/ # arguments
102             start = i + $1.size
103             level = 1
104             while level > 0 && i < code.size
105               i += 1
106               level += 1 if code[i] == ?\(
107               level -= 1 if code[i] == ?\)
108             end
109             line += $1.count("\n")
110             code_chunk = code[start..i-1]
111             message.args = parse_all(code_chunk, line)
112             line += code_chunk.count("\n")
113           when /\A,(\s*)/
114             line += $1.count("\n")
115             messages.concat parse_all(code[i+1..-1], line)
116             break
117           when /\A(\s+)/, # ignore whitespace
118                /\A(#.*$)/ # ignore comments
119             line += $1.count("\n")
120             i += $1.size - 1
121           else
122             raise "Unknown char #{code[i].inspect} at line #{line}"
123           end
124           i += 1
125         end
126         messages
127       end
128   end
129 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)`. Lazy evaluation also enables implementing macros in our language.

In file code/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.)

In file *code/mio/bootstrap.rb*

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

  object.def "clone" do |receiver, context|
    receiver.clone
  end
  object.def "set_slot" do |receiver, context, name, value|
    receiver[name.call(context).value] = value.call(context)
  end
  object.def "print" 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.
```

```
32    Lobby.def "method" do |receiver, context, message|
33      Method.new(context, message)
34    end
35  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.

In file *code/test/mio/oop.mio*

```
1  # Create a new object, by cloning the master Object
2  set_slot("dude", Object clone)
3  # Set a slot on it
4  dude set_slot("name", "Bob")
5  # Call the slot to retrieve its value
6  dude name print
7  # => Bob
8
9  # Define a method
10 dude set_slot("say_name", method(
11   # Print unevaluated arguments (messages)
12   arguments print
13   # => <Message @name="hello..."">
14
15   # Eval the first argument
16   eval_arg(0) print
17   # => hello...
18
19   # Access the receiver via `self`
20   self name print
21   # => Bob
22 ))
23
```

```
# Call that method
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.

In file *code/mio/boolean.mio*

```
# An object is always truthy

Object set_slot("and", method(
  eval_arg(0)
))
Object set_slot("or", method(
  self
))

# ... except nil and false which are false

nil set_slot("and", nil)
nil set_slot("or", method(
  eval_arg(0)
))

false set_slot("and", false)
false set_slot("or", method(
  eval_arg(0)
))
```

In file *code/test/mio/boolean.mio*

```
"yo" or("hi") print
# => yo

nil or("hi") print
# => hi

"yo" and("hi") print
# => hi
```

```
9
10 1 and(2 or(3)) print
11 # => 2
```

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

In file *code/mio/if.mio*

```
1  # Implement if using boolean logic
2
3  set_slot("if", method(
4    # eval condition
5    set_slot("condition", eval_arg(0))
6    condition and( # if true
7      eval_arg(1)
8    )
9    condition or( # if false (else)
10     eval_arg(2)
11   )
12 ))
```

And now… holy magical pony!

In file *code/test/mio/if.mio*

```
1  if(true,
2    "condition is true" print,
3    # else
4    "nope" print
5  )
6  # => condition is true
7
8  if(false,
9    "nope" print,
10   # else
11   "condition is false" print
12 )
13 # => 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)` which would then be translated into `set_slot("x", 1)`. This introduces syntactic sugar into our language without impacting its homoiconicity and awesomeness.

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`.

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 ideas to explore if you're interested in learning more.

HOMOICONICITY

That's the word you want to ostentatiously use 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 previous chapter 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 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.

Also, compiling to another language like JavaScript is a great option.

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.

CLASSES

I teach an online class based on this book called The Programming Language Masterclass. We spend the class creating a language similar to Awesome and then build a real VM in C, and even end up creating a JIT compiler from scratch. Use the discount *AWESOME* when booking to get $100 off.

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.

LuaJIT is a great example of state-of-the-art dynamic language machine code compiler for approaching native performance without sacrificing dynamism.

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, Esoteric programming languages.

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.

Each programming language is a unique piece of art that allows us humans to interact with machines. There are endless possibilities when designing a language, it can be optimized for performance, happiness, keystrokes, laughs. The only limit is your imagination.

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*

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 = [..., "while"]
```

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

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

In file *code/bracket_lexer.rb*

```
1  class BracketLexer
2    KEYWORDS = ["def", "class", "if", "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]
```

```
        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

      ######
      # All indentation magic code was removed and only this elsif was added.
      elsif chunk.match(/\A\n+/)
        tokens << [:NEWLINE, "\n"]
        i += 1
      ######

      elsif chunk.match(/\A /)
        i += 1

      else
        value = chunk[0,1]
        tokens << [value, value]
        i += 1

      end

    end

    tokens
  end
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

# ...

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

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

```
Constants["Number"].def :+ do |receiver, arguments|
  result = receiver.ruby_value + arguments.first.ruby_value
  Constants["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
    Constants["nil"]
  end
end
```