

The Perl Journal

Zoidberg: A Shell that Speaks Perl

Jaap Karssenberg • 3

Getting Started with Perl and MySQL

Thomas Valentine • 7

Creating Self-Contained Perl Executables, Part II

Julius C. Duque • 9

Grey Hats and Network Engineers

Simon Cozens • 12

A Wireless Popularity Contest

brian d foy • 16

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Jack J. Woehr:

***Perl 6 Now* • 18**

Source Code Appendix • 20

LETTER FROM THE EDITOR

Change is Good

You talk. We listen. Over the past few months, a great number of you have requested easier access to, and more frequent updates of, *The Perl Journal*. Consequently, effective with the next issue, *The Perl Journal* is moving from a PDF e-zine presentational format to a familiar HTML-based web site presentation. In doing so, *TPJ*'s content will be updated on a weekly, rather than monthly, basis.

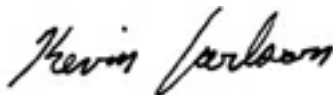
As you might guess, we're also redesigning the TPJ.com site to be more accessible and reflect the new format. For those of you with RSS newsreaders, an RSS feed will deliver up-to-date notification of new articles as they're posted.

In addition, we'll be bundling into PDF format the cream of the *TPJ* crop, packaging articles in a "Best of *TPJ*" compilation. We think this new incarnation of *TPJ* will give you the best possible experience as a *TPJ* reader.

The one thing that we aren't changing is the high-quality editorial content you've come to expect from *TPJ*. All the same great articles, book reviews, and columns that you've enjoyed and benefited from will continue to be available—but in a newer, easier-to-access package.

Your current subscription username and password will not change; you'll simply use those to access the new site. Look for the new *TPJ* in mid-May at <http://www.TPJ.com/>.

Okay, you know my e-mail address. Drop me a note and let us know what you think of the new approach. And if you have any other ideas of how we can improve *TPJ*, I look forward to hearing from you.



Kevin Carlson
Executive Editor
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to 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) 2005 by CMP Media LLC, unless otherwise noted. All rights reserved.



The Perl Journal

EXECUTIVE EDITOR

Kevin Carlson

MANAGING EDITOR

Della Wyser

ART DIRECTOR

Margaret A. Anderson

NEWS EDITOR

Shannon Cochran

EDITORIAL DIRECTOR

Jonathan Erickson

COLUMNISTS

Simon Cozens, brian d foy, Moshe Bar, 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

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT AND GENERAL COUNSEL

Sandra Grayson

SENIOR VICE PRESIDENT, COMMUNICATIONS Alexandra Raine

SENIOR VICE PRESIDENT, CORPORATE MARKETING

Kate Spellman

VICE PRESIDENT, GROUP DIRECTOR INTERNET BUSINESS

Mike Azzara

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

Perl 5.9.2 Released

Perl 5.9.2 was announced earlier this month and is available from CPAN. It's a development version, not intended for production environments; instead, it's designed to allow users to test out new features slated to appear in Perl 5.10. Core enhancements as described in the *perldelta* focus on Unicode, *suidperl* insecurities, and Malloc wrapping ("detecting attempts to assign pathologically large chunks of memory"). New modules include Autrijus Tang's *encoding::warnings*, "a module to emit warnings whenever an ASCII character string containing high-bit bytes is implicitly converted into UTF-8," and Richard Clamp's *Module::CoreList*, "a small, handy module that tells you what versions of core modules ship with any versions of Perl 5."

Under the hood, "the semantics of *pack()* and *unpack()* regarding UTF-8-encoded data has been changed. Processing is now by default character-per-character instead of byte-per-byte on the underlying encoding. Notably, code that used things like *pack("a*", \$string)* to see through the encoding of string will now simply get back the original *\$string*. Packed strings can also get upgraded during processing when you store upgraded characters. You can get the old behavior by using *use bytes*."

Pugs Hits Milestone

News on the Perl 6 front continues to be headlined by Pugs, Autrijus Tang's implementation of Perl 6 in Haskell, which has recently reached Version 6.2.1. That's not quite as advanced as it sounds: Under Pugs' naming scheme, which is based on 2π , the initial release was Version 6.0. But the 6.2 release, announced earlier this month, represented the project's first major milestone: "We are now reasonably confident that the basics of Perl 6 syntax and data structures are in place," Tang wrote in his online journal (<http://use.perl.org/~autrijus/journal/>). "We already have an object/type system now, and the 6.2.x series will make them available on the language level, together with a full-fledged class system." The 6.2.1 release of Pugs concentrates on stability and performance, with rewritten Context and Type code, a new OO core, and call-by-values bindings.

Tang has also been working with Leopold Totsch to bring Pugs into concert with Parrot. The plan, as reported in Tang's journal: "The evaluator and three independent compiler backends will now work in concert. Perl 6 source code will still be parsed to Pugs AST (Abstract Syntax Tree). The current evaluator in *Eval.hs* becomes a compiler, generating a Parrot AST. Parrot AST will then be compiled into Haskell, or pretty-printed into IMC (Intermediate Code)... That means Pugs can still run Perl 6 by itself, but enhancement on it can then be shared with Parrot and other languages targeting IMC."

NLNet Sponsors Parrot

More good news for Parrot: The Perl Foundation has announced that NLnet (<http://www.nl.net.nl/>), a Netherlands-based nonprofit organization dedicated to advancing open-source networking technology, has pledged \$70,000 to support Leopold Totsch and Chip Salzenberg in completing the first two Parrot development milestones. Leo has been pumping of Parrot since 2003; Chip re-

cently took over the duties of chief architect. Their work will complete the core of Parrot's grammar engine—which requires a Perl 6 grammar to be developed and tested against sample Perl 6 code—and extend that engine to product Parrot AST. Details of the Parrot milestones are listed at <http://www.perlfoundation.org/grants/2005-p6-proposal.html>.

Continuing the momentum, Chip has announced his intention to keep Parrot on a monthly release schedule. He suggested that code freezes should take place on the first of the month, and test builds made on Darwin, x86 Linux, and Win32 platforms. 64-bit Sparc may also be included in the mix. At the time of this writing, however, no preparations were underway for a May release.

On a more nuts-and-bolts level, the Parrot code was migrated this month from CVS to Subversion: "It was mostly painless, thanks to *cvs2svn*, and having tested out all the infrastructure and tools with other projects," Robert Spier reported on the new perl.org infrastructure weblog, log.perl.org. The Parrot source code now lives at <http://www.parrotcode.org/source.html>.

Ponie Plan Published

While Pugs and Parrot prance ahead, Ponie has been plodding behind in the Perl parade. As Nicholas Clark wrote on the *perl6-internals* list: "For various internal and external reasons work has been pretty much stalled for quite a while. I'm pleased to announce that it's now able to restart, and that I'm going to be able to allocate about 1 to 1 1/2 days per week to it on average. There is now a detailed roadmap breaking down the tasks needed between here and a first release, with estimates of times. It's checked into CVS at the top level, and available online as <http://opensource.fotango.com/software/ponie/plan>."

As the Ponie Plan explains, "The purpose of Ponie is to provide source compatibility between Perl extensions written in C and XS with Parrot, and thereby allow many important existing CPAN modules to be used unchanged with Perl 6." The four major steps broadly outlined in the plan are to progressively refactor all of the Perl data structures into PMCs (Parrot Magic Cookies); to move Perl 5's op code for type polymorphism into PMC methods; to replace Perl 5's "magic" features and behavior with Parrot implementations; and finally to migrate the Perl interpreter onto the Parrot runloop.

Upcoming Events

The Marseilles Perl Mongers will be hosting the second French Perl Workshop in Marseilles on June 9th and 10th. It's "the occasion for people to meet and talk about Perl—in French." See <http://conferences.mongueurs.net/fpw2005/> for details. The Second Austrian Perl Workshop (<http://conferences.yapceurope.org/apw2005/>) has been scheduled for the same days; it will take place in Vienna, and is organized around the theme of "using Perl." Autrijus Tang, Chip Salzenberg, and Leo Totsch will speak. Later in the month, the Pisa Perl Mongers will host the Second Italian Perl Workshop. It will be held at the Polo Fibonacci in the University of Pisa, on June 23rd and 24th; see <http://www.perl.it/workshop/> for more. Lastly, YAPC::EU::2005 (to be held in Braga, Portugal, August 31–September 2) has issued a second call for papers. Details are at <http://braga.yapceurope.org/index.cgi?CallForPapers>.

Zoidberg: A Shell That Speaks Perl

Allegedly, a certain Mr. Wall has claimed that “It is easier to port a shell than a shell script.” This, of course, is not true; shells are, although conceptually simple programs, in fact quite complex. Still, there are a few projects that try to improve the shell environment by adding Perl to the mix, most notably Psh and Zoidberg. In this article, I will show you some features of the Zoidberg Perl shell (see Figure 1).

There are two main ways in which a Perl shell can make your life easier: In the first place, you can use Perl syntax at the command prompt; and secondly, you can modify or extend your shell environment using Perl. UNIX users, in general, spend a lot of time working in a shell environment, so being able to tune this environment to your every wish can save you a lot of irritation. And of course, if you type Perl faster than shell script, it is nice to have a shell that understands you. Now, of course, the older shells allow for extensions: While bash has a notoriously obscure code base that is not easy to hack, zsh on the other hand even allows you some module interfaces. But these programs don’t have Perl bindings and adding them could well be more work than starting from scratch.

