Practical Perl Tools

Perl without Perl

DAVID N. BLANK-EDELMAN

David has over thirty years of experience in the systems administration/DevOps/SRE field in large multiplatform environments and is the author of the O'Reilly Otter book (new book on SRE forthcoming!). He is one of the co-founders of the now global set of SREcon conferences. David is honored to serve on the USENIX Board of Directors where he helps to organize and engineer conferences like LISA and SREcon. dnb@usenix.org

It's not exactly a Zen koan, but it will have to do for this issue's column. Today we're going to talk about how a method for parsing Perl without using the actual Perl interpreter can offer a whole host of benefits. This idea may be somewhat surprising because Perl has a reputation (perhaps deserved) as being a language where "the only thing which can parse Perl (the language) is perl (the binary)." For some detailed examples of this criticism, see one of the well-known essays at http://www.perlmonks.org/index.pl?node_id=44722, which is where that quote came from, originally attributed to Tom Christiansen.

Part of the problem is that there are some ambiguities in the language that only resolve themselves definitively during runtime. This reality harshed many the mellow of aspiring tool creators until at some point someone asked the question, "Could we write something that could parse enough Perl to be useful? Maybe we won't get 100% correct behavior, but how good does it have to be to let us get real work done?" Turns out, we can actually get really, really close. The leap in thinking that made this possible was a small shift in mindset: instead of thinking of the program/file as code that is executed, we can think of it as a static document we can parse. This lets us get close enough that very useful tools can be created, and that's what this column will focus on.

PPI

The heart of all of this work is the PPI module and the small ecosystem that surrounds it. PPI is a Perl module (so perhaps the title isn't entirely accurate) that knows how to parse existing Perl code. Even though this module is at the center of everything we're going to talk about today, we're going to do almost nothing with it directly. Directly using PPI is easy (excerpted from the docs):

```
use PPI;

# Create a new empty document
my $Document = PPI::Document->new;

# Load a Document from a file
$Document = PPI::Document->new('Module.pm');

# Does it contain any POD?
if ( $Document->find_any('PPI::Token::Pod') ) {
   print "Module contains POD\n";
}

# Remove all that nasty documentation
$Document->prune('PPI::Token::Pod');
$Document->prune('PPI::Token::Comment');
```

```
# Save the file
$Document->save('Module.pm.stripped');
```

Basically, you create the object that will represent the Perl document and then tell PPI to parse the file (or a chunk of Perl code in a string, not shown here). In the code above we tell PPI to find or remove different Perl structures from the parsed info and write this document back out to disk. This is pretty simple, so what can we build on this model?

Make It Pretty

One thing we can do once we "understand" the Perl document that has been parsed is to output the document in a way that improves its readability. One example of this is `PPI::Prettify`.

`PPI::Prettify` bootstraps on the work done by Google to produce a syntax highlighting tool that would make it easy to embed readable code in a Web page. This is the same software that Stack Overflow uses for its code examples. Their package, *prettify.js* (https://code.google.com/archive/p/google-code-prettify/), consists of a JavaScript module to do the highlighting and accompanying CSS that lets you "theme" the results. `PPI::Prettify` lets you skip the JavaScript part and just make use of the CSS themes that work with prettify.js (plus the highlighting is allegedly more accurate).

Using the module is basically a single call:

```
use File::Slurp;
use PPI::Prettify 'prettify';

my $document = read_file( $ARGV[0] );

print prettify( { code => $document } );
```

Here you see me using `File::Slurp` to pull an entire file into memory because `PPI::Prettify` expects to have code fed to it via a scalar.

The output looks a little like this:

```
<pre class="prettyprint"><span class="kwd">use</span
><span class="pln"> </span><span class="pln">WebService
::Spotify</span><span class="pun">;</span><span class="pln">
</span><span class="pln">
...
```

which is only "pretty" when displayed in an HTML document that references the right prettify.js CSS file for those classes.

