Building a parser with Perl

COMPILATION ARTIST

Lexers and parsers aren't only for the long-bearded gurus. We'll show you how you can build a parser for your own custom applications. **BY MICHAEL SCHILLI**

Lexers and parsers are everyday tools for compiler designers and inventors of new programming languages. Both lexers and parsers check arbitrarily complex expressions for syntactic validity and help to translate these complex expressions from a human-readable format to a machine language format.

Admittedly, having to write your own parser is quite uncommon these days, as data is often XML formatted, and there are enough easy-to-use parsers capable of handling XML data. But if you need to analyze and evaluate formulas entered by users, you have no alternative but to build your own parser.

Lex Me!

If you need to evaluate an expression such as *5+4*3*, you first have to isolate the operators and operands. As Figure 1 shows, a so-called lexer first extracts the symbols *5*, *+*, *4*, ***, and *3* from the string. These strings, which are also referred to as tokens, are fed to the parser, which then checks if they make mathematical sense. To do so, the parser typically creates a tree structure, which it then uses to check if the expression passed to it obeys the rules of a previously defined grammar. The grammar also specifies things like operator precedence (e.g., PEMDAS) or associativity (from left to right, or vice-versa).

After ascertaining the exact meaning of the expression, the computer can evaluate it. The bottom part of Figure 1 shows an example of a RPN processor (RPN: Reverse Polish Notation). The virtual machine pushes either numbers or operators onto the stack, and then it attempts to reduce operand-operand-operator combinations to single values. In Figure 1, first *4 3 ** is reduced to a value of 12, and the combination at the top of the stack, *5 12 +*, becomes 17, which is the correct result of the original computation task *5+4*3*. Of course, nothing stops you from passing a string like *5+4*3* to the Perl *eval* function, which would apply Perl's math rules to evaluate the expression. But if the expression contains variables, operators that Perl doesn't understand, or even if-else constructs, that is, if you are handling a miniature programming language, there is no alternative to a full-fledged parser.

Back to the lexer: we need to ignore blanks in the string we are evaluating; that is, the expression *5 +4 *3* has to produce the same symbols as *5+ 4*3*. However, lexing is not always as trivial as the example I just gave you. The operand could be a real number such as *1.23E40*, or even a function such as *sin(x)*, which we would need to break down into the tokens *sin(*, *x*, and *)*. CPAN has the *Parse::Lex* module for complex lexing activities like this. When you install the module, note that it needs

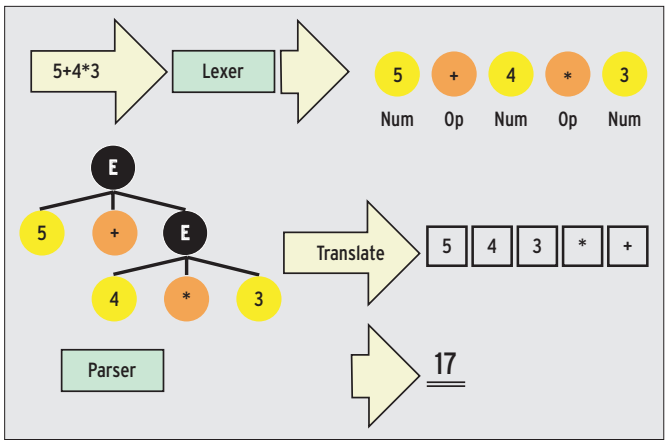

Figure 1: The lexer converts the string to tokens, and the parser creates the parse tree. The translator converts this to Reverse Polish Notation (RPN) and calculates the results by applying a simple algorithm.

at least version 0.37 of the *Parse::Template* module.

The *mathlexer* script (Listing 1) shows an example. It expects an arbitrarily complex mathematical expression as input and passes this on to the lexer; the lexer returns the token type and token content, which are then output for test purposes.

The module used by *mathlexer*, *MathLexer.pm*, defines the *MathLexer* class, which provides the *new* constructor to accept a string for lexical analysis (Listing 2). It then goes on to check if the string matches a number of regular expressions stored in the *@tokens* array. For each lexeme the *next* method then finds (a lexeme is a sequence of characters found by the lexer from which the lexer generates a token), the lexer returns two values.

The first element of the returned array reference is the name of the token the lexer has found (for example "NUM", "OPADD", "RIGHTP"). The second element then contains the actual value found in the analyzed text (e.g. "4.27e-14", " + ", ")"). Figure 2 shows the test output, which will be used as parser fodder in a real-life situation.

