

# *The Perl Journal*

## **Regex Arcana**

Jeff Pinyan • 3

## **XML Subversion**

Curtis Lee Fulton • 8

## **2004 OSCON Round-Up**

Andy Lester • 14

## **Molecular Biology with Perl**

Simon Cozens • 15

## **Pipelines and E-Mail Addresses**

brian d foy • 20

## **PLUS**

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Source Code Appendix • 22

## LETTER FROM THE EDITOR

### Just Regular

**R**egular expressions are the reason I learned Perl. The first time I saw an `s//` operator do its job, I was hooked. Given the mountain of data manipulation I was facing at the time, that one little operator sufficed to draw me in to the whole world of Perl. Even now, I love regexes. You might even say I love them a little too much. Looking back at some of my code, I find tools I've written that amount to little more than frameworks wrapped around my regexes. They're just data-delivery systems designed to pass various things through the real workhorses—the regular expressions that filter and transform the data. But I guess there's nothing really wrong with that.

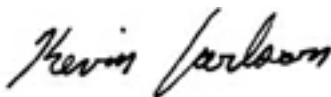
They're hard to resist because of their conciseness. A wealth of very subtle conditional activity can be contained in a very small syntactic space. Nowhere but in a regex do you get so much bang-per-character. Under the right circumstances, one-line pattern matches can do the work of 20 `if...else` conditions.

Regular expressions, like Perl itself, have deep ties to the past. They give UNIX graybeards a warm, fuzzy feeling, and for those of us who didn't come from UNIX, they just make us feel clever. Quick test: Can you use the phrase "zero-width negative look-ahead assertion" convincingly in a sentence? (Talking like this will either earn you the respect of your peers or make them not want to sit with you at lunch, depending on where you work.)

But even regex junkies should remember that conciseness does not equal efficiency. Perl's regular expression engine does things under the hood that aren't obvious. These can be pitfalls for the unwary. Too much backtracking, for instance, can cause exponential drag on the speed of your program. It's important to remember that a regular expression might not always be the best tool for the job. Examine your options, and don't get too dependent on any one technique.

This month, Jeff "japhy" Pinyan shows off some regex tricks that will let you maximize the power of your patterns without tipping the scales toward inefficiency. He'll show you how to use delayed-execution assertions to build some nifty self-constructing and recursive regexes, and introduce you to a whole stable of regex variables that I'll bet you didn't know about.

Regular expressions have been both praised as a vitally important tool and criticized as overly abstruse, arcane, and dense. Some see them as a deterrent to learning Perl. That's a little unfair, since you can do a lot in Perl without ever writing a regex, but it's true that regexes have become more closely associated with Perl than with other languages, even though some of those languages have regular expression capabilities that rival those of Perl. In any case, what's true of Perl is true of regular expressions: You can learn as little or as much as you need to do the job in front of you, without digging into a bloated API reference. And for that I'm thankful.



Kevin Carlson  
Executive Editor  
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to [editors@tpj.com](mailto:editors@tpj.com), faxed to (650) 513-4618, or mailed to *The Perl Journal*, 2800 Campus Drive, San Mateo, CA 94403.

THE PERL JOURNAL is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: \$18.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2004 by CMP Media LLC, unless otherwise noted. All rights reserved.



### The Perl Journal

#### EXECUTIVE EDITOR

Kevin Carlson

#### MANAGING EDITOR

Della Song

#### ART DIRECTOR

Margaret A. Anderson

#### NEWS EDITOR

Shannon Cochran

#### EDITORIAL DIRECTOR

Jonathan Erickson

#### COLUMNISTS

Simon Cozens, brian d foy, Moshe Bar, Randal Schwartz, Andy Lester

#### CONTRIBUTING EDITORS

Simon Cozens, Dan Sugalski, Eugene Kim, Chris Dent, Matthew Lanier, Kevin O'Malley, Chris Nandor

#### INTERNET OPERATIONS

##### DIRECTOR

Michael Calderon

##### SENIOR WEB DEVELOPER

Steve Goyette

##### WEBMASTERS

Sean Coady, Joe Lucca

#### MARKETING / ADVERTISING

##### PUBLISHER

Michael Goodman

##### MARKETING DIRECTOR

Jessica Hamilton

##### GRAPHIC DESIGNER

Carey Perez

#### THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403

650-513-4300. <http://www.tpj.com/>

#### CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND

MARKETING Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT/GROUP DIRECTOR INTERNET BUSINESS

Mike Azzara

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

VICE PRESIDENT, COMMUNICATIONS Alexandra Raine

PRESIDENT, CHANNEL GROUP Robert Faletra

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED

TECHNOLOGIES Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER NETWORK

COMPUTING ENTERPRISE ARCHITECTURE GROUP

Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE

DEVELOPMENT MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING

SOLUTIONS Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES

Marie Myers

# Perl News

*If you had a sense of déjà-vu reading last month's Perl News, you aren't crazy. Due to a version-control mix up, the July 2004 Perl News was a reprise of the July 2003 Perl News. Our apologies.*

—Ed.

## White Camel Awardees Announced

The sixth annual White Camel Awards were bestowed on three “unsung heroes of the Perl community” at the 2004 O'Reilly Open Source Convention. brian d foy, who started the White Camel tradition in 1999, takes his turn as a recipient this year, honored for—in addition to founding the Perl Mongers and publishing *The Perl Review*—his service in Iraq, where he managed to maintain Perl modules despite frequent lack of electricity and Internet access. Dave Cross, who started the first non-North American Perl Mongers group (London.pm) and now oversees the Perl Mongers groups worldwide, was also recognized this year. And finally, we at *TPJ* especially congratulate Jon Orwant, the original editor and publisher of *The Perl Journal*, on his White Camel award. Previous winners are listed at [http://www.perl.org/advocacy/white\\_camel/](http://www.perl.org/advocacy/white_camel/).

## The Value of Pie

Also at OSCON, Dan Sugalski did eventually take a pie in the face over his bet with Guido van Rossum that Python bytecode could be made to run faster on Parrot than it does in CPython. By the time the bet's deadline came due, the Parrot implementation—IronPython—was only complete enough to run four out of seven benchmarks, though it did beat CPython in three of those four benchmarks. Guido, in a gentlemanly fashion, declined the opportunity to pie Dan—much to the disappointment of the Perl crowd, which was howling for, if not blood, at least tasty whipped topping. Honor was satisfied on all sides when Dan agreed to auction off his own pie-ing at The Perl Foundation's fundraiser. A \$520 bid was mustered, and Guido received satisfaction: pictorial evidence is archived at [http://www.oreillynet.com/oscon2004/friday/Pages/Dan\\_Sugalski-1.html](http://www.oreillynet.com/oscon2004/friday/Pages/Dan_Sugalski-1.html), and Dan's own account of the escapade is at <http://www.sidhe.org/~dan/blog/archives/000372.html>.

IronPython development continues (see <http://ironpython.com/>) and Dan summarizes: “Getting Python going on Parrot has definitely been very useful. Besides shaking out some implementation issues (we may well be able to speed up our sub/method calls a lot because of this) it's been really useful in pointing out some areas we were weak in (like slices) and got us a good chunk of experience doing bytecode translation. All in all I'm glad we did it, and I think we'll have something pretty useful when we're done.”

## Perl Projects Progress

Development continues on Ponie, the Perl 5 interpreter rewritten to run on Parrot. Nicholas Clark has released Snapshot 3 of Ponie (<http://opensource.fotango.com/~nclark/ponie-3.tar.bz2>), demonstrating that “with this release the C type of the Perl core's data pointer, SV \*, is actually PMC \*, the C type of Parrot's data pointer.” The Ponie project is funded by Fotango, a consultancy group based in London, and is intended to help companies transition from Perl 5 to Perl 6.

Perl 5.8.5, the fifth maintenance release of Perl 5.8, is now available from CPAN. The core enhancements as described in the perldelta: “Perl's regular expression engine now contains support for matching on the intersection of two Unicode character classes. You can also now refer to user-defined character classes from within other user defined character classes.” Also, “The debugger can now emulate stepping backwards, by restarting and rerunning all but the last command from a saved command history.”

Patrick Michaud has been installed as the Perl 6 compiler pump-king, charged with “making the Perl 6 compiler happen” while Larry Wall concentrates on language design and Dan Sugalski handles the interpreter engine. Talk continues, as it has for a year, about forming a new parrot-compilers list, and Dan suggests renaming the perl6-internals list to parrot-internals. There may also eventually be a third list, parrot-library.

## Upcoming Events

A call for venues has been issued for YAPC::NA::2005; submissions are due by August 31st, with a decision tentatively scheduled for September 15th. For the first time, the YAPC Conference Committee has revealed the method by which it evaluates prospective venues; a scoring system is used that gives different weights to each of ten categories. The quality of the facilities, along with Internet access from both the conference location and the accommodations, are the most important considerations, closely followed by questions of cost. The scoring system also gives a bonus to new locations. You can see the full criteria at <http://yapc.org/yapc-crit.txt>, and the venue requirements are detailed at <http://www.yapc.org/venue-reqs.txt>. Venue proposals should be e-mailed to [yapc-venues@yetanother.org](mailto:yapc-venues@yetanother.org).

In other conference news worldwide, the first Brazilian Perl Users Meeting will take place October 18–20 at the Universidade do Rio de Janeiro (UNIRIO), under the auspices of the “2a Semana de Software Livre do Rio de Janeiro”: See <http://brasil.pm.org/>. Also, the Hungarian Perl Mongers are hosting a “biannual Micro Workshop” on August 28th in Budapest; <http://www.perl.org.hu/english/> has the details in English.



# Regex Arcana

```
# find any any doubled words
my $text = "Four score and and seven years ago";
while ($text =~ /\b(\w+)\W+1\b/g) {
    print "'$1' repeats itself";
    # 'and' repeats itself
}
```

I'm going to share two pieces of regex arcana with you—features of the Perl regex engine that aren't used often, but offer those who can master them great power.

The code evaluation assertion (`{ CODE }`) was introduced in Perl 5.005, but it was rough around the edges. In Perl 5.6, it was fine-tuned and more adjustments were made by 5.8. The code evaluation assertion allows a regex to execute an arbitrary block of Perl code during the course of its pattern matching. This device becomes particularly interesting when one realizes that the backtracking stack is much like a variable scoping stack.

The delayed execution assertion (`{?{ CODE }}`) is the heart of dynamic regexes. Using them is like walking across a bridge as you're

building it. Their true beauty is revealed when you create a *Regex* object (via *qr/*) with a delayed execution assertion in it. Inside that assertion is the self-same *Regex* object. Then you have created a recursive regex, one that is capable of matching nested data structures or acting like a proper grammar.

This article is going to use some regex variables you might not be familiar with, so I introduce them to you in Table 1.

```
"perl" =~ /(\\.\\.\\.\\.)/;

# $1 = "perl"
# $2 = "er"
# $+ = "er" (in this case, $2)
# $^N = "perl" (in this case, $1)
# @- = (0, 0, 1)
# @+ = (4, 4, 3)
```

```
substr($str, 0, $- [0]);           # produces $'
substr($str, $- [0], $+ [0] - $- [0]); # produces $&
substr($str, $+ [0]);               # produces $'
substr($str, $- [1], $+ [1] - $- [1]); # produces $1
substr($str, $- [2], $+ [2] - $- [2]); # produces $2 (etc.).
```

Variable	Description
\$+	The most recently opened capture group.
^N	The most recently closed capture group.
@-	The beginning offsets of \$& and capture groups.
@+	The ending offsets of \$& and capture groups.

**www.tpj.com**

might as well use them as many times as you can in that code, because you've already suffered the hit.

## Using (?{ ... })

Our first incantation is the code evaluation assertion. It will execute the code inside when it is encountered, and then continue with the regex; they always succeed, unless their contents interrupt the execution of your program.

Let's look at some applications of this assertion.

