DAVID N. BLANK-EDELMAN

practical Perl tools: let me help you get regular

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.

dnb@ccs.neu.edu

DIFFERENT PEOPLE HAVE VERY DIFFERent opinions of Perl as a language, but I think you might find a healthy majority who agree on the value of the regular expression engine it introduced. You probably get no better vote of confidence in the language world than to have a feature and sometimes its syntax copied almost verbatim. Python, Ruby, Java, .NET, the list goes on, all support some version of Perl-ish regular expressions. There's even a C library called "PCRE—Perl Compatible Regular Expressions," found at http://www.pcre.org/, which can bring that power to your program. Perl 6 aims to introduce further innovation on this front (see http://perlcabal.org/syn/ for more details).

In the meantime, there are a number of ways to make the existing support even more useful and powerful. This column will show you a few Perl regular expression–related modules in that vein.

Regexp Porn

If you want to get really serious about regular expressions, and I'd like to suggest that you do because they are often key to Perl programs, there's one book you need to read. Go buy Jeffrey Friedl's *Mastering Regular Expressions*. I'm not saying this just to shill for a fellow O'Reilly author. The book's a little short on plot and character development, but it is truly the best text on the subject. It will improve your ability to write and understand regular expressions in a number of languages and tools besides Perl (such as awk/grep).

Don't Write Your Own Regular Expressions

Regular readers of this column are familiar with this shtick where I say something is the best thing since split() bread in the first breath and then tell you not to use it in the second, unless . . .

Here's the latest one: don't write your own regular expressions for common items. First check to make sure it isn't already included in the Regexp::Common family of modules. Lots and lots of effort by smart people (certainly smarter than me) has gone into creating a collection of robust, reusable regular expressions for a whole slew of things. In just the Regexp::Common distribution itself, you can find regular expressions for matching:

- credit card numbers
- Social-Economical Numbers (e.g., social security numbers)
- URIs of all sorts
- strings with balanced delimiters
- lists
- IP addresses
- numbers
- profanity
- whitespace
- postal codes

Using Regexp::Common is pretty simple. First you load the module and specify which subset of regular expressions you'd like to use:

```
use Regexp::Common qw /net/;
```

Regexp::Common will then populate a tied hash called %RE that will be filled with the patterns you need. We can then use that hash in the regular expression match of our choice, like so:

```
/^$RE{net}{IPv4}$/ and print "$_ is a dotted decimal IP address\n";
```

The module uses further sub-hash syntax to select more specific options, such as:

```
/^$RE{net}{IPv4}{oct}{-sep => ':'}$/ # matches colon-separated octal IP addresses
```

Many of the pattern sets take an option -keep, as in:

```
$contains_ipaddr =~ /$RE{net}{IPv4}{-keep}/;
```

The -keep option lets you capture all or parts of the match. In this last example, $1 gets set to the full match and $2 through $5 store the components of the match. For example, if $contains_ipaddr was the string 'Your address is 192.168.0.5', $1 would contain 192.168.0.5, $2 would be 192, $3 would be 168, and so on.

And It's a Tie

The following idea is either incredibly useful or it is just a parlor trick, depending on your specific needs. I say that so you'll use it with caution. Mutating the standard hash semantics always makes your scripts a little harder to maintain, because it defies the usual expectations of the code reader. But perhaps it will be worth it to you.

There exist two modules, Tie::RegexpHash and Tie::Hash::Regex, that bring some regular expression magic to your hash data structures. The former lets you write code to store a regular expression as the hash key instead of a scalar. Here's the example from the documentation:

```
use Tie::RegexpHash;

my %hash;

tie %hash, 'Tie::RegexpHash';

$hash{ qr/^5(\s+|-)?gal(\.|lons?)?/i } = '5-GAL';

$hash{'5 gal'};      # returns "5-GAL"
$hash{'5GAL'};       # returns "5-GAL"
$hash{'5  gallon'};  # also returns "5-GAL"
```

Tie::Hash::Regex takes this idea in a different direction. Instead of storing the regular expression as the key, as we just saw, Tie::Hash::Regex first tries the usual exact match during a key lookup. If that fails, it then attempts a regular expression match to find that key. From its documentation:

```
use Tie::Hash::Regex;
my %h;

tie %h, 'Tie::Hash::Regex';

$h{key}      = 'value';
$h{key2}     = 'another value';
$h{stuff}    = 'something else';

print $h{key};   # prints 'value'
print $h{2};     # prints 'another value'
print $h{'^s'};  # prints 'something else'
```

Muchos Matching

There is a class of problems you are bound to run into at some point that entails having to run a (potentially large) number of matches over the same text. For example, if you need to find out if a mail message contains a certain set of keywords, you may find yourself initially writing code that looks like this:

```
my @keywords = qw( urgent acute critical dire exigent pressing serious grave );

foreach my $keyword in (@keywords){
  do_something() if $text =~ /$keyword/;
}
```