Note that *Parse::Lex* expects regular expressions as strings in the *@token* array. This means you need to escape backslashes as \\ if you want to avoid symbols such as * being interpreted as regex metacharacters. As expressions such as ** are difficult to decipher, MathLexer uses the slightly strange looking but identical regex, "*[*][*]*", for the first token definition.

A regular expression that covers the various ways of representing real numbers (for example, *1.23E40*, *.37*, *7*, *1e10*), isn't easy to formulate. Fortunately, the CPAN module *Regexp::Common* has prebuilt expressions for many tasks, including one for real numbers with all kinds of bits and bobs. After calling *use Regexp::Common* in the program, you can use a global hash to leverage these pearls of regex wisdom. The expression for real numbers can be retrieved by

simply writing *$RE{num}{real}*.

Incidentally, this expression also allows an optional minus sign in front of the real number. But due to the selected order of lexemes detected in *@tokens*, the lexer will always assume a preceding minus sign to be an *OP*. However, if a minus sign occurs in the real number's exponent, the lexer will take it to be a part of the *NUM* lexeme.

Additionally, the *skip* method called in Line 32 of Listing 2 ensures that the lexer ignores spaces and newline characters. However, if the *skip* method stumbles across a character sequence it doesn't recognize (such as }), the ERROR pseudo-token in Line 19 is used. This token defines an error-handling routine, which issues the *die* command to tell the lexer to quit.

Syntax Check, Tokens Please!

A parser then checks the syntactic validity of an expression. *4 + *3* would be invalid; we want the parser to report an error in this case and cancel processing. In many cases, parsers not only check the syntax of an expression, but also handle the translation work. After all, why not let the parser work out the results while it is poring over an arithmetic expression.

Listing 3, *AddMult.yp*, defines a grammar for the parser. It specifies how the parser combines the tokens streaming out of the lexer to predefined structures. The first so-called production, *expr: add | mult | NUM;*, specifies that the overall task of the parser is to reduce the sequence of all tokens to an *expr* type construct. If this is impossible, the tokens do not obey the grammar; a syntax error has occurred, and the parser quits.

Productions such as the one in Listing 3 have a so-called non-terminal on the left. The goal for the parser is to somehow match the lexer output with the right side of a production and then reduce it to the non-terminal on its left side. On its right side, a production can list lexed tokens (also known as terminals) but also other non-terminals, which are then resolved by other productions. In our example, *expr* can be three things, as the alternatives separated by the pipe sign, "|" on the right of the colon show: *add* (an addition), *mult* (a multiplication), or a terminal *NUM*, a real number that comes from the lexer.

The non-terminals *add* and *mult* are defined in the following productions in *AddMult.yp*. *add: expr OPADD expr* specifies that a non-terminal *add* comprises two *expr* non-terminals linked by the '+' operator. And as we already know, *expr* can contain additions, multiplications, or simple numbers.

The grammar file, *AddMult.yp*, provides an abstract description of a parser to the Parse::Yapp module available on CPAN. *AddMult.yp* is divided into three sections separated by the *%%* string. The header is at the top; it can contain

Listing 1: mathlexer

```
01 #!/usr/bin/perl -w
02 use strict;
03 use MathLexer;
04
05 my $str = ❯
"5*sin(x*-4.27e-14)**4*(e-pi)";
06 print "  $str\n\n";
07
08 my $lex =
09   MathLexer->new($str);
10
11 while (1) {
12
13  my ($tok, $val) =
14    $lex->next();
15
16  last unless defined $tok;
17
18  printf "%8s %s\n",
19         $tok, $val;
20 }
```

Listing 2: MathLexer.pm

```
01 ##################################
02 package MathLexer;
03 ##################################
04 use strict;
05 use Regexp::Common;
06 use Parse::Lex;
07
08 my @token = (
09  OPPOW  => "[*][*]",
10  OPSUB  => "[-]",
11  OPADD  => "[+]",
12  OPMULT => "[*]",
13  OPDIV  => "[/]",
14  FUNC   => "[a-zA-Z]\\w*\\(",
15  ID     => "[a-zA-Z]\\w*",
16  LEFTP  => "\\(",
17  RIGHTP => "\\)",
18  NUM    => "$RE{num}{real}",
19  ERROR  => ".*",
20  sub {
21   die qq(Can't lex "$_[1]");
22  },
23 );
24
25 ##################################
26 sub new {
27 ##################################
28  my ($class, $string) = @_;
29
30  my $lexer =
31    Parse::Lex->new(@token);
32  $lexer->skip("[\\s]");
33  $lexer->from($string);
34
35  my $self =
36    { lexer => $lexer, };
37
38  bless $self, $class;
39 }
40
41 ##################################
42 sub next {
43 ##################################
44  my ($self) = @_;
45
46  my $tok =
47    $self->{lexer}->next();
48  return undef
49    if $self->{lexer}->eoi();
50
51  return $tok->name(),
52    $tok->text();
53 }
54
55 1;
```

parser instructions or Perl code. The productions belonging to the grammar are in the middle and are followed by the footer, which can define more Perl code, although it is empty in Listing 3. To implement the parser, *AddMult.yp* is converted to a Perl module by the *yapp* utility that comes with Parse::Yapp.

