Practical Perl Tools
Parse Me, Amadeus

DAVID N. BLANK-EDELMAN

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010. dnb@ccs.neu.edu

In past columns we've had the pleasure of looking at configuration file processing of all sorts. We've discussed ways to work with simple file formats like .ini files and more complex formats like XML, YAML, and JSON. But what if you find you need something even more sophisticated? What if you find you need a config that is actually a mini-language (some would call it a DSL, or domain specific language)? In cases like that you'll have to write code that can parse this language so your program can work with the directives you've specified. This column is about one of the more popular and more powerful modules for this work.

The Basics

I should note that when you start to say words like "parse" the computer scientists in the room perk up their ears because they've all had the pleasure of studying compiler design at some point in their academic career. I personally haven't cracked open the canonical but actually really good tome on the subject ("the Dragon Book," aka *Compilers: Principles, Techniques, and Tools*) in quite a few years. My apologies if I am playing faster and looser with terminology around parsing than perhaps I should as a graduate of that august field. But let's talk about a few key ideas before actually seeing any code. The key things I want to get into are the "how" and the "what" of the process. But warning: we're going to only skim the surface of all of the subjects mentioned in this column.

Typically, a parser's job is to take in a set of "tokens" and decide if it makes sense in terms of some language definition (and if it does, the parser hands the program back some sort of data structure that contains the results of the parse). Let's see a simple language so I can show you what I mean by token. Most parsing tutorials start out with a calculator example (the tokens are "numbers" and "operators" where one of the operations might be "plus"), but let's use something slightly more interesting:

```
recipe strawberry lemonade popsicle
ingredient frozen lemonade - 12 ounces
ingredient cold water - 3 cups
ingredient frozen sliced strawberries - 16 ounces
direction stir lemonade + water
direction blend strawberries
direction stir strawberries + lemonade
direction freeze
```

In this case, I could say the first line above was made up of "recipe" followed by a NAME. The second line has "ingredient" an ingredient NAME, and a QUANTITY. Later on we see "direction," an ACTION and a set of OBJECTS. All of these things can be considered tokens.

As a related aside (if just to satisfy some of the other CS majors who are jumping up and down on the sidelines with their hands in the air waiting to point this out): there is a process that takes place before parsing, namely changing the plain stream of incoming text to tokens (r..e..c..i..p..e..<space>.. gets turned into "recipe" which is a RECIPE_LABEL token). That is

typically handled by lexer code. The Perl module we'll be using has a lexer built in so we won't need to explicitly do much lexing, but it is good to know what is going on when we get to that point.

Grammars Rock

We just discussed a bit of the "how" parsing works; now let's look into the "what" we are parsing. We need a way to tell a parser "here's the definition of the language to parse." More often than not, that definition takes the form of a grammar. Here's a simple grammar that matches the recipe language example we see above:

```
NAME INGREDIENT+ DIRECTION+
NAME: 'recipe' name
INGREDIENT: 'ingredient' name '-' amount UNIT
UNIT: 'ounces' | 'cups' | 'pounds'
DIRECTION: 'direction' action | 'direction' action name '+' name
```

Let's walk through this grammar one line ("rule") at a time. The first rule says a line consists of a NAME line followed by one or more INGREDIENT lines and then by one or more DIRECTION lines. Subsequent rules define what those kinds of lines contain. A NAME line starts off with the literal string 'recipe' followed by the name of the recipe. An INGREDIENT line starts with 'ingredient' followed by the name, a literal dash, and the amount of the ingredient in one of several possible units (as specified in the subsequent rule). Finally, we provide a DIRECTION line that can either specify just an action or an action that takes place between two of the ingredients.

One thing that may be a bit surprising about this grammar is the first line. You might be tempted to write it like this (as I did at first when writing this article):

```
NAME | INGREDIENT+ | DIRECTION+
```

because it might seem like we'll be parsing a recipe name line or some number of ingredient lines, or some number of direction lines. And indeed, we will be parsing one of those kinds of lines at a time. But if we want to specify that we are parsing one of those, followed by the next, followed by the next thing, we won't be specifying them as alternatives. If we do, then the parser can say, "Okay, let's match the first rule. The first rule says I need to find just one of those alternatives from the list. Found one. Okay, that rule has matched so I must be done parsing." Instead, we say we'll need to say we expect one thing after another.

