How To Create Your Own Freaking Awesome Programming Language

TABLE OF CONTENT

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

Revision #5, Published June 2013.

Cover background image © Asja Boros

Content of this book is © Marc-André Cournoyer. All right reserved. This eBook copy is for a single user. You may not share it in any way unless you have written permission of the author.

This is a sample chapter.
Buy the full book online at
http://createyourproglang.com

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.

This is a sample chapter.
Buy the full book online at
http://createyourproglang.com