

The Perl Review

Volume 0 Issue 0

February 1, 2002

Release early, release often

Publisher's note	i
About this issue	i
Community news	ii
Extreme Publishing	1
<i>brian d foy</i>	
Structured Classes in Perl	4
<i>Robby Walker</i>	
perlhacktut - so you want to be a Perl porter?	9
<i>Simon Cozens</i>	
Benchmarking Perl	30
<i>brian d foy</i>	
Book Review: <i>Learning XML</i>	37

Web Access <http://www.ThePerlReview.com/> **Email** letters@theperlreview.com **Publisher** brian d foy **Editor** David H. Adler **Technical Editors** Kurt Starsinic, Adam Turoff **Contributors** Robby Walker, brian d foy, David H. Adler, Simon Cozens **Copyright** Format Copyright © 2002 *The Perl Review*, article content Copyright © 2002 by their respective authors.

From the Publisher

A long time ago, in a city far, far away, a lowly MIT graduate student had an idea – “How can I avoid working on my thesis, fill up my apartment with cardboard boxes, and use up any remaining free time that I have?” Hence, *The Perl Journal* was born, and it was good. It created an almost religious following amongst its subscribers who came to feel that the magazine belonged to them. It was a very Readable Publication, after all.

But its publisher had to move on, and, as any good open source programmer would, looked for a worthy home for TPJ and found one. The forces of irrational exuberance, inflated stock prices, and some say plain stupidity and malicious intent at this new home almost destroyed TPJ before he was able to rescue it and find it another, better home right before the telecommunications industry bubble burst and Mr. Greenspan went on an eleven quarter interest rate cut streak that dried up advertiser dollars.

Yet Another Society had grown and grown and grown and taken over Perl Mongers, Perl Monks, and Perl guru Damian Conway. Larry Wall, the creator of Perl, showed his support of YAS with a \$14 donation. Things were looking good, leaving me looking for something new and exciting to do because like the MIT graduate student, I am too busy to run a magazine, so why not?

From the Editor

brian seems to have covered everything, so I guess that gives me license to ramble.

Welcome to Perl Magazine: The Next Generation. There have been various rumbblings about starting up another magazine devoted to Perl for some time now. I'm glad that not only is something finally coming of that, but that I can be a part of it.

As you'll see in the “Extreme Publishing” article in this issue, you're a part of it too. We want to do what we can to make this a quality publication and, in many ways, that's up to you. As time goes on, your input will help guide our path. Moreover, and here's the part for which I have some responsibility, you need to contribute.

This won't be much of a magazine without content. The editorial staff can't write *all* of it. In the months

to come, I'll be looking for articles. Bring us four, five, yes, even *twenty* articles! Well, all right, maybe that just applies to Damian. I am, of course just joking, Damian. Damian? Damian?? Medic!!

Seriously, I look forward to working with many of you. This should be a fun ride. . .

About this issue

We hope to create a print magazine devoted to Perl, but that is a lot of work. Our staff likes Extreme Programming (XP) methodologies quite a bit so we are applying it to this venture which means that we will release early so that you get a usable product as soon as possible and that we can build up to the final product in steps. Our first article, “Extreme Publishing” discusses this further. “Benchmarking Perl” connects us to the past since it previously appeared in *The Perl Journal*, although the author has updated it. Everything else we are learning as we go.

Submit your work

We need articles about Perl and what you are doing with with Perl. We want the articles to talk about more than just Perl, though. Can you connect Perl with extreme programming, design patterns, quality control, debugging, or other software engineering techniques? Can you explain how perl works? Specifically, we would like to devote future issues to these topics:

- Debugging Perl
- Testing Perl
- Parsing with Perl

Advertisers

Would you like to advertise? We would like to start including advertisements in the next issue, which will be another electronic version. Anyone who has donated or given favors to Perl Mongers gets free space in any of our digital issues. Contact us to discuss the details.

Contact us

Until we get things sorted out, send all enquiries to brian d foy at comdog@panix.com.

Community News

German Perl Workshop 4.0

<http://www.perlworkshop.de/2002/>

The original German Perl get together is right around the corner February 13-15 at the Polytechnic University of Applied Sciences Bonn-Rhein-Sieg, Sankt Augustin Campus (near Bonn), Germany.

YAPC::NorthAmerica

<http://www.yapc.org/America/>

After a long wait, the site for this year's YAPC::NorthAmerica has been announced. The conference will be held at Washington University in St. Louis, Missouri, June 26th through 28 2002. This year's theme is "Perl in Nature". The deadline for submissions is May 1, 2002.

O'Reilly Call for Papers

<http://conferences.oreillynet.com/os2002/>

The Open Source Convention 4 will be in San Diego again July 22-26, 2002. The convention, which started as the Perl Conference, now covers Perl, MySQL, PHP, Python, XML, Linux, and other key open source technologies. Proposals for talks and tutorials are due March 1. Registration opens in April.

YAPC::Europe

<http://www.yapc.org/Europe/2002/index.html>

Germany gets a second Perl events this year. The 2002 YAPC::Europe, hosted by the Munich Perl Mongers, will be at Technische Universität München September 18-20. Registration is now open, and the conference group is accepting proposals for papers and presentations. The conference fee is 89 euros.

The Perl Foundation

<http://perl-foundation.org/>

The Perl Foundation, an arm of Yet Another Society (YAS), has raised over \$60,000 for their Perl Development Fund. YAS support Perl guru Damian Conway for a year and is looking to do it again, along with new projects.

The Perl Review wants you

Our new Perl magazine needs help. Besides submissions, we also need people to adopt the news column, the Perl golf column, and other features that we might add. We would like to expand our list of technical and editorial reviewers. Contact us at com-dog@panix.com if you are interested.



Extreme Publishing

brian d foy

Abstract

The tenets of Extreme Programming include feature stories, simplicity, short release cycles, planning for change, and on-site customers. I illustrate these self-referentially by applying to them to how we publish this magazine, which I call Extreme Publishing, but you can apply them to almost any project on which you might work.

1 Short stories

In Extreme Programming (XP), customers and developers describe product features in terms of user stories which are short descriptions of the feature along with the value that the feature provides. The team divides the stories by the value that they add to the product, how long they take to implement, and how important they are to other features. The *Extreme Programming* series recommends using index cards for the stories so it is easy to move them around, annotate them, rip them up, or separate them into stacks.

Once the team sorts the stories, they decide how soon they can deliver at least some value. You do not have to completely finish the entire project to make something useful, and you should start with the least amount of work to finish the first story that provides value.

When I first thought about starting a new Perl magazine, I gave myself a six-month cooling off period to do my homework about what it would take to do everything. I started creating stories. Here is a list of some of the story titles I have at the moment.

- Get subscribers
- Get articles from authors
- Edit articles
- Layout magazine
- Send magazine to printers
- Send magazine to subscribers

Based on these stories, XP tells me to find out how to deliver the most value to the customer as soon as possible by dividing them into short release cycles, where “short” means something measured in a small number of weeks rather than months. Anything much longer than a month is too long. Let developers go off for months at a time without deliverables and you set yourself up for a lot of anguish.

I start going through the stories to see what I can put off until later. Do I need to send something to the printers and then subscribers or can I put that part off? That depends on what the customer and I think the value really is. Does the customer want content or form first? How much work do I have to do to deal

with printers and subscribers and how much immediate value does that add? It turns out that dealing with subscribers, printers, and the post office is a lot of work, and that particular work delays immediate value.

I decided, after talking to a lot of people, that if I could bootstrap the magazine to the point where I could publish a print version, then you, as the customer, get more value sooner. I maximize value by publishing an electronic version, but I also finish more stories by publishing PostScript (or its lesser, more accessible cousin, PDF) because I can send postscript to a service bureau which can turn it into a magazine. Thus, I make the first milestone an electronic “preview” issue.

2 Simplicity

The key to XP is simplicity, whether in design, process, or implementation. Work smarter rather than harder. Developers break up stories that are complex into two or more simpler stories. I broke down the goal, sending you a print magazine, into much more simple stories, such as laying out the content, so that I could complete stories quickly, which maintains a sense of accomplishment, which XP calls the *project velocity*. Higher project velocities keep teams motivated. A continuing series of small successes motivates more than the spectre of one big failure.

Simplicity also requires me to not go too far off the beaten path. I could have used software like Adobe Framemaker or Quark, which require me to worry about all sorts of details. Some printers require these formats, and have long troubleshooting lists for them. I could have re-invented the wheel by starting my publishing quest by writing a Perl module to handle the layout. After months of work on that maybe I would have something moderately useful. However, my goal is not to write Perl modules, or any code for that matter. None of my stories say that I need to produce code, and any time I start coding I am doing something that should support the project, rather than *be* the project.

I chose T_EX, or specifically L^AT_EX, which I already knew, I did not have to pay for, produces beautiful output almost completely on its own, and already has magazine modules on the Comprehensive T_EX Archive Network, which is the sort of thing Perl people like. Good service bureaus (and in this case, your printer) can deal with postscript without the hassle of some of the commercial formats. Additionally, L^AT_EX, in its simplest form, can be analogous to POD, which a lot of Perl people already know. A competent programmer could write a workable pod2latex in a half hour if it did not already come with Perl. The editors can accept submissions in POD, a perly thing to do, which lets the author think about the writing rather than the format.

3 You are the customer

Another major tenet of XP is the Onsite Customer, which I simply call the Customer. The Customer adds to the value because he can work directly with the developers, modify and enhance the stories, and learn about the product along the way. At the end of the process the Customer gets what he wants because he has been fully engaged along the way. Too many teams try to guess at what the Customer wants. They end up with what they want, and the Customer ends up with something they have to tolerate.