The module created by this process, *AddMult.pm*, implements a so-called bottom-up parser. This kind of parser reads a token stream from the lexer and attempts to create the parse tree shown in Figure 1 from the bottom upward. To do so, it combines the units it has read to create higher level constructs from tokens and lower level constructs. It continues this process, based on the rules of the grammar, until the results match the left side of the *first* production.

At each step, the parser does one of two things: shift or reduce. Shift tells the parser to get the next token from the input stream and push it onto the stack. Reduce tells the parser to combine the terminals and non-terminals on the stack to create higher level non-terminals, based on the rules of the grammar, thus reducing the height of the stack. If the input queue is empty, and if the last reduction has just left the parser with the left side of the initial production, the parser run has completed successfully.

Table 1 shows how a bottom-up parser implemented on the basis of the grammar in *AddMult.yp* processes tokens extracted from an input string of *5+4*3* step by step.

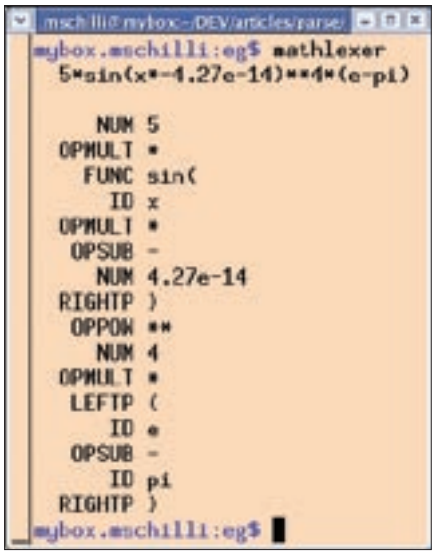

Figure 2: A mathematical expression lexed by MathParser.pm.

Listing 3: AddMult.yp

```
01 %left OPADD
02 %left OPMULT
03
04 %%
05 expr: add | mult | NUM;
06
07 add:  expr OPADD expr {
08         return $_[1] + $_[3]
09       };
10 mult: expr OPMULT expr {
11         return $_[1] * $_[3]
12       };
13 %%
```

In Step 0, the tokens *[NUM, "5"]*, *[OPADD, "+"]*, *[NUM, "4"]*, *[OPMULT, "*"]*, and *[NUM, "3"]* are available in the input queue. In Step 1, the parser pushes 5 (which is a NUM token) onto the stack (shift). In Step 2, it reduces the *NUM* terminal to *expr* based on the third alternative of the first production in the *AddMult.yp* grammar. The parser then processes the *[OPADD, "+"]* and *[NUM, "4"]* tokens from the input, shifts them onto the stack, and then reduces 4 to *expr*. But where to go from there? The parser could reduce *expr OPADD expr* on the stack to *expr*, following the second production of the grammar. On the other hand, it could fetch *[OPMULT, "*"]* from the input and hope to find another *expr* later to reduce *expr OPMULT expr* (third production).

Conflict

This kind of problem is common; grammars are often ambiguous. If we didn't have the traditional PEMDAS rule in math, the parser would be completely baffled by the shift-reduce conflict caused by the expression "5+4*3". The fact that algebraic operators have precedence, however, avoids the conflict. The parser has to wait before reducing 5+4, and needs to shift the * token onto the stack, as * is a stronger link between the operands than the weaker +.

If the same operators occur multiple times in succession, as in *5-3-2*, all operations have the same precedence, and another type conflict occurs. If the parser decides to reduce, after parsing *5-3*, it evaluates the operators from left to right, which is exactly according to the rules of algebra. A shift, on the other hand, would evaluate the expression as *5-(3-2)* instead, and this expression would lead to a surprising result of 6, instead of the 0 we might expect. The minus operator is thus referred to as being left-associative. We need to tell the parser about this, then it can resolve this type of conflict as well.