Zoidberg (or zoid, for friends) was written from scratch and is a pure Perl implementation. Due to an object-oriented and modular design, Zoidberg can be regarded as a framework as well as an application. So far, it has not yet seen a stable release, so some parts of the code are still a bit rough, but we work very hard to fix that. You should also keep in mind that once you start hacking your shell, it is in general a good idea to have another shell available just in case; after all, you don’t want to be locked out of your account.

To install zoid, try something like:

```
perl -MCPAN -e 'install q/Bundle::Zoidberg/'
```

Now run “zoid” to start the Zoidberg Perl shell.

Perl at the Command Prompt

Traditionally, shell languages have been ugly things. Their syntaxes are hacked together, resulting in an arcane language with lots of obscure exceptions (“for historic reasons” is an oft-used

phrase in the manuals). The sole objective of these languages is to be as expressive as possible with the minimum number of characters. In some respects, shell languages are not unlike Perl.

So why use Perl instead of shell script? An important reason is that Perl is a language that has a much broader scope than shell script; it is more powerful. The UNIX philosophy dictates that everything is a file, and I think no language is so tuned for text manipulation as Perl; so a lot of your daily administration tasks can be done with simple Perl statements that are the equivalent of many lines of shell script. In a Perl shell you can, for example, use regular expressions to modify the filenames in a certain directory.

The downside of using Perl syntax is that it was intended for scripts and isn’t really optimized for shell usage. Some people claim that the ultimate Perl shell is something like:

```
perl -e 'while (<STDIN>) { eval $_; warn if $? }'
```

But after a while, it gets very tiresome to keep typing *system "ls"* instead of the plain *ls* that most shells will understand. Of course, you can define a subroutine for this or even use Shell.pm’s AUTOLOAD function, but it won’t be as intelligent as you want

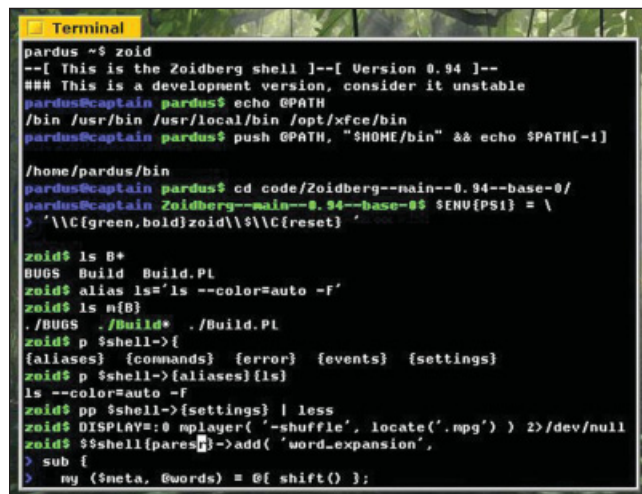


Figure 1: The Zoidberg Shell.

Jaap is currently a student in applied physics and philosophy. He can be contacted at pardus@cpan.org.

it to be. The next logical step is then to add a routine that tries to recognize which commands should be *system*'ed and which commands should be *eval*'ed; if you follow this way, you eventually end up with a program like *zoid*.

In *Zoidberg* you can use two different syntaxes. The first one is a partial implementation of the shell command language as specified by POSIX (full implementations include *bash*, *zsh*, etc.). The

*A lot of your daily
administration tasks can be done
with simple Perl statements
that are the equivalent of many
lines of shell script*

second syntax of course is Perl. This is just your normal Perl interpreter, but Perl code used at the command prompt undergoes a bit of source filtering to make it play better with the shell environment.

For example if you type:

```
zoid$ print map "\n" @PATH
```

then *@PATH* is imported from *Env.pm* before the evaluation of the command line. In this example, *Zoidberg* understands this is a Perl command because the word “print” is a reserved word.

Let's do another example: Say I want to change all filenames in a directory to their lowercase equivalent; this typically happens to me after using a brain-dead ftp client. Using Perl, this is a simple one-liner, while the shell equivalent would need some program, such as *sed*, *awk*, or *perl* to do the string manipulation.

```
zoid$ mv($_ => lc($_)) for grep /[A-Z]/, <*>
```

If you use *zoid* as your login shell, it will also will be used by other programs to execute commands. For example, from *vim* you can now use the following command to get a word count for the file you are editing.

```
:!cat % | {/^\\S/}g | wc -
```

All lines starting with a whitespace are supposed to contain example code so we *grep* them out. The second part of this pipeline shows *Zoidberg*'s notation to have a Perl *grep* in a pipeline. The curly braces force this command to be evaluated as Perl code instead of a filename and the *g* modifier at the end wraps the code in a loop.

Zoidberg also has support for multiline editing, which becomes very handy when you want, for example, to define a new subroutine on the command line. This is where *Term::ReadLine::Zoidberg* distinguishes itself. Other readline libraries allow you to edit only one line at a time—once you press return, you can't go back to the previous line. Not so in *zoid*, and what's more, you can even go to a “multiline mode” by pressing <ESC>-m.

Scripts

All scripts used by *zoid* are normal Perl scripts. This means that, for example, the rc files (“*/etc/zoidrc*” and “*\$HOME/.zoidrc*”) are just Perl scripts. To interface with the shell, these scripts use the *Zoidberg::Shell* module. This module exports an *AUTOLOAD* function that allows for shell commands to be called as Perl subroutines and a method called *shell()* that is the *zoid* equivalent to the Perl *eval* function. Below is an example *zoidrc* file that can be saved as “*~/.zoidrc*”:

```
use Zoidberg::Shell;
$shell = Zoidberg::Shell->current();

cat($ENV{HOME}.'/TODO')
    if $shell->{settings}{interactive};

unshift @INC, "$ENV{HOME}/lib/perl5/";
```

A Debugging Session

One of the things *zoid* can be used for is to test new modules interactively. It can be very useful to quickly test the output of some methods while you are coding a module. Take, for example, the following code and save it as “*~/lib/perl5/My/Module.pm*”:

```
package My::Module;

sub new { bless {} } # simple constructor

sub test { return 'foobar' }

1; # keep require happy
```

Now to load this module, we simply *use* it on the command prompt:

```
zoid$ use My::Module
zoid$ $mm = My::Module->new()
zoid$ p $mm->test()
```

Now you realize that there is a typo in the word “foobar,” so you go back to your editor, open the module, and correct it.

Next, we need a method to load the changed version of the module. At the moment, *zoid* has an undocumented built-in called “reload,” but it is broken, so let's define our own. The following code could be entered directly at the command prompt, but let's put it in your *zoidrc* file:

```
use Zoidberg::Shell;
my $shell = Zoidberg::Shell->current();

$shell->{commands}{reload} = sub {

    # transform module name to file name
    my $file = shift;
    $file .= '.pm' unless $file =~ /\.\w+$/;
    $file =~ s{::}{/}g;

    # look up the filename in %INC
    $file = $INC{$file} || $file;

    # load the file
    eval "do '$file'";

    # forward errors
    die if $@;
};
```

The hash `$$shell->{commands}` is tie'd to an object that helps to keep track of plug-ins, but for the moment, we can pretend that it is just a hash with anonymous subroutines. Now we can use this command to reload the module:

```
zoid$ reload My::Module
zoid$ p $mm->test()
```

This works fine for object-oriented modules, but how about libraries? If your module exports some functions, you can *use* it from the command prompt and call the exported routines. Another

Zoidberg has a built-in called mode, which is used to switch your default syntax

option is to change the namespace in which Perl commands are evaluated—this is done simply by using the “package” keyword:

```
zoid$ package My::Module
zoid$ p test()
```

To return to the default namespace, type *package Zoidberg::Eval*.

Extending Zoidberg

The parser already recognizes two different syntaxes, but how about adding another one? All you need for this is a plug-in that defines some rule to recognize the syntax, and a handler routine to execute commands given in this syntax. Let's teach Zoidberg SQL to see how this works.

To start with, we need to write a plug-in module that manages a DBI object. Save this module as “~/lib/perl5/SQLFish.pm” (remember that we added “~/lib/perl5” to `@INC` in the previously mentioned `zoidrc` file).

```
package SQLFish;

use strict;
use Zoidberg::Fish;
use Zoidberg::Utils qw/error output/;
use DBI;

our @ISA = qw/Zoidberg::Fish/;
```

Zoidberg::Fish is the base class for Zoidberg's plug-ins. It gives you a constructor and routines needed for the plug-in framework. If you use *Zoidberg::Utils*, your plug-in will blend in nicely with the rest of *zoid*. The plug-in is an object that lives below the main Zoidberg object. Use `$$plugin{shell}` to access the parent object.

```
sub connect_db { # create DBI object
    my $plugin = shift;
```

```
    $$plugin{db}->disconnect if ref $$plugin{db};
    $$plugin{db} = DBI->connect(@_);
```

```
    # make DBI survive forks
    $$plugin{db}{InactiveDestroy} = 1;
}
```

This routine will later be exported as a built-in command that is used to setup the database connection. Of course, you can call it from the `zoidrc` file if you want to be connected at all times.

