

BEST OF THE PERL JOURNAL

Games, Diversions & Perl Culture



O'REILLY®

Edited by Jon Orwant

BEST OF THE PERL JOURNAL

Games, Diversions, and Perl Culture

Related titles from O'Reilly

The Best of the Perl Journal Series

Volume 1: Computer Science and Perl Programming

Volume 2: Web, Graphics, and Perl/Tk

Volume 3: Games, Diversions, and Perl Culture

Other Perl titles

Advanced Perl Programming

Beginning Perl for Bioinformatics

CGI Programming with Perl

Embedding Perl in HTML with Mason

Learning Perl on Win32 Systems

Learning Perl

Mastering Algorithms with Perl

Mastering Perl/Tk

Mastering Regular Expressions

Perl & LWP

Perl & XML

Perl Cookbook

Perl for System Administration

Perl for Web Site Management

Perl Graphics Programming

Perl in a Nutshell

Perl Pocket Reference

Perl/Tk Pocket Reference

Practical mod_perl

Programming Perl, 3rd Edition

Programming the Perl DBI

Programming Web Services with Perl

Regular Expression Pocket Reference

Writing Apache Modules with Perl and C

Also available

The Perl CD Bookshelf

BEST OF THE PERL JOURNAL

Games, Diversions, and Perl Culture

Edited by Jon Orwant

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

Searching for Rhymes with Perl

Sean M. Burke

La poésie doit être faite par tous.

Poetry is for everyone to make.

—Lautréamont (Isidore Ducasse, 1846–1870)

Wherever I go, people always come up to me and say “Sean, you gotta help me—I need to find a three-syllable word that rhymes with toad.” And my answer is always the same; I always say “Well, we’re going to have to pull out the Perl for this one!”

Because, while TPJ articles constantly demonstrate that Perl is good at everything from designing sundials to peppering IRC with Eliza bots, one thing that it’s *really* good at is making short little programs for searching text. And that’s what this article is about—how to search text (specifically wordlists or pronunciation databases) for rhymes of various kinds.

Where to Look

If this article were about rhyming in Spanish, Italian, or Finnish, it’d be a whole lot shorter! Because for the most part, the way something is spelled in these languages tells you pretty well how to pronounce it; ending with the same letters may not be exactly the same thing as rhyming, but often you can start with the spelling and apply some trivial string replacement operations to get a phonetic form that can be searched for the presence of a rhyme. This can work even with French, where (for the most part) spelling tells you pronunciation, even though the pronunciation won’t tell you the spelling.

However, English isn’t that kind of language—not only does the English pronunciation of a word not tell you how to spell it, its spelling doesn’t tell you how to pronounce it. But luckily, lexicons exist that are basically simple databases, associating the normal written form of a word with some representation of its pronunciation. One of my favorite lexicons (partly because it’s free!) is Moby Pronunciator, available

at <http://www.dcs.shef.ac.uk/research/ilash/Moby/> for the downloading. It consists of about 177,000 entries, one word to a line, that look like this:

```
...
accipitrine      /&/k's/I/p/I/tr/I/n
Accius           '/&/k/S//i//@/s
acclaim          /@/'kl/eI/m
acclamation      ,/&/kl/@/'m/eI//S//@/n
acclamation_medal ,/&/kl/@/'m/eI//S//@/n_'m/E/d/-/l
acclamatory      /@/'kl/&/m/@/,t/oU/r/i/
acclimate        /@/'kl/aI/m/I/t
acclimation      ,/&/kl/@/'m/eI//S//@/n
acclimation_fever ,/&/kl/@/'m/eI//S//@/n_'f/i/v/@/r
acclimatise      /@/'kl/aI/m/@/,t/aI/z
acclimatize      /@/'kl/aI/m/@/,t/aI/z
acclivity        /@/'kl/I/v/I/t/i/
...
```

Ignoring the meanings of these symbols, you can see that (as the *README* will tell you), the format of each line is the word (or underscore-separated multiword phrase, like “acclamation_medal”), then a space, then the phonetic notation. What the slashes mean (and why there isn’t one between the /k/ and /l/) is something I’m unsure of. But I am sure that these slashes are annoying, since they get in the way of me trying to search. I have to remember to stick them in my search patterns, and I always worry that I stuck in one too many. The same goes for the commas and apostrophes, which indicate stress—and when I’m looking for a rhyme, I may not care about stress.