By the way, in the case of the power operator (** in Perl), algebra dictates the opposite approach: "4**3**2" ("4 to the power of 3 to the power of 2") is calculated as *4**(3**2)*. The power operator is right-associative! This is easy to check in Perl: *perl -le 'print 4**3**2'* gives us 262144 (4**9) and not 4096 (64**2).

Associativity and Precedence

The *yapp* parser generator also notices that the grammar is ambiguous. Here's how the *yapp* generator created the parser module *AddMult.pm* from the *AddMult.yp* file:

```
$ yapp -m AddMult AddMult.yp
4 shift/reduce conflicts
```

The first two lines in Listing 3 resolve the grammar confliect:

```
%left OPADD
%left OPMULT
```

Listing 4: addmult

```
01 #!/usr/bin/perl
02 use warnings;
03 use strict;
04
05 use MathParser;
06 use AddMult;
07
08 my $mp =
09   MathParser->new(
10  AddMult->new());
11
12 for (
13  qw( 5+4*3 5+4+3 5*4*3 5*4+3)
14   ) {
15  print "$_: ",
16    $mp->parse($_), "\n";
17 }
```

Listing 5: MathParser.pm

```
01 ##############################
02 package MathParser;
03 ##############################
04 use MathLexer;
05 use strict;
06 use warnings;
07
08 ##############################
09 sub new {
10 ##############################
11  my ($class, $parser) = @_;
12
13  my $self =
14    { parser => $parser };
15
16  bless $self, $class;
17 }
18
19 ##############################
20 sub parse {
21 ##############################
22  my ($self, $str, $debug) =
23    @_;
24
25  my $lexer =
26    MathLexer->new($str);
27
28  my $result =
29    $self->{parser}->YYParse(
30
31      yylex =>
32       sub { $lexer->next() },
33
34      yyerror =>
35       sub { die "Error" },
36
37      yydebug => $debug ?
38                 0x1F : undef
39  );
40 }
41
42 1;
```

These statements stipulate that both the + operator and the * operator are left-associative and, more importantly, that OPMULT has priority over *OPADD*, as *%left OPMULT* occurs later in the parser definition than *%left OPADD*.

If the parser were to define an OPMINUS operation using the - operator, it would be important to insert *%left OPMINUS before* the definition of OPMULT. If the yp file header had an entry for *%right OPMINUS* instead of *%left OPMINUS*, the parser would evaluate expressions such as *5-3-2* from right to left. And this would be disastrous, as *5-(3-2)* is *6*, in contrast to *5-3-2*, which gives us a value of 0. To teach the parser to raise numbers to powers, we would need a right-associative power operator *%right OPPOW*, after the *OPMULT* definition due to the high priority of the power operation and its right-associativity.

These tricks let the parser complete as shown in Table 2.

Besides the grammar, *AddMult.yp* defines some executable Perl code attached to the productions. For example,

```
mult: expr OPMULT expr {
       return $_[1] * $_[3]
      };
```

stipulates that the return value of the production (a companion to the non-terminal on the left side) is the product of the return values of the two *expr* expressions. This means that the parser will keep on pushing the result of the arithmetic expression it is evaluating upward until it reaches the start production, and the result can be returned to the caller by the parser. This automatically gives the syntax checker the ability to calculate formulas. Tables 1 and 2 show the return values of the current reduction in the "Return" column.

Note that *$_[1]* in the code segments refers to the first expression on the right side of the production (that is *expr*). Departing from the norm, the counter does not start at 0 here, as *$_[0]* in *Parse::Yapp* productions is always a reference to the parser. If a production contains multiple | separated alternatives, each alternative can define its own block of code. Note that a block of code only refers to the alternative it is attached to.

Before we can use the parser, just one more intermediate step: the *yapp* parser interface is slightly exotic, and as we will be using our previously defined *MathLexer* lexer, we can define a simpler interface in Listing 5. The *parse()* method in *MathParser* simply accepts the string to be parsed, and returns the arithmetic result. If an error occurs, the parser goes to the anonymous subroutine defined in Line 35 and quits.

Listing *mathparser* shows a simple application that uses *MathParser.pm* to parse and evaluate four different expressions:

```
5+4*3: 17
5+4+3: 12
5*4*3: 60
5*4+3: 23
```

Table 1: Parser Steps

Step	Rule	Return	Stack	Input
0				5+4*3
1	SHIFT		NUM	+4*3
2	REDUCE expr: NUM	5	expr	+4*3
3	SHIFT		expr OPADD	4*3
4	SHIFT		expr OPADD NUM	*3
5	REDUCE expr: NUM	4	expr OPADD expr	*3
	*Conflict: Shift/Reduce?			