Next, three methods follow that plug-in in several stages of the parser:

```
sub word_list { # claim commands
    my $plugin = shift;
    my ($meta, @words) = @{ shift() };
    return grep /^$words[0]/,
        qw/SELECT INSERT UPDATE DELETE/ if wantarray;
    return 'SQL' if $words[0] =~ /^[A-Z]+$/;
    return undef;
}
```

This method tells the parser that all commands written in all caps are SQL. The list context is used for tab completion of SQL commands.

```
sub parser { # perform selected expansions
    my ($plugin, $block) = @_;
    @$block =
        $$plugin{shell}->$expand_param(@$block);
    return $block;
}
```

This method overloads the default word expansions for the SQL syntax. We only expand for variables here, so we can use environment variables in our SQL statements. Path expansion, for example, is omitted here because we want to be able to use a “*” in the SQL command without escaping or quoting it.

And the last method:

```
sub handler { # execute SQL statements
    my $plugin = shift;
    my ($meta, @words) = @{ shift() };

    # catch connect command
    if ($words[0] =~ /^connect/i) {
        shift @words;
        return $plugin->connect_db(@words);
    }
    error "no db connection" unless ref $$plugin{db};
    my $sth =
        $$plugin{db}->prepare(join ' ', @words);
    $sth->execute;
    output [ map {join ' ', $_, @$}
        @{$sth->fetchall_arrayref} ];
}

1;
__END__
```

This method actually gets to execute the SQL commands. We use the utility functions *error* and *output* here. *error* is like *die* but has another output format and helps us with stack traces. *output* is like *print* but uses *Data::Dumper* when passed references; the way it is used here, it outputs data in multiple columns when possible. And we make an exception in the evaluation for statements starting with *connect*—this will prove useful later on.

Next, we define a plug-in configuration script to make Zoidberg use this module. Save this script as “~/zoid/plugins/SQL.pl”.

```
{
  module => 'SQLFish',
  export => ['connect_db'],
  parser => {
    word_list => 'word_list',
    parser    => 'parser',
    handler   => 'handler',
  }
}
```

This is just a Perl script containing only a hash defining the hooks that are in the module so that zoid can use them. We *export* the *connect_db* method as a built-in command and we list the parser hooks we defined. For more advanced plug-ins, one can also *import* events.

Now one might wonder what the advantage is of having SQL in your command shell instead of using a dedicated application for that. For one thing, you can combine SQL statements with general shell commands, so things like this will work:

```
zoid$ SELECT firstname, lastname FROM users
      | {s/\b(\w)/uc($1)/eg}p
zoid$ SELECT * FROM users | wc -l
zoid$ sql{ select from users where clue > 0 }
      > users.txt
```

In the last example we needed the parenthesis to protect the first “>”; this is a notation that works for all syntaxes in Zoidberg. Of course, once we tag the command as “sql” there is no need for caps to have it recognized as SQL.

You can also use the Zoidberg framework very easily to build a dedicated application, or at least an application that looks dedicated. This is demonstrated later.

Modes

Zoidberg has a built-in called *mode*, which is used to switch your default syntax. Say you want to type a series of SQL commands and you think typing commands in all caps is tiresome. Try this:

```
zoid$ mode SQL
zoid$ connect DBI:mysql:test
zoid$ select * from users
```

Here you see why we put that exception for *connect* in the handler routine of the plug-in. Once you are in a mode, you need the bang (“!”) to “shell out.” So in the SQL mode, you can still execute shell commands like this:

```
zoid$ !ls
zoid$ select * from users | !grep root
```

To get back to Zoidberg’s default mode, type:

```
zoid$ mode -
```

But wait, there is more! A mode can also be the name of a Perl module; in that case, you get to keep shell command syntax, but all commands are considered subroutines in the specified module.

```
zoid$ use CPAN
zoid$ mode CPAN::Shell->
zoid$ i /MimeInfo/
...
zoid$ install File::MimeInfo
```

```
...
zoid$ mode -
```

The “->” is there to tell zoid that *CPAN::Shell* always expects a class name as the first argument for a subroutine. Replace it with “::” for modules that don’t need this argument. Note that this is purely an example—there is already a CPAN plug-in for Zoidberg that can be used after typing *mode CPAN*; this plug-in also supports things such as tab expansion.

Write Your Own Shell Application

Say you want to write a program with a shell-like interface quickly; using Zoidberg can spare you the hassle of writing a command parser and all that. We again use the SQL plug-in of the previous section and write a simple script to start our SQL shell:

```
#!/usr/bin/perl

use Zoidberg::Shell;
use lib $ENV{HOME}.'./lib/perl5/';

# You have Env::PS1 ?
$ENV{PS1} = '\C{blue,bold}DBI\C{reset}: ';
$ENV{PS2} = '    : ';

my $shell = Zoidberg::Shell->new(
  settings => {

    # non-default rcfile
    rcfiles => [$ENV{HOME}.'./sqlshell'],
    mode    => 'SQL',
  }
);

$shell->main_loop; # run the shell
$shell->round_up;  # clean up pending objects
```

The trick here is the “mode” setting; actually you just run Zoidberg in a predefined mode. Of course, you still are allowed to use Perl, so you now have your own SQL Perl shell (but remember that default aliases like ‘p’ and ‘pp’ are not defined here). Also, you can still access system commands using the bang (“!”). Now try this script with a module name as mode string.

Ongoing Development

As I stated in the introduction, Zoidberg has not yet seen a stable release. This means it is neither feature complete nor bug free. But I think that the development has reached a point where it has become clear how the stable release might look, and I hope you have caught a glimpse of that in this article.

One of the things I’m currently working on is forking more functionality from Zoidberg into separate CPAN packages. At the moment, *Bundle::Zoidberg* consists of the Zoidberg package itself and two packages that have already forked from Zoidberg’s code base: *Term::ReadLine::Zoid* and *Env::PS1*. Especially forking *Term::ReadLine::Zoid* has been very good for the project. One of the subsystems I hope to release in a separate package is the parser and job-control code. Forking functionality makes the code more accessible for other applications and also forces us to have very clean interfaces between subsystems of the program and will thus allow for better test suites.

If you encounter any bugs while playing around with zoid, please report them through <http://rt.cpan.org/> and feel free to ask questions on the mailing list.

Getting Started with Perl and MySQL

One day at work, I was shocked to find that my junior co-workers didn't understand the ease and flexibility of Perl with regard to database machinations. They all had rudimentary Perl skills, so I decided to explain everything in one simple, easy to understand lesson. I'm sharing this basic tutorial here in hopes that more novice Perl programmers will make use of Perl's tight database integration, in this case with the MySQL database.

Background

Each example assumes you're already connected to a database and are using the *DBI* module to do so. *DBI* is a database-independent interface for Perl, which means that it provides a common interface to all kinds of databases. It also requires the *DBD* (or *database-dependent*) driver module that corresponds to the database you're using. In the case of MySQL, that would be the *DBD::mysql* module. This architecture allows you to change the underlying database that a Perl application uses, without having to change much of the application's code. *DBI* and all the *DBD::** modules are available on CPAN.

In all of the following statements, there appear the following three lines of code. Get to know them well, as you'll be writing them often.

```
$query = "SELECT col1, col2, col3, col4 FROM $thistable";
$stmt = $dbh->prepare($query);
$stmt->execute();
```

As with anything Perl, there are many ways to accomplish the basic logic of the database call. I've chosen this one because it shows simply and clearly the stages of the program execution. The MySQL query is contained in *\$query* and will fluctuate with every example. The statement is then prepared and executed. If there are problems in your SQL syntax, this is the moment in the program execution that an error will be generated. Check your logs or employ *CGI::Carp* to report errors to the browser (assuming your database query is part of a web-based application) if you run into this.

While these three lines are the starting point, what I'll focus on in this article are the ways to select the values required and get them into the sort of data structure that is most useful for a given

situation. Note also that the *\$thistable* variable can be anything you require. This is also true for every bit of information contained in the MySQL query (*\$query*).

This document presents the following constructs:

- Selecting one column into one variable using one variable.
- Selecting one column from many rows into one array using one variable.
- Selecting many columns from one row into one array using one array.
- Selecting many columns from many rows into many arrays using one array.

Each construct should fit with any number of applications. What we'll be exploring is the final form the values take.

Selecting one column into one variable using one variable. The starting point in any understanding of the easy integration of Perl and MySQL must begin with the simplest of all database calls—calling one known column and placing it into one known variable, like this:

```
$query = "SELECT thiscolumn FROM thistable
        WHERE id = 6";

$stmt = $dbh->prepare($query);
$stmt->execute();
$thisvariable = $stmt->fetchrow();
```

The query is loaded into *\$query* and the database selection is prepared and executed. The previously declared variable *\$thisvariable* is then loaded with the result of the database call via the *fetchrow()* function. Simple. Note that the MySQL statement in all examples reflects the need of the example.