Preparing the Data

So the first thing to do, whether it’s for the Moby Pronunciator wordlist or for any other wordlist you choose, is to strip out the parts you don’t want, take what’s left, and format it the way you like. Here, we can do that by deleting certain tokens in the pronunciation part:

- Slashes (used to separate phonemes?)
- Spaces and underscores (used to separate words)
- Apostrophes (used to precede syllables with primary stress)
- Commas (used to precede syllables with secondary stress)

Since these tokens are all single characters, we can delete them by just applying a `tr` operator to slashes, spaces, underscores, commas, and apostrophes. We use the `d` switch (“d” for delete):

```
tr\/_,'//d;
```

Personally, I find it disconcerting to have the backslash-escaped slash in there, so I tend to use different delimiters, like matching angle brackets:

```
tr</_,'><d;
```

Either way, you can build this into a program that reads the Moby Pronunciator database:

```
#!/usr/bin/perl
# mpron_convert -- Turn the mobypron.unc program into the mpron.dat
#                  format that we'll use.

use strict;
@ARGV = 'mobypron.unc' unless @ARGV;
my ($word, $pron, $meter, $next_stress_flag);
my $Debug = 0;
# $/ = "\cm"; # May be necessary

open(OUT, ">mpron.dat");

while (<>) {
    chomp;
    ($word, $pron) = split(' ', $_, 2);
    next unless $pron;

    $meter = '';
    $next_stress_flag = '0';
    foreach my $x ($pron =~ m<[',,][-\&yYaeiouAEIOU\@]+>g) {
        if ($x eq ',') {
            $next_stress_flag = '2'; # secondary stress
            next;
        } elsif ($x eq "'") {
            $next_stress_flag = '1'; # primary stress
            next;
        }
        $meter .= $next_stress_flag;
        $next_stress_flag = '0';
    }

    # So "stressless" one-syllable words all get stress. Also needed
    # for multiword phrases mode of monosyllabic words, like "base_load".
    $meter =~ tr/0/2/ if $meter =~ m/^0+$/s;

    # Remove stress marks, word separators, and the mystery slashes
    $pron =~ tr<', /_><>d;

    sleep(0), printf "%10s %-20s %s\n", $meter, $word, $pron if $Debug;
    print OUT join("\t", $word, $meter, $pron), "\n";
    last if $Debug and $. > 1000;
}
close(OUT);
exit;
```

Now, to search this database for rhymes (or any other phonetic information), there are two ways to go about it: use the code above, and once you've modified \$pron, search it for a pattern; or write \$word and the modified \$pron to a file, and then grep that file.

The benefit of the former is simplicity, but the benefit of the latter is efficiency—no need to constantly `chomp`, `split`, and `tr` for each line. Now, normally I say that program (as opposed to programmer) efficiency is overvalued in programming. But in this case, the Moby wordlist is so very large that the waste of the first approach is significant. So I say the second approach the one to take. We can save each line’s `$word` and `$pron` values to a file called *mpron.dat*, like so:

```
open(IN, '<mobypron.unc') or die $!;
open(OUT, '>mpron.dat') or die $!;
while (<IN>) {
    chomp;
    ($word, $pron) = split(' ', $_);
    $pron =~ tr</ _,'><>d;
    print OUT $word, "\t", $pron, "\n";      # Tab makes a nice delimiter
}
```

The resulting file, *mpron.dat*, begins like this:

```
...
accipitrine      &ksIpItrIn
Accius           &kSi@s
acclaim         @kleIm
acclamation     &k1@meIS@n
acclamation_medal &k1@meIS@nmEd-1
acclamatory     @k1&m@toUri
acclimate       @klaImIt
acclimation     &k1@meIS@n
acclimation_fever &k1@meIS@nfiv@r
acclimatise     @klaIm@taIz
acclimatize     @klaIm@taIz
acclivity       @klIvIti
...
```

