

Cooking flex with Perl

Alberto Manuel Simões, albie@alfarrabio.di.uminho.pt

Abstract

Perl has a lot of tools for parser generation using Perl. Perl has flexible data structures which makes it easy to generate generic trees. While it is easy to write a grammar and a lexical analyzer using modules like `Parse::Yapp` and `Parse::Lex`, these tools are not as fast as I would like. I use the `Parse::Yapp` syntax parser with the flex lexical analyzer to solve this.

1 Introduction

Some time ago, I needed to make a dictionary programming language parser faster. The original programmer had written the parser with a patched Berkeley Yacc (`byacc`), which can generate Perl from a grammar specification like `common yacc`. The only difference resides in the semantic actions, written with Perl. He wrote the lexical analyzer with simple Perl regular expressions. This combination works, but loading the complete dictionary took about 27 seconds, which was too long for me. I combined Perl with flex to cut the parsing time in half.

2 Parser structure

When I write a program in any language and a compiler interprets source code, a parser is involved. It takes the code I write, splits it out as strings of characters with a special meaning—tokens—and then tries to match these token sequences with a grammar. The process has two parts—lexing and parsing.

A lexical analyzer, or lexer, splits a program into pieces called tokens. Regular expressions match each token, which the lexer returns one by one to the caller program. Commonly, it returns the token type with the string which corresponds to the matched token.

Another program, a syntactic analyzer, or parser, puts these pieces together to check if their sequence makes any sense. The parser uses a grammar, or a set of rules, to construct a sentence and check if all the necessary tokens are used, and that no more are needed. It builds a tree of the program structure to generate or interpret code.

Traditional tools for generating parsers in C are `lex` and `yacc`, or their GNU variants, `flex` and `bison`. When using Perl, people implement parsers in many ways.

3 Why flex

On my first approach I converted the grammar to a full Perl parser. I could have chosen tools like `Parse::RecDescent`, `Parse::YALALR`, or others, but I chose `Parse::Yapp` because its syntax and functionality is very similar to the old `yacc`. In a half an hour I had a new, working version of the parser, but had only reduced the time for the parsing task by one or two seconds. This small speed-up occurs because `Parse::Yapp` creates full Perl programs, taking advantage of Perl facilities.

Next, since I could not make the syntax analyzer quicker, I tried to change the lexical one. I wanted to use `Parse::Lex`, but since it used Perl regular expressions I decided to find another option. I really like Perl, but started thinking about a C implementation for the Parser. That would take a long time but I heard about the `XSUB` concept and started working on a flex specification to glue with `Parse::Yapp`.

I could have implemented this glue using different techniques. I could write the lexical analyzer returning integers, where each one represents a different grammar terminal symbol, or I could return a string with the name of the symbol. While the second method can be slower than returning integers, the grammar is more legible, so I went with it.

I implemented my flex analyzer and glued it with `Parse::Yapp`. It ran in about half the original parsing time. Perl is very flexible, and flex is very fast. I can create simple and fast parsers using both of them together.

4 The Recipe

To demonstrate what I did I use something a lot more simple than my original problem, although I illustrate all of the major points in the process.

I want to create a parser for simple arithmetic problems like “1 + 2”. The lexer will break the string into tokens (“1”, “+”, “2”), perform the arithmetic operation, and return the result.

4.1 Writing the Grammar

I need to write two things—the grammar and the lexical analyzer. I like to start with the grammar, although that is personal preference.

The `Parse::Yapp` syntax is like `yacc`. I can write a grammar for arithmetic expressions, which I put in a file I name `myGrammar.y` shown in code listing 1.

```
----- Code Listing 1: myGrammar.y -----  
1 %token NUMBER NL  
2  
3 %left '*' '/'  
4 %left '-' '+'  
5  
6 %%  
7 command: exp NL { return $_[1] }  
8           ;  
9  
10 exp: exp '+' exp { return $_[1]+$[3] }
```

```
11 | exp '-' exp { return $_[1]-$_[3]}
12 | exp '*' exp { return $_[1]*$_[3]}
13 | exp '/' exp { return $_[1]/$_[3]} #forget x/0
14 | number      { return $_[1]}
15 | ;
16
17 number: NUMBER { return $_[1] }
18 | ;
19 %%
```

4.2 Writing the Lexical Analyser

Next, I write the lexical analyser in C. I create the file named myLex.l in which I put the instructions for lexing the source, shown in code listing 2.

Code Listing 2: myLex.l

```
1  %{
2  #define YY_DECL char* yylex() void;
3
4  char buffer[15];
5
6  %}
7
8  %%
9  [0-9]+ { return strcpy(buffer, "NUMBER"); }
10
11 \n      { return strcpy(buffer, "NL"); }
12
13 .      { return strcpy(buffer, yytext); }
14
15 %%
16 int perl_yywrap(void) {
17     return 1;
18 }
19 char* perl_yylextext(void) {
20     return perl_yytext;
21 }
```

The first three lines define the prototype for the lexical analyzer. I return strings instead of the integers returned by default. I have to put these strings somewhere. In this example, I allocate a char array named buffer where I will put the token information.

The next section is a normal lexical analyzer, returning the name of the token or the character found.

The perl_yywrap and perl_yylextext glue the parser to the lexical analyzer. The perl_yywrap function restarts the parsing task while perl_yylextext accesses the text which matched the regular expression through perl_yytext, which flex creates for me.

