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

**Introduction**
- Summary  
- About The Author  
- Before We Begin

**Overview**
- The Four Parts of a Language  
- Meet Awesome: Our Toy Language

**Lexer**
- Lex (Flex)  
- Ragel  
- Operator Precedence  
- Python Style Indentation For Awesome  
- Do It Yourself

**Parser**
- Bison (Yacc)  
- Lemon  
- ANTLR  
- PEG  
- Connecting The Lexer and Parser in Awesome  
- Do It Yourself

**Interpreter**
- Do It Yourself

**Runtime Model**
- Procedural  
- Class-based  
- Prototype-based  
- Functional  
- Our Awesome Runtime  
- Do It Yourself

**Compilation**
- Using LLVM from Ruby  
- Compiling Awesome to Machine Code

**Virtual Machine**
- Byte-code  
- Types of VM  
- Prototyping a VM in Ruby

**Going Further**
- Homoiconicity  
- Self-Hosting  
- What’s Missing?

**Resources**
- Events  
- Forums and Blogs  
- Interesting Languages

**Solutions to Do It Yourself**
This is a sample chapter.
Buy the full book online at
http://createyourproglang.com
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.

- JFlex for Java ([http://jflex.de/](http://jflex.de/))

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.


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:

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

will yield these tokens:

```plaintext
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.
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

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

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

      # Here's the indentation magic!
# We have to take care of 3 cases:
#
#   if true:  # 1) the block is created
#     line 1
#     line 2  # 2) new line inside a block
#   continue  # 3) dedent
#
# This elsif takes care of the first case. The number of spaces will determine
# the indent level.
elsif indent = chunk[\A\:\n( +)/m, 1] # Matches ": <newline> <spaces>"
    # When we create a new block we expect the indent level to go up.
    if indent.size <= current_indent
      raise "Bad indent level, got #{indent.size} indents, " +
           "expected > #{current_indent}"
    end
    # Adjust the current indentation level.
    current_indent = indent.size
    indent_stack.push(current_indent)
    tokens << [:INDENT, indent.size]
    i += indent.size + 2

# This 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.
eelsif indent = chunk[\A\n( *)/m, 1] # Matches "<newline> <spaces>"
    if indent.size == current_indent # Case 2
      # Nothing to do, we're still in the same block
      tokens << [:NEWLINE, "\n"
    elsif indent.size < current_indent # Case 3
      indent_stack.pop
      current_indent = indent_stack.first || 0
      tokens << [:DEDENT, indent.size]
      tokens << [:NEWLINE, "\n"
    else # indent.size > current_indent, error!
      # Cannot increase indent level without using ":", so this is an error.
      raise "Missing ':'"
    end
    i += indent.size + 1

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

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

```
require "lexer"

code = <<-EOS
  if 1:
    print "...
    if false:
      pass
    print "done!"
  print "The End"
EOS

p Lexer.new.tokenize(code)

# Output:
# [:IF, "if"], [:NUMBER, 1],
# [:INDENT, 2], [:IDENTIFIER, "print"], [:STRING, "..."], [:NEWLINE, "\n"],
# [:IF, "if"], [:IDENTIFIER, "false"],
# [:INDENT, 4], [:IDENTIFIER, "pass"],
# [:DEDENT, 2], [:NEWLINE, "\n"],
# [:IDENTIFIER, "print"], [:STRING, "done!"],
# [:DEDENT, 0], [:NEWLINE, "\n"],
# [: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.