Selecting one column from many rows into one array using one variable. When you need one array of values as the end product, this construct will suffice. There are several other ways an array (or even many arrays) can result from this processing block as well. More on that later.

```
$query = "SELECT col1 FROM thistable
        WHERE id >= 0";

$stmt = $dbh->prepare($query);
```

Thomas is a writer residing in Selkirk, Manitoba, Canada. He can be contacted at publications@forkedweb.com.


```
$sth->execute();
while ($thisvalue = $sth->fetchrow()) {
    push @thisarray, $thisvalue;
};
```

This example shows the normal preparation and execution procedures, as well as a good use of the *fetchrow()* function. The *while* loop stuffs values that are generated by the database call into *@thisarray*. The length and complexity of *@thisarray* can be whatever you require.

Selecting many columns from one row into one array using one array. This is the first time we've used the *fetchrow_array()* function. This function fetches entire rows and is one of the so-called "higher functions" that have been given priority of execution. This makes for a speedier database call for higher volumes of data.

```
$query = "SELECT col1, col2, col3, col4 FROM
          $thistable WHERE id = 6";
$sth = $dbh->prepare($query);
$sth->execute();
@thesevalues = $sth->fetchrow_array();
```

The normal preparation and execution takes place, followed by a *fetchrow_array()* call that is simply placed into the one array, *@thesevalues*. Each column is one index item.

Selecting many columns from many rows into many arrays using one array. This is perhaps the best implementation of the powerful combination that is the meshing of Perl and MySQL. The amount of data awaiting processing can be enormous. Your only limit is memory resources.

The columns called will be one array index item for each column fetched from the database. The complexity of the database call narrows the number of index items and thus the volume of data. Use a MySQL statement to narrow or order the search, as in the following example:

```
$query = "SELECT col1, col2, col3, col4 FROM
          $thistable WHERE id >= 0 ORDER BY id DESC";
$sth = $dbh->prepare($query);
$sth->execute();
while (@thesevalues = $sth->fetchrow_array()) {
    push @array1, $thesevalues[0];
    push @array2, $thesevalues[1];
    push @array3, $thesevalues[2];
    push @array4, $thesevalues[3];
};
```

The construct performs the database preparation and execution normally. What changes in this machination is that the array that is filled with *fetchrow_array()* is then pushed onto more arrays waiting within the *while* loop. For every successive result the *fetchrow_array()* function is able to return, the numbered arrays (*@array1*, *@array2*, etc.) are filled. You may then act upon all of the arrays in any way you see fit. Each array will reflect one column within the database for many rows, as befitting your MySQL database call. Using an index column such as the AUTO_INCREMENT ID column given in the example is handy to keep your database model simple and easy to retrieve.

This set of constructs is a great starting point in examining database calls. Each is simple and easy to use and conveys the mindset a database programmer employs. I hope these will get you started having fun with database calls using Perl.

TPJ

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!

Creating Self-Contained Perl Executables, Part II

You can convert a Perl script into a self-contained executable in more than one way. One way is to use *LibZip*, which I've already discussed in Part I of this article last month. *LibZip* builds an external self-contained library that you distribute along with your compiled Perl script. Another approach is to use *PAR*—the Perl Archive Toolkit.

As in *LibZip*, you need a working C compiler to compile Perl scripts. A sample Perl script, `sample1.pl` (shown in Listing 1), is provided with this article. This script will be compiled to showcase the capabilities of *PAR*. You may also need to install *Win32::Autoglob* before you can use `sample1.pl`.

Also, although the entire discussion here focuses on building executables for the Windows system, the steps presented here apply to other nonWindows platforms, too.

The Perl Archive Toolkit

PAR is the tool to use if you dislike having to bundle an executable with a separate library and prefer, instead, to have just a single executable. *PAR*, which is always available from <http://par.perl.org/index.cgi>, requires the following modules: *File::Temp*, *Compress::Zlib*, *Archive::Zip*, *Module::ScanDeps*, and *PAR::Dist*. Apart from these prerequisite modules, the *PAR* maintainers also recommend the following modules: *Parse::Binary* and *Win32::Exe*, if you're on a Windows system.

Assuming you have installed all the prerequisite modules, using *PAR* is easy.

Compiling Scripts Using the Command Line

If you prefer using the command line, the following command will compile the sample Perl script, `sample1.pl`, into a Windows executable:

```
pp -o sample1.exe sample1.pl
```

The resulting executable will be named `sample1.exe`, as specified by the `-o` option to the Perl Packer, *pp*. *pp* converts the source script, `sample1.pl`, into Windows native code.

If you want to compress the executable, you can invoke the `-z` option with a mandatory integer argument, from 0 to 9. 9 will use the maximum possible compression. If `-z` is not specified, the compression level defaults to 6.

```
pp -o sample1.exe -z 9 sample1.pl
```

You can also eliminate more excess baggage by using the filter option (`-f`), which requires an argument:

```
pp -o sample1.exe -z 9 -f PodStrip sample1.pl
```

This filter strips away all POD sections from the executable.

GUI-based Script Compilation

If you hate command lines, there is *tkpp*, a GUI-based version of *pp*. When you use *tkpp*, you must first set up some important paths: the locations of your Perl interpreter and *pp*. To set these, click on "File" and then choose "Preferences."

On the "Source file" text box, type the complete path to your script, or click on the icon beside the text box. In doing so, a file browser will pop up, where you can choose your script. On the "Output file" text box, type the name of the executable you want to create. Leave all other buttons in their default states. At the bottom center of the GUI, there is a "Build" button. Click this button to proceed with the compilation of your script. You should see the message "Building..." while your script is being processed. When compilation is finally completed, you should see the message "Ready."

You can also build an executable without a console window by clicking on the "GUI" checkbox. This is ignored, however, on nonWindows systems. The command-line version of this is the `-gui` option:

```
pp -o sample1.exe -z 9 -f PodStrip -gui sample1.pl
```

That's it!

Obfuscation

Like *LibZip*, *PAR* can also hide your source code from casual snooping by turning your script into comment-free Perl code with mangled variable names. Of course, this is not the same as encryption and surely will not discourage a determined cracker. Obfuscation cannot offer 100-percent protection of your Perl code.

To obfuscate your code, use the `-f` filter with *Obfuscate* as argument:

```
pp -o sample1.exe -z 9 -f PodStrip \  
-f Obfuscate sample1.pl
```

Julius is a systems administrator for Ayala Systems Technology. He can be contacted at jcdunque@lycos.com.

A harmless message will be displayed during compilation, which looks something like this:

```
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\6mBQIYqCex
syntax OK
```

Just ignore it.

But before you can use obfuscation, you must first install *B::Deobfuscate*, and before you can do that, you need to install *B::Keywords* and *YAML*, as well. Be sure to use Version 0.35 of *YAML*, and not the latest development release, or else you won't be able to install *B::Deobfuscate*.

Caveats

Using the diagnostics Module. Just like my experience with *LibZip*, I found out that having *use diagnostics* in my scripts will compile fine, but the resulting executables will not run properly. I often get error messages similar to this:

```
No diagnostics? at diagnostics.pm line 408.
Compilation failed in require at
xxxx/sample1.pl line 8.
BEGIN failed--compilation aborted at
xxxx/sample1.pl line 8.
```

I recommend that you remove or uncomment out *diagnostics* from your script prior to building its native code equivalent.

Using Obfuscation. One ugly problem with using obfuscation is that the generated executable is not always guaranteed to run. If I compile my script like this:

```
pp -o sample1.exe -f Obfuscate -z 9 \
-f PodStrip sample1.pl
```

and run the resulting executable on the current directory,

```
sample1.exe . -recursive -all
```

I sometimes get this error:

```
Undefined subroutine &main::performance called
at sample1.pl line 41.
```

Even weirder is when I rearrange the order of arguments I feed to *pp*:

```
pp -o sample1.exe -f PodStrip -z 9 \
-f Obfuscate sample1.pl
```

In which case I get a very different error:

```
String found where operator expected at
script/hash.pl line 10, near
```

```
"GetOptions 'digest=s'"
      (Do you need to predeclare GetOptions?)
syntax error at script/hash.pl line 10, near
"GetOptions 'digest=s'"
```

But I don't get these errors if I run the original Perl script!

It looks like *pp* itself gets confused when obfuscation is performed. I have informed the *PAR* maintainers about this. Let's hope this problem goes away in the next release of *PAR*.

*PAR works equally well on Linux.
As a matter of fact, PAR also runs
on FreeBSD, AIX, Solaris, HP-UX,
NetBSD, and Mac OS*

What can we learn from this? Aside from offering minimal source-code protection, use of obfuscation is also problematic. So, I discourage you from employing it, at least in its present state.

Using UPX. In the previous article where I discussed *LibZip*, I used UPX to compress the library and executable even more. This time, I attempted to use UPX on the executable generated by *PAR*. It compressed the executable all right, but when I ran the executable, I got a partial error:

```
I/O error: reading header signature :
at -e line 830
I/O error: reading header signature :
at -e line 830
```

But despite this imperfection, the executable still managed to proceed as usual.

Compiling Scripts For Linux. *PAR* works equally well on Linux. As a matter of fact, *PAR* also runs on FreeBSD, AIX, Solaris, HP-UX, NetBSD, and Mac OS. From the *PAR* FAQ: "The resulting executable will run on any platform that supports the binary format of the generating platform."

The steps enumerated here also apply to nonWindows systems without the slightest modification.

