

The Perl Journal

Perl and Inline Octave Code

Andy Adler • 3

Catching Cheats with the Perl Compiler

Deborah Pickett • 7

Grokking Web Archives

brian d foy • 12

Bayesian Analysis for RSS Reading

Simon Cozens • 15

A Better *Data::Dumper*

Randal Schwartz • 19

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Jack J. Woehr:

***Perl Template Toolkit* • 23**

Source Code Appendix • 24

LETTER FROM THE EDITOR

Simple Scripts to the Rescue

Automation is one of those futuristic promises, like personal jet packs and TV wristwatches, that never quite seems to be fulfilled. The personal computer has had more than two decades to deliver us from drudgery, and while it has undeniably made some tasks far easier, there's still a lot of drudgery going on out there, much of it done, ironically, on the computer.

Part of the problem is that computers don't solve every problem. And even for the problems they do solve, we have often underestimated the difficulty of devising algorithms to describe our tasks. It's a fundamental human/machine incompatibility, and those of us who are the most adept at resolving that incompatibility become programmers (ideally, that is).

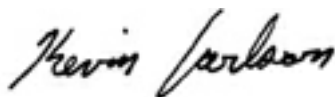
But there's a more subtle problem that holds back our potential to automate those tasks we really would rather that our machines did for us. The problem is that those who most need the automation are often those least able to provide it for themselves. The best person to devise an automatic process is the person who normally does the process manually and knows that process in agonizing detail. If it solves their own problem, they are highly motivated to make it work properly.

This is the whole idea behind scripting languages: Give the power of the *for()* loop and the *if/else* statement to mere mortals. Let the person who toils in a word processor or spreadsheet program write the code to take away their own burdens. That's the theory, anyway. The reality is that, even with simple scripting languages, the realm of the accomplished scripter is a rarified one. Tasks go unscripted because few people gain the skills to translate their detailed knowledge of a process into an algorithm for executing that process. Why the majority of people don't gain these skills is a whole separate matter that, even if I thought I fully understood, there wouldn't be enough room to discuss here.

Most people aren't programmers and aren't going to be anytime soon. What's interesting is the effect this has on little languages. They get used not by the general public, who need a really simple solution that they can tailor to their own needs, but by those of us who are already programmers. And what do we want? More features. So over time, complexity creeps in and puts these languages even further out of the perceived reach of novices. This also explains why some of the cleverest, best-designed software out there tends to solve problems that programmers have, not problems the general user population has.

Bottom line: The tasks that need automating either don't get automated at all or they get automated not by the people performing the task, but by white-coated laboratory wizards for whom the requirements of the project must be specified precisely. It's a fragile arrangement that often yields either brittle processes that can't change over time or too-general solutions that the end user has no idea how to implement.

The solution to this problem lies with perceptions. Some people are never going to like programming and will continue to do repetitive tasks just to avoid learning a programming tool. But, many others are kept from coding their own modest, yet effective, solutions simply by the perception that it's too hard or too boring. Or it just doesn't occur to them that there's a better way. Maybe it's just foolish optimism on my part, but I like to think they just haven't been introduced to the power and beauty of a simple *foreach()* loop.



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 (ISSN 1545-7567) is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: \$18.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2004 by CMP Media LLC, unless otherwise noted. All rights reserved.



The Perl Journal

EXECUTIVE EDITOR

Kevin Carlson

MANAGING EDITOR

Della Song

ART DIRECTOR

Margaret A. Anderson

NEWS EDITOR

Shannon Cochran

EDITORIAL DIRECTOR

Jonathan Erickson

COLUMNISTS

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

CONTRIBUTING EDITORS

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

INTERNET OPERATIONS

DIRECTOR

Michael Calderon

SENIOR WEB DEVELOPER

Steve Goyette

WEBMASTERS

Sean Coady, Joe Lucca

MARKETING / ADVERTISING

PUBLISHER

Timothy Trickett

MARKETING DIRECTOR

Jessica Hamilton

GRAPHIC DESIGNER

Carey Perez

THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403

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

CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND

MARKETING Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

PRESIDENT, TECHNOLOGY SOLUTIONS Robert Faletta

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 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