---

*The delayed execution assertion  
(?{ CODE }) is the heart of  
dynamic regexes. Using them is  
like walking across a bridge as  
you're building it*

---

## Inspecting a Regex's State

It has been said that "you can't observe the behavior of a system without affecting the system's behavior." Not so, at least in Perl 5.8. The following regex uses code assertions to show how a particular optimization in the engine works:

```
# the /x modifier allows embedded whitespace
# to help improve readability and clarity
"salad" =~ m{
    .* (?{ print "[%&" " " }) a
    .* (?{ print "(&" " " }) a
}x;
# output:
# [sal] [s] (sal)
```

We often see backtracking as a character-at-a-time procedure, but it's really much more efficient than that. In this case, something like `.*a` makes the engine jump from one "a" to the next when matching or backtracking. The code assertion doesn't get in the way of this, even though it's in-between the `.*` and `a`. (Sadly, in Perl 5.6, it did get in the way. Try that code in 5.6 and you'll see it stops the optimization from occurring.)

You have access to the regex variables inside a code assertion; my example above uses `&`, although it's general practice to avoid that variable. You can also access `$1` and other digit variables, `$^N`, `$+`, and the arrays `@-` and `@+`. One caveat, though, is that you can't access a capture group until it has been completed; that is, the closing parenthesis for `$1` must be passed before you can see the contents of `$1` in a code assertion, otherwise you'll be seeing its previous value. This code won't print anything inside the quotes because `$1` hasn't been closed before it's printed:

```
"perl" =~ m{
    ( (?{ print "'$1'\n" }) )+
}x;
```

In order to inspect the digit variables as they're being created, you need to be cunning: Create a variable inside a code evalua-

tion immediately prior to the capture group that stores the current location in the string. Then, use `substr()` on `$_`. This is a copy of the string you're matching against inside a code evaluation:

```
"perl" =~ m{
    (?{ $p = $_[0] })
    ( (?{
        .
        (?{ print substr($_, $p, $_[0] - $p), "\n" })
    })
}x;
```

That `$_` trick is undocumented as far as I can tell, which means it might not stay that way in the future. (But considering Perl 6 regexes will be radically different, we should have fun while we can.)

In this way, code evaluation assertions are good debugging tools because you can use them without interfering with the execution of the regex and you can get helpful diagnostic information, such as where you are in the string, what you've captured, and so on.

## Capturing Repetitions

I've often been asked why a regex like `/(\w)+/` only returns the last word character captured:

```
"japhy" =~ /(\w)+/ and print "<$1>"; # 'y'
```

The reason is because the regex is continually storing what `\w` matches to `$1`, and each time, it's overwriting the old contents. It won't create `$2`, `$3`, and so forth, and it doesn't create a `@1` array. So the question remains: How can I keep track of repeated capture groups?

The answer works on the principle that the matching and backtracking in a regex is very similar to a variable scoping stack if you use `local()` on your variables inside code evaluations. `local()` works differently in a regex; it gives a variable a value that remains until that point is backtracked past. Let's say we wanted to count how many characters we matched before the last "r" in a string. Here's code that doesn't use `local()`:

```
# not local()ized
"perl" =~ m{
    (?{ $x = 0 })
    (?{ \w (?{ ++$x }) }) *
    r
}x and print "x = $x\n";
```

It prints "4" as the value of `$x`, even though there are only two. This time, we'll use `local()` on a temporary variable and assign its value to `$x` only after the regex has succeeded:

```
# local()ized
"perl" =~ m{
    (?{ local $_x = 0 })
    (?{ \w (?{ local $_x = $_x + 1 }) }) *
    r
    (?{ $x = $_x })
}x and print "x = $x\n";
```

This time the value is reported correctly as "2."

The first code prints "4" because when the backtracking occurs, `$x`'s value isn't rewound or reverted to its previous one. The second code prints "2" for the opposite reason: Because of the use of `local()`, when backtracking occurs, the scoped variable `$_x` reverts to its previous value. Think of your regex as being a recursive function.

Although it requires a bit more coding on your part, this technique can be extended to let you keep track of capturing groups that have quantifiers on them:

```
"age=22 name=jeff state=NJ" =~ m{
  (?{ local (@_1, @_2) })
  (?
    ([^=]+) = (\\S+) \\s*
    (?{ local @_1 = (@_1, $1);
        local @_2 = (@_2, $2); })
  )*
  (?{ (@1 = @_1), (@2 = @_2) })
}x;
```

The code above produces two arrays, `@1` and `@2`, whose contents are “age,” “name,” “state,” and “22,” “jeff,” “NJ,” respectively.

If you’re wondering why I didn’t just write `push(@_1, $1)`, it’s because I have to make a new locally scoped array, and `push()` wouldn’t do that. You must use `local()` every time you need a variable to revert to its previous value when it’s backtracked past.

## Using `$_R` to Simplify Things

There’s another regex variable, `$_R`, which stores the value of the last `(?{ ... })` assertion. Perl is nice and makes `$_R` scope properly, so if your regex backtracks, `$_R` will revert to its previous value. This means we can write a much simpler looking regex:

```
"perl" =~ m{
  (?{ 0 })          # sets $_R to 0
  (?
    \w
    (?{ $_R + 1 })   # sets $_R to $_R + 1
  )*
  r
  (?{ $x = $_R })    # sets $x to $_R
}x and print "x = $x\n";
```

Here, we don’t need to worry about localizing our variables because `$_R` is properly dealt with by Perl. You need to be careful about how you modify it, though. We could not have written `(?{ $_R++ })` or `(?{ ++$_R })` in our regex because both of those explicitly assign to `$_R`. You usually don’t want to assign to `$_R` explicitly; just make the last value in your code assertion be the value you want `$_R` to have. Perl builds up a stack of the values returned by code evaluations and rolls that stack back when backtracking occurs. Setting `$_R` explicitly interrupts that stacking procedure.

We can use `$_R` with our example that creates `@1` and `@2` as well. We have to be careful in how our code assertions work:

```
"age=22 name=jeff state=NJ" =~ m{
  #   @1 @2
  (?{ [ [], [] ] })
  (?
    ([^=]+) = (\\S+) \\s*
    (?{ [
      [ @{$$_R->[0]}, $1 ],
      [ @{$$_R->[1]}, $2 ],
    ] })
  )*
  (?{
    @1 = @({ shift @{$$_R} });
    @2 = @({ shift @{$$_R} });
  })
}x;
```

You cannot rely on what value `$_R` will have outside of your regex, so you should always get the data from it at the very end of your regex, once you are sure it has succeeded.

## If-Then Assertions

You can also use code assertions as the condition to an if-then assertion. If you’re not familiar with them, they look like `(?(cond)true|false)`. If the `cond` part evaluates to a true value, then the true pattern is matched; otherwise, the false pattern is matched. If the selected pattern fails to match, the if-then assertion fails altogether. The `cond` part can be a look-ahead, a look-behind, a number referring to a capture group, or a code assertion. If it is a code assertion, `$_R` does not get its value changed.

*More often than not, you'll see these assertions used to create recursive regexes, ones that can match arbitrarily deeply nested parentheses, etc.*

Here’s a regex that matches `$_` if it is an integer less than 100:

```
my $less_than_100 = qr{
  ^ ( -?\d+ ) $      # match the entire number
  (?(?{ $1 >= 100 }) # if $1 >= 100...
    (?!              # then fail
    )                # otherwise it succeeds
  );
if ($n =~ $less_than_100) {
  # $n is less than 100
}
```

## Using `(??{ ... })`

Our second gimmick is the delayed execution assertion. It acts very much like a code evaluation assertion, but the return value is returned to the regex as something to be matched. Consider our first example, which finds words that are repeated—here’s a way to write it using a delayed execution assertion:

```
while ($text =~ /\b(\w+)\W+(??{ $1 })\b/g) {
  print "'$1' is repeated\n";
}
```

Because `$1` only contains word characters, I didn’t need to worry about any regex-specific characters being interpolated incorrectly. However, if we change from “words” to “non-whitespace chunks,” we’ll need to be safer:

```
while ($text =~ m{
  (?<!\S) (\S+)          # a chunk of non-spaces
  \s+                    # some whitespace
  (??{ quotemeta $1 }) (?!\\S) # that chunk again
}xg) {
  print "'$1' is repeated\n";
}
```

If you’re curious why I didn’t use `\b` in this regex, it’s because I’m not looking for word boundaries, so I can’t count on it working properly.

But that's a very silly use of a very powerful feature. It's not very economically sound to use your car to drive one city block and this assertion is an SUV. So let's go off-roading.

## Self-Constructing Regexes

One of the first uses I found for this assertion was to match a chunk of unique characters in a string. Take, for example, “the quick brown fox jumped over the lazy dog.” If I want to grab the first chunk of unique characters, I use an assertion that builds a character class based on the characters it has already matched:

```
my $unique = qr{
    .                # match any character
    (??{             # evaluate and match...
        "[^\Q$&\E]"  # a self-modifying char class
    })*              # zero or more times
}sx;
my ($uniq_str) = $string =~ /(($unique)/;
```

It's important to realize that we're not matching the same character class over and over again. The delayed execution assertion is being reexecuted each time. We can see what the character class looks like if we want:

```
my $unique = qr{
    .
    (??{
        print my $cc = "[^\Q$&\E]";
        $cc;
    })*
}sx;
my ($uniq_str) = $string =~ /(($unique)/;
```

On “the quick brown...,” this gives the output:

```
[^t]
[^th]
[^the]
[^the\ ]
[^the\ q]
[^the\ qu]
[^the\ qui]
[^the\ quic]
[^the\ quick]
```

It stops when it reaches the second occurrence of a space because the character class fails to match something it hasn't seen before. If we wanted to ignore spaces, we could simply add the ability to match any whitespace characters before we try matching a new character:

```
qr{
    . (?: \s+ | (??{ "[^\Q$&\E]" }) ) *
}xs
```

This time we would match “the quick brown f.”

## Dynamic Quantifiers

You can use this assertion to make a quantifier based on text you've matched. If you have a string like “2aaaa 6a 3aaa 5 1a,” and you want to match a number if it is followed by exactly that many a's, you capture the number and then make use of a delayed execution assertion, which uses that number as its quantifier:

```
@matches = $data =~ m{
    \b (\d+)
    (??{ "a{$1}" }) }
```

```
(?! a )
}gx;
```

You could be more efficient and use “a”x \$1 instead of “a{\$1}”, but the example can be expanded to match a lower and upper limit:

```
# match X,Y, then match between X and Y a's
$data = "1,3aa 2,5aaaaaaaa 2,7aaaaa";
@matches = $data =~ m{
    \b (\d+) , (\d+)
    (??{ "a{$1,$2}" })
    (?! a )
}xg;
```

Of course, you'd have to make sure \$1 isn't greater than \$2, but that's left as an exercise to the reader.

Let's look back to our “is X less than 100?” example. We can rewrite this with (??{ ... }) instead:

```
if ($num =~ m{
    ^ (~?\d+) $
    (??{ $1 >= 100 and '(?!)' })
}x) {
    # $1 is less than 100
}
```

Here, if \$1 is equal to or greater than 100, we insert (?!) into the regex, an empty negative look-ahead, which always fails. If \$1 is less than 100, then our >= operation returns "" as its false value, and the regex succeeds.

## Recursive Regexes

More often than not, you'll see these assertions used to create recursive regexes, ones that can match arbitrarily deeply nested parentheses, etc. To get this to work, you create a *Regexp* object and then put that object inside a delayed execution assertion:

```
# if you're using a lexical, it MUST be
# declared before it is assigned to
my $px;
$px = qr{
    (?
        (?> [^\(\)]+ | \\\. )
        |
        \ ( (??{ $px }) \ )
    ) *
}xs;
```

Now we can make sure a string is properly ()-balanced:

```
my $good = 'a + (b / (c - 2)) * (d ^ (e+f))';
my $bad1 = 'a + (b / (c - 2) * (d ^ (e+f))';
my $bad2 = 'a + (b / (c - 2)) * (d) ^ (e+f))';
print "not " if $good !~ /^$px$/; print "ok\n";
print "not " if $bad1 !~ /^$px$/; print "ok\n";
print "not " if $bad2 !~ /^$px$/; print "ok\n";
```

The first is reported OK, the second and third are reported as not OK (which is what we would expect).