Count It

Another helpful class of things built on top of PPI are the modules that can give us some stats about our code. For example, the Perl::Metrics::Simple package comes with a *countperl* utility, which gives the following output:

```
$ countperl spotify2.pl
Perl files found: 1

Counts
--------
total code lines:          14
lines of non-sub code:     8
packages found:            0
subs/methods:              1

Subroutine/Method Size
----------------------------
min:                       6
max:                       6
mean:                      6.00
std. deviation:            0.00
median:                    6.00

McCabe Complexity
---------------------
Code not in any subroutine
min:                       5
max:                       5
mean:                      5.00
std. deviation:            0.00
median:                    5.00

Subroutines/Methods
min:                       2
max:                       2
mean:                      2.00
std. deviation:            0.00
median:                    2.00

List of subroutines, with most complex at top
----------------------------------------------------
complexity sub                                 path          size
  5        {code not in named subroutines} ./spotify2.pl    8
  2        print_artists                   ./spotify2.pl    6
```

If you really get into this sort of static analysis, there are more complex modules like `Perl::Metrics` and `Code::Statistics` you may want to explore.

Find It

A perhaps more exciting consequence of being able to parse Perl from Perl is the ability to create utilities that can selectively operate on source files based on the semantics of the code they contain. For example, we can now write programs that only work on Perl files that contain documentation (or better yet, are missing documentation). We can search for text just in the comments of the code (looking at only real comments vs. a crude guess that looks at strings that start with a #). We can look for code that has "barewords" in it, and so on. PPI opens this all up for us.

Two easy ways to get into this are using the Find::File::Rule:PPI module and utilities like App::Grepl. Let's look at both.

We've talked about the Find::File::Rule family before in this column (I'm very fond of it), but let's do a quick review anyway. Find::File::Rule is a module family meant to extend the functionality of the Find::File module that ships with Perl and also make it easier to use. Instead of writing a special subroutine whose job it is to determine whether an object found when traversing a directory tree is of interest, you write code that looks more like this (from the doc):

```
# find all the subdirectories of a given directory
my @subdirs = File::Find::Rule->directory->in( $directory );
```

and

```
# find all the .pm files in @INC
my @files = File::Find::Rule->file()
                            ->name( '*.pm' )
                            ->in( @INC );
```

Basically, you string together a bunch of methods that express rules for determining the files or directory names of interest. I find it easiest to read the code backwards—in the last code sample, it says to look in the directories listed in the @INC array. In those directories, collect the names of all of the files and directories that have a name ending in .pm. Of these, return those that are files.

Find::File::Rule:PPI adds a ppi_find any method that lets you specify the same sort of selectors we saw at the very beginning of the column. So, for instance, if we wanted a list of all of the Perl files in a directory (and its subdirectories) that have embedded POD documentation, we could write:

```
use File::Find::Rule;
use File::Find::Rule::PPI;

my @podfiles =
  File::Find::Rule
   -> file()
   -> name('*.pm')
   -> ppi_find_any('PPI::Token::Pod')
   -> in('.');

print join("\n", @podfiles), "\n";
```

App::Grepl takes this a little further in that you can search for text (à la grep) in specific Perl structures. For example, you could look for the string "USENIX" in just the POD part of files with code like this:

```
use App::Grepl;

my $grepl = App::Grepl->new( {
    dir      => ".",
    look_for => [ 'pod' ],
    pattern  => 'USENIX'
} );
$grepl->search;
```

or from the command line:

```
grepl --dir . --pattern 'USENIX' --search pod
```

There are modules that can do less general scanning as well. For example, App::ScanPrereqs offers a nice CLI that can show all of the prerequisites for the code in a directory:

```
$ scan-prereqs .
+----------------------------------------+-------------+
| module                                 | version     |
+----------------------------------------+-------------+
| blib                                   | 1.01        |
| ExtUtils::MakeMaker                    | 0           |
| File::Find                             | 0           |
| File::Spec                             | 0           |
| Filename::Backup                       | 0           |
| IO::Handle                             | 0           |
| IPC::Open3                             | 0           |
| Log::ger                               | 0           |
| Module::CoreList                       | 0           |
| Perinci::CmdLine::Any                  | 0           |
| perl                                   | 5.010001    |
| Perl::PrereqScanner                    | 0           |
| Perl::PrereqScanner::Lite              | 0           |
| Perl::PrereqScanner::NotQuiteLite      | 0           |
| Pod::Coverage::TrustPod                | 0           |
| strict                                 | 0           |
| Test::More                             | 0           |
| Test::Pod                              | 1.41        |
| Test::Pod::Coverage                    | 1.08        |
| warnings                               | 0           |
+----------------------------------------+-------------+
```

It's a little meta, but that's what the module reports for itself when I run the scanner.