Searching the Prepared Data

With *mpron.dat* prepared, we can `grep` it for whatever pattern we want in the pronunciation. Suppose we’re still after a three-syllable word that rhymes with “toad.” The idea of rhyme in English is a pretty straightforward matter: if two words rhyme, they end in the same sounds (generally the last vowel and any consonants following it). If I were quite familiar with the phonetic notation for Moby Pronunciator (or whatever alternate pronunciation database you might use), I could, off the top of my head, say how to represent the sound “-oad” from “toad.” However, I’ve never bothered, since it’s so easy to just look up the word you want to rhyme with, and see how it’s represented:

```
% grep '^toad' mpron.dat
toad          toUd
toad's-mouth  toUdzmouT
toadeater     toUdit@r
toadfish      toUdfIS
toadflax      toUdf1&ks
```

```

toadstone      toUdstoUn
toadstool      toUdstul
toadstool_disease toUdstuldIziz
toady          toUdi

```

oUd it is!

```

% grep 'oUd$' mpron.dat
abode          @boUd
access_road    &ksEsroUd
acnode         &knoUd
Aeolian_mode   ioUli@nmoUd
alamode        &l@moUd
Alexis_Claude  ALEksikloUd
all-hallowed   Olh&loUd
anchor_rod     &Nk@rroUd

```

and 281 other matches, ending with `zip_code` (`zIpkouD`). There are so many because we haven't limited our search to three-syllable words. So how do we do that?

Counting Syllables

As with most models of syllables in most languages, an English syllable is basically a vowel sound with some number of consonants before and after it. Now, actually settling on what consonants go with what vowels is a sticky subject (is `rostrum` `rAs-tr@m` or `rA-str@m?`), but since all we want to do now is count the syllables, we merely need to count the number of vowel sounds.

You've seen that some vowel sounds, like the long "o" sound in "toad," are represented by a pair of ASCII characters, `oU`. That means that we can't simply count the number of vowel characters in the pronunciation string, because then `oU` would count as two. We could count the number of times we find a sequence of some number of vowel characters, but that would match only once in each of these two-syllable words:

```

eon   i@n   (one sequence: "i@")
Noah  noU@  (one sequence: "oU@")

```

(The `@` character here represents the "uh" sound in unstressed syllables.) However, if we go back to the format of the original Moby Pronunciator file (as opposed to our cooked `mpron.dat` file), we see that those slashes can do us some good:

```

eon  '/i//@/n
Noah 'n/oU//@/

```

One consistency is that there's at least one slash between vowels in different syllables. So where the vowels in the two syllables in "Noah" in our prepared file run together, they are still separate in the original file. This means that if we start with the original form of the pronunciation entry, and count the number of occurrences of sequences of vowel characters:

```

eon  '/i//@/n   (two sequences: "i", "@")
Noah 'n/oU//@/ (two sequences: "oU", "@")

```

then we get a correct syllable count. All we need to know now is what “vowel characters” means. The Moby Pronunciator documentation says that it uses all of the following characters (or sequences of them):

```
a e i u o A E I O U y Y & @ -
```

We can count syllables by seeing how often this matches:

```
m/[-\&yAeiouAEIOU\@]+/g
```

We can simply write that into our program that produces *mpron.dat*, by matching it against `$pron` before we delete the slashes.

Coping with (Syllabic) Stress

Let’s say this three-syllable word to rhyme with “toad” is needed not merely for its austere artistic potency, but because we need it to complete our Baudelairean opus magnum, which ends:

```
I chanced upon a lovely toad,  
It gleamed and danced like ____!
```

DUM-duh-DUM. In technical terms, you’ve got eight-syllable lines, with this metrical pattern (where slash means stressed, and underscore means unstressed):

```
I chanced upon a lovely toad,  
_ / _ / _ / _ /  
  
It gleamed and danced like ____!  
_ / _ / _ / _ /
```

So not only do you want the word you’re after to have three syllables, but you want it to have a particular stress pattern. A word like:

```
electrode  
_ / _
```

has the exactly the wrong stress pattern, even though it *is* three syllables long, and rhymes with “toad.” (That’s aside from “danced like electrode” being a bit ungrammatical—hey, this is *poetry!*) Since we’re about to rebuild *mpron.dat* to contain each entry’s syllable count, we might as well note syllable stress patterns too.