4.3 Writing the glue

Now I have the two main tools and I am only missing the glue. The easiest way I can do this is creating a module for my parser. I start with `h2xs` which creates most of the module structure and files for me.

```
h2xs -n myParser
```

I have to edit some of the files that `h2xs` creates, and add the files that I just created. I need to add some XSUB code to `myParser.xs` to the lexer in `myLex.l` to Perl, add a line to the `typemaps` file, and adjust the `Makefile.PL` to correctly compile everything.

I copy the flex source to `myParser/myLex.l` and the `Parse::Yapp` grammar to `myParser/myGrammar.y` to the module directory.

`Parse::Yapp` expects a `yylex` function that returns a pair—the name of the token and the matched text—so I add a `perl_lex` function to `myParser.pm`, shown in code listing 3. I wrote `perl_yylextext` in `myLex.l` (code listing 2), and flex generates `perl_yylex` for me.

```
_____ Code Listing 3: The perl_lex subroutine in myParser.pm _____  
1 sub perl_lex {  
2     my $token = perl_yylex();  
3     if ($token) {  
4         return ($token, perl_yylextext())  
5     } else {  
6         return (undef, "");  
7     }  
8 }
```

I also need an error-handling function in `myParser.pm`. My error function in code listing 4 will simply print the token read and the token it expected if it encounters an error.

```
_____ Code Listing 4: An error message for unrecognized tokens _____  
1 sub perl_error {  
2     my $self = shift;  
3     warn "Error: found ", $self->YYCurtok,  
4         " and expecting one of ", join(" or ", $self->YYExpect);  
5 }
```

Once I have that done, I edit `myParser.xs` file to map the C functions into Perl ones. I leave the code that `h2xs` generated alone, and add another header file with the others.

```
#include "myLex.h"
```

At the end of the file I add the parts that connect my lex functions with the functions that I call from Perl, shown in code listing 5. The spaces and newlines are significant. The first line of each function is the return type, followed by a line with the name of the function. Then I have two lines showing Perl where to get the return value from the C functions.

Code Listing 5: Function glue in myParser.xs

```

1 char*
2 perl_yylex()
3     OUTPUT:
4         RETVAL
5
6 char*
7 perl_yylextext()
8     OUTPUT:
9         RETVAL

```

I create the myLex.h file, shown in code listing 6, which holds the prototypes for the functions in myLex.l.

Code Listing 6: Lexer function prototypes in myLex.h

```

1 char* perl_yylex(void);
2 char* perl_yylextext(void);

```

I have to map data types from C to Perl. Integers are trivial and handled directly by perl, but I also have pointers to characters. I create a file named typemap, shown in code listing 7 that translates pointers to characters to the built-in T_PV Perl type.

Code Listing 7: The typemap file

```

1 char* T_PV

```

4.4 Putting it all together

The code is now complete, but I need to modify Makefile.PL so it can compile it. The yapp command from the Parser::Yapp distribution creates myGrammar.pm from myGrammar.y, and flex creates lex.perl.yy.c. I add an additional target to the Makefile through the MY::postamble subroutine. I also add the flex library, fl, to the library list, and name the produced library myLexer.so. Code listing 8 shows my final Makefile.PL.

Code Listing 8: My modified Makefile.PL

```

1 use ExtUtils::MakeMaker;
2
3 $YACC_COMMAND = "( yapp -o myGrammar.pm -m 'myGrammar' myGrammar.y)";
4
5 # This is needed before WriteMakefile
6 '$YACC_COMMAND';
7
8 WriteMakefile(
9     'NAME'           => 'myParser',
10    'VERSION_FROM'   => 'myParser.pm',
11    'LIBS'           => ['-lfl'], # This is for flex
12    'MYEXTLIB'       => 'myLexer.so', # Our lexer
13    );
14

```

```
15 sub MY::postamble {
16   "
17   \$(MYEXTLIB): lex.perl.yy.c myParser.pm myGrammar.pm
18   \t\$(CC) -c lex.perl.yy.c
19   \t\$(AR) cr \$(MYEXTLIB) lex.perl.yy.o
20   \tranlib \$(MYEXTLIB)
21
22   myGrammar.pm: myGrammar.y
23   \t\$(YACC_COMMAND)
24
25   lex.perl.yy.c: myLex.l
26   \tflex -Pperl.yy myLex.l
27   ";
28 }
```

I compile my new module with the standard Perl module make sequence.

```
perl Makefile.PL
make
```

4.5 Testing

I can also test my module with the standard Perl test harness, although I have not added any real tests to test.pl.

```
make test
```

This does not really test if my module really works—only that it compiles and loads without error. I add a test, shown in code listing 9, which takes a string from standard input, parses it, and returns the result of the arithmetic operation.

```
_____ Code Listing 9: My Grammar test _____
1 use myGrammar;
2 my $parser = new myGrammar;
3 my $result = $parser->YYParse(yylex => \&myParser::perl_lex,
4                                     yyerror => \&myParser::perl_error);
5 print "The result is $result\n";
```

Now, I run `make test` again and enter “1+4*2+3”, press enter, and end standard input with `^D` (or `^Z` on Windows), and I get the answer 25.

5 Conclusion

While Perl is really flexible, some tools are quicker than their Perl equivalents. I can use these tools from Perl with a little bit of work.

6 References

Manual pages:

`flex(1)`, `perlguts(1)`, `perlxs(1)`, `perlxstut(1)`

Perl Module documentation:

`Parse::Yapp`, `ExtUtils::MakeMaker`