Using this recursive mechanism, we can create a regex that matches a palindrome. A palindrome is a string that reads the same forwards and backwards, like “toot,” or “madam, I'm adam.” Punctuation and case are ignored.

```
my $pdrome;
$pdrome = qr{
    (\w) \w*
```

```

(??{ $pdrome })?
\W* \1
}xi;

```

This matches “toot,” but not “rotor.” Can you see why?

There is a boundary case: If the string has an odd number of letters, then the turning point letter won’t be repeated. Let’s add support for that:

```

my $pdrome;
$pdrome = qr(
    (\W) \W*
    (??{ $pdrome })?
    \W* \1
    |
    \W
)xi;

```

Now we can match “Sit on a potato pan, Otis.”

## The Future: Perl 6

So those are the two workhorses of Perl’s eccentric regex implementation. In Perl 6 (<http://dev.perl.org/perl6/>), these features won’t disappear. In fact, the code evaluation assertion will probably become increasingly prevalent. Here’s a quick taste of how Perl 6 will make these assertions easier to write and simpler to understand.

## Code Evaluation

In Perl 6, regex metacharacters will change meaning to make things clearer. Code evaluation assertions are now just enclosed by curly braces. Our regex earlier that matches a number that is less than 100 could be written in Perl 6 like this:

```

/ ^ (\d+) $ { $1 < 100 or fail } /

```

The first thing you should notice is that, with all that whitespace, there is no `/x` modifier on my regex. That’s because it will be the default. Next, you can see the code evaluation is `{ code }`, which looks like a closure (because it is). Finally, even though this is merely a code evaluation, it can affect the regex by calling `fail`.

## Delayed Execution

But the syntax `{ COND or fail }` has a shorthand in Perl 6, by way of the new look of the delayed execution assertion: `<...>`. You’ll be able to use `<$rx>` to get the same effect as `(??{ $rx })`. A special case of this assertion is `<(...)>`, which regards its internals as an expression that causes the match to fail at its current point if it evaluates to false:

```

/ ^ (\d+) $ < ( $1 < 100 ) > /

```

## Backtracking Control

Finally, a note on efficiency. Our regex here is currently less efficient than it should be. After a number like “300” fails, Perl backtracks the `\d+` to match “30,” which fails because “30” is followed by “0,” not the end of the string. But Perl still does this needless backtracking. In Perl 5, we would use `(?> (\d+))`, the “cut” assertion, which prevents internal backtracking.

In Perl 6, you’ll be able to say it more succinctly with a colon (or two, or three). In our case, we’ll want two colons, which says that this set of alternatives is all to fail if we have to backtrack past the `::`. Since our regex has no alternatives in it, it means this match will fail if the `::` is backtracked past.

```

/ ^ (\d+) $ :: < ( $1 < 100 ) > /

```

We can get even sleeker—if we use `<commit>` instead of `::`, the entire regex will fail, not just the match at the current location. This will eliminate our need for anchors:

```

/ (\d+) <commit> < ( $1 < 100 ) > /

```

## Food for Thought

With your new knowledge, you should try tackling these two tasks. First, construct a regex that determines the longest sequence of unique characters in a string. Now, decipher the goal of this regex. It uses both kinds of code assertions, the if-then assertion, and the `$/R` variable:

```

my %data = $str =~ m{
    (?{ [ "", "", "" ] })
    ( [ ^\s=]+ ) \s* = \s*
    (?{ (?=[ "' ])
        (?{
            my $c = substr($_, $+[0], 1);
            [ $c, "[^\Q$c\E]**", $c ]
        })
        |
        (?{ [ "", '\s+', "" ] })
    })
    (??{ $^R->[0] })
    ((??{ $^R->[1] })))
    (??{ $^R->[2] })
}xg;

```

To see its dissection, go to <http://japhy.perlmonk.org/regexes/>.

TPJ

**Fame & Fortune  
Await You!**

**Become a  
TPJ  
author!**

*The Perl Journal* is on the hunt for articles about interesting and unique applications of Perl (and other lightweight languages), updates on the Perl community, book reviews, programming tips, and more.

If you’d like to share your Perl coding tips and techniques with your fellow programmers – *not to mention becoming rich and famous in the process* – contact Kevin Carlson at [kcarlson@tpj.com](mailto:kcarlson@tpj.com).



# XML Subversion

**Y**ou remember when you first learned about relational databases—the seductive simplicity of the grid found a special place in your heart. Its unwavering uniformity assured you that you would always, always, find what you are looking for.

But since then, reality has set in. Those rigid little boxes have become oppressive. As your projects grow, your data often doesn't fit into the perfect little rows that you have crafted.

As an alternative, the free-form structure of XML is tantalizing. You'd love to be able to adjust the structure of your data at a moment's notice. But what about speed and efficiency? You can't parse an entire XML tree every time you need some data. You need the speed of a relational database, but crave the organic structure of XML.

Your solution is *XML::EasySQL*, which is a two-way SQL/XML base class for Perl. Here are some of the benefits it provides:

- Two-way transforms between XML and relational data.
- Smart SQL updates: Only altered tables are updated.
- Unlimited tree depth.
- Multiple tables can merge into one XML tree, then back again.
- Precise control over how data is translated.
- Either an easy XML interface or plain DOM.
- Database independency.

*XML::EasySQL* works by first taking data that is output from DBI and turning it into an XML tree. The programmer is then free to modify the data using the easy XML interface that's provided, or start hacking directly on the underlying *XML::DOM*. When you're ready to dump the changed data back to the database, you only have to call one method. The tree is stored as shown in Figure 1.

*XML::EasySQL* consists of two classes: *XML::EasySQL* and *XML::EasySQL::XMLnode*. *XML::EasySQL* is the actual data ob-

ject class. Its methods transform data between XML and SQL forms. You probably want to use *XML::EasySQL* as the base class for your data objects.

*XML::EasySQL::XMLnode* is optional, although it's highly recommended. It's really just a simplified DOM interface for Perl. The class is derived from a fork of Robert Hanson's excellent *XML::EasyOBJ* module, which seeks to offer a more "Perl-ish" interface to the *XML::DOM*. (So far, Perl doesn't have native support for the DOM, which seems the natural ascendant of its built-in hashes. But a hacker can dream, can't he?)

Listing 1 shows a synopsis from the *XML::EasySQL* manual, which shows both classes working together.

## Installing *XML::EasySQL*

The easiest way to install *XML::EasySQL* is with the CPAN module:

```
$ su -l
$ perl -MCPAN -e shell
cpan> install XML::EasySQL
```

Or you could manually download and install the module. Just snag the latest version from CPAN at <http://search.cpan.org/~curtisf/>.

After you've downloaded it, invoke the standard incantation:

```
$ tar fxzv XML-EasySQL-version.tar.gz
$ cd XML-EasySQL-version
$ perl Makefile.PL
$ make
$ su
$ make install
```

## The *XML::EasySQL* Class

I recommend that you use *XML::EasySQL* as a base class for your data objects. You don't have to, but if you do use the class directly, you'll wind up polluting your constructor with some pretty hairy parameters.

---

*Curtis is a journalist and programmer who lives in Portland, Oregon. He can be reached at [curtisf@fultron.net](mailto:curtisf@fultron.net).*

Listing 2 shows a simple base class (*User*) that's derived from *XML::EasySQL*. See what I mean about the constructor parameters? You're not going to want to pass that chunk of XML to your data object constructor every time you need a new object. If you simply use *XML::EasySQL* as a base class, you can neatly tuck all that XML inside the package.

But what is all that XML anyway? We'll get to that later. For now, just keep in mind that the XML chunk in Listing 1 is used to define how your SQL maps to XML.

So the new class *User* has inherited all of *XML::EasySQL*'s methods. The constructor passes that XML chunk to the base class constructor, *XML::EasySQL::new*. You'd use the *User* class as shown in Listing 3.

Note that the *User* class, like its base class, still needs its data argument passed through the constructor. You will probably find that too messy and want to make a base class of your own that handles all of the SQL communication, and use that as the base class for all your data objects. For example, a base class called "*Base*" could look something like Listing 4. Our new *User* class, with its new base class, is shown in Listing 5.

Now that the SQL query is hidden, the *User* object could be constructed this way:

```
my $user = User->new((db=>$db, user_id=>2, comment_id=>183);
```

And to save any changes made to the XML, all that is needed is:

```
$user->save();
```

The rest of the *User* interface remains unchanged. If you are writing a large program with many different types of data objects, you'll probably want to make more than one base class.

## The XML Schema

Remember that XML chunk we keep passing to the *XML::EasySQL* constructor? Here's the skinny: Every *XML::EasySQL* object needs an XML schema. The schema tells *XML::EasySQL* how each column is supposed to map in and out of the XML tree.

Table columns can map to an XML tree one of three ways:

- **attrib.** This simply applies the column value as an XML attribute on the root node. For example, if a column named *id* was mapped as an *attrib*, the resulting XML document would look like this: `<root id="23">...`
- **string.** This is an XML string that's a child of the root node. If a column named *headline* was mapped as a string, the resulting XML document would look like this: `<root><headline>Man Bites Dog</headline>...`
- **element.** This is the most important feature in the schema. If a column is mapped as an element, the data is parsed into an XML branch and grafted onto the root node of the XML document. This means you can add new nodes on the fly, but they'll still be recorded in the database. For example, say a column named *bio* in a database contained this string: `<bio><name>Curtis Lee Fulton</name></bio>`. It would get parsed into XML and grafted onto the DOM. You could then manipulate it like any other XML document, including adding new branches.

Here's a simple schema:

```
<schema name="users" default="string"></schema>
```

Here's a more complex one:

```
<schema name="users" default="attrib" default_table="users">
<columns>
<id type="attrib"/>
<group_id type="attrib"/>
```

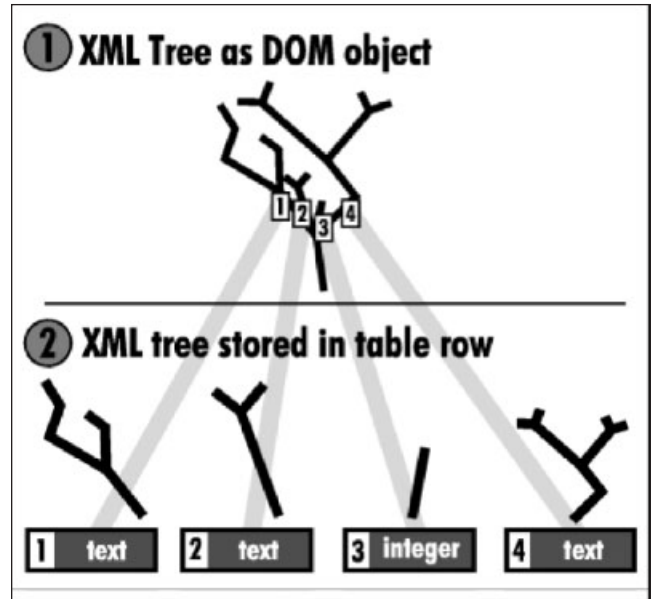


Figure 1: How EasySQL stores an XML tree in a relational database.

```
<email type="string"/>
<bio type="element"/>
<history table="comments" type="element"/>
</columns>
</schema>
```

The *XML::EasySQL* schema can have three root attributes: *name*, *default*, and *default\_table*.

- **name.** Sets the name of the root XML element. If missing, it defaults to "xml."
- **default.** The default type, which controls how incoming SQL column data is processed. If *default* is missing, then *XML::EasySQL* will ignore SQL columns that aren't specified in the schema. See "type" in the following list of column elements for more details on the possible types.
- **default\_table.** The default table a column belongs to. If missing, it defaults to what the *name* attribute is set to.

