Practical Perl Tools

From the Editor

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

The vast majority of the columns we've spent together so far have focused on how to improve your life within the bubble of the programming experience. We've looked at tools to make programming easier, more efficient, perhaps even a little more fun. For this column, let's try something different and bust out of our usual snow globe. We're going to look at three ways we can call out to Perl or Perl-based tools from within the editor we are using to improve our lives. So still Perl, but perhaps a little bit more at the periphery than before.

Reflowing and Reformatting Text

Once upon a time, Damian Conway, one of the leading lights of the Perl community, decided he didn't like any of the existing tools for reformatting and reflowing plain text. They couldn't handle bulleted lists, indentation, quoting, embedded structures like lists within quoted text, and so on. Or if they handled them, they didn't handle all of them simultaneously. As the documentation for the module we are about to see notes, if you take this sample text:

```
In comp.lang.perl.misc you wrote:
: > <CN = Clooless Noobie> writes:
: > CN> PERL sux because:
: > CN>    * It doesn't have a switch statement and you have to put $
: > CN>signs in front of everything
: > CN>    * There are too many OR operators: having |, || and 'or'
: > CN>operators is confusing
: > CN>    * VB rools, yeah!!!!!!!!!
: > CN> So anyway, how can I stop reloads on a web page?
: > CN> Email replies only, thanks - I don't read this newsgroup.
: >
: > Begone, sirrah! You are a pathetic, Bill-loving, microcephalic
: > script-infant.
: Sheesh, what's with this group - ask a question, get toasted! And how
: *dare* you accuse me of Ianuphilia!
```

and run it through the UNIX fmt tool (or even the Perl module Text::Wrap), you get this:

```
In comp.lang.perl.misc you wrote:  : > <CN = Clooless Noobie> writes:  : > CN>
PERL sux because:  : > CN>    * It doesn't have a switch statement and you
have to put $ : > CN>signs in front of everything : > CN>    * There are too
```

```
many OR operators: having |, || and 'or' : > CN>operators is confusing : > CN>
* VB rools, yeah!!!!!!!!!  : > CN> So anyway, how can I stop reloads on a web
page?  : > CN> Email replies only, thanks - I don't read this newsgroup.  : >
: > Begone, sirrah! You are a pathetic, Bill-loving, microcephalic : > script-
infant.  : Sheesh, what's with this group - ask a question, get toasted! And
how : *dare* you accuse me of Ianuphilia!
```

Not exactly an improvement. Conway decided to write a Perl module that would grok all of these things, and so the modules Text::Autoformat and Text::Reform were born. Text::Autoformat tries to determine the various structures found in text and then call Text::Reform to reformat them in a pleasing fashion. How pleasing? Here are the results when we run them on our sample text above:

```
In comp.lang.perl.misc you wrote:
: > <CN = Clooless Noobie> writes:
: > CN> PERL sux because:
: > CN>  *It doesn't have a switch statement and you
: > CN>    have to put $ signs in front of everything
: > CN>  *There are too many OR operators: having |, ||
: > CN>    and 'or' operators is confusing
: > CN>  *VB rools, yeah!!!!!!!!! So anyway, how can I
: > CN>    stop reloads on a web page? Email replies
: > CN>    only, thanks - I don't read this newsgroup.
: >
: > Begone, sirrah! You are a pathetic, Bill-loving,
: > microcephalic script-infant.
: Sheesh, what's with this group - ask a question, get toasted!
: And how *dare* you accuse me of Ianuphilia!
```

When Conway wrote the Text::Autoformat module, I believe his main desire was not to call it from within a larger Perl program, but, rather, to let it be used more handily from your favorite text editor of choice. To do that, you need to pass the text you want to reformat out of your text editor into an invocation of the Perl interpreter that looks like this:

```
perl -MText::Autoformat -e "{autoformat{all=>1,right=>75};}"
```

To break this down, it says:

- load the Text::Autoformat module
- then call the autoformat subroutine with the following options:

  all = 1 to instruct the module to reformat all of the text (vs. just the first paragraph)
  right = 75 to instruct the module to reformat things with a right margin of 75

I use that command all the time from within a TextMate macro (for example, on the very text you are reading), but you could map a key in vim to do the same thing:

```
map <C-J> !G perl -MText::Autoformat -e "{autoformat{all=>0,right=>75};}"<cr>
```