Now that I have completed my first milestone, which was to put this into your hands, you get to be a Virtual Customer. Although XP would prefer that you actually come over to sit in my apartment while I do this, perhaps as a pair publisher, is just not going to happen. Later we might have some place which you can visit, and then you can be an Onsite Customer. Until then, its your job to help me modify and enhance the stories for the next milestone. What can we do to provide the most value in the next release cycle, which we measure in weeks? How has your idea about our end product, a magazine devoted to Perl, changed?

The stories in the pile for the next release cycle include some of the same stories from the first cycle, which we have to repeat, but now have some practice doing, as well as some new ones. We incrementally add new features so we do not be bogged down at the start of the project.

- Produce an A4 layout
- Sell some ads to cover production costs
- Collect subscribers for future issues
- Include a book review column
- Include an ongoing Perl Golf column

If you do your part, and we do our part, at the end of the process, when we have a completed product, you should have something close to your dream Perl magazine. One editor has already told me that if I do not include an article about beer in every issue, he will be disappointed. So, maybe most people will get their dream Perl magazine at least.

4 Future directions

I want to include at least something that mentions XP, or other things that help us work smarter rather than harder, in each issue. Some XP concepts I did not discuss include unit and acceptance tests (with PerlUnit perhaps), planning for change, the 40 hour work week, or pair programming. If you would like to write about these, and can relate them to Perl, let us know.

Remember to be our Virtual Onsite Customer, too. We do not plan too far into the future so that we can be more flexible. If you do not speak up, we can not give you what you want.

If you have not experienced XP before, check out some of the references. Get your managers to accept the XP methodology, then, once they have signed-off on XP, show them the chapter on the 40-hour work week.

5 References

You can read more about XP in these fine books:

Extreme Programming Explained - Kent Beck

Extreme Programming Installed - Ron Jeffries, *et al*

Extreme Programming Examined - Giancarlo Succi and Michele Marchesi

Structured Classes in Perl

Robby Walker *

Abstract

Perl's class system is like no other – its utter lack of structure makes it very simple and elegant. Unfortunately, this design also robs Perl of some power that more structured class systems can offer. While I do not suggest that Perl's class system should be changed, as its down and dirty approach can be really useful, sometimes a more orderly design is useful. For this purpose, I present the `Class::Structured` module, and discuss its implementation.

1 Conflicting class structures

Consider a super class `Text`, which stores some string variable `contents` containing the `Text` object's string data. `Book` inherits from `Text`, but `Book` also defines a variable `contents`, which in this case refers to the book's table of contents.

One of the nicest features of a Perl object is that its reference may point to anything, be it an array, hash, or simple scalar. This allows for a nice flexible set of options for a class developer. If both `Text` and `Book` use a hash for their internal storage, then both `Text` and `Book` write to the same location `$self->{'contents'}`, thus overwriting each others data, which neither module expects to happen. Even worse, if `Text` expects its subclasses to use a hash, but `Book` uses an array for internal storage, the program will complain when it tries to access an array like it was a hash, and vice versa.

Assuming this variable name clash can be avoided, `Book` still may need to call `Text`'s constructor. Perl does not call the constructor of a super class automatically and pushes this responsibility off to the subclass. Constructors look just like regular functions to the Perl.

Finally, what if I want to define an abstract function (a function whose implementation is deferred to subclasses, although it is declared in the base class) `search` in `Text`, and ensure that all subclasses of `Text` implement this function? Perl provides no way to enforce this check automatically.

To address these problems I wrote `Class::Structured`, which provides solutions to these problems by creating private variables, constructors, and abstract functions. It is still in the early phases of development, and does not implement inheritance types (like `private`, `protected`, and `public`), protected variables, or destructors yet. However, it provides a more structured approach to classes than standard Perl.

Before I get too wrapped up in the benefits of `Class::Structured`, I need to first consider its one severe drawback: speed. Using `Class::Structured` introduces some amount of overhead into your programs, and should not be used if speed is critical. However, where design is favored over speed, perhaps to aid in code maintenance, `Class::Structured` can be a very useful tool.

*webmaster@cd-lab.com

2 Using Class::Structured

Using Class::Structured is easy. Code listing 1 defines our Book class.

Code Listing 1

```

1 package Book;
2
3 # inherit from Text
4 use Text;
5 our @ISA = qw(Text);
6
7 # use the structured class module
8 use Class::Structured qw(:all);
9
10 # define the private type variable
11 define_variables 'contents' => 'private';
12
13 # define our constructor
14 constructor 'new',
15 { 'Text' => 'new' },
16 implementation {
17     my $self = shift;
18     $self->contents = shift;
19 };
20
21 # declare the abstract draw method
22 declare_abstract 'search';

```

I start by inheriting from class Text. Line 8 imports the Class::Structured module and all of its symbols.

At line 11 I define my class variables using the `define_variables` function. In general, the syntax for this command is a list of variable name and access type pairs.

```
define_variables 'varname' => 'accesstype', 'varname' => 'accesstype', ...;
```

where `varname` refers to the variable being defined and `accesstype` must be `private`. I include the `accesstype` field in order to allow for `protected` variables in the future.

Once I have called this function, I can access the private `contents` variable to retrieve its value or to set it by using it as an lvalue subroutine. Class::Structured makes use of a fairly recent Perl feature to define subroutines so that they can be assigned to. If you try this without Class::Structured, or something else that defines lvalue subroutines for you then Perl will rightly complain that it cannot compile the program.

```
my $x = $self->contents;
$self->contents = "Three blind mice...";
```

Class::Structured supports public variables through its `public` method. Public variables are available to all of the classes in the hierarchy.

```
my $x = $self->public("name");
$self->public("name") = 'A Shape';
```

The public variable space is shared by all classes in a hierarchy, and public and private variables of the same name can co-exist. A normal method call to `contents` accesses the private variable, while the `public` method can still access public variables, even if they are the same name.

```
$self->contents = "A long time ago...";
$self->public("contents") = "Chapter 1...";
```

Going back to code listing 1, I defined my constructor (lines 14 - 19) next. The syntax for constructor definition includes the constructor's name, a declaration of its super class, and its implementation.

Code Listing 2

```
1 constructor 'name',
2 { 'superclass' => 'constructor', ... },
3 implementation {
4     my $self = shift;
5     ... your code here ...
6 };
```

In code listing 2, the `name` string refers to the constructor's name, which in code listing 1 was the traditional `new`. The `superclass` part specifies which parent class's constructors to use, since Perl cannot recognize a constructor on its own and a class may have as many as it likes. The `superclass` declaration can be omitted, in which case the parent's default constructor will be used. A default constructor is either the first constructor to be defined by a class, or the constructor defined with the `default_constructor` keyword instead of the normal `constructor`.

To finish my constructor declaration, I use the keyword `implementation` followed by a code block and terminated with a semicolon. Within the `implementation` section, I do not bless my own class variable. The `Class::Structured` module handles this and passes the result as the first argument instead of the class type. If I do not use a semicolon my program will not compile because `constructor` is a regular function unlike Perl's keyword `sub`.

3 A Review of Symbol Tables

In order to explain how `Class::Structured` works, I first must recall some facts about Perl's symbol tables. The majority of `Class::Structured`'s implementation deals with adding functions to a package's symbol table.

Perl's symbol table works essentially just like a hash so it allows variables with names that are not valid Perl variable names. For instance, while `!structured!.abstracts` is not a legal variable name, I can use a symbolic reference to create it anyway.

```
${ '!structured!.abstracts' } = 'foo'; # aha!
```

In the implementation of `Class::Structured` I often use variables in this way so that I can create variables that `Class::Structured` can use while also minimizing the chance of collisions with actual variables used by the package whose symbol table I manipulate. The general format of a name shows its package and variable name.

```
$package . '::!structured!.' . $varname
```

So, in `Class::Structured`, if I am dealing with the `abstracts` variable for the `Foo::Bar` package, the variable name is

```
{ 'Foo::Bar::!structured!.abstracts' }
```

I can also add a function to the `$package` package. This is especially useful because it lets me use a closure as a package function. A Perl closure is simply a subroutine which refers to a lexical variable that has gone out of scope. It does not have a name and does not usually have a symbol table entry, but through `Class::Structured` it can.

```
*{ $package . '::' . $function_name } = sub { my $x = 'foo'; $x };
```

4 How Class::Structured Works

With the review of symbol tables out of the way, I can explain the guts and innards of `Class::Structured`.

Recall that I am adding three concepts to Perl's class system: constructors, private variables, and abstract functions. The easiest of these to implement, and the one I will discuss here, is the abstract function. Remember that an abstract function is a function whose implementation is deferred by a class to its subclasses. A class with abstract functions left to be defined should not be instantiable.

Code listing 3 shows the implementation of the function used to define a new abstract function.

Code Listing 3

```

1 use Carp;
2 use Set::Scalar;
3
4 sub declare_abstract {
5     my $function_name = pop; # get last param as function name
6     my $package = caller;
7
8     # update the abstract list
9     # (use a weird name so we don't collide with a real variable)
10    my $list_name = $package.'::'. '!structured!.abstracts';
11
12    ${ $list_name } = Set::Scalar->new() unless defined ${ $list_name };
13    ${ $list_name }->insert( $function_name );
14
15    # declare the function
16    *{ $package.'::'.$function_name } =
17        sub {
18            croak "$function_name in class $package is declared abstract, " .
19                " and cannot be called";
20        };
21 }
```

This code determines what package to modify by using the `caller` function, which, when called with no arguments, will return the name of the package that called the subroutine. If code in package `A` calls the subroutine, `caller` will return `A`.

Lines 12 and 13 update a `Set::Scalar` object that contains the list of defined abstract function names. This variable is used later to check that all abstracts have been implemented.

Finally, on lines 15 - 20 I define the abstract function with the same name that will cause the program to stop with an appropriate error message.