The schema can have multiple column entries. Each entry must have a unique tag name that matches a real column name in a SQL table. Column elements can have two attributes:

- **type.** This describes how the SQL data will map onto the XML tree. There are three types: *attrib*, *string*, and *element*.
- **table.** The table the column belongs to. If missing, it defaults to "default\_table."

## The Node Interface

EasySQL's built-in XML node interface is *XML::EasySQL::XMLobj*. The interface is a fork of Robert Hanson's wonderful *XML::EasyOBJ* module, which he wrote because he wanted a more "Perl-ish" interface to *XML::DOM*. The best part about Hanson's interface is that its underlying DOM is always available. You can get at the underlying *XML::DOM* element from any *XMLobj* object by calling the *getDomObj* method.

*XMLobj* can do almost anything *XML::DOM* can do, but with less demand on the user. Listing 6 gives a synopsis.

Here's an example with actual XML. Say you've got a tree that looks like this:

```

<root>
  <branch>
    <twig some_attr="I am">
      <leaf1>leaf1a</leaf1>
      <leaf2>leaf2b</leaf2>
      <leaf3>leaf3c</leaf3>
    </twig>
    <twig some_attr="the Walrus">
      <leaf1>leaf1d</leaf1>
      <leaf2>leaf2e</leaf2>
      <leaf3>leaf3f</leaf3>
    </twig>
  </branch>
</root>

```

Navigating an XML tree with *XML::EasySQL::XMLObj* is almost like pawing through a bunch of nested hash references, except that the syntax is different. Also, hashes have a 1:1 ratio of name-to-value pairs, while each XML tag can have a string value and any number of attributes.

Let's say we wanted to print the value "leaf2e" from the above XML fragment. Assuming that *\$doc* is the root element of the XML page ("root"), your code is going to look like this:

```
print $doc->branch->twig(2)->leaf2->getString();
```

The *getString* method returns the text within an element.

The *XMLObj* manual is fairly straightforward, but to get you started, here's a brief rundown of how to accomplish XML tasks with *XMLObj*:

**Retrieve an element.** There are two ways to get at an element. The easiest way is to just call it as a method:

```
$doc->element_i_want;
```

Sometimes, you'll have a tag name that won't fly as a method call. If so, just use the method *getTagName*:

```
$doc->getTagName('element_i_want');
```

**Remove an element.** To remove an element, just call the method *remElement*. It removes a child element of the current element:

```
remElement( TAG_NAME, INDEX )
```

The name of the child element and the index must be supplied, although leaving the INDEX parameter undefined will simply remove the first occurrence of the named element.

**Retrieve an attribute.** Not rocket science, this one. Call *getAttr* like so, and you'll get the value of the named attribute:

```
$element->getAttr('attribute_name');
```

**Fetch a tag name.** If you're enumerating through a bunch of child elements, you might want to know their names. To do so, just call *getTagName*. Here's how:

```
foreach my $element ( $doc->getElement() ) {
  my $name = $element->getTagName();
}
```

**Set an attribute.** There's a good chance that at some point in your life, you'll want to set an attribute value. *setAttr* is the way to go. In return for its services, all it asks for is a series of name/value pairs. Provide that, and it will do the deed:

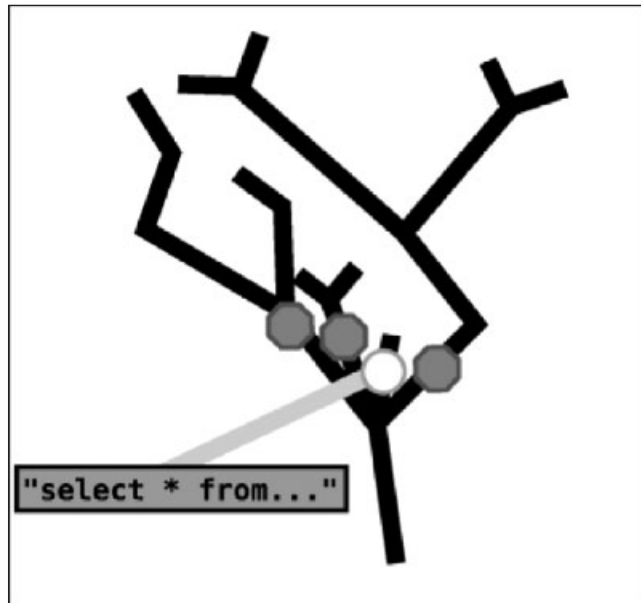


Figure 2: Only strings and attributes living in the root level of an XML tree can be referenced in an SQL query.

```
$element->setAttr('attrib_name_1', 'what do we want?', 'attrib_name_2', 'XML',
'attrib_name_3', 'When do we want it', 'attrib_name_4', 'Now!');
```

**Remove an attribute.** When the attribute has served its purpose, send it to to sleep with the fishes, like so:

```
$element->removeAttr('Don Corleone');
```

If you find you need more explicit control over the underlying DOM, you can get to it from any node with the *getDomObj* method:

```
$doc->getDomObj
```

The method simply returns the DOM object associated with the current node. This is useful when you need fine access via the DOM to perform a specific function. If you end up working directly with *XML::DOM*, you're going to have to flag parts of the XML yourself as needing updates. If you don't, changes you make to the XML tree won't get saved in the database.

If you've used *XML::DOM* to make a change anywhere in the XML tree except for an attribute of the root element, call *EasySQL::flagSync(base\_name)*. *Base name* is the name of the table (or subroot element) that holds the changes. If you have changed an attribute of the root element, call *EasySQL::flagAttribSync(attribute\_name)*.

If you don't want to muck with *flagAttribSync* and *flagSync* and you don't mind the performance hit, just call *EasySQL::getSQL(true)* before you write to your database. Calling *getSQL* with a *true* parameter will cause the entire table to be updated, whether or not it's changed.

## Designing Tables for EasySQL

A good way to design a relational database for *EasySQL* is to start with the rudimentary XML tree we think we'd like to work with and then work backwards.

*EasySQL* allows for fairly flexible database design, although there is one rule you are constrained to: Strings and attributes can only be referenced in a SQL query if they exist on the root level of the XML tree (see Figure 2). The result is that any data living deeper than root level in the XML tree can't be explicitly queried.

Let's say we decided on an XML tree that looks like this:

```
<user id="23">
<bio>
<username>curtisf</username>
<password>secret</password>
<name>Curtis Lee Fulton</name>
<resume>xxxxxxxxxx</resume>
</bio>

<plan>
<entry date="6/01/04">hello world</entry>
<entry date="6/12/04">goodbye cruel world</entry>
</plan>
</user>
```

---

*Every XML::EasySQL object needs an XML schema. The schema tells XML::EasySQL how each column is supposed to map in and out of the XML tree*

---

Pretty simple. It looks good, but remember that only root level trees get their own tables in *EasySQL*. That's fine for most of the data here, but not for all.

It's a good bet that the *id* attribute, as well as the *name* and *username* strings, are going to be queried. *id* is okay where it is because it's an attribute of the root tag, but the other two elements must be moved. Let's reformat the tree to look like this:

```
<user id="23">
<username>curtisf</username>
<name>Curtis Lee Fulton</name>
<bio>
<password>secret</password>
<resume>xxxxxxxxxx</resume>
</bio>
<plan>
<entry date="6/01/04">hello world</entry>
<entry date="6/12/04">goodbye cruel world</entry>
</plan>
</user>
```

Not bad, but *username* probably doesn't need its own tag, so let's tighten things up a bit by making it an attribute of the root element:

```
<user id="23" username="curtisf">
<name>Curtis Lee Fulton</name>
<bio>
<password>secret</password>
<resume>xxxxxxxxxx</resume>
</bio>
```

```
<plan>
<entry date="6/01/04">hello world</entry>
<entry date="6/12/04">goodbye cruel world</entry>
</plan>
</user>
```

Looks good. Now, let's design our XML schema for our data object:

```
<schema name="users" default="attrib" default_table="users">
<columns>
<id type="attrib"/>
<username type="attrib"/>
<name type="string"/>
<bio type="element"/>
<plan type="element"/>
</columns>
</schema>
```

Of course, since we've set *attrib* as the default, we don't need to explicitly set the first two:

```
<schema name="users" default="attrib" default_table="users">
<columns>
<name type="string"/>
<bio type="element"/>
<plan type="element"/>
</columns>
</schema>
```

So now we just need to design our database table. Here's a PostgreSQL schema to match the example:

```
CREATE TABLE users (
id integer,
username character varying,
name character varying,
bio text,
plan text
);
```

We know we're going to use the attribute *id* as the unique ID for each record, so let's tweak the SQL schema a bit:

```
CREATE TABLE users (
id serial NOT NULL,
username character varying,
name character varying,
bio text,
plan text
);
ALTER TABLE ONLY users
ADD CONSTRAINT users_pkey PRIMARY KEY (id);
```

Sweet. Now *id* is constrained as a unique primary key and will autoincrement whenever a new record is added.

Once we start coding, we'll have a pretty flexible data object *User*. We can add as many new branches to *bio* and *plan* as we want, and the branches will get stored in the database.

Just for kicks, let's create a data object for our new schema:

```
package User;
use XML::EasySQL;
@ISA = ('XML::EasySQL');

use strict;
```



```

sub new {
    my $proto = shift;
    my $params = shift;

    # the XML schema string
    $params->{schema} = q(
<schema name="users" default="attrib" default_table="users">
<columns>
<name type="string"/>
<bio type="element"/>
<plan type="element"/>
</columns>
</schema>
);
    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new($params);
    bless $self, $class;
};

1;

```

Let's put our new object into action:

```

# fetch the data from the database
my $data->{users} = $db->selectrow_hashref('select * from users where username =
                                                                    'curtisf');

# construct the data object
my $user = User->new((data=>$data));

my $xml = $user->getXML();

# fetch some data
my $id = $xml->getAttr('id');
my $password = $xml->bio->getString('password');

# add and modify some data
$xml->bio->setAttr('age', 22);
$xml->bio->city->setAttr('zip', 97201);
$xml->bio->city->setString('Portland');

# write the changes to the database
my $sql = $user->getSQL();
my $q = "update users set ".$sql->{users}." where id = 23";
$db->do($q);

```

## Conclusion

There's no reason to shoehorn your data into rigid tables. The *EasySQL* approach allows you to access and modify your data from a unified XML tree, while using any relational database to quickly find the data you need.

TPJ

(Listings are also available online at <http://www.tpj.com/source/>.)

## Listing 1

```

# fetch a database row as hash ref
my $data = {};
$data->{users} = $db->selectrow_hashref('select * from users where id = 2');

# init the new EasySQL data object
my $data_object = EasySqlChildClass->new((data=>$data));

# get the root XML element
my $xml = $data_object->getXML();

# make changes to the XML document
$xml->username->setString('curtisleefton');
$xml->bio->setAttr('age', 22);
$xml->bio->city->setString('Portland');

```

# 101 Perl Articles!



From the pages of *The Perl Journal*, *Dr. Dobb's Journal*, *Web Techniques*, *Webreview.com*, and *Byte.com*, we've brought together 101 articles written by the world's leading experts on Perl programming. Including everything from programming tricks and techniques, to utilities ranging from web site searching and embedding dynamic images, this unique collection of *101 Perl Articles* has something for every Perl programmer.

Plus, this collection of articles is fully searchable, and includes a cross-platform search engine so you can immediately find answers you're looking for. Delivered as HTML files in a ZIP archive or CD-ROM image, download *101 Perl Articles* and burn your own CD-ROM or store it on hard disk.