If you have a large set of keywords or a large set of repetitions, this gets old/inefficient very quickly, because you are spinning up the regexp engine and forcing it to traipse through the same text over and over again. One standard way to improve on this method is to use regular expression alternation and do a single match on the text, as in:

```
my @keywords = qw( urgent acute critical dire exigent pressing serious grave );
# quotemeta is used to neuter regexp chars in the keyword list
my $match = join '|', map { quotemeta } @keywords;
do_something() if $text =~ /$match/;
```

This is far more efficient even (and especially) if the keyword list is very large. But we can do better than this. The Text::Match::FastAlternatives module is meant to handle exactly this case. It will analyze your list and create a "matcher" which you can use on the text you are checking:

```
use Text::Match::FastAlternatives;
my @keywords = qw( urgent acute critical dire exigent pressing serious grave );
my $keymatch = Text::Match::FastAlternatives->new(@keywords);
do_something() if $keymatch->match($text);
```

People who follow the latest developments in Perl might say at this point, "Wait! But what about the trie-based optimization improvements in 5.10? Don't they make the regexp alternative code we just saw fast too?" It is an excellent question, albeit incomprehensible for those people who don't follow the latest developments in Perl. One of the cool things the Perl developers added in the 5.10 release was some modifications to the regular expression engine that would automatically handle alternation cases like this using a more efficient internal representation. If you use 5.10 and above, you get this speedup for free. Text::Match::FastAlternatives is actually faster than the improved regular expression engine, so it is still potentially the best option for even 5.10+ users. See the Text::Match::FastAlternatives documentation for more details.

But what if we're dealing with something a little more complicated than a list of keywords? What if, instead, we had a set of regular expressions we needed to check against a piece of text? If you need something more in that direction, you would be well served to look at the Regexp::Assemble module. Its documentation says:

> Regexp::Assemble takes an arbitrary number of regular expressions and assembles them into a single regular expression (or RE) that matches all that the individual REs match.
>
> As a result, instead of having a large list of expressions to loop over, a target string only needs to be tested against one expression. This is interesting when you have several thousand patterns to deal with. Serious effort is made to produce the smallest pattern possible.
>
> It is also possible to track the original patterns, so that you can determine which, among the source patterns that form the assembled pattern, was the one that caused the match to occur.

The example from the documentation looks like this:

```
use Regexp::Assemble;

my $ra = Regexp::Assemble->new;
$ra->add( 'ab+c' );
$ra->add( 'ab+-' );
$ra->add( 'a\w\d+' );
$ra->add( 'a\d+' );
print $ra->re; # prints a(?:\w?\d+|b+[-c])
```

Turning on pattern tracking (so you can figure out which regexp matched) is a matter of adding a `track => 1` option to the new() call above and using the source() method. There is one fiddly bit related to pattern tracking and security for people running versions of Perl earlier than 5.10, so be sure to read the documentation before you start to use this feature. When you do consult the docs, you'll discover that the module has a fairly rich set of features. For example, it can read the list of patterns to assemble directly from a file using add_file(). It can also return the assembled pattern as a string so you can store it for later use.

One last Regexp::Assemble tip to mention before moving on to our last module of this column: Regexp::Assemble does a good job of creating "the smallest pattern possible," but another author has written an add-on module called Regexp::Assemble::Compressed which purports to "assemble more compressed regular expressions." It is a subclass of Regexp::Assemble, so you would use it in the same way as its parent module. I haven't had a chance to test it, but you might want to give it a look if smaller results would be helpful.

Do It All at Once

So far we've only talked about using regular expressions for matching purposes. For the last module I'd like to mention, let's consider the other main use of regular expressions: substitution. One cool module you may not have heard of is Regexp::Subst::Parallel, which claims to "safely perform multiple substitutions in parallel." Let's take a simple example of how this could be useful. Imagine we had to change the gender of the English words in a piece of text. If we wanted to do this by running a set of regular expressions against the text, we'd quickly run into trouble if our code looked like this:

```
$text =~ s/\bshe\b/he/;
$text =~ s/\bhe\b/she/;
$text =~ s/\bher\b/him/;
$text =~ s/\bhim\b/her/;
$text =~ s/\bfemale\b/male/;
$text =~ s/\bmale\b/female/;
```

. . . and so on

Ordinarily we'd be forced to switch to a different parsing and transform approach, but Regexp::Subst::Parallel lets us write code that will do the intended substitutions:

```
use Regexp::Subst::Parallel;
my $text = subst($text,
   qr/\bshe\b/       => 'he',
   qr/\bhe\b/        => 'she',
   qr/\bher\b/       => 'him',
   qr/\bhim\b/       => 'her',
   qr/\bfemale\b/    => 'male',
   qr/\bmale\b/      => 'female',
);
```

Hopefully, after this set of tips you are feeling more regular already. Take care, and I'll see you next time.