Cooking flex with Perl

Alberto Manuel Simões
(albie@alfarrabio.di.uminho.pt)

Abstract

There are a lot of tools for parser generation using Perl. As we know, 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`, this pair of tools is not as efficient as I would like. In this document I'll present a way to cook quickly `Parse::Yapp` with the better lexical analyzer I know: `flex`.

1 Parser structure

Before showing my results, I should make a brief description of what's the standard built method for a parser, and how it works. As you should use Perl or any other programming language, I can assume you know or at least have used a parser. When I write a program in any language and it's compiler generates code or interpret it, this process has a parser involved. It takes the code you write, splits it on tokens (strings of characters with a special meaning in the language) and then tries to match these token sequence with a grammar.

We call a lexical analyzer to a program which splits the program into pieces. We use regular expressions to match each token, which will be returned, one by one, to the caller program. Commonly, we return not only the token type (a name) but also the string which correspond to the matched token.

The other step, putting these pieces together and check if their sequence makes any logic, is done by another program, called a syntactic analyzer. It uses a grammar (a sequence of rules) to construct a sentence and check if all tokens are used, and no more are needed. While checking this, it is built a tree in memory with the program structure. To generate code, or interpret it, the program will use make traversals over this tree.

Traditional tools for generating parsers in `C` are `lex` and `yacc` (or their variants, `flex` and `bison`). When using Perl, I can say there are no traditional tools. Indeed, there are a lot of different ways people implement parsers.

2 Why flex

Some time ago, I needed make a dictionary programming language parser more efficient. The original parser was written with a patched Berkeley Yacc (`byacc`) which can generate Perl from a grammar specification like common `yacc`. The only difference resides on the semantic actions, written with Perl. The lexical analyzer was written with simple Perl regular expressions. This combination works, but loading the complete dictionary took a lot of time.

The first approach was to convert the grammar to a full Perl parser. I could choose tools like `Parse::RecDescent`, `Parse::YALALR`, `Parse::Yapp` or others. I choose `Parse::Yapp` not because of the efficiency but because its syntax and functionality is very similar to the old `yacc`. An half an hour and I had a new working version of the parser, but only less one or two seconds of time for the parsing task. This small speed-up occurs because `byacc` was generating a Perl program with `C` structure. `Parse::Yapp` creates full Perl programs, taking advantage of Perl facilities.

Next, if I can't make the syntax analyzer quicker, I can try to change the lexical one. The first idea was to use `Parse::Lex` but since it used Perl regular expressions I decided to find another option. I really like Perl, but started thinking on a C implementation for the Parser. That would take a long time but I heard about the XSUB concept and start working on a flex specification to glue with Parse::Yapp.

This glue can be implemented using different techniques. For example, I could write the lexical analyzer returning integers, where each one represents a different grammar terminal symbol. Another approach was returning a string with the name of the symbol. While this can be less efficient than the first one, it makes the grammar more legible. This is the reason why I choose it.

I implemented this flex analyzer and glued it with Parse::Yapp. It ran in less than 50% of parsing time.

byacc + perl re	yapp + flex
27.625 sec	*14.161 sec*

flex is very efficient, but Perl is very flexible. Putting together these two tools makes parse generation simple, quick and efficient.

3 The Recipe

Because the dictionary programming language is very complex, I decided to give a simple arithmetic parser example to make it simple. This recipe assumes you know the basics about parser generation.

3.1 Writing the Grammar

First, like in other languages, we need to write the grammar and the lexical analyzer. Which to write first, depend on your way of working but, normally we start with the grammar to find the terminal symbols.

The Parse::Yapp[1] syntax is like yacc:

```
%{
   # Perl initialization code
%}
# Yapp commands
%%
# The Grammar with perl semantic actions
%%
# Perl functions
```

Let's write a grammar for arithmetic expressions. Isn't that exciting?

I hope this is a straightforward grammar, with the only objective of showing the Parse::Yapp syntax. Edit a myGrammar.yp file and write:

```
%token NUMBER NL

%left '-' '+'
%left '*' '/'

%%
command: exp NL  { return $_[1] }
       ;

exp: exp '+' exp { return $_[1]+$_[3]}
```

[1]As Parse::Yapp module does not come installed with Perl, you have to get it from CPAN.

```
    | exp '-' exp { return $_[1]-$_[3]}
    | exp '*' exp { return $_[1]*$_[3]}
    | exp '/' exp { return $_[1]/$_[3]} #forget x/0
    | number      { return $_[1]}
    ;

number: NUMBER   { return $_[1] }
      ;
%%
```

3.2 Writing the Lexical Analyser

Next, let's write the lexical analyser in C. I hope you remember this assembler language `:-)` Create the file `myLex.l` with:

```
%{
 #define YY_DECL char* yylex() void;
%}

 char buffer[15];

%%
[0-9]+   { return strcpy(buffer, "NUMBER"); }

\n       { return strcpy(buffer, "NL"); }

.        { return strcpy(buffer, yytext); }

%%
int perl_yywrap(void) {
  return 1;
}
char* perl_yylextext(void) {
    return perl_yytext;
}
```

The first three lines define the prototype for the lexical analyzer. As said before, I want to return strings instead of the normal integers returned by default. Then, I have to put these strings somewhere. One option could be to allocate memory each time a token is found. In this example, I allocate one buffer where I will put the token information. You must remember that using a hardcoded 15 character buffer implies that you use smaller strings to terminal symbols.