`Class::Structured` defines two functions, `list_abstracts` and `check_abstracts` that can be called with a package name to list all outstanding abstract functions and to check that all abstract functions have been implemented, respectively. The latter is used by constructor functions to make sure a class has defined all abstract functions before instantiating an object.

The implementation of such constructors as well as private variables can be found in `Class::Structured`'s source code, which is available on the Comprehensive Perl Archive Network. Please feel free to ask me any questions regarding how they work, or to offer suggestions on this module since it is still in the early phases of development.

5 References

Comprehensive Perl Archive Network – <http://www.cpan.org>

lvalue subroutines are discussed in the `perlsub` man page

`Set::Scalar`

perlhacktut – so you want to be a Perl porter?

Simon Cozens

Abstract

Getting involved in Perl development isn't as difficult as it might seem. Most of the barriers are actually psychological – an experienced Perl programmer with a smattering of C should have no problems at all picking up enough about the internals to get involved with development. Of course, there are some things you'll need to know. The purpose of this document is to help you find them.

[Editor's note: parts of this tutorial appear in various places in the Perl documentation, mostly in the perlhack manual page (<http://www.perldoc.com/perl5.6.1/pod/perlhack.html>) which first appeared in perl 5.6. Simon originally wrote it as one tutorial, and we publish it in its entirety. You can also read this on Simon's web site <http://simon-cozens.org/writings/perlhacktut.html>]

1 Getting started

Here are some things a Perl porter will need to get hold of:

1.1 The latest Perl source

It's a lot easier to hack on the Perl source if you've got a copy of it, right? So, download `src/perl-devel.tar.gz` from CPAN. Later on, you'll see that we sometimes refer to an even more bleeding-edge version of Perl than the development release called "perl-current". For the time being, though, `perl-devel` is fine.

1.2 A C compiler and debugger

On a Unix system, you'll probably have a C compiler installed; on Windows or similar, my personal recommendation is to install the Cygwin development environment, (<http://sourceware.cygwin.org/cygwin>) although Perl can be built with Microsoft Visual C or Borland C. Mac people will want to get MPW.

You'll want a source-level C debugger which allows you to step through the execution of a program and evaluate expressions much like the built-in Perl debugger does for Perl programs. `gdb` is the standard console debugger on non-commercial Unices and also comes with Cygwin. (<http://sourceware.cygwin.com/gdb/>)

1.3 A subscription to perl5-porters

perl5-porters is the central development list for Perl. If you're interested in how Perl works, where it's going, or how you can contribute to it, you will want to subscribe: send mail to perl5-porters-subscribe@perl.org

You'll also need to read the following things:

1.3.1 perlguts

This is of paramount importance, since it's the documentation of what goes where in the Perl source. Read it over a couple of times and it might start to make sense – don't worry if it doesn't yet, because the best way to study it is to read it in conjunction with poking at Perl source, and we'll do that later on.

You might also want to look at Gisle Aas's illustrated perlguts – there's no guarantee that this will be absolutely up-to-date with the latest documentation in the Perl core, but the fundamentals will be right. (<http://gisle.aas.no/perl/illguts/>)

1.3.2 perlxs and perlxs

A working knowledge of XSUB programming is incredibly useful for core hacking; XSUBs use techniques drawn from the PP code, the portion of the guts that actually executes a Perl program. It's a lot gentler to learn those techniques from simple examples and explanation than from the core itself.

1.3.3 perlapi

The documentation for the Perl API explains what some of the internal functions do, as well as the many macros used in the source.

1.3.4 Porting/pumpkin.pod

This is a collection of words of wisdom for a Perl porter; some of it is only useful to the pumpkin holder, but most of it applies to anyone wanting to go about Perl development.

1.3.5 The perl5-porters FAQ

This is posted to perl5-porters at the beginning of every month, and should be available from <http://perlhacker.org/p5p-faq>; alternatively, you can get the FAQ emailed to you by sending mail to perl5-porters-faq@perl.org. It contains hints on reading perl5-porters, information on how perl5-porters works and how Perl development in general works.

2 Finding Your Way Around

Perl maintenance can be split into a number of areas, and certain people (pumpkins) will have responsibility for each area. These areas sometimes correspond to files or directories in the source kit. Among the areas are:

2.1 Core modules

Modules shipped as part of the Perl core live in the `lib/` and `ext/` subdirectories: `lib/` is for the pure-Perl modules, and `ext/` contains the core XS modules.

2.2 Documentation

Documentation maintenance includes looking after everything in the `pod/` directory, (as well as contributing new documentation) and the documentation to the modules in core.

2.3 Configure

The configure process is the way we make Perl portable across the vast myriad of operating systems it supports. Responsibility for the configure, build and installation process, as well as the overall portability of the core code rests with the configure pumpkin – others help out with individual operating systems.

The files involved are the operating system directories, (`win32/`, `os2/`, `vms/` and so on) the shell scripts which generate `config.h` and `Makefile`, as well as the `metaconfig` files which generate `Configure`. (`metaconfig` isn't included in the core distribution.)

2.4 Interpreter

And of course, there's the core of the Perl interpreter itself. Let's have a look at that in a little more detail.

Before we leave looking at the layout, though, don't forget that `MANIFEST` contains not only the file names in the Perl distribution, but short descriptions of what's in them, too. For an overview of the important files, try this:

```
perl -lne 'print if /^[^\|]+\.[ch]\s+/' MANIFEST
```

3 Elements of the interpreter

The work of the interpreter has two main stages: compiling the code into the internal representation, or bytecode, and then executing it. `perlguts/Compiled code` explains exactly how the compilation stage happens.

Here is a short breakdown of perl's operation:

3.1 Startup

The action begins in `perlmain.c`. (or `miniperlmain.c` for `miniperl`) This is very high-level code, enough to fit on a single screen, and it resembles the code found in `perlembed`; most of the real action takes place in `perl.c`.

First, perlmain.c allocates some memory and constructs a Perl interpreter:

```

1  PERL_SYS_INIT3(&argc,&argv,&env);
2
3  if (!PL_do_undump) {
4      my_perl = perl_alloc();
5      if (!my_perl)
6          exit(1);
7      perl_construct(my_perl);
8      PL_perl_destruct_level = 0;
9  }
```

Line 1 is a macro, and its definition is dependent on your operating system. Line 3 references `PL_do_undump`, a global variable – all global variables in Perl start with `PL_`. This tells you whether the current running program was created with the `-u` flag to `perl` and then undump, which means it's going to be false in any sane context.

Line 4 calls a function in `perl.c` to allocate memory for a Perl interpreter. It's quite a simple function, and the guts of it looks like this:

```
my_perl = (PerlInterpreter*)PerlMem_malloc(sizeof(PerlInterpreter));
```

Here you see an example of Perl's system abstraction, which we'll see later: `PerlMem_malloc` is either your system's `malloc`, or Perl's own `malloc` as defined in `malloc.c` if you selected that option at configure time.

Next, in line 7, we construct the interpreter; this sets up all the special variables that Perl needs, the stacks, and so on.

Now we pass Perl the command line options, and tell it to go:

```

exitstatus = perl_parse(my_perl, xs_init, argc, argv, (char **)NULL);
if (!exitstatus) {
    exitstatus = perl_run(my_perl);
}
```

`perl_parse` is actually a wrapper around `S_parse_body`, as defined in `perl.c`, which processes the command line options, sets up any statically linked XS modules, opens the program and calls `yyparse` to parse it.

3.2 Parsing

The aim of this stage is to take the Perl source, and turn it into an op tree. We'll see what one of those looks like later. Strictly speaking, there's three things going on here.

`yyparse`, the parser, lives in `perly.c`, although you're better off reading the original YACC input in `perly.y`. (Yes, Virginia, there is a YACC grammar for Perl!) The job of the parser is to take your code and 'understand' it, splitting it into sentences, deciding which operands go with which operators and so on.

The parser is nobly assisted by the lexer, which chunks up your input into tokens, and decides what type of thing each token is: a variable name, an operator, a bareword, a subroutine, a core function, and so on.

The main point of entry to the lexer is `yylex`, and that and its associated routines can be found in `toke.c`. Perl isn't much like other computer languages; it's highly context sensitive at times, it can be tricky to work out what sort of token something is, or where a token ends. As such, there's a lot of interplay between the tokeniser and the parser, which can get pretty frightening if you're not used to it.

As the parser understands a Perl program, it builds up a tree of operations that it needs to perform. The routines which construct and link together the various operations are to be found in `op.c`, and will be examined later.

3.3 Optimization

Now the parsing stage is complete, and the finished tree represents the operations that the Perl interpreter needs to perform to execute our program. Next, Perl does a dry run over the tree looking for optimisations: constant expressions such as `3 + 4` will be computed now, and the optimizer will also see if any multiple operations can be replaced with a single one. For instance, to fetch the variable `$foo`, instead of grabbing the glob `*foo` and looking at the scalar component, the optimizer fiddles the op tree to use a function which directly looks up the scalar in question. The main optimizer is `peep` in `op.c`, and many ops have their own optimizing functions.

3.4 Running

Now we're finally ready to go: we have compiled Perl byte code, and all that's left to do is run it. The actual execution is done by the `runops_standard` function in `run.c`; more specifically, it's done by these three innocent looking lines:

```
while ((PL_op = CALL_FPTR(PL_op->op_ppaddr)(aTHX))) {
    PERL_ASYNC_CHECK();
}
```

You may be more comfortable with the Perl version of that:

```
PERL_ASYNC_CHECK() while $Perl::op = &{$Perl::op->{function}};
```

Well, maybe not. Anyway, each op contains a function pointer, which stipulates the function which will actually carry out the operation. This function will return the next op in the sequence – this allows for things like if which choose the next op dynamically at run time. The `PERL_ASYNC_CHECK` makes sure that things like signals interrupt execution if required.