Stress is noted in the original data file with commas and apostrophes:

```
acclamatory /@/'kl/&/m/@/,t/oU/r/i/  
acclimate /@/'kl/aI/m/I/t  
acclimation ,/&/kl/@/'m/eI//S//@/n
```

Unfortunately, the apostrophe (primary stress) or comma (secondary stress) that marks the following syllable as stressed isn’t right before the vowel that we’d match in order to count that syllable. If it were, we could come up with a single regex that would match any vowel cluster as well as its stress notation:

```
m/([, '])[-\&yAeiouAEIOU\@]+/g
```

Each time this matched, we could just look in \$1 to see what kind of stress the syllable had. However, that’s not the way the data is. As it is, we have to match the stress marks wherever they are, and then set a flag so that the following syllable will be marked as stressed (and in the absence of the flag, marked unstressed). We can combine this with the syllable counter that works its way through the word, based on this regex:

```
m/[',,][-\&yYaeiouAEIOU\@]+/g
```

We can work this into part of the main loop for our converter program, so that it can cook up a field representing the meter of each word for each line in *mpron.dat*.

(Usually “meter” is used for talking about the consistent stress pattern of whole lines of poetry—but I’m using it here to refer to just the stress pattern of particular words, mostly because \$meter is easier to type than \$metrical_structure or \$stress_pattern!)

```
while (<IN>) {
  chomp;
  ($word, $pron) = split(' ', $_);

  # This is where we'll stack up a '0', '1', or '2', one for each
  # vowel-character-group in this word, as seen in $pron
  $meter = '';
  $next_stress_flag = '0'; # Initial value

  # Loop over the vowels and accent marks in $pron before we change it
  foreach my $x ($pron =~ m/[',,][-\&yYaeiouAEIOU\@]+/g) {
    if ($x eq ',') { # Secondary stress
      $next_stress_flag = '2';
    } elsif ($x eq "'") { # Primary stress
      $next_stress_flag = '1';
    } else { # It's a vowel
      $meter .= $next_stress_flag; # Note it as another syllable
      $next_stress_flag = '0'; # Clear flag for next time
    }
  }

  $pron =~ tr</ _, '><>d; # Okay, NOW we can change it

  print OUT join("\t", $word, $pron, $meter), "\n";
}
```

The whole business of \$next_stress_flag being set in one iteration for use in the next may not make much sense. Here’s a rough English summary of how \$meter is devised for each word:

Each time a vowel-character cluster is found in this word’s \$pron, add a character to \$meter representing the stress level of this syllable. If this syllable was preceded by an apostrophe, note this syllable as “1”. If it was preceded by a comma, note this syllable as a “2”. Otherwise, note it as a “0”.

What this gives us is a *mpron.dat* file like this:

```
accipitrine      0100    &ksIpItrIn
Accius           100     &kSi@s
acclaim         01      @kleIm
acclamation      2010    &kl@meIS@n
acclamation_medal 201010  &kl@meIS@nmEd-1
acclamatory     01020   @kl&m@toUri
acclimate       010     @klaImIt
acclimation     2010    &kl@meIS@n
acclimation_fever 201010  &kl@meIS@nfiv@r
acclimatise     0102    @klaIm@taIz
acclimatize     0102    @klaIm@taIz
acclivity       0100    @klIvIti
```

There are three tab-separated fields to each line. If we merely want to know the number of syllables in a word, we just count the number of characters in the second field. But if we want to know more (say, to stipulate the stress pattern of those syllables), we have the data to do that, too.

Now recall that we’re looking for a word that meets these criteria:

- Rhymes with “toad”
- Has three syllables
- Has the stress pattern / _ / (stressed, unstressed, stressed)

We figured out that we could formalize “rhymes with toad” as a matter of matching the regex `m/oUd$/`. But when it comes to matching the stress pattern of the word, we’re thinking in terms of stressed and unstressed—a two-term distinction—but the data we’ve got (from the Moby Pronunciator, but most pronunciation databases do it this way) represents stress in terms of primary stress, secondary stress, and unstressed—a three-term distinction.

