

**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. [Do It Yourself IV](#)
8. [Compilation](#)
 1. [Using LLVM from Ruby](#)
 2. [Compiling Awesome to Machine Code](#)
9. [Virtual Machine](#)
 1. [Byte-code](#)
 2. [Types of VM](#)
 3. [Prototyping a VM in Ruby](#)
10. [Going Further](#)
 1. [Homoiconicity](#)
 2. [Self-Hosting](#)
 3. [What's Missing?](#)
11. [Resources](#)
 1. [Books & Papers](#)
 2. [Events](#)
 3. [Forums and Blogs](#)
 4. [Interesting Languages](#)
12. [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](#)
13. [Appendix: 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](#)
14. [Farewell!](#)

Published November 2011.

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>

PARSER

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

Lets take those tokens from previous section:

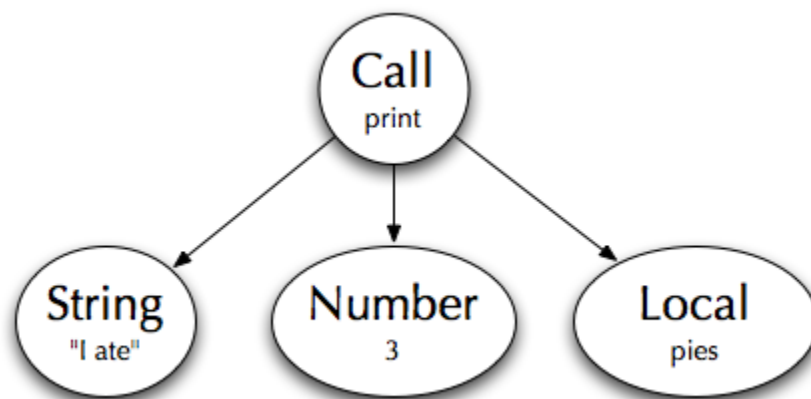
```
1 [IDENTIFIER print] [STRING "I ate"] [COMMA]
2     [NUMBER 3] [COMMA]
3     [IDENTIFIER pies]
```

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:

```
1 [<Call name=print,
2     arguments=[<String value="I ate">,
3         <Number value=3>,
4         <Local name=pies>]
5 >]
```

Or as a visual tree:

Figure 2



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

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

BISON (YACC)

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

- [Racc for Ruby](#)
- [Ply for Python](#)
- [JavaCC for Java](#)

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

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

On the left is defined how the rule can be matched using tokens and other rules.

On the right side, between brackets is the action to execute when the rule matches.

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

LEMON

[Lemon](#) is quite similar to Yacc, with a few differences. From its website:

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

For more information, refer to the [the manual](#) or check real examples inside [Potion](#).

ANTLR

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

PEGS

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

[Treetop](#) is an interesting Ruby tool for creating PEG.

OPERATOR PRECEDENCE

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

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

Here's the operator precedence table for our language, based on the [C language operator precedence](#):

```
1 left  '.'
2 right '!'
3 left  '*' '/'
4 left  '+' '-'
5 left  '>' '>=' '<' '<='
6 left  '==' '!='
7 left  '&&'
8 left  '||'
9 right '='
10 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:

```
1 expression:      equality
2 equality:        additive ( ( '=' | '!=' ) additive )*
3 additive:       multiplicative ( ( '+' | '-' ) multiplicative )*
4 multiplicative: primary ( ( '*' | '/' ) primary )*
5 primary:        '(' expression ')' | NUMBER | VARIABLE | '-' primary
```

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

CONNECTING THE LEXER AND PARSER IN AWESOME

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

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

```
1 class Parser
2
3 # Declare tokens produced by the lexer
4 token IF ELSE
5 token DEF
6 token CLASS
7 token NEWLINE
8 token NUMBER
9 token STRING
10 token TRUE FALSE NIL
11 token IDENTIFIER
12 token CONSTANT
13 token INDENT DEDENT
14
```

grammar.y

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