The actual functions called are known as PP code, and they're spread between four files: `pp_hot.c` contains the 'hot' code, which is most often used and highly optimized, `pp_sys.c` contains all the system-specific functions, `pp_ctl.c` contains the functions which implement control structures (if, while and the like) and `pp.c` contains everything else. These are, if you like, the C code for Perl's built-in functions and operators.

4 Internal Variable Types

You should by now have had a look at `perlguts`, which tells you about Perl's internal variable types: SVs, HVs, AVs and the rest. If not, do that now.

These variables are used not only to represent Perl-space variables, but also any constants in the code, as well as some structures completely internal to Perl. The symbol table, for instance, is an ordinary Perl hash. Your code is represented by an SV as it's read into the parser; any program files you call are opened via ordinary Perl filehandles, and so on.

The core `Devel::Peek` module lets us examine SVs from a Perl program. Let's see, for instance, how Perl treats the constant "hello".

```

0  % perl -MDevel::Peek -e 'Dump("hello")'
1  SV = PV(0xa041450) at 0xa04ecbc
2  REFCNT = 1
3  FLAGS = (POK,READONLY,pPOK)
4  PV = 0xa0484e0 "hello"\0
5  CUR = 5
6  LEN = 6

```

Reading `Devel::Peek` output takes a bit of practise, so let's go through it line by line.

Line 1 tells us we're looking at an SV which lives at 0xa04ecbc in memory. SVs themselves are very simple structures, but they contain a pointer to a more complex structure. In this case, it's a PV, a structure which holds a string value, at location 0xa041450. Line 2 is the reference count; there are no other references to this data, so it's 1.

Line 3 are the flags for this SV – it's OK to use it as a PV, it's a read-only SV (because it's a constant) and the data is a PV internally. Next we've got the contents of the string, starting at location 0xa0484e0.

Line 5 gives us the current length of the string – note that this does not include the null terminator. Line 6 is not the length of the string, but the length of the currently allocated buffer; as the string grows, Perl automatically extends the available storage via a routine called `SvGROW`.

You can get at any of these quantities from C very easily; just add `Sv` to the name of the field shown in the snippet, and you've got a macro which will return the value: `SvCUR(sv)` returns the current length of the string, `SvREFCOUNT(sv)` returns the reference count, `SvPV(sv, len)` returns the string itself with its length, and so on. More macros to manipulate these properties can be found in `perlguits`. Let's take an example of manipulating a PV, from `sv_catpv`, in `sv.c`

```

1  void
2  Perl_sv_catpv(pTHX_ register SV *sv, register const char *ptr, register STRLEN len)
3  {
4      STRLEN tlen;
5      char *junk;
6      junk = SvPV_force(sv, tlen);
7      SvGROW(sv, tlen + len + 1);
8      if (ptr == junk)
9          ptr = SvPVX(sv);
10     Move(ptr, SvPVX(sv)+tlen, len, char);
11     SvCUR(sv) += len;
12     *SvEND(sv) = '\0';
13     (void)SvPOK_only_UTF8(sv);          /* validate pointer */
14     SvTAINT(sv);
15 }

```

This is a function which adds a string, `ptr`, of length `len` onto the end of the PV stored in `sv`. The first thing we do in line 6 is make sure that the SV has a valid PV, by calling the `SvPV_force` macro to force a PV. As a side effect, `tlen` gets set to the current value of the PV, and the PV itself is returned to junk.

In line 7, we make sure that the SV will have enough room to fit the old string, the new string and the null terminator. If `LEN` isn't big enough, `SvGROW` will reallocate space for us. If `junk` is the same as the string we're trying to add, we can grab the string directly from the SV; `SvPVX` is the address of the PV in the SV.

Line 10 does the actual catenation: the `Move` macro moves a chunk of memory around: we move the string `ptr` to the end of the PV – that's the start of the PV plus its current length. We're moving `len` bytes of type `char`. After doing so, we need to tell Perl we've extended the string, by altering `CUR` to reflect the new length. `SvEND` is a macro which gives us the end of the string, so that needs to be a `"\0"`.

Line 13 manipulates the flags; since we've changed the PV, any IV or NV values will no longer be valid: if we have `$a=10; $a.="6"`; we don't want to use the old IV of 10. `SvPOK_only_utf8` is a special UTF8-aware version of `SvPOK_only`, a macro which turns off the `IOK` and `NOK` flags and turns on `POK`. The final `SvTAINT` is a macro which launders tainted data if taint mode is turned on.

AVs and HVs are more complicated, but SVs are by far the most common variable type being thrown around. Having seen something of how we manipulate these, let's go on and look at how the op tree is constructed.

5 Op Trees

First, what is the op tree, anyway? The op tree is the parsed representation of your program, as we saw in our section on parsing, and it's the sequence of operations that Perl goes through to execute your program, as we saw in Running.

An op is a fundamental operation that Perl can perform: all the built-in functions and operators are ops, and there are a series of ops which deal with concepts the interpreter needs internally – entering and leaving a block, ending a statement, fetching a variable, and so on.

The op tree is connected in two ways: you can imagine that there are two “routes” through it, two orders in which you can traverse the tree. First, parse order reflects how the parser understood the code, and secondly, execution order tells perl what order to perform the operations in.

The easiest way to examine the op tree is to stop Perl after it's finished parsing, and get it to dump out the tree. This is exactly what the compiler backends `B::Terse` and `B::Debug` do. Let's have a look at how Perl sees `$a = $b + $c`:

```

0  % perl -MO=Terse -e '$a=$b+$c'
1  LISTOP (0x8179888) leave
2    OP (0x81798b0) enter
3    COP (0x8179850) nextstate
4    BINOP (0x8179828) sassign
5      BINOP (0x8179800) add [1]
6        UNOP (0x81796e0) null [15]
7          SVOP (0x80fafe0) gvsv GV (0x80fa4cc) *b
8            UNOP (0x81797e0) null [15]
9              SVOP (0x8179700) gvsv GV (0x80efeb0) *c
10        UNOP (0x816b4f0) null [15]
11      SVOP (0x816dcf0) gvsv GV (0x80fa460) *a

```

Let's start in the middle, at line 4. This is a BINOP, a binary operator, which is at location 0x8179828. The specific operator in question is `sassign` – scalar assignment – and you can find the code which implements in the function `pp_sassign`; in `pp_hot.c`. As a binary operator, it has two children: the add operator, providing the result of `$b+$c`, is uppermost on line 5, and the left hand side is on line 10.

Line 10 is the null op: this does exactly nothing. What is that doing there? If you see the null op, it's a sign that something has been optimized away after parsing. As we mentioned in Optimization above, the optimization stage sometimes converts two operations into one, for example when fetching a scalar variable. When this happens, instead of rewriting the op tree, it's faster just to replace the redundant operation with the null op. Originally, the tree would have looked like this:

```
SVOP (0x816b4f0) rv2sv [15]
      SVOP (0x816dcf0) gv GV (0x80fa460) *a
```

That is, fetch the `a` entry from the main symbol table, and then look at the scalar component of it: `gvsv` (`pp_gvsv` into `pp_hot.c`) happens to do both these things.

The right hand side, starting at line 5 is similar to what we've just seen: we have the add op (`pp_add` also in `pp_hot.c`) add together two `gvsv`s.

Now, what's this about?

```
LISTOP (0x8179888) leave
      OP (0x81798b0) enter
      COP (0x8179850) nextstate
```

`enter` and `leave` are scoping ops, and their job is to perform any housekeeping every time you enter and leave a block: lexical variables are tidied up, unreferenced variables are destroyed, and so on. Every program will have those first three lines: `leave` is a list, and its children are all the statements in the block. Statements are delimited by `nextstate`, so a block is a collection of `nextstate` ops, with the ops to be performed for each statement being the children of `nextstate`. `enter` is a single op which functions as a marker.

That's how Perl parsed the program, from top to bottom:

```
Program
 |
Statement
 |
 =
 / \
 /   \
$a   +
     / \
    $b $c
```

However, it's impossible to perform the operations in this order: you have to find the values of `$b` and

\$c before you add them together, for instance. So, the other thread that runs through the op tree is the execution order: each op has a field `op_next` which points to the next op to be run, so following these pointers tells us how perl executes the code. We can traverse the tree in this order using the `exec` option to `B::Terse`:

```
% perl -MO=Terse,exec -e '$a=$b+$c'
OP (0x8179928) enter
COP (0x81798c8) nextstate
SVOP (0x81796c8) gvsv GV (0x80fa4d4) *b
SVOP (0x8179798) gvsv GV (0x80efeb0) *c
BINOP (0x8179878) add [1]
SVOP (0x816dd38) gvsv GV (0x80fa468) *a
BINOP (0x81798a0) sassign
LISTOP (0x8179900) leave
```

This probably makes more sense for a human: enter a block, start a statement. Get the values of \$b and \$c, and add them together. Find \$a, and assign one to the other. Then leave.

The way Perl builds up these op trees in the parsing process can be unravelled by examining `perly.y`, the YACC grammar. Let's take the piece we need to construct the tree for `$a = $b + $c`.

```
1 term      :   term ASSIGNOP term
2             { $$ = newASSIGNOP(OPf_STACKED, $1, $2, $3); }
3           |   term ADDOP term
4             { $$ = newBINOP($2, 0, scalar($1), scalar($3)); }
```

If you're not used to reading BNF grammars, this is how it works: You're fed certain things by the tokeniser, which generally end up in upper case. Here, `ADDOP`, is provided when the tokeniser sees `+` in your code. `ASSIGNOP` is provided when `=` is used for assigning. These are 'terminal symbols', because you can't get any simpler than them.

