How To Create Your Own Freaking Awesome Programming Language

By Marc-André Cournoyer
Published August 2009

Thanks to Jean-Pierre Martineau, Julien Desrosiers and
Thanh Vinh Tang for reviewing early drafts of this book.

TABLE OF CONTENTS

LEXER

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

Let's say you have the following code:

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

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

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

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

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

LEX (FLEX)

Flex is a modern version of Lex (that was coded by Eric Schmidt, CEO of Google, by the way) for generating C lexers. Along with Yacc, Lex is the most commonly used lexer for parsing and it has been ported to many target languages.

It has been ported to several target languages.

- Rex for Ruby (http://github.com/tenderlove/rexical/)
- JFlex for Java (http://jflex.de/)

More details in the Flex manual (http://flex.sourceforge.net/manual/)

RAGEL

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

More details in the Ragel manual (http://www.complang.org/ragel/ragel-guide-6.5.pdf).

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

- Min's lexer in Java (http://github.com/macournoyer/min/blob/master/src/min/lang/Scanner.rl)
- Potion's lexer in C (http://github.com/whymirror/potion/blob/fae2907ce1f4136-da006029474e1cf761776e99b/core/pn-scan.rl)

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 as an operator precedence table, often based on mathematics order of operations. Several ways to handle this exists. Yacc-based parser implement the Shunting Yard algorithm (http://en.wikipedia.org/wiki/Shunting_yard_algorithm) in which you give a precedence level to each kind of operator. Operators are declared with `%left` and `%right`, more details in Bison's manual (http://dinosaur.compilertools.net/bison/bison_6.html#SEC51).

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 other will produce the desired result:

```
expression:                 equality-expression
equality-expression:        additive-expression ( ( '==' | '!=' )
                              additive-expression )*
additive-expression:        multiplicative-expression ( ( '+' | '-' )
                              multiplicative-expression )*
multiplicative-expression:  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-expression` is the last rule called in the parsing process, it will have greater precedence.

PYTHON STYLE INDENTATION FOR AWESOME

If you intend to build a fully-functioning language, you should use one of the two previous tools. Since Awesome is a simplistic language and we just 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, just like 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:

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

will yield these tokens:

```
1  [IDENTIFIER if] [IDENTIFIER tasty] [EQUAL] [IDENTIFIER True]
2  [INDENT] [IDENTIFIER print] [STRING "Delicious!"]
3  [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 on the next page.

lexer.rb

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

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

    # Current character position we're parsing
    i = 0

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

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

    # This is how to implement a very simple scanner.
    # Scan one caracter at the time until you find something to parse.
    while i < code.size
      chunk = code[i..-1]

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

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

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

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

      # Here's the indentation magic!
```

```
54        #
55        # We have to take care of 3 cases:
56        #
57        #   if true:  # 1) the block is created
58        #     line 1
59        #     line 2  # 2) new line inside a block
60        #   continue  # 3) dedent
61        #
62        # This elsif takes care of the first case. The number of spaces will determine
63        # the indent level.
64        elsif indent = chunk[/\A\:\n( +)/m, 1] # Matches ": <newline> <spaces>"
65          # When we create a new block we expect the indent level to go up.
66          if indent.size <= current_indent
67            raise "Bad indent level, got #{indent.size} indents, " +
68                  "expected > #{current_indent}"
69          end
70          # Adjust the current indentation level.
71          current_indent = indent.size
72          indent_stack.push(current_indent)
73          tokens << [:INDENT, indent.size]
74          i += indent.size + 2
75
76        # This elsif takes care of the two last cases:
77        # Case 2: We stay in the same block if the indent level (number of spaces) is the
78        #         same as current_indent.
79        # Case 3: Close the current block, if indent level is lower than current_indent.
80        elsif indent = chunk[/\A\n( *)/m, 1] # Matches "<newline> <spaces>"
81          if indent.size == current_indent # Case 2
82            # Nothing to do, we're still in the same block
83            tokens << [:NEWLINE, "\n"]
84          elsif indent.size < current_indent # Case 3
85            indent_stack.pop
86            current_indent = indent_stack.first || 0
87            tokens << [:DEDENT, indent.size]
88            tokens << [:NEWLINE, "\n"]
89          else # indent.size > current_indent, error!
90            # Cannot increase indent level without using ":", so this is an error.
91            raise "Missing ':'"
92          end
93          i += indent.size + 1
94
95        # Ignore whitespace
96        elsif chunk.match(/\A /)
97          i += 1
98
99        # We treat all other single characters as a token. Eg.: ( ) , . !
100       else
101         value = chunk[0,1]
102         tokens << [value, value]
103         i += 1
104
105       end
106
```

```
107       end
108
109       # Close all open blocks
110       while indent = indent_stack.pop
111         tokens << [:DEDENT, indent_stack.first || 0]
112       end
113
114       tokens
115     end
116   end
```

You can test the lexer yourself by extracting the `code.zip` file included with the book. Run ruby `lexer_test.rb` and it will output the tokenized version of the code.

lexer_test.rb

```
1   require "lexer"
2
3   code = <<-EOS
4   if 1:
5     print "..."
6     if false:
7       pass
8     print "done!"
9   print "The End"
10  EOS
11
12  p Lexer.new.tokenize(code)
13
14  # Output:
15  # [[:IF, "if"], [:NUMBER, 1],
16  #  [:INDENT, 2], [:IDENTIFIER, "print"], [:STRING, "..."], [:NEWLINE, "\n"],
17  #                [:IF, "if"], [:IDENTIFIER, "false"],
18  #                [:INDENT, 4], [:IDENTIFIER, "pass"],
19  #                [:DEDENT, 2], [:NEWLINE, "\n"],
20  #                [:IDENTIFIER, "print"], [:STRING, "done!"],
21  #  [:DEDENT, 0], [:NEWLINE, "\n"],
22  #  [:IDENTIFIER, "print"], [:STRING, "The End"]]
```

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

A. Modify the lexer to parse: while condition: ... control structures.

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