TPJ

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

Listing 1

```
#!/usr/local/bin/perl
```

```
# sample1.pl
# Julius C. Duque
```

```
#use diagnostics;
#use strict;
#use warnings;
use Cwd;
use Getopt::Long;
use File::Find;
```

```
use Win32::Autoglob;
use Digest::MD5;

my $VERSION = "1.0.0 (for TPJ)";

my ($showfiles, $showdigests, $recursive, $all, $quiet, $help) = ();

GetOptions(
    "showfiles" => \$showfiles,
    "showdigests" => \$showdigests,
    "recursive" => \$recursive,
    "all" => \$all,
    "quiet" => \$quiet,
    "help" => \$help
);
```

```

$showfiles = $showdigests = 1 if ($all);

syntax() if ($help or !$ARGV);

foreach my $infile (@ARGV) {
    chomp $infile;
    if (!-e $infile) {
        print "**** ERROR: $infile does not exist, skipping it...\n" if
(!$quiet);
        next;
    } elsif (-d $infile) {
        if ($recursive) {
            find({wanted => sub {
                if (-f) {
                    print make_digest($_, 'MD5');
                    print " $_" if ($showfiles);
                    print " [MD5]" if ($showdigests);
                    print "\n";
                }
            }, no_chdir => 1}, $infile);
        } else {
            if (!$quiet) {
                print "**** ERROR: $infile is actually a directory, skipping
it.\n";
                print "**** ERROR: Use --recursive if you want to process
$infile recursively\n";
            }
            next;
        }
    } else {
        print make_digest($infile, 'MD5');
        print " $infile" if ($showfiles);
        print " [MD5]" if ($showdigests);
        print "\n";
    }
}

sub make_digest
{
    my ($file, $tmd) = @_;

```

```

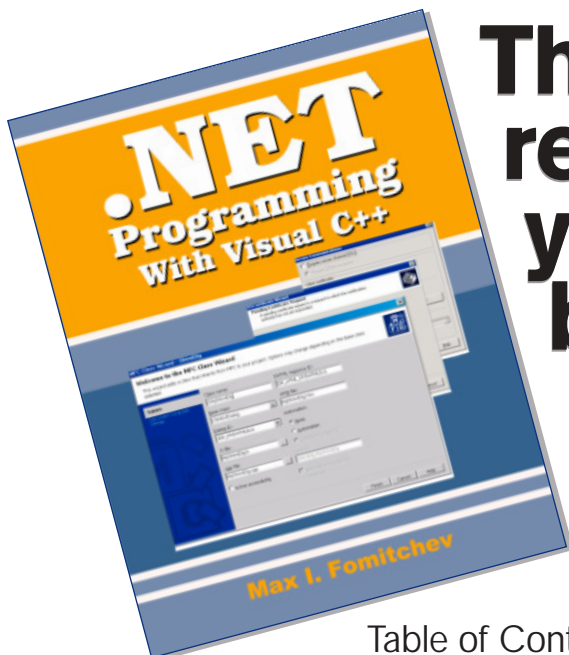
    my $digest_obj;
    open INFILE, $file or die "Cannot open $file: $!";
    binmode INFILE;
    $tmd = $tmd =~ /^Digest::/? $tmd : "Digest::$tmd";
    eval "require $tmd";
    $digest_obj = new $tmd;
    $digest_obj->addfile(*INFILE);
    close INFILE;
    return $digest_obj->hexdigest;
}

sub syntax
{
    if ($^O eq "MSWin32") {
        $0 =~ s/.*\\//g; # Windows
    } else {
        $0 =~ s/.*///g;
    }

    print "$0 $VERSION\n\n";
    print "Usage: $0 file1 [file2 ...]\n";
    print "\n";
    print "Other options:\n";
    print " --showfiles      print filenames\n";
    print " --showdigests      print digests used\n";
    print " --recursive         recursively descend into directory\n";
    print " --all               implies --showfiles and --showdigests output\n";
    print " --quiet             suppress error messages\n";
    print " --help              print this help message\n";
    exit 1;
}

```

TPJ



The .NET resource you've been waiting for!



- Delivered in PDF format.
- Packed with C++ code examples.
- Thousands of lines of source code.
- A complete reference to the .NET Framework

Table of Contents and sample chapter available at:
<http://www.ddj.com/dotnetbook/>

Get your copy now!

Available via **download** for just **\$19.95**
 or
 on **CD-ROM** for only **\$24.95** (plus s/h).



Grey Hats and Network Janitors

Simon Cozens

For a while, at college, I was unable to hide my computing abilities and got roped into the position of student computer representative. This was no more than a glorified helpdesk role, until one day it all went horribly wrong.

For a few days, students had been complaining that they couldn't send e-mail. It was strange, but investigation revealed it was a problem with the upstream ISP's SMTP servers. Nothing I could do about it. Then, at about 10 PM, the entire network went down. Nothing worked. I got hold of the poor IT officer, who cleared out the 4G firewall log, and restarted the firewall. We promised to look over the log at a more sensible time.

At this point, I had already guessed what had happened—some virulent machine had been mass-mailing, filling the firewall log with connections, filling up the upstream mail servers, and getting us on some DNS blacklists. This needed to stop; not only did we need to detect which machines were being unpleasant but, if possible, stop them in advance of anything nasty happening.

Of course, I was only the student computer representative, and didn't have access to the firewall, the network infrastructure, or any of the servers. So this is where things got decidedly "grey-hat."

I was already playing with ethereal (<http://www.ethereal.com/>) to look for strange network traffic—excessive SMTP activity, unusual DNS queries, or connections to IRC servers in Eastern Europe, for instance—but someone on #perl mentioned ArachNIDS, a set of rules for the snort Intrusion Detection Software, which picked up virus activity. Perfect.

Of course, finding the IP address of the culprit is only half the problem—from there, we have to physically locate the machine and its owner, and, uh, reeducate them. That's where Perl comes in.

Enter Blart

The college network is entirely wireless—which is good, because I can see all the traffic without requiring particular privileges—

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.

and entirely DHCP—which is bad, because it's a little harder to track which IP addresses belong to which computers. Even when you've gone from IP address to MAC, you then need to keep track of who that computer belongs to, and where they can be found.

I set up a SQLite database with three tables. The owner table deals with human beings, and where they live:

```
CREATE TABLE owner (  
    name varchar(255) primary key,  
    room varchar(255)  
);
```

The machine table keeps track of MAC addresses and who owns them. Later it could be expanded to keep track of when I last visited the machine, what antivirus software it has on it, any comments, and so on:

```
CREATE TABLE machine (  
    mac varchar(18) primary key not null,  
    owner varchar(255)  
);
```

And now we need to relate IP addresses to MAC addresses; we use a lease table that tells me when DHCP leases were allocated and when they were cancelled again:

```
CREATE TABLE lease (  
    id integer not null primary key,  
    mac varchar(18),  
    address varchar(22),  
    issued varchar(255),  
    annulled varchar(255)  
);
```

I wanted to make my register, which I whimsically called "Blart" (as readers of The Register's "Bastard Operator From Hell" may appreciate), as easy to get going as possible. If the database file didn't exist, it should create it and the three tables on the fly. I've developed a little trick for doing this:

```

package Blart;
use DBI;
use constant BLART_DB => "/var/lib/blart.db";
-f BLART_DB || do {
    my $dbh =
        DBI->connect("dbi:SQLite:dbname=".BLART_DB)
        or die DBI->errstr;
    local $/ = "";
    $dbh->do($_) for grep /\S/, <DATA>;
}

```

Some virulent machine had been mass-mailing, filling the firewall log with connections, filling up the upstream mail servers, and getting us on some DNS blacklists

The database's schema is stored in the DATA section of the Perl module. The schema being SQL, individual statements are delimited by semicolons. So, if the database doesn't exist, we connect to it (SQLite will automatically give us an empty database) and then execute all the SQL statements we see in the DATA block.

Next, we use my favorite database handling module, *Class::DBI*, to allow us to get at the database through Perl classes. In fact, we'll use *Class::DBI::Loader*, which handles reading the database schema and setting up the classes for us:

```

Class::DBI::Loader->new(
    dsn => "dbi:SQLite:".BLART_DB,
    namespace => "Blart"
);
Blart::Machine->has_a(owner => "Blart::Owner");
Blart::Owner->has_many(
    machines => "Blart::Machine"
);
Blart::Lease->has_a(mac => "Blart::Machine");

```

Now we have our database; we now need some data.