The grammar, lines one and three of the snippet above, tells you how to build up more complex forms. These complex forms, 'non-terminal symbols' are generally placed in lower case. `term` here is a non-terminal symbol, representing a single expression.

The grammar gives you the following rule: you can make the thing on the left of the colon if you see all the things on the right in sequence. This is called a "reduction", and the aim of parsing is to completely reduce the input. There are several different ways you can perform a reduction, separated by vertical bars: so, `term` followed by `=` followed by `term` makes a `term`, and `term` followed by `+` followed by `term` can also make a `term`.

So, if you see two terms with an `=` or `+`, between them, you can turn them into a single expression. When you do this, you execute the code in the block on the next line: if you see `=`, you'll do the code in line 2. If you see `+`, you'll do the code in line 4. It's this code which contributes to the op tree.

```
|   term ADDOP term
{ $$ = newBINOP($2, 0, scalar($1), scalar($3)); }
```

What this does is creates a new binary op, and feeds it a number of variables. The variables refer to the tokens: \$1 is the first token in the input, \$2 the second, and so on – think regular expression backreferences.

\$\$ is the op returned from this reduction. So, we call newBINOP to create a new binary operator. The first parameter to newBINOP, a function in op.c, is the op type. It's an addition operator, so we want the type to be ADDOP. We could specify this directly, but it's right there as the second token in the input, so we use \$2. The second parameter is the op's flags: 0 means 'nothing special'. Then the things to add: the left and right hand side of our expression, in scalar context.

6 Stacks

When perl executes something like addop, how does it pass on its results to the next op? The answer is, through the use of stacks. Perl has a number of stacks to store things it's currently working on, and we'll look at the three most important ones here.

6.1 Argument stack

Arguments are passed to PP code and returned from PP code using the argument stack, ST. The typical way to handle arguments is to pop them off the pack, deal with them how you wish, and then push the result back onto the stack. This is how, for instance, the cosine operator works:

```
NV value;
value = POPn;
value = Perl_cos(value);
XPUSHn(value);
```

We'll see a more tricky example of this when we consider Perl's macros below. POPn gives you the NV (floating point value) of the top SV on the stack: the \$x in cos(\$x). Then we compute the cosine, and push back an NV. The X in XPUSHn means that the stack should be extended if necessary – it can't be necessary here, because we know there's room for one more item on the stack, since we've just removed one! The XPUSH* macros at least guarantee safety.

Alternatively, you can fiddle with the stack directly: SP gives you the first element in your portion of the stack, and TOP* gives you the top SV/IV/NV/etc. on the stack. So, for instance, to do unary negation of an integer:

```
SETi(-TOPi);
```

Just set the integer value of the top stack entry to its negation.

Argument stack manipulation in the core is exactly the same as it is in XSUBS – see perlxsut, perlxs and perlgluts for a longer description of the macros used in stack manipulation.

6.2 Mark stack

I say “your portion of the stack” above because PP code doesn't necessarily get the whole stack to itself: if your function calls another function, you'll only want to expose the arguments aimed for the called function, and not (necessarily) let it get at your own data. The way we do this is to have a “virtual” bottom-of-stack, exposed to each function. The mark stack keeps bookmarks to locations in the argument stack usable by each function. For instance, when dealing with a tied variable, (internally, something with “P” magic) Perl

has to call methods for accesses to the tied variables. However, we need to separate the arguments exposed to the method to the argument exposed to the original function – the store or fetch or whatever it may be.

Here's how the tied push is implemented; see `av_push` in `av.c`:

```
1  PUSHMARK(SP);
2  EXTEND(SP,2);
3  PUSHs(SvTIED_obj((SV*)av, mg));
4  PUSHs(val);
5  PUTBACK;
6  ENTER;
7  call_method("PUSH", G_SCALAR|G_DISCARD);
8  LEAVE;
9  POPSTACK;
```

The lines which concern the mark stack are the first, fifth and last lines: they save away, restore and remove the current position of the argument stack.

Let's examine the whole implementation, for practise:

```
PUSHMARK(SP);
```

Push the current state of the stack pointer onto the mark stack. This is so that when we've finished adding items to the argument stack, Perl knows how many things we've added recently.

```
EXTEND(SP,2);
PUSHs(SvTIED_obj((SV*)av, mg));
PUSHs(val);
```

We're going to add two more items onto the argument stack: when you have a tied array, the `PUSH` subroutine receives the object and the value to be pushed, and that's exactly what we have here – the tied object, retrieved with `SvTIED_obj`, and the value, the `SV val`.

```
PUTBACK;
```

Next we tell Perl to make the change to the global stack pointer: `dSP` only gave us a local copy, not a reference to the global.

```
ENTER;
call_method("PUSH", G_SCALAR|G_DISCARD);
LEAVE;
```

`ENTER` and `LEAVE` localise a block of code – they make sure that all variables are tidied up, everything that has been localised gets its previous value returned, and so on. Think of them as the `do` and `end` of a Perl block.

To actually do the magic method call, we have to call a subroutine in Perl space: `call_method` takes care of that, and it's described in `percall`. We call the `PUSH` method in scalar context, and we're going to discard its return value.

```
POPSTACK;
```

Finally, we remove the value we placed on the mark stack, since we don't need it any more.

6.3 Save stack

C doesn't have a concept of local scope, so perl provides one. We've seen that ENTER and LEAVE are used as scoping braces; the save stack implements the C equivalent of, for example:

```
{
    local $foo = 42;
    ...
}
```

See perlguits/Localising Changes for how to use the save stack.

7 Millions of Macros

One thing you'll notice about the Perl source is that it's full of macros. Some have called the pervasive use of macros the hardest thing to understand, others find it adds to clarity. Let's take an example, the code which implements the addition operator:

```
1  PP(pp_add)
2  {
3      djSP; dATARGET; tryAMAGICbin(add,opASSIGN);
4      {
5          dPOPTOPnnrl_ul;
6          SETn( left + right );
7          RETURN;
8      }
9  }
```

Every line here (apart from the braces, of course) contains a macro. The first line sets up the function declaration as Perl expects for PP code; line 3 sets up variable declarations for the argument stack and the target, the return value of the operation. Finally, it tries to see if the addition operation is overloaded; if so, the appropriate subroutine is called.

Line 5 is another variable declaration – all variable declarations start with d – which pops from the top of the argument stack two NVs (hence nn) and puts them into the variables right and left, hence the rl. These are the two operands to the addition operator. Next, we call SETn to set the NV of the return value to the result of adding the two values together. This done, we return – the RETURN macro makes sure that our return value is properly handled, and we pass the next operator to run back to the main run loop.

Most of these macros are explained in perlapi, and some of the more important ones are explained in perlxs as well. Pay special attention to perlguits/Background and PERL_IMPLICIT_CONTEXT for information on the [pad]THX_? macros.

8 Poking at Perl

To really poke around with Perl, you'll probably want to build Perl for debugging. Let's get hold of the latest source tree and build it.

We noted earlier that there was a version of Perl that's even more up-to-date than the latest development release. If you've got rsync installed, you can get it like this:

```
rsync -auvz rsync://ftp.linux.activestate.com/perl-current/ bleedperl/
```

Otherwise, you'll have to download the whole of it via FTP:

```
ftp://ftp.linux.activestate.com/pub/staff/gsar/APC/perl-current/
```

Alternatively, you can download it patch-by-patch:

```
ftp://ftp.linux.activestate.com/pub/staff/gsar/APC/diffs/
```

Now type

```
./Configure -d -D optimize=-g
make
```

-g is a flag to the C compiler to have it produce debugging information which will allow us to step through a running program. Configure will also turn on the DEBUGGING compilation symbol which enables all the internal debugging code in Perl. There are a whole bunch of things you can debug with this: `perlrun` lists them all, and the best way to find out about them is to play about with them. The most useful options are probably

```
l Context (loop) stack processing
t Trace execution
o Method and overloading resolution
c String/numeric conversions
```

Some of the functionality of the debugging code can be achieved using XS modules.

```
-Dr => use re 'debug'
-Dx => use O 'Debug'
```

9 Using a source-level debugger

If the debugging output of -D doesn't help you, it's time to step through perl's execution with a source-level debugger.¹

To fire up the debugger, type

```
gdb ./perl
```

¹We'll use gdb for our examples here; the principles will apply to any debugger, but check the manual of the one you're using.

You'll want to do that in your Perl source tree so the debugger can read the source code. You should see the copyright message, followed by the prompt.

```
(gdb)
```

help will get you into the documentation, but here are the most useful commands:

run [args]

Run the program with the given arguments.

break function_name

```
break source.c:xxx
```

Tells the debugger that we'll want to pause execution when we reach either the named function (but see Function names!) or the given line in the named source file.

step

Steps through the program a line at a time.

next

Steps through the program a line at a time, without descending into functions.

continue

Run until the next breakpoint.

finish

Run until the end of the current function, then stop again.

Just pressing Enter will do the most recent operation again – it's a blessing when stepping through miles of source code.

print

Execute the given C code and print its results. WARNING: Perl makes heavy use of macros, and gdb is not aware of macros. You'll have to substitute them yourself. So, for instance, you can't say

```
print SvPV_nolen(sv)
```

but you have to say

```
print Perl_sv_2pv_nolen(sv)
```

You may find it helpful to have a "macro dictionary", which you can produce by saying `cpp -dM perl.c` — sort. Even then, `cpp` won't recursively apply the macros for you.

10 Dumping Perl Data Structures

One way to get around this macro hell is to use the dumping functions in `dump.c`; these work a little like an internal `Devel::Peek`, but they also cover OPs and other structures that you can't get at from Perl. Let's take an example. We'll use the `$a = $b + $c` we used before, but give it a bit of context: `$b = "6XXXX"; $c = 2.3`. Where's a good place to stop and poke around?