**\$9.95** For subscribers to *The Perl Journal*

**\$12.95** For nonsubscribers to *The Perl Journal*

**\$24.95** To subscribe to *The Perl Journal* and receive *101 Perl Articles*

Go to  
<http://www.tpj.com/>  
 now!

```
$xml->history->access->setAttr('last', time());

# output entire XML doc as string to STDOUT
print $xml->getDomObj->toString();

# update the database
my $sql = $data_object->getSQL();
my $q = "update users set ".$sql->{users}." where id = 2";
$db->do($q);
```

## Listing 2

```
package User;
use XML::EasySQL;
@ISA = ('XML::EasySQL');

use strict;

sub new {
    my $proto = shift;
    my $params = shift;

    # the XML schema string
    $params->{schema} = q(
<schema name="users" default="attrib" default_table="users">
<columns>
<id type="attrib"/>
<group_id type="attrib"/>
<email type="string"/>
<bio type="element"/>
<history table="comments" type="element"/>
</columns>
</schema>
);
    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new($params);
    bless $self, $class;
};

1;
```

## Listing 3

```
# fetch the data from the database
my $data = {};
$data->{users} = $db->selectrow_hashref('select * from users where id = 2');
$data->{comment_data} = $db->selectrow_hashref('select * from comments where
                                              id = 183');

# construct the data object
my $user = User->new({data=>$data});

# modify the data
my $xml = $user->getXML();
$xml->username->setString('curtisleefton');
$xml->bio->setAttr('age', 22);
$xml->bio->city->setString('Portland');
$xml->history->access->setAttr('last', time());
```

```
# write the changes to the database
my $sql = $user->getSQL();
my $q = "update users set ".$sql->{users}." where id = 2";
$db->do($q);
my $q = "update comments set ".$sql->{comments}." where id = 183";
$db->do($q);
```

## Listing 4

```
package Base;
use XML::EasySQL;
@ISA = ('XML::EasySQL');

use strict;

sub new {
    my $proto = shift;
    my $params = shift;
    my $db = $params->{db};
    my $schema = $params->{schema};
    my $data = {};
    foreach my $table (keys %{$params->{query}}) {
        my $d = $db->selectrow_hashref("select * from $table
                                         where id = ".$params->{query}->{$table});
        foreach my $k (keys %{$d}) {
            $data->{$k} = $d->{$k};
        }
    }

    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new({data=>$data, schema=>$schema});
    bless $self, $class;
};
```

```
}

sub save {
    my $self = shift;
    my $sql = $self->getSQL();
    foreach my $table (keys %{$sql}) {
        my $q = "update $table set ".$sql->{$table}."
                where id = ".$self->{query}->{$table};
        $db->do($q);
    }
}

1;
```

## Listing 5

```
package User;
use Base;
@ISA = ('Base');

use strict;

sub new {
    my $proto = shift;
    my $params = shift;

    # the XML schema string
    $params->{schema} = q(
<schema name="users" default="attrib" default_table="users">
<columns>
<id type="attrib"/>
<group_id type="attrib"/>
<email type="string"/>
<bio type="element"/>
<history table="comments" type="element"/>
</columns>
</schema>
);

    # get the SQL data
    $params->{query}->{users} = $params->{user_id};
    $params->{query}->{comments} = $params->{comment_id};

    # save the ids
    $self->{query} = $params->{query};
    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new($params);
    bless $self, $class;
};

1;
```

## Listing 6

```
# fetch the root XML::EasySQL::XMLobj object from your data object.
my $doc = $easy_sql_object->getXML();

# read from document
my $text = $doc->root->some_element($index)->getString();
my $attr = $doc->root->some_element($index)->getAttr('attrib');
my $element = $doc->root->some_element(34);
my @elements = $doc->root->some_element;

# first "some_element" element
my $elements = $doc->root->some_element;
# list of "some_element" elements
my @elements = $doc->root->some_element;

# write to document
$doc->root->an_element->setString('some string')
$doc->root->an_element->setAttr('attrib', 'val')
$doc->root->an_element->setAttr('attrib1' => 'hello', 'attrib2' => 'world')

# access elements with getElement, rather than a method call
my $element = $doc->root->getElement('element_name')->getElement(
    'another_element_name');

# get at the XML::DOM object
my $dom = $doc->root->foobar->getDomObj;

# get elements with unknown tag names
my @elements = $doc->root->getElement();
my $fourth_element = $doc->root->getElement('', 4);

# remove elements and attributes
$doc->root->removeElement('kill_this_element', 2);
$doc->root->tag_name->removeAttr('kill_this_attribute');
```

TPJ

# 2004 OSCON Round-Up

*Andy brings us the news from the 2004 O'Reilly Open Source Convention in Portland, Oregon. If you're going to an upcoming conference and would like to write a conference report for TPJ, drop me a line at [kcarlson@tpj.com](mailto:kcarlson@tpj.com) —Ed.*

Walking in to the Portland Marriott this year, I felt like I was home again. It's like nothing had changed, starting with the dozen geeks in the lobby, mostly on Mac laptops, sucking up the free OSCON wireless goodness.

Monday morning started with a bang, with Damian Conway's tutorial "Perl Best Practice." For three hours, he lectured on how to write good, maintainable code in a language that makes it so easy to write bad code. It was the most pragmatic talk I'd seen from someone whose talks are more often aimed at the "proof of crazed concept" angle. Most interesting, or at least most Perl-specific, was a discussion on writing maintainable regular expressions, including the importance of building up regular expressions from components using the `qr//` operators, and always using the `/s`, `/m`, and `/x` flags.

Damian also mentioned lesser known modules like `IO::Prompt`, which makes the drudgery of getting user input slick and handy, and `Readonly`, which makes variables nonmodifiable. (Actually, `IO::Prompt` is so little-known that Damian hasn't published it yet.) It was interesting to see a few items on which we disagree, like the use of mutator methods instead of setter/getter pairs and inline POD. I firmly believe that POD belongs interspersed throughout the code (see `WWW::Mechanize` for an example) so that changes can be made to the docs easily when the code changes; Damian says it all begins after the `__END__` token. TMTOWTDI, indeed.

This session effectively paid for the entire trip, since I've been working on getting departmental coding standards, and this session covered 90 percent of what I wanted to have in them. If you can see Damian give this talk, do so. If not, don't worry. He'll be turning it into a book for O'Reilly. (And by the way, O'Reilly & Associates is now O'Reilly Media.)

After the lunch break, it was back to Damian talking about how to give a quality presentation. Although there was no Perl content, I'd have been a fool to miss a master of presentation give away his secrets. He discussed every aspect of giving a talk, from what to call it and what to talk about, to what to wear. Key concepts included the importance of telling a story, rather than presenting a hierarchical tree of ideas; that the presentation is meant to inspire and give a taste of the topic, since the talk will probably not be the primary source of information; and the importance

of "stair-stepping" your talk, leaving "landings" of less content to let the audience catch its breath.

Larry Wall's yearly State Of The Onion address was a love letter to the Perl community. No big announcements, just a recap of the last year of Larry's life based on the theme of screensavers, and a heartfelt thanks to the people that keep Perl going. After Larry's talk, David Adler and Ann Barcomb presented the Perl Foundation's yearly White Camel awards. Awards went to Dave Cross, the current leader of the Perl Mongers user groups (<http://www.pm.org/>), and to two Perl magazine publishers: Jon Orwant is the original publisher of the magazine you're currently reading, and brian d foy is the publisher of *The Perl Review*, as well as the founder of the Perl Mongers. brian also had copies of *TPR*'s first print issue. It's great to see another magazine about Perl. It makes me feel like we're increasing our collective strength and mindshare.

Speaking of mindshare, Tim O'Reilly's talk on "What Book Sales Tell Us" gave some interesting computer trends reflected in publishing sales trends. Most notable is that PHP books are 50–100 percent better sellers than Perl books, which reinforces what I've been seeing: PHP's lower barrier to entry means more bodies, but more beginner programmers. Perhaps Perl will be seen as more of an "adult" programming language.

The session calendar also backed up the popularity of PHP. PHP was all over the place, especially with the recent release of PHP 5. PHP still doesn't have namespaces, but darned if people aren't doing a lot of cool stuff with it. Ruby was all but nonexistent this year. I hope Ruby hasn't had its day in the sun, fading away into obscurity.

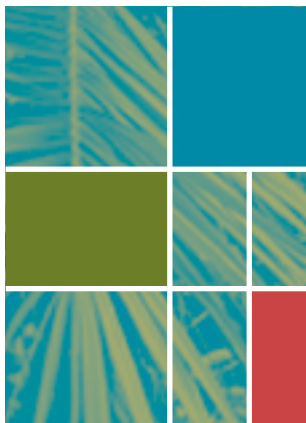
Finally, this year's Lightning Talks were a little different. Normally, Mark-Jason Dominus presents 16 talks in 90 minutes. This year, the reins were in the capable hands of Geoff Avery, who gave us 45 minutes of Lightning Talks, and 45 minutes of Current Projects talks. He referred to these project talks as "lightning talks without the time limit," where project leaders were able to get up and give a call for help on their current projects. Autrijus Tang discussed `svk`, a Perl-Subversion integration project he's involved with, and Joke Visser's `pVoice` generated a lot of interest. Joke's daughter Krista is unable to speak, and `pVoice` allows her to speak using the computer. See <http://pvoice.org/> for details. (And of course, I plugged the Phalanx project: <http://qa.perl.org/>.)

As always, OSCON was a blast, and well worth the time away from the family and the mountain of e-mail awaiting my return. It's amazing to watch the interaction of these people who make Perl happen. As immediate as e-mail and IRC and blogs may be, there's no comparison to having people meet in person. I ask everyone interested in Perl development to get somewhere that they can meet other Perl folks, whether it's OSCON, YAPC, or just a local Perl Mongers meeting. I'll see you next year!

TPJ

---

*Andy manages programmers for Follett Library Resources in McHenry, IL. In his spare time, he works on his CPAN modules and does technical writing and editing. Andy is also the maintainer of Test::Harness and can be contacted at [andy@petdance.com](mailto:andy@petdance.com).*



# Molecular Biology With Perl

*Simon Cozens*

A week or so ago, I got an e-mail from one of my housemates—yes, it’s that sort of house—showing me some Perl code she’d written. After June’s *The Perl Journal* article by Ivan Tubert-Brohman on Perl and chemistry, this code made me sit up and take notice. Now don’t worry if you don’t know any chemistry. Before this morning, neither did I. We’re going to look at a bit of chemistry, a bit of text processing, a little bit of optimizing basic Perl programs, and quite a lot of graphics programming.

You see, my housemate is an immunology researcher and spends her days (and most of her nights) playing with big glass columns full of proteins. Every so often, her proteins do something exciting and we get a step closer to understanding why the human body can suffer from autoimmune diseases like lupus and multiple sclerosis. Meanwhile, I’m writing web-based search engines or something similarly meaningless. It makes you think.

The Perl code that she’d been writing measures the distance between two atoms in one particular protein, a bit of a human with some Epstein-Barr virus mixed in. She’d got a file that told her where all the atoms were, and applied some trigonometry to work out the distance.

## PDB Format

Protein structure data is usually stored in a file format called PDB, after the Protein Data Bank, which collects a huge number of these protein files. A PDB file looks like this:

HEADER	AMINO ACID
ATOM 1 N ALA 0001	-1.053 1.300 0.614
ATOM 2 CA ALA 0001	-0.304 0.032 0.746
ATOM 3 C ALA 0001	0.770 -0.014 -0.311
ATOM 4 O ALA 0001	1.952 -0.167 -0.047
ATOM 5 H ALA 0001	-1.805 1.385 1.386
ATOM 6 O ALA 0001	0.354 0.125 -1.567
ATOM 7 H ALA 0001	-1.522 1.368 -0.358
ATOM 8 H ALA 0001	0.176 0.013 1.740
ATOM 9 C ALA 0001	-1.237 -1.200 0.610
ATOM 10 H ALA 0001	-2.007 -1.183 1.397
ATOM 11 H ALA 0001	-0.655 -2.129 0.709

*Simon is a freelance programmer and author, whose titles include Beginning Perl (Wrox Press, 2000) and Extending and Embedding Perl (Manning Publications, 2002). He’s the creator of over 30 CPAN modules and a former Parrot pumping. Simon can be reached at [simon-cozens.org](mailto:simon-cozens.org).*

```
ATOM 12 H ALA 0001 -1.737 -1.199 -0.371
ATOM 13 H ALA 0001 1.100 0.082 -2.154
```

This is a pretty simple one, for the amino acid alanine, one of the compounds that makes up DNA. There are a bunch of other lines that might be in this file, such as an AUTHOR line, some REMARK lines, and CONECT lines, which tell you how the atoms bond together. As we’ll see later, that information isn’t as important as it might appear.