A similar tool comes from the Perl::MinimumVersion module which can output information like:

```
 ----------------------------------------------------------------------
| file                          | explicit  | syntax   | external |
| -------------------------------------------------------------------- |
| Makefile.PL                   | v5.10.1   | v5.10.0  | n/a      |
| bin/scan-prereqs              | v5.101    | v5.6.0   | n/a      |
| lib/App/ScanPrereqs.pm        | v5.10.1   | v5.10.0  | n/a      |
| t/00-compile.t                | v5.6.0    | v5.6.0   | n/a      |
| t/author-pod-coverage.t       | ~         | ~        | n/a      |
| t/author-pod-syntax.t         | ~         | v5.6.0   | n/a      |
| t/release-rinci.t             | ~         | ~        | n/a      |
| -------------------------------------------------------------------- |
| Minimum explicit version      : v5.10.1                             |
| Minimum syntax version        : v5.10.0                             |
| Minimum version of perl       : v5.10.1                             |
 ----------------------------------------------------------------------
```

If you want to go one step fancier, there's the App::PrereqGrapher module which makes pretty pictures like the one in Figure 1.

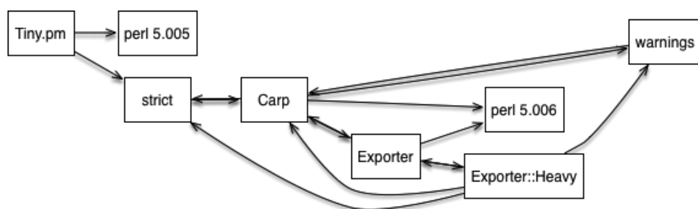

Figure 1: Output from App::PrereqGrapher

Make Your Code Better

Okay, the last set of PPI-powered modules to cover: those that help us write better code. Examples of this are the several modules like Log::Report::Extract::PerlPPI that make it easier to find, extract, and replace translatable strings in the code (e.g., error messages) for when you need to write code that will work in several languages.

Even more fun is software like Code::DRY, which calls itself "Cut-and-Paste-Detector for Perl code" and says, "The module's main purpose is to report repeated text fragments (typically Perl code) that could be considered for isolation and/or abstraction in order to reduce multiple copies of the same code (aka cut and paste code)." This can produce reports like (from the doc):

```
1 duplicate(s) found with a length of 8 (>= 2 lines) and 78 bytes
    reduced to complete lines:
1.  File: t/00_lowlevel.t in lines 57..64 (offsets 1467..1544)
2.  File: t/00_lowlevel.t in lines 74..81 (offsets 1865..1942)
===================
...<duplicated content>
===================
```

As a last and perhaps most useful module to visit, we return to something we've seen in past columns: Perl::Critic. Perl::Critic attempts to "critique Perl source code for best-practices." I think its doc says it best:

> Perl::Critic is an extensible framework for creating and applying coding standards to Perl source code. Essentially, it is a static source code analysis engine. Perl::Critic is distributed with a number of Perl::Critic::Policy modules that attempt to enforce various coding guidelines. Most Policy modules are based on Damian Conway's book Perl Best Practices. However, Perl::Critic is not limited to PBP and will even support Policies that contradict Conway. You can enable, disable, and customize those Policies through the Perl::Critic interface. You can also create new Policy modules that suit your own tastes.

When I write code, I tend to use a few tools to improve it, even as I'm writing. First, there's the internal checks of use strict. Then there is perltidy (which doesn't use PPI because it existed a couple of years before PPI came into being, but there are bug reports that suggest it should) for aligning and generally pretty printing the code. And finally, there's perlcritic, the command line tool that calls Perl::Critic on the code to look for best practices being violated by the code. For example, here's a run on the CSS::Tiny module file:

```
$ perlcritic Tiny.pm
Bareword file handle opened at line 27, column 2.  See pages
  202,204 of PBP.  (Severity: 5)
Don't modify $_ in list functions at line 53, column 16.  See
  page 114 of PBP.  (Severity: 5)
Expression form of "eval" at line 69, column 19.  See page
  161 of PBP.  (Severity: 5)
Expression form of "eval" at line 69, column 46.  See page
  161 of PBP.  (Severity: 5)
Bareword file handle opened at line 90, column 2.  See pages
  202,204 of PBP.  (Severity: 5)
```

Not everything it complains about is crucial to change, but it does occasionally point out flaws in the code that can and should be easily remedied.

With this tip, I'll say take care, and I'll see you next time.