Watching and Discovering DHCP

First we'll watch out for any DHCP offers and leases flying over the network. The *tcpdump* utility is good enough for this, since in its verbose mode, it can dump out the IP address returned by the server to the client. The output of *tcpdump -tnv port bootpc or port bootps* will look like this:

```

1112543649.359207 IP (...) 0.0.0.0.68 >
255.255.255.255.67: BOOTP/DHCP, Request from
00:08:a1:7c:4f:06
Client Ethernet Address: 00:08:a1:7c:4f:06
[|bootp]

```

```

1112543650.400357 IP (...) 192.168.0.254.67 >
192.168.1.121.68: BOOTP/DHCP, Reply
Your IP: 192.168.1.121
Server IP: 192.168.0.254
Client Ethernet Address: 00:08:a1:7c:4f:06
[|bootp]

```

From this, we can extract the client's IP address and MAC address as returned by the server; we can also look at the requests to see if there are some from a real IP address—that is, a client attempting to renew a lease—and store the information about its MAC and IP if we're not currently aware of the lease. First, we add a new method to Blart to help us register a lease:

```

sub lease {
    my ($self, $mac, $ip, $time) = @_;
    $mac = uc $mac;

```

If we already know about this existing lease, give up:

```

return 0
    if Blart::Lease->search(
        address => $ip,
        mac => $mac,
        annulled => 0
    );

```

If we know about the IP address or the MAC address and there's a current lease associated with them, that lease is now invalidated:

```

for (Blart::Lease->search(
    address => $ip,
    annulled => 0),
    Blart::Lease->search(
        mac => $mac,
        annulled => 0)
) {
    $_->annulled($time); $_->update;
}

```

Finally, create the allocation:

```

Blart::Lease->create({
    mac => $mac,
    address => $ip,
    annulled => 0,
    issued => $time});
return 1;
}

```

This will put the appropriate row in the leases table. With this, it's now easy to register leases:

```

Blart->lease( "00:05:5D:F0:E5:8B"
    => "192.168.1.140", time );

```

All we have to do is parse what's coming out of *tcpdump*, and we can note any DHCP allocations as they happen. The CPAN module that's going to help us with this is *Regexp::Common*. *Regexp::Common* contains a huge number of precooked regular expressions for matching all sorts of useful things—among them, IP addresses and MAC addresses. We'll use it to simplify handling the *tcpdump* output:

```

use Blart;
use Regexp::Common;
my ($ip, $mac, $time);
my $ip_re = $RE{net}{IPv4};

```

The `%RE` hash comes from `Regexp::Common`. We've asked it to provide us a regular expression, `$ip_re`, which handles IP Version 4 network addresses. Now we watch what comes out of `tcpdump`:

```
my $file = shift ||
    "tcpdump -ttvv port bootpc or port bootps |";
open IN, $file or die "$file: $!";
while (<IN>) {
```

The first line we want to look at will tell us the timestamp, the two IP addresses (client and server), and whether this is a request or response:

```
if (/^(\\d+).*?(\\$ip_re)\\.\\d+
    > (\\$ip_re)\\.\\d+.*(Request|Reply)/
    ) {
    $time = $1;
    $ip = $4 eq "Request" ? $2 : $3;
}
```

If it is a request, then the client IP is the first one we see because it's talking to a server. If it's a reply, the client is the second IP address, the one that's being replied to. Next we want to look for lines containing the MAC address, and again `Regexp::Common` comes up with the goods:

```
elsif (
    /Client Ethernet Address: (\\$RE{net}{MAC})/i
) {
    $mac = $1;
    Blart->lease($mac, $ip, $time)
    if $ip ne "0.0.0.0" and $ip !~ /^255.255/;
    $time = "";
    $ip = "";
}
```

If this is a real IP—that is, a broadcast made by a host that doesn't have an IP yet—then we record it, reset the variables, and wait for the next DHCP query or reply.

But I found that, actually, lease renewals didn't happen very often, and if I set Blart up and running with an empty database, there was a lot of traffic that it didn't know about. I wanted to get the leases table up and running very quickly, seeded with at least some data. To do this, I again made use of one of the convenient properties of a wireless network—because I can see every packet, I can look down at the link layer as well as the IP layer, and pick up both IP and MAC addresses out of a single packet. The `-e` option to `tcpdump` does that, producing output like so:

```
1112544236.491527 00:08:a1:7c:4f:06 >
00:05:49:d9:44:03 IP 192.168.1.180.3304 >
66.xx.yy.zz.80
```

Now we have two sets of MAC/IP pairs to look at. The problem is, not all of this is going to represent machines on our network. If I go to look at Google's site, then I'll see a connection like the one above; the `66.xx.yy.zz` address is external to our network, and the MAC address given is that of the gateway. We obviously don't want to store this as a DHCP lease; we are not in the business of giving DHCP leases to Google's web servers. So we need to ensure that IP addresses we store are part of our network. The `NetAddr::IP` module is a handy one for comparing network blocks:

```
use NetAddr::IP;

my $home_network =
```

```
NetAddr::IP->new("192.168.0.0/16");
my $address = NetAddr::IP->new("127.0.0.1");
print "Ours" if $home_network->contains($address);
```

So, as before, we look at the output of `tcpdump`, and extract the IPs and MACs from the incoming lines:

Because I can see every packet, I can look down at the link layer as well as the IP layer, and pick up both IP and MAC addresses out of a single packet

```
open IN, "tcpdump -l -en ip|" or die $!;
my $net = qr/^192.168.*/;
my (@macs, @ips);
while (<IN>) {
    next unless
        ($mac[0], $mac[1], $ip[0], $ip[1]) =
            /(\\$RE{net}{MAC}) > (\\$RE{net}{MAC})* IP
            (\\$RE{net}{IPv4})*.* > (\\$RE{net}{IPv4})/i;
```

Notice that we're using an array rather than the more obvious `$mac1` and `$mac2`, because we plan to look at both MAC/IP pairs and do exactly the same thing with each of them. As usual, though, we don't want to repeat code, so we use an array and a *for* loop:

```
for (0..1) {
    next unless $home_network->contains(
        NetAddr::IP->new($ip[$_])
    );
    Blart::Machine->find_or_create($mac[$_]);
    Blart->lease($mac[$_], $ip[$_], time);
}
```

Now we can fill up the leases table a bit faster. In the next step, we want to go from a computer to its owner.

Taking Names

It's at this stage that we need a bit more of a human interface to Blart—we want a tool that allows us to register users, their machines, and to query the database. Because we're performing multiple related tasks with the same tool, I used my `Getopt::Auto` module to handle the command-line processing. This module allows you to define “subcommands,” like so:

```
use Getopt::Auto
[adduser => "Add a user"],
```

```
[registermachine =>
  "Add a machine to a user's profile"],
[what => "Display information on a thing
  (IP/MAC/Name)"],
[unknown => "List computers with no known owner"]
];
```

Now if I say *blartadmin adduser "Joe Bloggs" "Room 10"*, *Getopt::Auto* arranges for the *adduser* subroutine to be called with the parameters from the command line. We're not going to go into all of those functions—we'll just look at the first two, which allow us to register machines and users. Thanks to *Class::DBI*, they're pretty simple:

```
sub adduser {
  my ($user, $room) = @_;
  Blart::Owner->create({ name => $user,
                        room => $room });
  warn "Added user $user, $room\n";
}
```

The second one, registering a machine, is marginally more tricky; we want to be able to register a machine either by IP address or MAC address, which in reality means we need to perform, an IP address lookup and then register it by MAC address. First we write a quick helper method in *Blart* to look up an active lease given IP address:

```
sub ip_to_mac {
  my ($self, $ip) = @_;
  my ($lease) = Blart::Lease->search({
    address => $ip, annulled => 0 });
  if ($lease) { return $lease->mac }
}
```

And now we can register a machine:

```
sub registermachine {
  my ($mac, $user, $room) = @_;
  if ($mac =~ /\./) {
    die "Couldn't find a MAC address for that IP"
      unless $mac = Blart->ip_to_mac($mac);
  }
  my $user_id = Blart::Owner->retrieve($user);
  my $machine = Blart::Machine->find_or_create({
                                          mac => $mac });
  $machine->owner($user_id);
  $machine->update;
}
```

If the "MAC address" has a period in it, it's really an IP address, so we do our lookup to see who that IP address is currently owned by, and register that computer to a particular owner.

At this stage, we have a database that goes from user to computer to IP address. Now things can get interesting.

"Friendly" Intrusion Detection

Suppose we have *Snort*, *tcpdump*, or some other network utility up and running telling us about the state of the network; it'll give us a list of IP addresses merrily talking to other IP addresses. We want to make this a little more personal. If we add this method to *Blart*:

```
sub annot {
  my ($self, $ip) = @_;
  return unless my $mac = Blart->ip_to_mac($ip);
  if($mac->owner) {
```

```
    return "("$mac->owner->name.",
             "$mac->owner->room.")";
  } else { return "("$mac->mac.")" }
```

we have something that takes an IP, and returns something like (Simon Cozens, OH 4). And if our *tcpdump* or *Snort* logs are piped through this command:

```
| perl -MBlart -MRegex::Common -pe
  's/($RE{net}{IPv4})(.\d+)?/
  $1.Blart->annot($1).$2/g'
```

they become marked up with names and room numbers of the guilty parties, and we can watch *Snort* print out logs, like so:

```
02/10-16:34:28 [*] [1:1699:7] P2P
Fastrack kazaa/morpheus traffic
{TCP} 192.168.1.197(John Xxxxx, OH**):
1827 -> 82.41.xx.yy:1409
```

A swift pop up the stairs and a quiet word usually does the job at this point.

There are a few other things I'd like to add to the machine table: Maybe I could automatically run *queso* or *nmap* on hosts joining the network, so we can keep a note of their operating system and potentially vulnerable ports; noting down the network name that Windows machines advertise themselves as can sometimes be helpful to identify unknown machines. (The virus-ridden computer that started off this whole caper was only found because it sent out browser announcements identifying it as LI-BRARYA.)