Bring on the Perl

Now that we have a grammar that specifies what we want to parse and a sample document to parse, let's put tab A into slot B. There are a number of Perl modules for parsing grammars, but the one we're going to look at is Parse::RecDescent. Parse::RecDescent has been around since 1997 and is one of the grand dames of the Perl parsing world at this point. We'll turn everything we've seen so far into a Perl program using that module:

```
use Parse::RecDescent;

my $grammar = q {
   startrule: recipename ingredient(s) direction(s)
   recipename: 'recipe' name
   ingredient: 'ingredient' name '-' amount unit
   unit: 'ounces'
        | 'cups'
        | 'pounds'
   direction: 'direction' action name '+' name
            | 'direction' action name
            | 'direction' action
   action: /\w+/
   amount: /\d+/
   name: /[a-zA-Z0-9 ]+/
};

my $heredoc = <<END;
recipe strawberry lemonade popsicle
ingredient frozen lemonade - 12 ounces
ingredient cold water - 3 cups
ingredient frozen sliced strawberries - 16 ounces
direction stir lemonade + water
direction blend strawberries
direction stir strawberries + lemonade
direction freeze
END

my $parser = new Parse::RecDescent($grammar);

print defined $parser->startrule($heredoc) ? 'OK' : 'NOT OK', "\n";
```

The major parts of this program are pretty simple: first we list the grammar we're going to use (more on this in a moment), followed by the sample document we're going to parse. We request a Parse::RecDescent object that we next used to start the parse at the rule marked 'startrule' and perform a parse, printing the results.

Now that we're looking at actual code (finally!) it would probably be useful to compare the code to the previous grammar in our text because the differences will be illustrative. The first difference is our first line gets marked "startrule" so we know where to begin a parse. It would be reasonable to have a convention that a parse starts at the first rule listed, but no such convention exists for the module. This makes more sense if there could be two potential starting places for a parse, for example a "debug rule" and the real "start rule." The only problem with this explanation is I'm making this reason up. I've never seen people actually do this, but it sure sounds plausible, doesn't it?

```
1 |startrule   |Trying subrule: [recipename]                                    |
2 |recipename |Trying rule: [recipename]                                       |
2 |recipename |Trying production: ['recipe' name]                              |
2 |recipename |Trying terminal: ['recipe']                                     |
2 |recipename |>>Matched terminal<< (return value: [recipe])                   |
2 |recipename |                                                                |
2 |recipename |                                                                |" strawberry lemonade
  |           |                                                                |popsicle\ningredient frozen
2 |recipename |                                                                |lemonade - 12
  |           |                                                                |ounces\ningredient cold
2 |recipename |                                                                |water - 3 cups\ningredient
  |           |                                                                |frozen sliced strawberries -
2 |recipename |                                                                |16 ounces\ndirection stir
  |           |                                                                |lemonade + water\ndirection
2 |recipename |                                                                |blend
  |           |                                                                |strawberries\ndirection stir
2 |recipename |                                                                |strawberries +
  |           |                                                                |lemonade\ndirection
2 |recipename |                                                                |freeze\n"
2 |recipename |Trying subrule: [name]                                          |
3 |   name    |Trying rule: [name]                                             |
3 |   name    |Trying production: [/[a-zA-Z0-9 ]+/]                            |
3 |   name    |Trying terminal: [/[a-zA-Z0-9 ]+/]                              |
3|    name    |>>Matched terminal<< (return value: [strawberry lemonade popsicle])|
3 |   name    |                                                                |
3 |   name    |                                                                |"\ningredient frozen
  |           |                                                                |lemonade - 12
3 |   name    |                                                                |ounces\ningredient cold
  |           |                                                                |water - 3 cups\ningredient
3 |   name    |                                                                |frozen sliced strawberries -
  |           |                                                                |16 ounces\ndirection stir
3 |   name    |                                                                |lemonade + water\ndirection
  |           |                                                                |blend
3 |   name    |                                                                |strawberries\ndirection stir
  |           |                                                                |strawberries +
3 |   name    |                                                                |lemonade\ndirection
  |           |                                                                |freeze\n"
3 |   name    |                                                                |
3 |   name    |>>Matched production: [/[a-zA-Z0-9]+/]<<                        |
3 |   name    |                                                                |
3 |   name    |>>Matched rule<< (return value: [strawberry lemonade popsicle])  |
3 |   name    |                                                                |
3 |   name    |(consumed: [strawberry lemonade popsicle])                      |
3 |   name    |                                                                |
2 |recipename |>>Matched subrule: [name]<< (return value: [strawberry lemonade popsicle]  |
2 |recipename |                                                                |
2 |recipename |>>Matched production: ['recipe' name]<<                         |
2 |recipename |                                                                |
2 |recipename |>>Matched rule<< (return value: [strawberry lemonade popsicle])  |
2 |recipename |                                                                |
2 |recipename |(consumed: [recipe strawberry lemonade popsicle])                |
2 |recipename |                                                                |
```

Figure 1: If you set $::RD_TRACE variable in Parse::RecDescent to 1, you will get debugging output like this when parsing the first line in our example.