Each of these ATOM lines contain a number of space-separated fields; again, not all of which are in use in this sample. Here’s a fuller one, which is actually part of an alanine acid in a bigger protein:

```
ATOM 65 CB ALA A 10 94.137 2.952 8.447 1.00 16.71
```

To make things a little clearer, I’ll add a ruler on top of it, like so:

```
12345612345x1212x123121234xxxx123456781234567812345678123456
ATOM 65 CB ALA A 10 94.137 2.952 8.447 1.00 16.71
```

Here we have an atom with a unique ID of 65; the next field tells us that this is C, a carbon atom, and then things skip about a little.

As well as identifying the atom with a unique ID, we want to give it a structural description that helps us understand what it’s doing in the molecule. So the fourth field, A, is a top-level structural marker—we’re in the first “bit” of the molecule. Then the third and fifth fields (ALA 10) tell us that this atom belongs to an ALANine acid, (the “residue”), which is the 10th residue in this bit of the molecule. To further qualify the atom, we give it a unique suffix, which tells us which particular carbon atom this is in that acid; the suffixes tend to go in a particular sequence: blank for the first such carbon atom, then “A” for alpha, “B” for beta, and so on.

Then we have the three numbers that are what it’s all about—the three-dimensional coordinates of that atom in the protein, as revealed by X-ray crystallography. The other two fields are more properties of the atom that we don’t need to deal with for now.

## Heather’s Code

What my housemate needed to do was to find the distance between alpha carbon atom in residue number 65 of the first part of the molecule, and alpha carbon atom in residue number 67 of the second part. Her code, slightly reformatted for brevity, looked like this:

```
opendir(DIR, ".");
@files = grep(/\.pdb$/, readdir(DIR));
```



```

closedir(DIR);

foreach $file (@files) {
    print "$file\n";
    open (FILE,$file) || die "Cannae dae it\n";
    while ($line = <FILE>) {
        if ($line =~ /CA ... A 65/) {
            $ax = substr($line, 31, 8);
            $ay = substr($line, 39, 8);
            $az = substr($line, 47, 8);
        }
        if ($line =~ /CA ... B 67/) {
            $bx = substr($line, 31, 8);
            $by = substr($line, 39, 8);
            $bz = substr($line, 47, 8);
        }
    }

    $distance = sqrt(($ax - $bx)*($ax-$bx) + ($ay - $by)*($ay-$by)
        + ($az - $bz)*($az-$bz));

    print "$distance \n";
}

```

When run in a directory containing Heather's protein, it produces the following output:

```

1H15.pdb
18.2286209297357

```

And that's the answer: 18.2 Angstroms between the two atoms.

## Basic Refinements

Now I hate myself for this, but it seems like my natural reaction when I look at someone else's Perl code is to see if I can do it better. So before I start mucking around with the program, let me say it has two major advantages over any other implementation: First, my housemate was able to get it written quickly and easily, and second, it gives the right answer. Anything else is window-dressing.

But permit me a bit of window-dressing.

When I sat down to write this article, I had gotten a directory with quite a few PDB files in it, and was quite surprised to see this output from the program:

```

1BNA.pdb
0
1H15.pdb
18.2286209297357
adna.pdb
18.2286209297357
ala.pdb
18.2286209297357

```

So the first problem is that we neither stop when we've found the answer nor reset our coordinates every time we open a new file. We'll do both of these things as our first refinement:

```

foreach $file (@files) {
    print "$file\n";
    open (FILE,$file) || die "Cannae dae it\n";
    my ($ax, $ay, $az, $bx, $by, $bz);
    while ($line = <FILE>) {
        # ...

    }

    $distance = sqrt(($ax - $bx)*($ax-$bx) + ($ay - $by)*($ay-$by)
        + ($az - $bz)*($az-$bz));

    print "$distance \n";
}

```

```

        last if $distance;
    }
}

```

(# ... represents parts we haven't changed.)

Additionally, since we have all the values we need once we've found the second atom, we can put a last in there, too, to make things more efficient:

```

while ($line = <FILE>) {
    if ($line =~ /CA ... A 65/) {
        $ax = substr($line, 31, 8);
        $ay = substr($line, 39, 8);
        $az = substr($line, 47, 8);
    }
    if ($line =~ /CA ... B 67/) {
        $bx = substr($line, 31, 8);
        $by = substr($line, 39, 8);
        $bz = substr($line, 47, 8);
        last;
    }
}

```

Now there's a lot of repetition in those 10 lines, and repetition is something that a computer programmer should try to avoid at all costs. I'll say that again: Repetition is something that a computer programmer should try to avoid at all costs.

Why? Suppose we got one of those offsets wrong—suppose the coordinates start at position 30, not 31. We have to change 31 to 30, and then 39 to 38, and then 47 to 46—and, to make things worse, we have to go down a couple of lines and change it all again. If something appears more than once, there's a reasonable chance that we'll end up forgetting to change it one of those times. In short: More repetitions means more maintenance.

Thankfully, sometime in the '60s, computer scientists came up with the concept of a subroutine to help us avoid repetition:

```

use constant COORDS_OFFSET => 31;

sub co_ords {
    my $line = shift;
    (substr($line, COORDS_OFFSET, 8),
     substr($line, COORDS_OFFSET + 8, 8),
     substr($line, COORDS_OFFSET + 16, 8))
}

while ($line = <FILE>) {
    if ($line =~ /CA ... A 65/) {
        ($ax, $ay, $az) = co_ords($line);
    }
    if ($line =~ /CA ... B 67/) {
        ($bx, $by, $bz) = co_ords($line);
        last;
    }
}

```

Similarly, we can use the exponentiation operator (\*\*) to avoid repeating terms when calculating the distance:

```

$distance = sqrt(($ax - $bx) ** 2 + ($ay - $by) ** 2
    + ($az - $bz) ** 2);

```

This isn't just trying to shave characters off a program for the sake of it—this is reducing the amount of maintenance we have to do in the future. And now we've got the kind of program as I'd write it—although, to be honest, I'd still be casting an eye over those repeated *substrs*.

## Handling PDB Files

The *substrs* would be bothering me because Perl has a bunch of tools for extracting data from lines of text and, while *substr* is a perfectly good one, it's not great at doing several things at once. Since we're using regular expressions to match the atoms we want, a better way might be to also use them to extract the coordinates.

```
while ($line = <FILE>) {
    if ($line =~ /CA ... A 65 (.8){.8}){.8})/) {
        ($ax, $ay, $az) = ($1, $2, $3);
    }
    if ($line =~ /CA ... B 67 (.8){.8}){.8})/) {
        ($ax, $ay, $az) = ($1, $2, $3);
        last;
    }
}
```

This is a lot better for the simple reason that it keeps the information about the structure of a given PDB file line in one place, where previously we had the structure encoded in a regexp and also in offsets in a subroutine.

Perhaps better still, though, might be to use the *unpack* function—this is used primarily to unpack fixed-length records from a binary file, but it works just as well to unpack fixed-length records from a text file, such a PDB file. Let's look at the PDB line with its ruler again:

```
1234561234561212x123121234xxxx123456781234567812345678123456123456
ATOM      65  CB  ALA  A  10      94.137   2.952   8.447   1.00  16.71
```

Now, to create an *unpack* format from this, we simply turn our field length specifiers (*123456*) into the letter *A* (for ASCII) followed by the length: *A6*. As for the *x*'s, which represent unused columns—well, they just stay as *x*'s, to give us:

```
my $pdb_format = "A6A5xA2A2xA3A2A4xxxxxA8A8A8A6A6";
while ($line = <FILE>) {
    my @fields = unpack($pdb_format, $line);
    ...
}
```

This is great if we're going to use all the information in the line, but because we're only using a little bit of it, we actually end up doing a bit more work:

```
if ($fields[0] eq "ATOM" and $fields[2] eq " C"
    and $fields[3] eq "A " and $fields[5] eq " A" and
    $fields[6] == 65) {
    ($ax, $ay, $z) = @fields[7,8,9];
}
```

Yuck. So in this case, it looks like using regular expressions is the right way to do it. Of course, being a lazy Perl programmer, the first thing I really did when looking at this program was to look on <http://search.cpan.org/> for PDB, and I turned up *Chemistry::File::PDB* and *Chemistry::Mol*, the subject of Tubert-Brohman's June *TPJ* article.

Now in this instance, *Chemistry::File::PDB* turns out to not be that useful—the columns in the PDB file that deal with the residue sequence number are biology specific and aren't picked up by the more generic *Chemistry* module.

But it did give me another idea.

## Visualization with OpenGL

While Heather was busy working on her molecules, another friend was playing with writing some new OpenGL bindings for Perl. OpenGL is a graphics library that makes writing 3D visualization

applications really easy. I had been thinking about visualizing molecules into those lovely ball-and-stick models that you used to get in high-school chemistry, but I thought it would be way too difficult to work out how all the atoms fit together and where to place them.

But here was my housemate with her PDB files, which told me the precise coordinates of where each atom was supposed to go. The coordinates were being handed to me on a plate, and OpenGL provided a simple way of drawing spheres at a particular set of

---

*My housemate is an  
immunology researcher and  
spends her days (and most of her  
nights) playing with big glass  
columns full of proteins*

---

coordinates, so shouldn't it be trivial to write my own molecular visualization software to rival *rasmol* or *pymol*?

Well, it was that simple. The first thing I needed to do was set up the OpenGL framework—get the window up and running, set light and material parameters, and tell OpenGL what to do when the mouse moved or the window resized. Thankfully, most of the hard work is hidden in the *OpenGL::Simple::Viewer* module, which provides default handlers for OpenGL events. We begin by using the relevant modules and setting up GLUT, the OpenGL toolkit:

```
glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );

glutInit;

my $window = glutCreateWindow("PerlMOL");
glutDisplayFunc( basic_displayfunc(\&visualize) );
glutReshapeFunc( \&basic_reshapefunc );
glutMouseFunc( \&basic_mousefunc );
glutMotionFunc( \&basic_motionfunc );
```

The *basic\_...* functions are provided by *OpenGL::Simple::Viewer*. Apart from *basic\_displayfunc*, they provide handlers that allow you to resize, rotate, translate, zoom, and image using the mouse. This is pretty much all we need for a basic molecular visualizer. *basic\_displayfunc* provides a wrapper around a subroutine we provide, which will do all the actual drawing.

We also need to set up some shading and lighting:

```
my @LightAmbient = ( 0.4, 0.4, 0.4, 1.0 );
my @LightDiffuse = ( 0.9, 0.9, 0.9, 1.0 );
my @LightSpecular = ( 0.1, 0.1, 0.1, 0.1 );
my @LightPos = ( 0, 0, 2, 1.0 );
glShadeModel( GL_SMOOTH );

glLight( GL_LIGHT1, GL_AMBIENT, @LightAmbient );
glLight( GL_LIGHT1, GL_DIFFUSE, @LightDiffuse );
glLight( GL_LIGHT1, GL_SPECULAR, @LightSpecular );
glLight( GL_LIGHT1, GL_POSITION, @LightPos );
```

```
glEnable(GL_LIGHT1);
glEnable(GL_LIGHTING);
glEnable(GL_DEPTH_TEST);

glColorMaterial( GL_FRONT, GL_AMBIENT_AND_DIFFUSE );
glEnable(GL_COLOR_MATERIAL);
```

And the final thing we need to do before diving into writing the drawing code is to call the main loop:

```
glutMainLoop;
```

---

*You don't have to be a biologist to play with OpenGL—any kind of structure or surface can be modeled, displayed, and then viewed from multiple angles*

---

So far, we've brought up the window, set the parameters, and told it to go, calling out visualize subroutine to work out what to do.