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

The last pair or functions are used to glue the parser with the lexical analyzer. The first one is used only to restart the parsing task while the second is used to access the text which matched the regular expression.

3.3 Writting the glue

At this point I have the two main tools written. It misses, only, the glue. The easiest way to do this is creating a module for our parser with the `h2xs` tool:

```
h2xs -n myParser
```

Then, copy the flex source and the `Parse::Yapp` grammar to the module directory (`myParser/myLex.l` and `myParser/myGrammar.yp`).

Parse::Yapp expects a yylex function to return a pair, the name of the token and the matched text. To accomplish this, add the following function to your module (myParser.pm), just before the line containing "1;".

```
sub perl_lex {
  my $token = perl_yylex();
  if ($token) {
    return ($token, perl_yylextext())
  } else {
    return (undef, "");
  }
}
```

If you remember, I wrote perl_yylextext, but perl_yylex is generated by flex.

I will need an error handling function, too! The error function will print what token was read and the token it was expecting. This function should be added after the perl_lex function.

```
sub perl_error {
  my $self = shift;
  warn "Error: found ",$self->YYCurtok,
    " and expecting one of ",join(" or ",$self->YYExpect);
}
```

Now, edit your myParser.xs file. It contains glue code to map C function into Perl ones. Don't bother the already written code. Add only an include file after the existing ones:

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

and add this glue at the end of the file:

```
char*
perl_yylex()
    OUTPUT:
         RETVAL

char*
perl_yylextext()
    OUTPUT:
        RETVAL
```

Note that spaces and new lines are important. The first line for each function is the return type. It follows a line with the name of the function and, in the case of arguments, one line for each argument (like old C style). Then, there has to be two lines showing Perl where to get the return value from the C functions.

If you noticed the include directive I added, the file does not exist! Create the myLex.h file and insert these prototypes:

```
char* perl_yylex(void);
char* perl_yylextext(void);
```

We are done with the mapping of C functions to Perl ones. Now, we have to map data types. Integers are trivial and handled directly by perl, but we have pointers to characters. In this case, we have to create the file typemap in the module root directory with the text:

```
char*  T_PV
```

This maps pointers to characters to the built-in T_PV Perl type.

3.4 Putting it all together

Allright, you done it! The only problem is that the standard `Makefile.PL` does not knows how to compile this stuff. I changed my Makefile.PL to look like this:

```
use ExtUtils::MakeMaker;

$YACC_COMMAND = "( yapp -o myGrammar.pm -m 'myGrammar' myGrammar.yp)";

# This is needed before WriteMakefile
`$YACC_COMMAND`;

WriteMakefile(
              'NAME'             => 'myParser',
              'VERSION_FROM'     => 'myParser.pm',
              'LIBS'             => ['-lfl'], # This is for flex
              'MYEXTLIB'         => 'lib.so', # Our lexer
             );

sub MY::postamble {
  "
\$(MYEXTLIB): lex.perl_yy.c myParser.pm
\t\$(CC) -c lex.perl_yy.c
\t\$(AR) cr \$(MYEXTLIB) lex.perl_yy.o
\tranlib \$(MYEXTLIB)

myGrammar.pm: myGrammar.yp
\t$YACC_COMMAND

lex.perl_yy.c: myLex.l
\tflex -Pperl_yy myLex.l
          ";
}
```

3.5 Testing

Now, compile it the standard way:

```
perl Makefile.PL
make
make test
```

The test stuff didn't show off if this really works. Let us add some lines to the end of the test script (`test.pl` file):

```
use myGrammar;
my $parser = new myGrammar;
my $result = $parser->YYParse(yylex => \&myParser::perl_lex,
                              yyerror => \&myParser::perl_error);
print $result;
```

Now, make test again and write `1+4*2+3`, press enter and hit `C^d`. You did it!

4 Conclusions

While Perl is really flexible, there are tools which do some tasks quicker than Perl. This is because Perl is an interpreted language, and tools like `flex` generate an optimized finite automata program written in `C`. This means we should not re-invent the wheel, and use these tools from Perl.

4.1 See also

Check these man pages: flex(1), perlguts(1), perlxs(1), perlxstut(1) and Parse::Yapp(3).