What about `pp_add`, the function we examined earlier to implement the `+` operator:

```
(gdb) break Perl_pp_add
Breakpoint 1 at 0x46249f: file pp_hot.c, line 309.
```

Notice we use `Perl_pp_add` and not `pp_add` – see Function Names. With the breakpoint in place, we can run our program:

```
(gdb) run -e '$b = "6XXXX"; $c = 2.3; $a = $b + $c'
```

Lots of junk will go past as `gdb` reads in the relevant source files and libraries, and then:

```
Breakpoint 1, Perl_pp_add () at pp_hot.c:309
309      djSP; dATARGET; tryAMAGICbin(add,opASSIGN);
(gdb) step
311      dPOPTOPnnr1_ul;
(gdb)
```

We looked at this bit of code before, and we said that `dPOPTOPnnr1_ul` arranges for two NVs to be placed into left and right – let’s slightly expand it:

```
#define dPOPTOPnnr1_ul NV right = POPn; \
                      SV *leftsv = TOPs; \
                      NV left = USE_LEFT(leftsv) ? SvNV(leftsv) : 0.0
```

`POPn` takes the `SV` from the top of the stack and obtains its `NV` either directly (if `SvNOK` is set) or by calling the `sv_2nv` function. `TOPs` takes the next `SV` from the top of the stack – yes, `POPn` uses `TOPs` – but doesn’t remove it. We then use `SvNV` to get the `NV` from `leftsv` in the same way as before – yes, `POPn` uses `SvNV`.

Since we don’t have an `NV` for `$b`, we’ll have to use `sv_2nv` to convert it. If we step again, we’ll find ourselves there:

```
Perl_sv_2nv (sv=0xa0675d0) at sv.c:1669
1669      if (!sv)
(gdb)
```

We can now use `Perl_sv_dump` to investigate the `SV`:

```
SV = PV(0xa057cc0) at 0xa0675d0
REFCNT = 1
FLAGS = (POK,pPOK)
PV = 0xa06a510 "6XXXX"\0
CUR = 5
LEN = 6
$1 = void
```

We know we’re going to get 6 from this, so let’s finish the subroutine:

```
(gdb) finish
Run till exit from #0 Perl_sv_2nv (sv=0xa0675d0) at sv.c:1671
0x462669 in Perl_pp_add () at pp_hot.c:311
311      dPOPTOPnnr1_ul;
```

We can also dump out this op: the current op is always stored in `PL_op`, and we can dump it with `Perl_op_dump`. This'll give us similar output to `B::Debug`.

```

{
13  TYPE = add  ===> 14
      TARG = 1
      FLAGS = (SCALAR,KIDS)
      {
          TYPE = null  ===> (12)
          (was rv2sv)
          FLAGS = (SCALAR,KIDS)
          {
11      TYPE = gvsv  ===> 12
          FLAGS = (SCALAR)
          GV = main::b
          }
      }
}

```

11 Internal functions

Earlier we remarked that `pp_add` was actually known as `Perl_pp_add` to the debugger – all of Perl's internal functions which will be exposed to the outside world are prefixed by `Perl_` so that they will not conflict with XS functions or functions used in a program in which Perl is embedded. Similarly, all global variables begin with `PL_`. (By convention, static functions start with `S_`.)

Inside the Perl core, you can get at the functions either with or without the `Perl_` prefix, thanks to a bunch of defines that live in `embed.h`. This header file is generated automatically from `embed.pl`. `embed.pl` also creates the prototyping header files for the internal functions, generates the documentation and a lot of other bits and pieces. It's important that when you add a new function to the core or change an existing one, you change the data in the table at the end of `embed.pl` as well. Here's a sample entry from that table:

```

Apd |SV**  |av_fetch      |AV* ar|I32 key|I32 lval

```

The second column is the return type, the third column the name. Columns after that are the arguments. The first column is a set of flags:

A

This function is a part of the public API.

p

This function has a `Perl_` prefix; ie, it is defined as `Perl_av_fetch`.

d

This function has documentation using the `apidoc` feature which we'll look at in a second.

Other available flags are:

s

This is a static function and is defined as `S_whatever`.

n

This does not use `aTHX_` and `pTHX` to pass interpreter context. (See `perlguts/Background` and `PERL_IMPLICIT_CONTEXT`.)

r

This function never returns; croak, exit and friends.

f

This function takes a variable number of arguments, printf style. The argument list should end with ..., like this:

```
Afprd |void |croak |const char* pat|...
```

o

This function should not have a compatibility macro to define, say, Perl_parse to parse. It must be called as Perl_parse.

j

This function is not a member of CPerlObj. If you don't know what this means, don't use it.

x

This function isn't exported out of the Perl core.

If you edit embed.pl, you will need to run make regen_headers to force a rebuild of embed.h and other auto-generated files.

12 Source Documentation

There's an effort going on to document the internal functions and automatically produce reference manuals from them – perlapi is one such manual which details all the functions which are available to XS writers. perlintern is the autogenerated manual for the functions which are not part of the API and are supposedly for internal use only.

Source documentation is created by putting POD comments into the C source, like this:

```
/*
=for apidoc sv_setiv

Copies an integer into the given SV. Does not handle 'set' magic. See
C<sv_setiv_mg>.

=cut
*/
```

Please try and supply some documentation if you add functions to the Perl core.

13 Patching

All right, we've now had a look at how to navigate the Perl sources and some things you'll need to know when fiddling with them. Let's now get on and create a simple patch. Here's something Larry suggested: if a U is the first active format during a pack, (for example, pack "U3C8", @stuff) then the resulting string should be treated as UTF8 encoded.

How do we prepare to fix this up? First we locate the code in question – the pack happens at runtime, so it’s going to be in one of the pp files. Sure enough, pp_pack is in pp.c. Since we’re going to be altering this file, let’s copy it to pp.c .

Now let’s look over pp_pack: we take a pattern into pat, and then loop over the pattern, taking each format character in turn into datum_type. Then for each possible format character, we swallow up the other arguments in the pattern (a field width, an asterisk, and so on) and convert the next chunk input into the specified format, adding it onto the output SV cat.

How do we know if the U is the first format in the pat? Well, if we have a pointer to the start of pat then, if we see a U we can test whether we’re still at the start of the string. So, here’s where pat is set up:

```
STRLEN fromlen;
register char *pat = SvPVx(*++MARK, fromlen);
register char *patend = pat + fromlen;
register I32 len;
I32 datumtype;
SV *fromstr;
```

We’ll have another string pointer in there:

```
STRLEN fromlen;
register char *pat = SvPVx(*++MARK, fromlen);
register char *patend = pat + fromlen;
+ char *patcopy;
register I32 len;
I32 datumtype;
SV *fromstr;
```

And just before we start the loop, we’ll set patcopy to be the start of pat:

```
items = SP - MARK;
MARK++;
sv_setpvn(cat, "", 0);
+ patcopy = pat;
while (pat < patend) {
```

Now if we see a U which was at the start of the string, we turn on the UTF8 flag for the output SV, cat:

```
+ if (datumtype == 'U' && pat==patcopy+1)
+   SvUTF8_on(cat);
if (datumtype == '#') {
    while (pat < patend && *pat != '\n')
        pat++;
```

Remember that it has to be patcopy+1 because the first character of the string is the U which has been swallowed into datumtype!

Oops, we forgot one thing: what if there are spaces at the start of the pattern? `pack(" U*", @stuff)` will have U as the first active character, even though it's not the first thing in the pattern. In this case, we have to advance `patcopy` along with `pat` when we see spaces:

```
if (isSPACE(datumtype))
    continue;
```

needs to become

```
if (isSPACE(datumtype)) {
    patcopy++;
    continue;
}
```

OK. That's the C part done. Now we must do two additional things before this patch is ready to go: we've changed the behaviour of Perl, and so we must document that change. We must also provide some more regression tests to make sure our patch works and doesn't create a bug somewhere else along the line.

The regression tests for each operator live in `t/op/`, and so we make a copy of `t/op/pack.t` to `t/op/pack.t`. Now we can add our tests to the end. First, we'll test that the U does indeed create Unicode strings:

```
print 'not ' unless "1.20.300.4000" eq sprintf "%vd", pack("U*",1,20,300,4000);
print "ok $test\n"; $test++;
```

Now we'll test that we got that space-at-the-beginning business right:

```
print 'not ' unless "1.20.300.4000" eq
    sprintf "%vd", pack(" U*",1,20,300,4000);
print "ok $test\n"; $test++;
```

And finally we'll test that we don't make Unicode strings if U is not the first active format:

```
print 'not ' unless v1.20.300.4000 ne
    sprintf "%vd", pack("COU*",1,20,300,4000);
print "ok $test\n"; $test++;
```

Musn't forget to change the number of tests which appears at the top, or else the automated tester will get confused:

```
-print "1..156\n";
+print "1..159\n";
```

We now compile up Perl, and run it through the test suite. Our new tests pass, hooray!