For the black-hats, maybe the table could be marked up with the output of *dsniff* to record any passwords that the unfortunate user was spraying across the network; I, on the other hand, would merely like to add in the last time I checked that the machine in question was properly protected against viruses.

You believe me, right?

TPJ





A Wireless Popularity Contest

brian d foy

If I can detect a wireless access point, I can figure out who made the hardware. Plenty of people like to make maps of wireless hot spots, and now that every coffee shop, bookstore, and McDonalds seems to have wireless access, the maps are pretty well covered. I'm interested in which hardware people are using, though.

In a cab on the way from the airport last week, I had a lot of juice left in my Powerbook, so I opened it and started doing some work. I was immediately prompted to join a wireless network.

I declined the invitation, of course, but I decided to see what was available. I had a half-hour until I would get home, and I would cover about 20 miles of a main surface street along the way. I fired up MacStumbler to see what was around. This is usually called "war driving," and although I'm not a war driver (I don't even have a car, a basic requirement), I gave it a try from the cab.

I got a lot of hits. Every time I passed a school, whether elementary school or college, MacStumbler would start pinging wildly, and within a couple of blocks the list of access points had scrolled the screen.

Every hit reveals a lot of detail, and I could get even more if I connected a GPS receiver. I can export the information to a text file in wiscan format. There are several columns in the text format, but I've pulled out the first several columns because the full record is too long to fit here. The first two columns are earth coordinates: I've left out the actual coordinates. The next column, contained in parentheses, is the access point name (SSID in wireless parlance). The next column contains the mode: The BSS says that this access point is set up as a Base Station Subsystem. The last column that I show is the one I want because it has information about the hardware manufacturer.

```
N 0 E 0 (Otter Office) BSS (00:11:24:00:c5:a1) ...
N 0 E 0 (Otter Base) BSS (00:0d:93:8b:7b:3d) ...
```

That last column is the MAC address, which uniquely identifies the hardware that handles the network interface (NIC). The first 3 bytes identify the manufacturer, and come from the Institute of Electrical and Electronics Engineers (IEEE). This is also known as the Organizational Unit Identifier (OUI). The IEEE maintains a list of all of the OUIs, along with the manufacturer and address.

I don't have to worry about the IEEE listing, though, because I have a module, *Net::MAC::Vendor*, that handles it for me. I originally made this module to explore my home network and identify machines by their NIC manufacturers. Now I can use it to identify the vendors of wireless access points I came across on the way home.

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.

First, I need to get the MAC out of the text file. I could write a full program to do this, but I want to do this from the command-line because it's more fun that way. I saved the data in a file I named wiscan.txt:

```
% perl -nle 'm/\s([0-9a-f:]{17})\s/i and \
print $1' wiscan.txt
```

I just want the MAC portion. I could create a full parser to get me every column, but I'm in a cab and I have a half hour before I get home. I want to write the program and run it several times before I have to get out of the cab. I cheat in the regular expression a bit because I know that the 17-character sequence of hexadecimal digits and colons is almost always going to be the MAC. I certainly hope no one has named their access point with 17 colons in a row. If the match works, I continue through the *and* short circuit and print the value of *\$1*, which is the part of the regular expression wrapped in parentheses. The *-n* switch wraps a *while(<>){ }* loop around the program I specified on the command line with *-e*, and the *-l* switch adds a newline to the end of my print statement (it also does a *chomp* on the input line, but I don't care about that).

My output is a list of MAC addresses. I'll want to use these to get the OUI information, and then I can count the different manufacturers:

```
00:11:24:08:5e:25
00:11:24:00:c5:a1
00:0d:93:8b:7b:3d
00:0c:41:79:04:95
```

For the next part, I use the *lookup()* function from *Net::MAC::Vendor*. I'll pipe a list of MACs into another program to translate them to vendor names. The *lookup()* function uses the IEEE web site to get the OUI for the MAC, then returns an anonymous array of that vendor's information. The first element in that array is the vendor name. For those four MACs, I get these vendor names. Three of those access points belong to Apple, but that's no big deal: Those are the access points I saw right before I got out of the cab, and they all live in my apartment. That Linksys is downstairs.

```
Apple Computer
Apple Computer
Apple Computer
The Linksys Group, Inc.
```

I start with a simple program that takes a line of input and passes it to the *lookup()* function then prints the first element in the anonymous array it gets back. Before I do any of that, I use the *load_cache()* function, which fetches the entire list of OUIs and parses it so *lookup()* doesn't have to use the network every time I call it. I pay the network penalty just once with *load_cache()*:

```
#!/usr/bin/perl

use Net::MAC::Vendor;

package Net::MAC::Vendor;

load_cache();

while( <> )
{
    my $oui = lookup( $_ );

    print $oui->[0], "\n";
}
```

I save this program as “oui” and can add it to my command pipeline to get that list of vendor names:

```
% perl -nle 'm/\s([0-9a-f:]{17})\s/i and \
print $1' wiscan.txt | oui
```

Before I get too far, I should state this caveat: The vendor name does not necessarily tell you whose name is on the outer case of the product. I’ll ignore that in this article. I want to count the number of times each vendor appears. That’s a bit tricky, though, since the same vendor can have several different blocks of numbers, and it seems that each block has a different form of the vendor name. Here are a few examples:

```
2WIRE, INC.
2Wire, Inc
Apple Computer
Apple Computer, Inc.
Cisco Systems
Cisco-Linksys
Cisco-Linksys, LLC
D-Link Corporation
D-Link Systems, Inc.
```

I modified my OUI program to try to clean up the data a bit. I lowercase everything then uppercase letters after a word boundary, and I get rid of anything after a comma or anything that looks like INC, LTD, and so on. It’s not going to be perfect, because I can already see I can’t handle something like D-Link’s different representations.

```
#!/usr/bin/perl

use Net::MAC::Vendor;

package Net::MAC::Vendor;

load_cache();

while( <> )
{
    my $oui = lookup( $_ );

    my $vendor = lc $oui->[0];

    $vendor =~ s/,.*//;
    $vendor =~ s/\b(llc|inc|ltd|co|corp)(\.\|\\b)//i;
    $vendor =~ s/\s+$/i;
    $vendor =~ s/\b(\w)\U$1/g;

    print $vendor, "\n";
}
```

That cleans up that list quite nicely, save for a few entries such as D-Link. Now I need to count them. I modified my program again to accumulate counts with a hash, then report the results after it has seen all the lines. In the *foreach* loop, I sort the hash keys by their values so I get a descending list of keys based on the number of times I saw an access point from that vendor. I also keep track of the total number of access points that I saw so I can compute a percentage:

```
#!/usr/bin/perl

use Net::MAC::Vendor;

package Net::MAC::Vendor;

load_cache();

my %count;
my $total;

while( <> )
{
    my $oui = lookup( $_ );

    my $vendor = lc $oui->[0];

    $vendor =~ s/,.*//;
    $vendor =~ s/\b(llc|inc|ltd|co|corp)(\.\|\\b)//i;
    $vendor =~ s/\s+$/i;
    $vendor =~ s/\b(\w)\U$1/g;

    $count{ $vendor }++;
    $total++;
}

foreach my $key (
    sort { $count{$b} <=> $count{$a} } keys %count )
{
    printf "%3d  %2d%%  %s\n",
        $count{$key}, $count{$key} / $total * 100, $key;
}
```

Finally, I get my results, and the top entries don’t surprise me: LinkSys has the most routers, and its name shows up in the first two entries. Chicago public schools use those, and there are a lot of schools between my apartment and the airport. Even if I subtract the several Apple Airports I have, Apple is still doing pretty well, and much better than I expected, considering they are probably the most expensive (even if they are the easiest on the eyes).

25	19%	The Linksys Group
22	17%	Cisco-Linksys
21	16%	2wire
16	12%	Apple Computer
8	6%	Cisco Systems
7	5%	D-Link Corporation

If I cared enough, I could do the same sort of thing with the mapping work other war drivers have done and locate clusters of different sorts of access points, but I’ll leave that up to you. Drive safely.

References

Net::MAC::Vendor: <http://search.cpan.org/dist/Net-MAC-Vendor/OUI>: <http://standards.ieee.org/regauth/oui/oui.txt>
 War driving: <http://faq.wardrive.net/>
 MacStumbler: <http://www.macstumbler.com/>

TPJ



Perl 6 Now

Jack J. Woehr

Perl 6 Now sets out to teach Perl 6, the Perl of the future, in terms of Perl 5, the Perl of today. This book is the first I've seen that can legitimately claim to be a genuine Perl 6 book. It does a creditable job of language coverage, but the author is logorrheic and continually core dumps. The combination of good tech and supercilious prose makes *Perl 6 Now* go down something like chili-garlic sauce in Cream of Wheat. It's still a pretty darned funny book, one quite informative on the subject at hand. And you can't fault the good heart of the author, who pledges twelve-and-a-half percent of the royalties from the book to the Electronic Frontier Foundation (EFF).