After some experimentation, I settled on this as the best way to reconcile these two systems: When I say “stressed,” I mean having primary (“1”) or secondary (“2”) stress. When I say “unstressed,” I mean having secondary (“2”) stress, or no stress (“0”).

So we can now formulate “I want the word to go DUM-duh-DUM” as a matter of its meter string matching the regex `/[12][02][12]/`.

Now, to pull off a search with these criteria, we could go back to our command-line `grep` pattern:

```
% grep 'oUd$' mpron.dat
```

and amend it with:

```
% grep 'oUd$' mpron.dat | grep '[12][02][12]' | more
```

But all this grepping is getting rather cumbersome, and won't work terribly nicely with increasingly complex search patterns. In the end, it'd be so much simpler if we just wrote a custom (and therefore customizable!) search tool in Perl.

A Simple mpron Searcher

Since there are three fields in our database, it makes sense to be able to provide search criteria for any of those three fields. And regular expressions are the most powerful way to express search patterns. Each of our searches could be thought of as specified by three regular expressions: the first to match the spelling form of the word (probably not your primary interest, but it could be useful), the second to match the meter of the word, and the third to match the pronunciation of the word.

I figure this search tool (which we might as well call `mpron`) could have this command-line syntax:

```
% mpron spelling_re stress_re pron_re
```

with the assumption that if we stipulate nothing for one or any of these regexes, then we're not imposing any limitation on that field. So "rhymes with toad" would be just a matter of:

```
% mpron '' '' 'ouD$'
```

We can implement this simply with a program like this:

```
($word_re, $meter_re, $pron_re) = @ARGV[0,1,2];
open(IN, '<mpron.dat') or die "Can't read-open mpron.dat: $!";

print "# Word RE: <$word_re> Meter RE: <$meter_re> Pron RE: <$pron_re>\n";

# Loop over every line
while (<IN>) {
    chomp;
    print $_, "\n" # the matching line
    if (...it meets all our criteria...)
        ...then do something...
}
```

Now, how do we formalize "it meets all our criteria"? We could just say:

```
if ($bits[0] =~ m/$word_re/oi      # /i means case insensitivity
    && $bits[1] =~ m/$meter_re/o    # and /o means "compile only once"
    && $bits[2] = m/$pron_re/o)
```

However, that makes sense only if we've provided all three criteria. We don't want to bother trying to match an element of `@bits` against the contents of a variable like `$meter_re` if there's nothing in that variable (that is, if the search criterion it corresponds to is no criterion at all).

For each kind of test, we want the comparison to succeed if there was a criterion and it matches, or if there was no search criterion at all. In terms of logical operators, this is an “or” relationship. Specifically,

```
pass this test if:
    there was no criterion specified OR I pass the criterion
```

Passing each of the three criteria is a matter of matching the appropriate regex, as with:

```
$bits[1] =~ m/$meter_re/o
```

As for how to express “there was no criterion specified,” we can simply test the string length of the variable containing the regex:

```
!length($meter_re)
```

This is true when `$meter_re` is empty. Put it all together and you get:

```
!length($meter_re) || $bits[1] =~ m/$meter_re/o
```

For all the tests put together:

```
print $_, "\n" if (!length($word_re) || $bits[0] =~ m/$word_re/oi)
                && (!length($meter_re) || $bits[1] =~ m/$meter_re/o)
                && (!length($pron_re) || $bits[2] =~ m/$pron_re/o);
```

Incidentally, you can use the and operator (the low-precedence variant of `&&`) to minimize the number of parentheses:

```
print $_, "\n" if !length($word_re) || $bits[0] =~ m/$word_re/oi
                and !length($meter_re) || $bits[1] =~ m/$meter_re/o
                and !length($pron_re) || $bits[2] =~ m/$pron_re/o;
```

And that’s all we’ve got to do for a fully featured program that searches any of the fields in *mpron.dat*.