Since the molecule is not going to change while we're rotating or zooming around it, we build up and cache a model outside of doing the drawing. First, we want to open the molecule file using *Chemistry::File::PDB*, and then find all the atoms:

```
use Chemistry::File::PDB;

my $mol = Chemistry::MacroMol->read(shift);
my @atoms = $mol->atoms;
for my $atom (@atoms) {
```

We're going to need to know three things about each atom: the coordinates (where to put it), the mass (which relates to how big we're going to make it), and the element type (which relates to what color we're going to make it). Thankfully, *Chemistry::Mol* already handles each of these.

First the mass. The problem with the mass is that hydrogen atoms are very, very light; so much lighter than anything else that if we draw the atoms to a linear scale, they look tiny. So we scale the atomic mass by taking a logarithm to smooth out the differences in atoms, and then scale again by a linear factor to make it look reasonable:

```
my $mass = log( 1 + $atom->mass ) / $mass_scale;
```

Next, the color. We set up two color conversion tables, for convenience. The first maps names of colors to OpenGL vectors:

```
my %color = (
    red    => [ 1, 0, 0, 1 ],
    yellow => [ 1, 1, 0, 1 ],
    orange => [ 1, 0.5, 0, 1 ],
    green  => [ 0, 1, 0, 1 ],
    cyan   => [ 0, 1, 1, 1 ],
```

```
    blue   => [ 0, 0, 1, 1 ],
    magenta => [ 1, 0, 1, 1 ],
    grey   => [ 0.5, 0.5, 0.5, 1 ],
    white  => [ 1, 1, 1, 1 ],
);
```

And the second maps elements to colors; I've used a relatively standard color map here:

```
my %element_colors = (
    C => $color{grey},
    O => $color{red},
    N => $color{blue},
    H => $color{white},
    S => $color{yellow},
    P => $color{orange},
);
```

Now for each atom, we can determine the color:

```
my $color = $element_colors{ $atom->symbol } || $color{cyan};
```

We use cyan as a default, in case there are any unexpected atoms like potassium cropping up in our molecule. Finally, we can extract the coordinates from the atom object:

```
my @coords = $atom->coords->array;
```

So for each atom, we put all these things into an array reference and place it in an array:

```
push @ballpoints, [ $color, $mass, @coords ];
}
```

This is the model of our molecule. Now we need to draw it! Thankfully, OpenGL makes things very simple:

```
sub visualize {
    if ( !@ballpoints ) { make_model() }
    for (@ballpoints) {
        my ( $color, $mass, @coords ) = @$_;
        glColor($color);
        glPushMatrix;
        glTranslate(@coords);
        glutSolidSphere( $mass, $sphericity, $sphericity);
        glPopMatrix;
    }
}
```

We set the color, translate the drawing frame to the appropriate coordinates, and draw a circle with the given mass and sphericity (sphericity is a constant that tells OpenGL how many segments to make into a circle). In the OpenGL world, everything is made up of polygons; the more polygons in a sphere, the more spherical it looks, but the more complicated it is to render. I've found that a sphericity of about eight provides attractive but not overly resource-heavy atoms.

And, well, that's it. We can load up our molecules, view the atoms on the screen, and rotate and zoom them with the mouse. Great. Except for one thing: Where are the bonds?

## Bonds

We've got the balls in our ball-and-stick diagram, but no sticks—we don't know how the atoms fit together. The PDB file has some CONECT records that purport to tell us this, but *Chemistry::File::PDB* does not read them—in part because they are not always complete. Instead, we can work out the bonds between atoms by applying science.

Just like two large objects have a gravitational attraction to each other, atoms that are close to each other stay in position by means of something called the “Van der Waals forces.” Practically, this means that we can tell if two atoms are bonded by seeing if the distance between them is less than a particular boundary value, which is a function of the mass of the two atoms in question.

Of course, to check the distance and mass of every pair of atoms to see if they’re bonded is a horrendous task, and so we have two ways to avoid this: We can either use a clever algorithm or we can let someone else do the work. I prefer the second solution, and a couple of moments searching on CPAN brought me to *Chemistry::Bond::Find*:

```
use Chemistry::Bond::Find qw(:all);
find_bonds($mol);
```

And now we can query *\$mol->bonds* to link our atoms together. Inside our *build\_model* routine, we put:

```
for my $bond ( $mol->bonds ) {
    my ( $from, $to ) = $bond->atoms;
    my @from = $from->coords->array;
    my @to   = $to->coords->array;
    push @ballsticks, [ \@from, \@to ];
}
```

Each bond consists of a pair of atoms, so we just take note of the coordinates of each atom in the pair so we can draw a line between them. Once we have this data, we can add the following to the *visualize* routine:

```
glColor( @{ $color{'grey'} } );
glLineWidth(4);
for (@ballsticks) { glBegin(GL_LINES); glVertex(@$_) for @$_; glEnd; }
```

And that’s all—for each stick we start a *GL\_LINES* object, pass in the coordinates of our two vertices, and then end it. Now we get to look at molecule diagrams like Figure 1.

## Perlmol’s Progress

There’s a lot more than can be done with this project, which I’ve called “perlmol” (as a nod to Python’s “pymol,” a similar visualization tool). These future extensions come in two categories: things to do with biology and things to do with OpenGL.

For instance, a simple biology change could be to alter the way atoms are colored; another way molecular biologists like to visualize their molecules is to color each set of atoms according to the residue to which they belong. We can do this by altering our *build\_model* routine:

```
my $color = gen_color($atom->{attr}{"pdb/residue_name"});
```

*gen\_color* is a handy little routine that dishes out consistent colors to each textual “tag” (in this case, the residue name) that comes its way: The first one will be red, the second yellow, the next orange, and so on. It works by maintaining a mapping between tags and colors, and using a sequence to assign a color to a tag that doesn’t already have one:

```
{
    my %ccache;
    my @colors = values %color;
    my $iter = 0;
    sub gen_color {
        my $key = shift;
        $ccache{$key} ||= $colors[$iter++ % $#colors];
    }
}
```

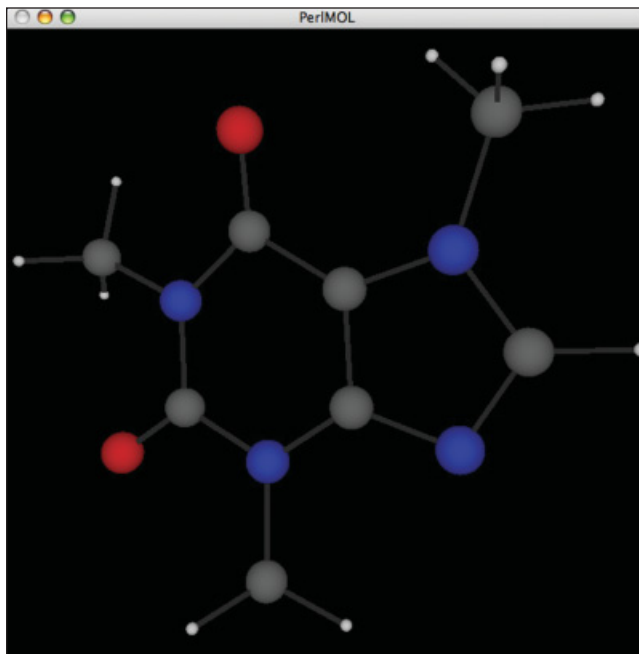


Figure 1: A molecule diagram. (This one happens to be caffeine.)

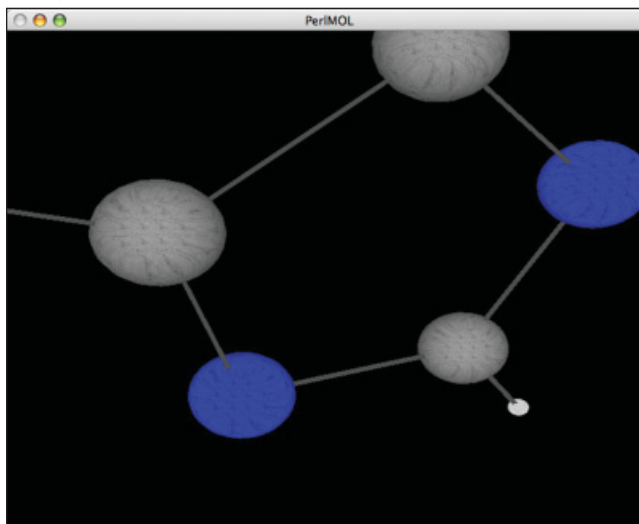


Figure 2: A texture-coated molecule.

An OpenGL change, on the other hand, would be something like allowing you to coat your molecules in different textures, as shown in Figure 2.

One planned change that covers both areas is that biologists don’t usually look at the whole swarm of balls-on-sticks for a molecule of many thousands of atoms; instead, they track what’s called the “backbone,” a particular series of atoms that runs from residue to residue. Normally, this is displayed as a “ribbon” swirling around the molecule. Offering this as an option would make visualizing large molecules much more comfortable.

But you don’t have to be a biologist to play with OpenGL—any kind of structure or surface can be modeled, displayed, and then viewed from multiple angles in a variety of textures, materials, or lighting conditions, and the *OpenGL::Simple* modules make it incredibly easy to do that. Happy visualizing!





# Pipelines and E-Mail Addresses

*brian d foy*

Lately, I've had to do a bit more work on my e-mail inbox to separate the good messages from spam. With some pipelines, some Perl, and a little bit of procmail, I get most of the work done although I have to keep trying new things to keep up with the spammers' tricks.

Spam is getting to be a bigger and bigger problem and many people are saying that most of the e-mail they get is now spam. That's certainly true for me: I get over a hundred spam messages for every valid one.

The trick, so far, is filtering the messages to pick the few good messages out of the torrent of spam and there are plenty of tools to do this. I use a procmail tool called SpamBouncer by Catherine Hampton, and the Spam Assassin Perl package is very popular.

I filter the spam to a folder that I can check later and, when I do, I usually find that a message was incorrectly identified as spam. The sender may use a service provider that I identify as a spam relay, the message may contain an unfortunate combination of keywords (like "On my vacation, I was on the bank of this Nigerian river") or many other things that inadvertently trigger the spam. Each time I find one of these false positives, I add that address to the white list.

Before I start the other spam filters on my incoming mail, I check if the From address is in my local white list. I filter those directly into my inbox:

```
:0 f
* ? formail -x"From:" | egrep -is -f whitelist.txt
| /path/to/my/inbox
```

I seeded my white list with all of the addresses that I had answered in the past three months by using a quick and dirty command pipeline with three parts. Each part does a little bit of processing then hands off the result to the next, which does another little part. This comes from the UNIX philosophy to use a lot of small tools that do specific jobs rather than larger programs that try to do several things at once. I can string together small programs in many ways to do many different tasks:

---

*brian has been a Perl user since 1994. He is founder of the first Perl Users Group, NY.pm, and Perl Mongers, the Perl advocacy organization. He has been teaching Perl through Stonehenge Consulting for the past five years, and has been a featured speaker at The Perl Conference, Perl University, YAPC, COMDEX, and Builder.com. Contact brian at comdog@panix.com.*

```
grep ^From: answered-mail | perl -ple 'chomp; s/^From:\s+//; s/\s*(.*?)\s+//;
s/.*<(.@.*)>.*$/1/;' | sort -u
```

The *grep* pulls out the *From:* line in each message (and that picks up the original address in forwarded messages, too, but I decided to keep those—if I trust the person who forwarded it, I should probably trust the original sender as well). Out of the *grep* command flows a list of *From:* lines that goes through the pipe operator and becomes the input of the *perl* command.