Finally, the documentation. The job is never done until the paperwork is over, so let's describe the change we've just made. The relevant place is `pod/perlfunc.pod`; again, we make a copy, and then we'll insert this text in the description of `pack`:

```
=item *
```

```
If the pattern begins with a C<U>, the resulting string will be treated
as Unicode-encoded. You can force UTF8 encoding on in a string with an
initial C<U0>, and the bytes that follow will be interpreted as Unicode
characters. If you don't want this to happen, you can begin your pattern
with C<C0> (or anything else) to force Perl not to UTF8 encode your
string, and then follow this with a C<U*> somewhere in your pattern.
```

All done. Now let's create the patch. `Porting/patching.pod` tells us that if we're making major changes, we should copy the entire directory to somewhere safe before we begin fiddling, and then do

```
diff -ruN old new > patch
```

However, we know which files we've changed, and we can simply do this:

```
diff -u pp.c~          pp.c          > patch
diff -u t/op/pack.t~  t/op/pack.t  >> patch
diff -u pod/perlfunc.pod~ pod/perlfunc.pod >> patch
```

We end up with a patch looking a little like this:

```
--- pp.c~          Fri Jun 02 04:34:10 2000
+++ pp.c          Fri Jun 16 11:37:25 2000
@@ -4375,6 +4375,7 @@
     register I32 items;
     STRLEN fromlen;
     register char *pat = SvPVx(++MARK, fromlen);
+   char *patcopy;
     register char *patend = pat + fromlen;
     register I32 len;
     I32 datumtype;
@@ -4405,6 +4406,7 @@
...

```

And finally, we submit it, with our rationale, to `perl5-porters`. Job done!

14 Conclusion

We've had a brief look around the Perl source, an overview of the stages perl goes through when it's running your code, and how to use a debugger to poke at the Perl guts. Finally, we took a very simple problem and

demonstrated how to solve it fully – with documentation, regression tests, and finally a patch for submission to p5p.

I'd now suggest you read over those references again, and then, as soon as possible, get your hands dirty. The best way to learn is by doing, so:

- Subscribe to perl5-porters, follow the patches and try and understand them; don't be afraid to ask if there's a portion you're not clear on – who knows, you may unearth a bug in the patch...
- Keep up to date with the bleeding edge Perl distributions and get familiar with the changes. Try and get an idea of what areas people are working on and the changes they're making.
- Find an area of Perl that seems interesting to you, and see if you can work out how it works. Scan through the source, and step over it in the debugger. Play, poke, investigate, fiddle! You'll probably get to understand not just your chosen area but a much wider range of perl's activity as well, and probably sooner than you'd think.

The Road goes ever on and on, down from the door where it began.

If you can do these things, you've started on the long road to Perl porting. Thanks for wanting to help make Perl better – and happy hacking!

15 AUTHOR

Copyright © 2000 Simon Cozens; All rights reserved.

This document may be distributed under the same terms as Perl itself.

Benchmarking Perl

brian d foy

Abstract

Perl's motto is "There Is More Than One Way To Do It". Some ways are easier to read, some are faster, and some are just plain incomprehensible. Eventually, I want to know how long it takes for my Perl code to execute, and I can use the Benchmark module to find out. The Benchmark module comes with the standard Perl distribution and is written completely in Perl, so you should be able to use it right away.

1 Trouble with time()

[*Author's note: This article originally appeared in The Perl Journal 11. The builtin module is now List::Util, and the run times on the original test machine seem quite slow compared to my Powerbook. I updated the output and some of the run times for modern technology, except where noted.*]

Before I start with the Benchmark module, let's think about what I must do to time an event. I need to know when the event started and when the event ended. If I know those I can determine the difference between them which I call the duration of the event. There is actually quite a bit to consider.

If I wanted to time one of my scripts, I could look at my watch when the script started, and again when it finished. If my scripts takes more than several seconds I actually might be able to do that. I do not need to rely on my watch, though, since Perl already provides a way to do this with the built-in `time()` function which returns the system time. In code listing 1 I record the system time twice and take the difference of the two.

Code Listing 1

```
1 my $start_time = time;
2
3 # My code here
4
5 my $end_time   = time;
6
7 my $difference = $end_time - $start_time;
8
9 print "My code took ($difference) seconds\n";
```

Since `time()` returns seconds, this method can only record times and differences with a resolution of seconds, which might be too coarse a granularity for the really fast code that I have written. Also, remember that my script is not the only process running. The CPU does other things before it finishes my script, so the stopwatch approach does not tell me how long the CPU actually spent on *my* script. When the CPU is more loaded, the time to execute might even be longer, or shorter when less loaded.

2 Better resolution with `times()`

The built-in `times()` function returns a list of the user time, system time, children's user time, and children's system time with a finer granularity than `time()` (it relies on `time(2)` – see your system's documentation for details), and only records the time the spent on my process. In code listing 2 can use the same technique that I used in code listing 1:

Code Listing 2

```

1 my @start_time = times;
2
3 # My code here
4
5 my @end_time    = times;
6
7 my @differences = map { $end_time[$_] - $start_time[$_] }
8                     (0..$#start_time);
9
10 my $difference = join ', ', @differences;
11 print "My code took ($difference) seconds\n";

```

But computers are pretty fast and that code might run a lot faster than the smallest time that I can measure, even with `times()`. To get around this I can run my code several times and measure the time it takes to run it several times then take the average. This makes the situation much more complicated. Not only I would need to make a loop to run the code several times while timing it, but I would need to figure out how the addition of the loop affected the time. You won't have to worry about any of this if you use the Benchmark module.

3 The Benchmark module

Now I want to rewrite my previous examples using the Benchmark module. In code listing 3, I construct a Benchmark object to record the time. The constructor creates a list of the times returned by `time()` and `times()`, although I do not need to worry about that since I just want to use the abstract interface.

Code Listing 3

```

1 use Benchmark;
2
3 my $start_time = new Benchmark;
4
5 # my really slow code here
6 my @array = (1 .. 1_000_000);
7 foreach my $element ( @array ) { $_ += $element }
8
9 my $end_time    = new Benchmark;
10
11 my $difference = timediff($end_time, $start_time);
12
13 print "It took ", timestr($difference), "\n";

```

I also need a way to determine the time difference, which I can do with Benchmark's `timediff()` function which returns another Benchmark object. When I want to see the times that I have measured, I use Benchmark's `timestr()` method which turns the information into a human-readable string.

This function provides several ways to print the time by using additional parameters which the module's documentation explains. The code in code listing 3 produces output listing 3:

```
----- Output for Code Listing 3 -----  
It took 16 wallclock secs ( 3.83 usr + 0.00 sys = 3.83 CPU)  
-----
```

The first number, 16 secs, is the real time it took to execute, which should be the same as if I watched the clock on the wall. The module takes this directly from `time()`. The next numbers are the values from `times()` giving the user and system times, which, when summed, give the total CPU time.

I can also measure the time it takes to do several iterations of the code by using the `timeit()` method, which takes either a code reference or a string to `eval()`. The function returns a Benchmark object that I can examine as before. Code listing 4 shows an example with a string representation of the code.

```
----- Code Listing 4 -----  
1  #!/usr/bin/perl  
2  use Benchmark;  
3  
4  my $iterations = 1_000;  
5  
6  my $code = 'foreach my $element ( 1 .. 1_000 ) { $_ += $element }';  
7  
8  my $time = timeit($iterations, $code);  
9  
10 print "It took ", timestr($time), "\n";  
-----
```

```
----- Output for Code Listing 4 -----  
It took 2 wallclock secs ( 1.71 usr + 0.00 sys = 1.71 CPU) @ 584.80/s (n=1000)  
-----
```

I could do the same thing with a code reference as I show in code listing 5.

```
----- Code Listing 5 -----  
1  #!/usr/bin/perl  
2  use Benchmark;  
3  
4  my $iterations = 1_000;  
5  
6  my $code = sub { foreach my $element ( 1 .. 1_000 ) { $_ += $element } };  
7  
8  my $time = timeit($iterations, $code);  
9  
10 print "It took ", timestr($time), "\n";  
-----
```

Caution: Don't compare benchmarks of code references and strings! Use the same for each technique that you compare since there is extra overhead with the `eval()` needed to benchmark the code reference. This is true for `timeit()` and any of the Benchmark functions I demonstrate.

As I mentioned before, running a snippet of code several times might have additional overhead unassociated with the actual bits of code that I care about – the looping constructs of the benchmark, for instance. One of the advantages of the Benchmark module is that `timeit()` will run an empty string for the same number of iterations and subtract that time from the time to run your code. This should take care of any extra overhead introduced by the benchmarking code. The Benchmark module several functions that let you have a finer control over this feature and the module documents each one.

The function `timethis()` is similar to `timeit()`, but has optional parameters for `TITLE` and `STYLE`. The `TITLE` parameter allows you to give your snippet a name and `STYLE` affects the format of the output. The results are automatically printed to `STDOUT` although `timethis()` still returns a Benchmark object. Internally, `timethis()` uses `timeit()`. Code listing 6 does the same thing as code listing 5 but saves a couple of lines of code. It produces the same output, although the `TITLE` precedes it.

Code Listing 6

```

1  #!/usr/bin/perl
2  use Benchmark;
3
4  my $iterations = 1_000;
5
6  my $code = sub { foreach my $element ( 1 .. 1_000 ) { $_ += $element } };
7
8  timethis($iterations, $code, 'Foreach');
```

Output for Code Listing 6

```
Foreach:  2 wallclock secs ( 1.65 usr +  0.00 sys =  1.65 CPU) @ 606.06/s (n=1000)
```

4 Example: some sums

Now that I know how long it took to run my bit of code, I am curious if I can make the time shorter. Can I come up with another way to do the same task, and, if I can, how does its time compare to other ways? Using the Benchmark module, I can use `timeit()` for each bit of code, but Benchmark anticipates this curiosity and provides me a function to compare several snippets of code.

The `timethese()` function is a wrapper around `timethis()`. The hash `%Snippets` contains snippet names as keys and either code references or strings as values. The function returns a list of Benchmark objects for each snippet.

```
my @benchmarks = timethese($iterations, \%Snippets);
```

As with anything else that deals with hashes, Benchmark uses an apparently random order `timethese()` to go through the snippets, so I have to keep track of which order `timethis()` reports the results. If I wanted to do further programmatic calculations with the times, I could store the list returned by `timethese()`, but for now I will simply rely on the information printed from `timethis()`. But first, I need something to compare.