The second and perhaps more important difference between the grammar versions is the stuff at the bottom of the grammar. In the first appearance of our grammar we mentioned things like "name," "amount," and "action" without ever saying just what those things are (or more importantly, how the parser would know one if it bumped into one in a dark alley). If we left out those lines from our grammar, our program would throw the following errors:

```
Warning: Undefined (sub)rule "action" used in a production.
Warning: Undefined (sub)rule "name" used in a production.
Warning: Undefined (sub)rule "name" used in a production.
Warning: Undefined (sub)rule "name" used in a production.
Warning: Undefined (sub)rule "amount" used in a production.
```

Parse::RecDescent makes defining these parts of the grammar easy; we just need to provide a Perl regular expression that will match that part. We say a name can be a letter/number (plus a space if desired), an action is a single word, and an amount is one or more digits. And, yes, we are actually providing direction to the Parse::RecDescent lexer so it knows how to construct those tokens.

One last thing to point out is that Parse::RecDescent has a very legible (for English speakers) way of saying, "One or more of these rules." We see that in action in the grammar where it uses this English pluralization idiom when it mentions "ingredient(s)" and "direction(s)" to indicate it is standing in for one or more of those things.

With all of this build up, what happens if we run this program? It outputs (oh, the suspense is delicious):

```
OK
```

If we changed the sample document so it said:

```
ingredient frozen lemonade - 12 bounces
```

it would print:

```
NOT OK
```

instead. Okay, maybe not so exciting, but actually this is useful. Now you know how to write a program that validates a document based on your mini-language. We'll see how to actually capture the info in the document in just a second. Before we do, I want to mention a super- helpful Parse::RecDescent feature that you may find yourself using during development. If you add the following line to your code:

```
$::RD_TRACE = 1;
```

it spits out a ton of really useful debugging information about the parse. In the interest of space, let me show you a very small excerpt of the debug output.

In Figure 1, you can see the parse began with its start rule trying to match the subrule about the recipe name. The rule it is trying to match is found in the second column. In the trace in Figure 1, we can see that the parser looks for the literal string 'recipe', finds it, and then sees whether it can find the input it needs to collect a recipe name from the input it has available (shown in the third column). It succeeds, showing you the result of the matches and what part of the input it was able to consume.

So how do we use the information that Parse::RecDescent presumably could gather as it parses merrily along? To do that we have to discuss what the module calls "actions." With Parse::RecDescent, you can specify what should happen at each step in the parse. For example, you might want to have the parser return the values it matched along the way so you can construct a data structure that the rest of your program will traverse. The simplest way to get into the action game is to use a feature called autoactions that lets you set a single action to automatically take place after every rule has been parsed. It gets specified something like this:

```
$::RD_AUTOACTION = q { [@item] };
```

(or you can sneak it into the grammar itself using a special tag). The @item array in an action holds info on the items that are being matched ($item[0] is the actual name of the rule that is being matched; the rest of the array specifies the other parts of what is found). There are other magic variables that can be referenced; see the doc for more information. If we took our previous program and added that autoaction line (plus loading Data::Dumper) and said instead:

```
my $parseresults = $parser->startrule($heredoc);

print Dumper $parseresults,"\n";
```

we would see output that began this way:

```
$VAR1 = [
          'startrule',
          [
            'recipename',
            'recipe',
            [
              'name',
              'strawberry lemonade popsicle'
            ]
          ],
          [
            [
              'ingredient',
              'ingredient',
              [
                'name',
                'frozen lemonade '
```

```
        ],
        '-',
        [
          'amount',
          '12'
        ],
        [
          'unit',
          'ounces'
        ]
      ],

      ...
```

For a more complex but precise parse tree, we can slip an <autotree> tag ahead of the startrule in the grammar, and Parse::RecDescent will create a data structure that begins like this:

```
$VAR1 = bless( {
  '__RULE__' => 'startrule',
  'recipename' => bless( {
      '__RULE__' => 'recipename',
      'name' => bless( {
                        '__VALUE__' => 'strawberry lemonade
                             popsicle'
                      }, 'name' ),
     '__STRING1__' => 'recipe'
  }, 'recipename' ),
  'ingredient(s)' => [
      bless( {
               'unit' => bless( {
                                  '__VALUE__' => 'ounces'
                                }, 'unit' ),
               'amount' => bless( {
                                    '__VALUE__' => '12'
                                  }, 'amount' ),
               '__STRING2__' => '-',
               '__RULE__' => 'ingredient',
               'name' => bless( {
                                  '__VALUE__' => 'frozen
                                       lemonade '
                                }, 'name' ),
               '__STRING1__' => 'ingredient'
             }, 'ingredient' ),

            ...
```

Now, what you do with that data structure once you get it is truly up to you. In our case, you could have something that engages your fully automated kitchen to make a popsicle for you.

I want to leave you pondering this little bit of free will, but before I go I think I would be remiss if I didn't mention that there are other really cool parsing modules available. The two that I have my eye on in particular are the Regexp::Grammars module (builds on the super-powerful regexp constructs in Perl 5.10+) and the Marpa::R2 module, which uses a very different parsing algorithm than Parse::RecDescent and can do some cool stuff that Parse::RecDescent can't. Do check them both out if parsing is in your future.

Take care and I'll see you next time.