I apologize if this seems obvious, but if you attempt to run a command like this in vim (or another editor), and instead of returning nicely reformatted text, your original paragraph is replaced with something that looks like this:

```
Can't locate Text/Autoformat.pm in @INC (@INC contains:
/opt/local/lib/perl5/site_perl/5.14.1/darwin-multi-2level
```

```
/opt/local/lib/perl5/site_perl/5.14.1 /opt/local/lib/perl5/vendor_perl/5.14.1/
darwin-multi-2level
/opt/local/lib/perl5/vendor_perl/5.14.1
/opt/local/lib/perl5/5.14.1/darwin-multi-2level
/opt/local/lib/perl5/5.14.1
/opt/local/lib/perl5/site_perl
/opt/local/lib/perl5/vendor_perl/5.14.0
/opt/local/lib/perl5/vendor_perl .).
BEGIN failed--compilation aborted.
```

it means that you will need to install the Text::Autoformat module before you can proceed. For those of you who have multiple versions of Perl installed on your machine (e.g., because you have both the Perl that ships with the system and the one you installed through MacPorts/Homebrew/Fink), sometimes you will find you will get this message because your editor configuration is picking up the wrong Perl (the one without Text::Autoformat installed in its @INC) from your path. An easy fix is to change the command being run to include a full path to the right Perl interpreter (e.g., /opt/local/bin/perl -MText::Autoformat...).

Tidy Your Lousy Code

Although we are not actually doing any programming in this column, this seems like a natural place to point out two other tools that can be called from an editor to improve the programming process. Both of these have made at least one appearance in this column, but I love them too much not to mention them again: Perl::Tidy and Perl::Critic. Both of these things are modules designed to work on Perl code and both come with a script that runs on the command line.

In the case of Perl::Tidy, or, more precisely, when using its accompanying command-line perltidy, code can get read in from stdin and printed out again in a much, much prettier form to stdout. As a demonstration, here's some sample code found embedded in the Perl::Tidy documentation:

```
use strict;
my @editors=('Emacs', 'Vi  '); my $rand = rand();
print "A poll of 10 random programmers gave these results:\n";
foreach(0..10) {
my $i=int ($rand+rand());
print " $editors[$i] users are from Venus" . ", " .
"$editors[1-$i] users are from Mars" .
"\n";
```

If I run it through perltidy from my editor (the command I call is `perltidy -st -q $FILENAME`, but for vi we could use just `:%!perltidy`) using some defaults (more on that in a moment), I get:

```
use strict;
my @editors = ( 'Emacs', 'Vi  ' );
my $rand = rand();
print "A poll of 10 random programmers gave these results:\n";
foreach ( 0 .. 10 ) {
        my $i = int( $rand + rand() );
        print " $editors[$i] users are from Venus" . ", "
          . "$editors[1-$i] users are from Mars" . "\n";
}
```

If you look at the difference between the two, there are lots of little cleanups going on (e.g., the space around arguments in parenthesis). I realize it is a particularly geeky thing to say this, but when I start with code that looks like this:

```
my %a = (
        $a => 1,
        $apple => 2,
        $bigapple => 3,
        $verylargeapple => 'new york',);
```

and I turn it into this using a single keystroke:

```
my %a = (
    $a              => 1,
    $apple          => 2,
    $bigapple       => 3,
    $verylargeapple => 'new york',
);
```

such that the arrows all line up it is deeply satisfying. If you've noticed that all, or at least most, of the arrows in this column have lined up over the years, that's not my doing. I have Perl::Tidy to thank. One last note before I move on to Perl::Critic: I mentioned running perltidy with defaults. Perl::Tidy has a ton of configurable options. Don't like it if your arrows line up? (Of course you do!) Prefer to leave a closing parenthesis at the end of a line of code without wrapping it as above? All of these things can be set as options. By default, if you create a .perltidyrc, Perl::Tidy will attempt to read it to set your favorite options. At the moment I use the following .perltidyrc file, which was recommended in Conway's excellent *Perl Best Practices*:

```
# PBP .perltidyrc file

-l=78       # Max line width is 78 cols
-i=4        # Indent level is 4 cols
-ci=4       # Continuation indent is 4 cols
-st         # Output to STDOUT
-se         # Errors to STDERR
-vt=2       # Maximal vertical tightness
-cti=0      # No extra indentation for closing brackets
-pt=1       # Medium parenthesis tightness
-bt=1       # Medium brace tightness
-sbt=1      # Medium square bracket tightness
-bbt=1      # Medium block brace tightness
-nsfs       # No space before semicolons
-nolq       # Don't outdent long quoted strings
-wbb="% + - * / x != == >= <= =~ < > | & **= += *= &= <<= &&= -= /= |=+ >>=
||= .= %= ^= x="  # Break before all operators
```

(The last line, beginning `-wbb`, should be all one line.)