To demonstrate `timethese()`, I want to compare five methods of summing an array of numbers. I give each snippet a name based on my impression of it. The **Idiomatic** method is the standard use of `foreach()`. The **Evil** use of `map` in a void context seems like it might be clever, but how fast is it? The **Iterator** technique uses the `sum()` function from `List::Util`, which uses XS to connect the code written in C to my Perl program. I think **Iterator** to be competitive, so I came up with two more techniques, **Curious** and **Silly**, to spread out the field. Code listing 7 shows these techniques as the values in `%Snippets`.

Code Listing 7

```

1  #!/usr/bin/perl
2
3  use Benchmark qw(timethese);
4  use List::Util qw(sum);
5
6  my $Iterations = 100_000;
7
8  @array = ( 1 .. 10 );
9
10 my \%Snippets = (
11   Idiomatic => 'foreach ( @array ) { $sum += $_ }',
12   Evil      => 'map { $sum += $_ } @array;',
13   Iterator  => '$sum = sum @array;',
14   Curious   => '$sum=0; grep { /\^(d+)\$/ and $sum += $1 } @array',
15   Silly     => q|$sum=0; $_ = join 'just another perl journal', @array;
16               while( m/(\d+)/g ) { $sum += $1 }|
17               );
18
19 timethese($Iterations, \%Snippets);

```

On my Powerbook G3 running Mac OS 10.1.2 with perl5.6.1, I get the following output.

Output for Code Listing 7

```

Benchmark: timing 100_000 iterations of Curious, Evil, Idiomatic, Iterator, Silly...
  Curious: 20 wallclock secs (12.84 usr + 0.00 sys = 12.84 CPU) @ 7788.16/s (n=100000)
    Evil:  6 wallclock secs ( 3.87 usr + 0.00 sys =  3.87 CPU) @ 25839.79/s (n=100000)
  Idiomatic: 1 wallclock secs ( 1.01 usr + 0.00 sys =  1.01 CPU) @ 99009.90/s (n=100000)
  Iterator: 1 wallclock secs ( 0.25 usr + 0.00 sys =  0.25 CPU) @ 400000.00/s (n=100000)
            (warning: too few iterations for a reliable count)
    Silly: 15 wallclock secs ( 9.52 usr + 0.00 sys =  9.52 CPU) @ 10504.20/s (n=100000)

```

As I see, the `sum()` function from `List::Util` is *very* fast. In fact, it was so fast that for 10,000 iterations the Benchmark module could not measure a reliable time. The **Idiomatic** method is slightly faster than the clever use of `map`, but both are significantly slower than `sum()`, and the other methods, which I never expected to be fast, are indeed quite slow.

This comparison does not satisfy me though. What happens as the size of the array and the number of iterations changes? I ran several combinations of array size and iterations for each of the methods using a Sparc20 running Solaris 2.6 with perl5.004. Since I don't really care about the **Curious** and **Silly** methods, I only report the results for the **Idiomatic**, **Evil**, and **Iterator** methods if. I ran each with arrays of sizes from 10 to 10,000 elements and iterations from 1,000 to 1,000,000 times. The longest time took about 86,000 CPU seconds. Do not try this without telling the system administrator what you are doing, especially if you

named your script `test` – it is not nice to get email nastygrams from an administrator who thinks you have a script running amok when it is really doing exactly what you want it to do. Not that this happened to me and you cannot prove it anyway.

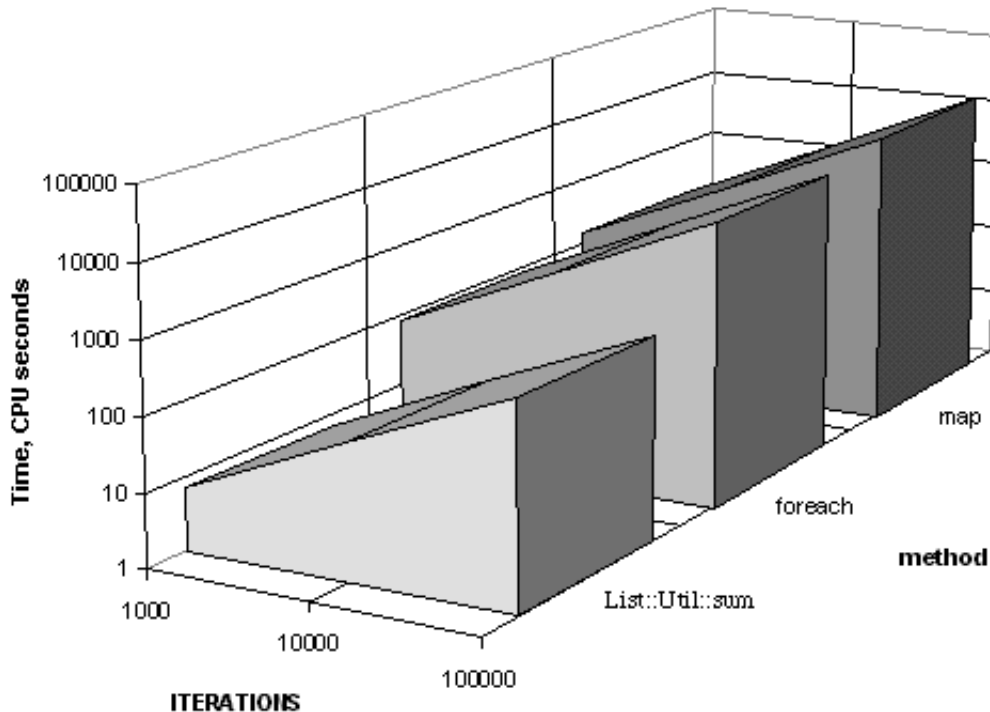


Figure 1: Run times versus iterations for three functions

5 So what have I learned?

The “stopwatch” approach is not very effective for timing a piece of code because the CPU can work on other things before it finishes the one I want to time. The Perl builtin `times()` is a little better, but my code might run faster than `times()` can measure. I do not have to worry about these issues when I use the Benchmark module, which can average the time to run bits of code over many iterations and even compare different bits of code.

In my summing example, I discovered that the clever use of `map()` was consistently slower than the idiomatic `foreach()`, which was much slower than `sum()`. Although `foreach()` would be the idiomatic way to sum an array, I am hard-pressed to explain why absorbing an order of magnitude speed penalty is good – I will be using `sum()`. If you think that you have a faster method, you now have the tools to test it. If your method beats `sum()`, send me a note!

I did not show all of the things that you can do with the Benchmark module. Since I first published this article, another set of functions appeared. You can turn some of these benchmarks inside out by measuring the number of iterations that the computer completes in a set time which makes life much more satisfying when Benchmarks seemingly do not quit.

6 References

- *The Perl Journal* 11

Find the following modules on CPAN – the Comprehensive Perl Archive Network ([URL:http://search.cpan.org](http://search.cpan.org))

- List::Util module
- Benchmark module

Book Review: *Learning XML*



Eric T. Ray; O'Reilly & Associates; 0-596-00046-4; 368 pages; January 2001; reviewed by brian d foy

Finally O'Reilly & Associates published a beginner's XML book, and one thin enough (368 pages) not to cause major shoulder pain if you carry it home from the bookshop. Most XML books I have had the displeasure to review try to disguise their author's inexperience as "definitive" and "authoritative" with excess verbiage, figures, large fonts, and reprinted W3C documents that result in 800 page bricks. Eric T. Ray actually uses XML (as O'Reilly "XML guru") and wrote *Learning XML* in DocBook, which has its own XML DTD.

This book delivers what it advertises, learning XML, and does not try to do more. The author says as much as he needs to say and leaves the reader plenty of references to follow for more information, rather than trying to explain everything. The author's depth of understanding of XML – that is, the philosophy and concept rather than just the section numbers in the standards – is obvious.

Eric explains XML rather than the particular, often single, tool which represent the range of most other XML author's experience. Indeed, he devotes few words to any tools which gives the text remarkable clarity and focus. Chapter 8 shows simple and digestible Perl examples and surveys the current state of Perl XML modules well enough to get the reader on his way, but the rest of the book is strictly XML. Readers who want to learn about various tools or pro-

gramming techniques for XML can graduate to other O'Reilly & Associates books.

Appendix B not only lists all of the relevant resources for further study of XML, but also gives a paragraph description of each reference and, since XML evolves, the status of the document at the time of publication. The glossary, absent in most other XML books I have reviewed, is well laid out and clearly defines the technical language of XML without sounding like XML itself.

Robert Romano did a superb job with the illustrations and thier connection with the example XML text. Generous line leading around figures and example text, along with simple illustrations rather than pixelated Windows screen shots, make the text easy to read and effortlessly connect the concepts in the prose with the examples by way of numbered bullets. O'Reilly's usual high standard of typography which uses three fonts and clean layout contrasts sharply with *XML Bible* in which I found six different fonts on one page in a ragged right monstrosity, or *The XML Handbook* which lets its heavy, word-processed typeface text fall into the gutter or superimposes it on dark gray gradients. O'Reilly & Associates books are, no matter the subject matter, a pleasure to view, and with their superb editing and remarkable restraint the publisher of first choice on almost any technical subject. •

Perl Golf

We have not figured out the rules, chosen the judges, or calculated what your chances of winning really are, but we do have the prizes – Perl Mongers hats or t-shirts along with a chance for fame and glory in the next issue of *The Perl Review*.

Solve the following problem with a ridiculously low number of keystrokes, uses Perl in some clever or devious way, or is otherwise interesting and send it to comdog@panix.com.

Convert a base 36 number, with the digits [0-9A-Z], to its base 10 representation

If you would like to be a judge, or the maintainer of this column, or have an interesting golf problem, let us know. We can send you a hat or a t-shirt too. •