The *perl* one-liner looks a little too simple, and although I tried more complex Perl, I decided that simpler was better. I do a *chomp* and three substitutions to get rid of everything in the line but the e-mail address: one to get rid of the *From:* at the front of the line, and the other two to get rid of everything else but the e-mail address, which usually comes in one of three formats:

- barney@example.com
- barney@example.com (Barney Rubble)
- "Barney Rubble" <barney@example.com>

The second substitution, *s/\s\*(.\*?)\s\*//*, gets rid of the bits in the parentheses, including the whitespace around them, leaving the first and second examples as just the e-mail address. The last substitution, *s/.\*<(.@.\*)>.\*\$/1/*, matches the bits in angle brackets and gets rid of everything else. There are some cases that get past these substitutions, but in the rare case that happens, I can fix it by hand quicker than I can write code that will handle every possible case.

Out of the Perl portion flows an unordered list of e-mail addresses in whatever order they appear in the input and probably with a lot of duplicates. The *sort(1)* command takes care of those. It puts the addresses in order and the *-u* switch removes duplicates.

This is where things get interesting because, even though I've seeded my file, I don't like the order that *sort(1)* spits out. The various sort functions (command-line and *perl*'s) sort "alphabetically." They don't know anything about what the data represents or why I care about it. The default sort functions simply compare the inputs character-by-character and, if I use *sort(1)*, my white list is going to be sorted as meaningless strings.

The lines in my white list do have meaning, though. Each line is an e-mail address and, in each address, there is a user name and a domain name. Instead of sorting as meaningless strings, I want to order the addresses by their domain, and within each

domain, by the user. For instance, I want all of the “stonehenge.com” addresses to show up before the “tpj.com,” and with-in all the “stonehenge.com” addresses, I want “brian” to show up before “merlyn.” That makes it easy for me to see all of the addresses in a particular domain, which I sometimes need if I want to quickly inspect the white list for all of the addresses in a particular domain.

This kind of sort is easy in Perl and it is another pipeline, sometimes called a “Schwartzian Transform.” I start with an unsorted list of addresses and I want to end up with a list of unique addresses sorted by domain name. I am going to make my program “pipeline-able.” I take input from either the files specified on the command line or standard input. At the end of processing, I send the result to standard output so I can pipe it into another command if I like:

```
#/usr/bin/perl

$, = "\n";

print
map { $_->[0] }
sort { $a->[2] cmp $b->[2] or $a->[1] cmp $b->[1] }
map { [ $_, split /\@/ ] }

keys %{
    map { chomp; s/\\s*(.*?)\\s*//; s/.*(.*@.*)*/$1/; ( lc $_, 1 ) }
    grep { s/^From:\\s+// }
    <>
};
```

There are pipelines within the program, too. I use several Perl list operators and they do the same thing that pipelines do: Take an input list, do something to it, and output another list. I can line these up just like I did with programs on the command line.

I find it easiest to read Perl pipelines from bottom to top (or right to left). Each part of the pipeline depends on the part that comes after it, so *perl* has to evaluate that part first.

Knowing that, I have two parts to my program: the chunk after the *print()* and the part after the *keys()*. The *keys()* portion gets the addresses and pulls out the unique ones. It is the same thing that I did on the command line, but all in Perl. The *print()* portion handles the ordering using a map-sort-map pipeline. Each one is a pipeline and the *keys()* pipeline flows into the one above it.

Inside the *keys()* pipeline, I have to do a bit of trickery. The *keys()* function takes a hash, so I need a hash. I know I want to avoid intermediate variables; thus, I start with `%{ }`. That dereferences a hash reference. Inside that dereference, I need a hash reference, so I use the anonymous hash constructor, `{ }`. When I put those together, I get doubled curly braces `%{ { }`.

Inside my doubled curly braces, I have a pipeline built from two list operators and the file input operator (aka, the diamond operator), `<>`. Without command-line arguments, the `<>` reads from standard input. It spits out lines of input that become the input for the *grep()*, which takes the place of the command-line program of the same name. The *grep()* only passes through lines that make its condition, `s/^From:\\s+//`, return True. The *s///* operator returns True if it was able to make the substitution, and I match lines beginning with *From:* at the same time that I take that string off of the line. The matching lines flow into the *map()*, which handles most of the stuff I did in my command-line version. At the end of the *map()*, I return a short list, the e-mail address, and 1, which I use as a key-value pair. All of these key-value pairs come out of the *map()*, and the anonymous hash constructor puts them together for the *keys()* function.

The *keys()* pulls off the unique e-mail address. If I had duplicate addresses in the input, I got rid of the extras when I used the

address as a hash key. The *keys()* command turns the hash into a list I pass into the next major part: the Schwartzian Transform that will order the address.

The Schwartzian Transform has already been dissected at length elsewhere, so I’ll skip a lengthy discussion. In short, the transform is a map-sort-map and something other than the original input to order the elements. In the first *map()*, I create an anonymous array of three elements: the original string in `$_`, and the user and domain portions of the address I get from splitting the original string. I pass those anonymous arrays to the sort, which compares the domain (`$a->[2]`) and, if those match, compares the user (`$a->[1]`). Once ordered, the last *map()* extracts from each anonymous array the first element, which is just the original string and the address that I want.

The last step is a *print()*. I might want to use this program in another pipeline, so I output all of the addresses, one address between line. I previously set the Perl special variable `$`, which inserts its value between each element of the list I give to *print()*. In this case, I put a newline between each address and send them to standard output.

Now with this program, I can add new addresses to my white list with a much shorter command line. With a bit of output redirection (`>>`), I can add those directly to my white list:

```
% sort_addresses >> white_list.txt
```

And, I can even use it to reorder my white list.

```
% mv white_list.txt white_list.bak
% sort_addresses white_list.bak > white_list.txt
```

Because I wrote my program as a pipeline, I can use it to do other things, too, such as list everyone who sent me mail in a particular month:

```
% sort_addresses read-mail-jun-2004
```

I can even use this program directly from my newsreader. I use PINE, which has an *enable-unix-pipe-cmd* option (although your system administrator may have turned this off). While I highlight a message in the index view or read a message, the `|` key leads to a prompt and, as long as my program is in my path, I can type just its name. PINE shows me the program output:

```
Pipe message 37 to : sort_addresses
```

I can also redirect that output directly to my white list:

```
Pipe message 37 to : sort_addresses >> white_list.txt
```

Now, I have a program that I created to update my white list so I could pick out the good e-mail addresses from the flood of spam but, since I wrote it to work in a pipeline, I can use it for other things also. Perl lets me do this because it has list operators that also act as pipelines. You may not want to make your entire program one long pipeline, but you can certainly use smaller versions of this example to streamline parts of your program.

## References

procmail: <http://www.procmail.org/>  
 SpamBouncer: <http://www.spambouncer.org/>  
 PINE: <http://www.washington.edu/pine/>  
 Schwartzian Transform: <http://www.stonehenge.com/merlyn/UnixReview/col06.html>

---

# Source Code Appendix

---

## Curtis Lee Fulton “XML Subversion”

### Listing 1

```
# fetch a database row as hash ref
my $data = {};
$data->{users} = $db->selectrow_hashref('select * from users where id = 2');

# init the new EasySQL data object
my $data_object = EasySqlChildClass->new({data=>$data});

# get the root XML element
my $xml = $data_object->getXML();

# make changes to the XML document
$xml->username->setString('curtisleeefulton');
$xml->bio->setAttr('age', 22);
$xml->bio->city->setString('Portland');
$xml->history->access->setAttr('last', time());

# output entire XML doc as string to STDOUT
print $xml->getDomObj->toString();

# update the database
my $sql = $data_object->getSQL();
my $q = "update users set ".$sql->{users}." where id = 2";
$db->do($q);
```

### Listing 2

```
package User;
use XML::EasySQL;
@ISA = ('XML::EasySQL');

use strict;

sub new {
    my $proto = shift;
    my $params = shift;

    # the XML schema string
    $params->{schema} = q(
<schema name="users" default="attrib" default_table="users">
<columns>
<id type="attrib"/>
<group_id type="attrib"/>
<email type="string"/>
<bio type="element"/>
<history table="comments" type="element"/>
</columns>
</schema>
);
    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new($params);
    bless $self, $class;
};

1;
```

### Listing 3

```
# fetch the data from the database
my $data = {};
$data->{users} = $db->selectrow_hashref('select * from users where id = 2');
$data->{comment_data} = $db->selectrow_hashref('select * from comments where id = 183');

# construct the data object
my $user = User->new({data=>$data});

# modify the data
my $xml = $user->getXML();
$xml->username->setString('curtisleeefulton');
$xml->bio->setAttr('age', 22);
$xml->bio->city->setString('Portland');
$xml->history->access->setAttr('last', time());

# write the changes to the database
my $sql = $user->getSQL();
my $q = "update users set ".$sql->{users}." where id = 2";
$db->do($q);
my $q = "update comments set ".$sql->{comments}." where id = 183";
$db->do($q);
```

### Listing 4

```

package Base;
use XML::EasySQL;
@ISA = ('XML::EasySQL');

use strict;

sub new {
    my $proto = shift;
    my $params = shift;
    my $db = $params->(db);
    my $schema = $params->($schema);
    my $data = {};
    foreach my $table (keys %{$params->(query)}) {
        my $d = $db->selectrow_hashref("select * from $table
            where id = ".$params->(query)->($table));
        foreach my $k (keys %{$d}) {
            $data->{$k} = $d->{$k};
        }
    }

    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new((data=>$data, schema=>$schema));
    bless $self, $class;
}

sub save {
    my $self = shift;
    my $sql = $self->getSQL();
    foreach my $table (keys %{$sql}) {
        my $q = "update $table set ".$sql->($table).
            " where id = ".$self->(query)->($table);
        $db->do($q);
    }
}

1;

```

## Listing 5

```

package User;
use Base;
@ISA = ('Base');

use strict;

sub new {
    my $proto = shift;
    my $params = shift;

    # the XML schema string
    $params->(schema) = q{
<schema name="users" default="attrib" default_table="users">
<columns>
<id type="attrib"/>
<group_id type="attrib"/>
<email type="string"/>
<bio type="element"/>
<history table="comments" type="element"/>
</columns>
</schema>
};

    # get the SQL data
    $params->(query)->(users) = $params->(user_id);
    $params->(query)->(comments) = $params->(comment_id);

    # save the ids
    $self->(query) = $params->(query);
    my $class = ref($proto) || $proto;
    my $self = $class->SUPER::new($params);
    bless $self, $class;
}

1;

```

## Listing 6

```

# fetch the root XML::EasySQL::XMLobj object from your data object.
my $doc = $easy_sql_object->getXML();

# read from document
my $text = $doc->root->some_element($index)->getString;
my $attr = $doc->root->some_element($index)->getAttr('attrib');
my $element = $doc->root->some_element(34);
my @elements = $doc->root->some_element;

# first "some_element" element
my $elements = $doc->root->some_element;
# list of "some_element" elements
my @elements = $doc->root->some_element;

```



---

```
# write to document
$doc->root->an_element->setString('some string')
$doc->root->an_element->setAttr('attrib', 'val')
$doc->root->an_element->setAttr('attrib1' => 'hello', 'attrib2' => 'world')

# access elements with getElement, rather than a method call
my $element = $doc->root->getElement('element_name')->getElement('another_element_name');

# get at the XML::DOM object
my $dom = $doc->root->foobar->getDomObj;

# get elements with unknown tag names
my @elements = $doc->root->getElement();
my $fourth_element = $doc->root->getElement('', 4);

# remove elements and attributes
$doc->root->remElement('kill_this_element', 2);
$doc->root->tag_name->remAttr('kill_this_attribute');
```

***TPJ***