Let’s put it to work. Our command line for “find three syllable word, rhyming with toad, and having a DUM-duh-DUM stress pattern” is:

```
% mpron '' '^[12][02][12]$_' 'ouD$'
```

The `^` and `$` in `^[12][02][12]$_` ensure that the stress pattern string consists *entirely* of that stress pattern, instead of merely having that stress pattern in the word somewhere. Here we go!

```
% mpron '' '^[12][02][12]$_' 'ouD$'
alamode      102      &l@moUd
antinode     102      &ntInoUd
antipode     102      &ntIpoUd
arillode     102      &r@loUd
autocode     102      At@koUd
a_la_mode    201      &l@moUd
calicoed     102      k&l@koUd
discommode   201      dIsk@moUd
episode      102      EpIsoUd
hemipode     102      hEmIpoUd
```

incommode	201	Ink@moUd
internode	102	Int@rnoUd
keratode	102	kEr@toUd
Kozhikode	101	koUZIKoUd
manucode	102	m&nj@koUd
megapode	102	mEg@poUd
microcode	102	maIkroUkoUd
nematode	102	nEm@toUd
Nesselrode	102	nEs@lroUd
overstowed	201	oUv@rstoUd
palinode	102	p&lInoUd
pigeon-toed	102	pIdZ@ntoUd
porticoed	102	poUrt@koUd
staminode	102	st&m@noUd
superload	102	sup@rloUd
trematode	102	trEm@toUd
waggonload	102	w&g@nloUd

Poetry in motion—or rather, in automation! The complete code is shown below.

```
#!/usr/bin/perl
# mpron -- search for words matching a given phonetic pattern

use strict;
my $Debug = 0;
my($word_re, $meter_re, $pron_re) = @ARGV[0,1,2];

if ($meter_re =~ m<^[/_]+$>) {
    $meter_re =~ s</><[12]>g;
    $meter_re =~ s<_><[20]>g;
    $meter_re = '^' . $meter_re . '$';
}

print "# Word RE: <$word_re> Meter RE: <$meter_re> Pron RE: <$pron_re>\n";

die "You need at least one stipulation for word, meter, or pronunciation!"
    unless length $word_re or length $meter_re or length $pron_re;

my $search_file = 'mpron.dat';
open(IN, $search_file) or die "Can't open $search_file: $!";

my @bits;
my $matches = 0;
my $lines = 0;
while (<IN>) {
    chomp;
    @bits = split "\t", $_;
    next unless @bits == 3;
    ++$lines;
    ++$matches, print $_, "\n"
        if (!length($word_re) || $bits[0] =~ m/$word_re/oi) and
            (!length($meter_re) || $bits[1] =~ m/$meter_re/o) and
            (!length($pron_re) || $bits[2] =~ m/$pron_re/o);
    if ($Debug) { last if $. > 2000 }
}
```

```

}
print "# $matches matches across $lines lines in ", time - $^T, " seconds.\n";
exit;

```

Accommodating Another Notation

One minor quibble, though: it’s a bit cumbersome converting our `/_/` (DUM-duh-DUM) notation into the regex `^[12][02][12]$`. We should have our program accept the slash and underscore notation. We can do that by just adding, very early in our program, some code to convert from that notation (if that’s what it sees) into regex notation. Namely:

```

# If the string consists entirely of slashes and underscores...
if ($meter_re =~ m<^[/_]+$>) {
    $meter_re =~ s</><[12]>g;
    $meter_re =~ s<_><[20]>g;
    $meter_re = '^' . $meter_re . '$';
}

```

This translates `/_/` to `^[12][02][12]$` as the second argument:

```

% mpron '' '/_/' 'oUd$' | less
# Word RE: <> Meter RE: <^[12][20][12]$> Pron RE: <oUd$>
alamode      102      &l@mOUd
...and so on...

```

By the way, if you want `/_/` to mean “ends in DUM-duh-DUM” instead of specifically “consists entirely of DUM-duh-DUM,” then you could change that last line to this instead:

```

$meter_re = $meter_re . '$';          # No '^' at the beginning

```

The only question left to answer is: what exactly did our poetic toad gleam and dance like? No program can tell you which of the twenty-six matching words (three-syllable, `/_/`, rhyming with toad) that we found is *le mot juste*, but given the circumstances, the choice is clear:

```

I chanced upon a lovely toad,
It gleamed and danced like microcode!

```