Author Scott Walters appears from his photo to be one of Jerry's Kids (Garcia, not Lewis). He wandered through Forth and Snobol before reaching Perl, which should have given me a hint of what was coming, your reviewer having written the first book about ANS Forth in a similar style (*Forth: The New Model*. M&T Publishing, 1992, ISBN 1558512772). Walters is a genuine language nut, which renders him entertaining as a raconteur and somewhat infuriating as a tech writer. He's the smitten enthusiast, and he's not letting go of your arm before you've viewed his entire private collection.

In the first section of the book, Walters attempts to reconstruct some of the history of computer science that transpired before his advent on the scene in a fashion most clever and entertaining, albeit exaggerated, tendentious, and inaccurate. For example:

"Perl 5.00, released in 1994, introduced objects, a feature first realized in Simula (1968), where it languished in academia for nearly 20 years before it was popularized by C++."

He means "finally adopted" rather than "introduced," i.e., "introduced" into the autonomous universe of Perl the author inhabits. Though Walters later acknowledges Smalltalk, he seems unaware that it so far from "languished in academia" that it became IBM's primary means of applying GUI technology to enterprise applications for roughly a decade. And:

"From PL/I (1965), Perl 5.005 took multiple concurrent threads of execution."

Jack J. Woehr is an independent consultant and team mentor practicing in Colorado. He can be contacted at <http://www.softwoehr.com/>.

Perl 6 Now: The Core Ideas Illustrated with Perl 5

Scott Walters

Apress, 2005

424 pp., \$39.99

ISBN 1590593952

This is rapture rather than reportage. Perl took threads from the platforms it was executing on and from overwhelming sentiment in the developer/user community that it should do so like every other significant contemporary programming language already did.

The book has a didactic concept, or rather, several. Didactic concept #1 is that you're going to grasp Perl 5, or you will pause to grasp Perl 5, with precious little assistance from the book (perl-doc is suggested), and then proceed to absorb a literary treatment of Perl 6 as a delta of evolution from Perl 5. After the first chapter, didactic concept #2 emerges: Namely, as long as you came to visit, the author will sit down and chattily teach you Perl 6 in its entirety, but you really ought to know Perl 5 because it will be around for a very long while in maintenance, and don't say the author didn't warn you. Yet a third concept is that the book will teach you Perl by teaching you all of computer science that might conceivably be relevant to the particular language construct under discussion, and explain how Perl embodies the theoretical abstract, or strives to, or fails to, or used to fail but now conquers the heights victoriously in Perl 6.

In the Introduction, Walters means to teach you, in a general way, how to install and maintain Perl, but he has such a wonderful time telling the reader about many other things he has studied that the fragmentary and idiosyncratic choice of practical content for the chapter is squeezed exclusively into the last few paragraphs.

Part 3, “Threads and Objects,” commences with Chapter 12, “CPAN Modules,” the “Summary” of which intones thusly:

“Remember that (like so many modules documented in this book), *B::Bytecode* isn’t stable but is considered a working prototype. Generally, when it fails, it fails completely rather than subtlety. The status of other modules mentioned in this section varies between experimental, proof of concept, and production. This doesn’t mean that bug reports aren’t welcome, but reported bugs may not necessarily be fixed in a timely fashion.”

Is that really the item that should be foremost in the mind of the reader at this juncture of the exposition?

Chapter 10, “Block Structure,” commences:

“Assembly language, COBOL, and old versions of BASIC don’t indent code to distinguish the code run in a loop or run conditionally from the normal flow. Block structure defines scope. Scope is the area (approximately the starting line and ending line or the file or the package) in which a variable, method, or subroutine is valid.”

This sort of treatment is the norm in *Perl 6 Now*. The third sentence (with some work) should have been the first, the first should never have gotten past the editor, and the word “defines” should be “delimits.”

Chapter 14 “Objects” starts out:

“Objects are instances of classes. Classes are reusable logic. The logic may be reused not only in different programs but multiple times in the same program. Each instance of reuse has its own private data, cleverly named instance data. This chapter is about the *Perl6::Classes* module, the *Attribute::Property* module, and Apocalypse 12, the object specification for Perl 6.”

Is there any reader who understands this chapter-opening paragraph who needed to hear the first two sentences? Another discussion informs the reader:

“given blocks topicalize a parameter as `$_`, the default variable, for when blocks to test against. In topicalizing its parameter to `$_`, *given* behaves like the *for* statement. *when* is a specialized version of if that knows how to compare different types to each other and executes the equivalent of a *last* when a match is found.”

As the “Who This Book Is For” acknowledges in a roundabout way, the book is written primarily for people who are into Perl qua Perl. Surely, they will lavish *Perl 6 Now* with praise in the newsgroups. Some may even read it, if they have time to spare from programming Perl.

The chapters can be cross-referential, with knowledge from a later chapter needed to grasp the discussion of the current chapter. An example is Chapter 20, “Continuations,” which commences with a few confiding observations that imply you understand coroutines, discussed in Chapter 21. Admittedly, a chart of chapter dependencies is found in the introduction. But more coherent and disciplined technical writing would have rendered this superfluous.

Anecdotal irrelevancies and superfluities lard the text of *Perl 6 Now*. Yet there’s real substance everywhere. You just have to pick out the signal from carrier wave, which can be difficult when an author imposes so much of his intellectual context on the reader. Walters quotes Albert Einstein, “Intellectuals solve problems: geniuses prevent them.” The good doctor might also have observed that intellectuals striving towards genius cause almost as many problems as they solve, which would explain a lot about this book.

TPJ

Subscribe now to

Dr. Dobb's E-mail Newsletters

They're Free! <http://www.ddj.com/maillists/>

- ✓ **AI Expert Newsletter.** Edited by Dennis Merritt; the AI Expert Newsletter is all about artificial intelligence in practice.
- ✓ **Dr. Dobb's Linux Digest.** Edited by Steven Gibson, a monthly compendium that highlights the most important Linux newsgroup discussions.
- ✓ **Dr. Dobb's Software Tools Newsletter.** Having a hard time keeping up with new developer tools and version updates? If so, Dr. Dobb's Software Tools e-mail newsletter is just the deal for you.
- ✓ **Dr. Dobb's Data Compression Newsletter.** Mark Nelson reports on the most recent compression techniques, algorithms, products, tools, and utilities.
- ✓ **Dr. Dobb's Math Power Newsletter.** Join Homer B. Tilton and expand your base of math knowledge.
- ✓ **Dr. Dobb's Active Scripting Newsletter.** Find out the most clever Active Scripting techniques from Mark Baker.

Sign up now at <http://www.ddj.com/maillists/>

Source Code Appendix

Julius C. Duque “Creating Self-Contained Perl Executables, Part II”

Listing 1

```
#!/usr/local/bin/perl

# sample1.pl
# Julius C. Duque

#use diagnostics;
#use strict;
#use warnings;
use Cwd;
use Getopt::Long;
use File::Find;
use Win32::Autoglob;
use Digest::MD5;

my $VERSION = "1.0.0 (for TPJ)";

my ($showfiles, $showdigests, $recursive, $all, $quiet, $help) = ();

GetOptions(
    "showfiles" => \$showfiles,
    "showdigests" => \$showdigests,
    "recursive" => \$recursive,
    "all" => \$all,
    "quiet" => \$quiet,
    "help" => \$help
);

$showfiles = $showdigests = 1 if ($all);

syntax() if ($help or !@ARGV);

foreach my $infile (@ARGV) {
    chomp $infile;
    if (! -e $infile) {
        print "**** ERROR: $infile does not exist, skipping it...\n" if (!$quiet);
        next;
    } elsif (-d $infile) {
        if ($recursive) {
            find({wanted => sub {
                if (-f) {
                    print make_digest($_, 'MD5');
                    print " $_" if ($showfiles);
                    print " [MD5]" if ($showdigests);
                    print "\n";
                }
            }, no_chdir => 1), $infile);
        } else {
            if (!$quiet) {
                print "**** ERROR: $infile is actually a directory, skipping it.\n";
                print "**** ERROR: Use --recursive if you want to process $infile recursively\n";
            }
            next;
        }
    } else {
        print make_digest($infile, 'MD5');
        print " $infile" if ($showfiles);
        print " [MD5]" if ($showdigests);
        print "\n";
    }
}

sub make_digest
{
    my ($file, $tmd) = @_;
    my $digest_obj;
    open INFILE, $file or die "Cannot open $file: $!";
    binmode INFILE;
    $tmd = $tmd =~ /^Digest::/? $tmd : "Digest::$tmd";
    eval "require $tmd";
    $digest_obj = new $tmd;
    $digest_obj->addfile(*INFILE);
    close INFILE;
    return $digest_obj->hexdigest;
}
```

```

sub syntax
{
    if ($^O eq "MSWin32") {
        $0 =~ s/.*\///g; # Windows
    } else {
        $0 =~ s/.*\///g;
    }

    print "$0 $VERSION\n\n";
    print "Usage: $0 file1 [file2 ...]\n";
    print "\n";
    print "Other options:\n";
    print "  --showfiles    print filenames\n";
    print "  --showdigests  print digests used\n";
    print "  --recursive    recursively descend into directory\n";
    print "  --all          implies --showfiles and --showdigests output\n";
    print "  --quiet        suppress error messages\n";
    print "  --help        print this help message\n";
    exit 1;
}

```

TPJ