Listing 6: UnAmb.yp

```
01 # Unambiguous +/* grammar
02 %%
03 expr: expr OPADD term {
04         return $_[1] + $_[3];
05       }
06       | term {
07         return $_[1];
08       };
09
10 term: term OPMULT NUM {
11         return $_[1] * $_[3];
12       }
13       | NUM {
14         return $_[1];
15       };
16 %%
```

Table 2: Final Stage of Parser Run

Step	Rule	Return	Stack	Input
6	SHIFT		expr OPADD expr OPMULT	3
7	SHIFT		expr OPADD expr OPMULT NUM	
8	REDUCE expr: NUM		expr OPADD expr OPMULT expr	
9	REDUCE expr: expr OPMULT expr	12	expr OPADD expr	
10	REDUCE expr: expr OPADD expr	17	expr	

This shows that the parser honors precedence rules and evaluates expressions such as *5+4*3* and *5*4+3* correctly.

There is another way of resolving precedence conflicts. If you formulate a grammar such as the one in Listing 6, the higher precedence of the '*' operator derives from the relationships between the productions. A multiplication is first reduced in the non-terminal *term*, before any addition reductions are performed.

This approach also allows us to implement the behavior for parentheses, if they are are allowed in the input string; to force "(5+4)*3", for example. To do this, we simply redefine the *term* production and add another production for *force*, which jumps on any parantheses and immediately reduces the expressions between the brackets:

```
term:  term OPMULT force
         { ... }
       | force
force: LEFTP expr RIGHTP
         { return $_[2]; }
       | NUM
```

Instead of evaluating the arithmetic expression directly, it makes sense to convert it to a format that is easier to compute, such as RPN. Listing 7 shows the grammar for doing this. We have only changed the production code segments, which, rather than passing on calculated values, now write numbers and operations to an array, which is passed back as a reference, finally arriving where the parser was called.

rpn is the calling script; as you might expect, it produces completely different conversions of *5+4*3* and *5+4+3*:

```
5+4+3: [5, 4, +, 3, +, ]
5+4*3: [5, 4, 3, *, +, ]
```

With the top expression, the translator simply processes the expression from left to right and adds the individual values, first adding 5 and 4, and then adding 3 to the result.

With the bottom expression, *5+4* cannot be reduced straight away due to PEMDAS rules. Instead, the translator first pushes the next number, *3*, onto the RPN stack, then it performs the multiplication, and only then does it add the result 12 to the 5 located lower down on the stack.

There are numerous books on the subject of parsing; the Dragon Book [2] is the classic work. It may not be easy to read, but it is indispensable. Besides the bottom-up parser generator, *Parse::Yapp*, which is based on techniques used by the Unix tools *lex*, and *yacc* ([5]), CPAN also has a top-down parser generator, *Parse::RecDescent*. *Parse::RecDescent*. has completely different characteristics due to the differences in parsing technology employed. [4] gives a few examples on how to use *Parse::Yapp* and *Parse::RecDescent*. Finally, you can write parsers manually. This option of manually writing the parser is particularly effective with functional programming, as described at [3] and [6]. ■

Listing 7: RPN.yp

```
01 %left OPADD
02 %left OPMULT
03
04 %%
05 expr: add
06       | mult
07       | NUM { return
08               [ $_[1] ]; };
09
10 add:  expr OPADD expr {
11           return [
12               @{$_[1]},
13               @{$_[3]},
14               $_[2]
15           ];
16       };
17
18 mult: expr OPMULT expr {
19           return [
20               @{$_[1]},
21               @{$_[3]},
22               $_[2]
23           ];
24       };
25 %%
```

Listing 8: rpn

```
01 #!/usr/bin/perl
02 use strict;
03 use warnings;
04
05 use MathParser;
06 use RPN;
07
08 my $mp =
09   MathParser->new(
10  RPN->new());
11
12 for my $string (
13  qw(5+4+3 5+4*3)) {
14
15  print "$string: [";
16
17  for( @{ $mp->parse($string)
18        }) {
19   print "$_, ";
20  }
21
22  print "]\n";
23 }
```

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/66/Perl*

[2] *Compilers*, Aho, Sethi, Ullman, Addison Wesley, 1986

[3] *Higher Order Perl*, Mark Jason Dominus, Morgan Kaufmann, 2005

[4] *Pro Perl Parsing*, Christopher M. Frenz, Apress, 2005

[5] *lex & yacc*, Levine, Mason & Brown, O'Reilly, 1990

[6] "Parser Combinators in Perl," Frank Antonsen, theperlreview.com, Summer 2005