If you don't feel like setting up a .perltidyrc as I did many moons ago when I first read the book, you can now use a -pbp argument to perltidy and it will set these parameters for you.

The second Perl-based command I mentioned above was perlcritic, installed as part of the Perl::Critic module. The mention of *Perl Best Practices* above is a good

segue because that book basically helped spawn Perl::Critic. Perl::Critic is meant to analyze Perl code and determine if it is complying with certain policies meant to enforce coding best practices. The original rules were based on the Conway book, but more have been added over time. Perl::Critic also lets you use add-on modules to add all sorts of different policies to the checking process. When it finds anything that violates any of these rules it will spit out warning messages. If you would like to see examples of these messages, take a peek back at the December 2009 column where I first mentioned both Perl::Critic and Perl::Tidy.

These error messages have a similar form to those you might expect to see emitted from another language's compiler. As a result, most of the editors that offer perlcritic integration do so using a variation of their already existing functionality that lets a user try to compile code from within the editor (jumping to the lines with errors if any are found). There are add-on packages for a number of the more popular editors/IDEs, including vim, Emacs, Komodo, Eclipse (within the Eclipse Perl Integration project), BBEdit, Padre, and so on.

And You Thought Grep Was Cool

For the last tool that we are going to see which you can integrate into your editor, I want to introduce you to three little letters that may significantly improve how you find things in your ever-increasing mountain of data: ack. ack is a grep-ish utility by Andy Lester. Like grep, it was designed to help you find data within files. It is just a bit smarter (okay, a lot smarter). I don't think I can do any better describing why you might want to use it than to quote from the documentation:

Top 10 reasons to use ack instead of grep.

1. It's blazingly fast because it only searches the stuff you want searched.
2. ack is pure Perl, so it runs on Windows just fine. It has no dependencies other than Perl 5.
3. The standalone version uses no non-standard modules, so you can put it in your ~/bin without fear.
4. Searches recursively through directories by default, while ignoring .svn, CVS, and other VCS directories.
   Which would you rather type?
   ```
   $ grep pattern $(find . -type f | grep -v '\.svn')
   $ ack pattern
   ```
5. ack ignores most of the crap you don't want to search:
   o VCS directories
   o blib, the Perl build directory
   o backup files like foo~ and #foo#
   o binary files, core dumps, etc.
6. Ignoring .svn directories means that ack is faster than grep for searching through trees.
7. Lets you specify file types to search, as in --perl or --nohtml. Which would you rather type?
   ```
   $ grep pattern $(find . -name '*.pl' -or -name '*.pm' -or -name '*.pod' \
       | grep -v .svn)
   $ ack --perl pattern
   ```
   Note that ack's `--perl` also checks the shebang lines of files without suffixes, which the find command will not.

8. File-filtering capabilities usable without searching with ack -f. This lets you create lists of files of a given type.
   ```
   $ ack -f --perl > all-perl-files
   ```
9. Color highlighting of search results.
10. Uses real Perl regular expressions, not a GNU subset.
11. Allows you to specify output using Perl's special variables. To find all #include files in C programs:
    ```
    ack --cc '#include\s+<(.*)>' --output '$1' -h
    ```
12. Many command-line switches are the same as in GNU grep:
    `-w` does word-only searching
    `-c` shows counts per file of matches
    `-l` gives the filename instead of matching lines
    etc.
13. Command name is 25% fewer characters to type! Save days of free-time! Heck, it's 50% shorter compared to `grep -r`.

So there you go, 13 of the top 10 reasons why ack may replace grep as a command you type on a regular basis. TextMate, Vim, Emacs and other add-ons let you do things like conduct fast searches from within the editor and then jump to the places in the files where your search text was found.

With that high note, I think we'll end our exploration of Perl utilities that can be called from an editor. If you have a particularly cool example of this sort of thing that you use all the time, please write me a note so I can include it in a future column. Take care, and I'll see you next time.