Your Regularly Scheduled Apocalypse

Larry Wall posted an outline of future Apocalypses to the perl6-language list, with brief comments on the unwritten chapters. The schedule posits some 33 Apocalypses, but many of these are being written in parallel, and most contain significantly fewer RFCs than the first six chapters. “Perl 6 now is pretty much A1—6 plus 12,” Larry commented. “That’s actually most of the earth-shaking stuff.” He also mentioned that Apocalypse 12, covering objects, “is mostly written.” The Apocalypse outline is at <http://www.mail-archive.com/perl6-language@perl.org/msg15276.html>.

Meanwhile, Damian Conway published Exegesis 7, explaining how text formatting will work in Perl 6. “If you’re a regular user of Perl 5’s *format*,” he notes, “you might like to try the *form* function instead. It’s available right now in the *Perl6::Form* module, which waits upon thy pleasure at the CPAN.” (For those who can’t remember Apocalypse 7, titled “Formats”—it was appended to Apocalypse 6 and consisted of: “Gone from the core. See Damien.”)

A String of Perls

“Ladies and gentlemen, I give you...objects!” wrote Leo Tötsch, announcing the Parrot 0.1.0 release. Dubbed the “Leaping Kakapo” release—as it was made public on February 29—Parrot 0.1.0 also boasts basic multithreading support, better documentation, and many more supported platforms as well as lots of other features and bug fixes. A kakapo, in case you were wondering, is a rare flightless parrot from New Zealand.

In other release news, the Perl 5 developer team has completed Perl 5.0005_04, a maintenance release that allows Perl 5.005 to compile and run under UNIX systems with recent Berkeley DB libraries or GCC 3 as well as under Mac OS X. According to pumpking Leon Brocard, 5.005_05 will be released later in the year and will feature more portability fixes.

Haiku Contest Winners Announced

Hundreds of haikus were submitted to the ActiveState Perl Haiku Contest, in both the categories of Haikus Written About Perl and Haikus Written *In* Perl. Netherlander Ed Snoeck took top honors in the first category with a poem that evokes both pearls and Perl:

ugliness that grows
into beauty inside of
your favorite shell

Dallas programmer James Tilley won the grand prize in the second category with his rigorous poesy:

no less can I say;
require strict, close attention
while you...write haiku

A couple of poets had the temerity to submit haikus extolling the virtues of C or taking digs at “unreadable” code, but most of the entries seem as heartfelt as they are witty. You can read them all at <http://aspn.activestate.com/ASPN/Perl/Haiku/>.

Upcoming Events

YAPC::Europe has issued a call for papers. The theme of the conference is “Perl for Profit,” but talks on other topics will also be considered. Proposals must be submitted by Friday, April 30th. The conference will be held in Belfast, Northern Ireland, September 15–17. See <http://belfast.yapc.org/> for details.

The First Italian Perl Workshop will be held in Pisa, July 19–20. “Most of the workshop will probably be in Italian, but talks in English are welcome, too,” one organizer noted on perlmonks.org. The web site is <http://www.perlworkshop.it/>.

An Austrian Perl Workshop is scheduled for May 20–22 in Vienna. Leo Tötsch is giving a Parrot tutorial, and other talk proposals are welcomed at http://vienna.pm.org/en_ws_talk.html.

Not to be confused with the Austrian workshop is the first Australian YAPC, which will be held this December in Melbourne. The conference is still taking shape; details will appear at <http://yapc.dlist.com.au/>.

Perl Foundation Announces Grants Committee Membership

The Perl Foundation has announced new members of the Grants Committee. Dan Sugalski, who received a grant in 2002, will be cochairing the committee with Hugo van der Sanden. The other voting members of the committee are Leon Brocard, Rafael Garcia-Suarez, Nicholas Clark, Brigitte Jellinek, and Stas Bekman. Allison Randal, the president of Yet Another Society, will have a nonvoting place on the committee as board liaison. Committee members cannot be awarded grants.

The Source of the Search

Randy Kobes noted on use.perl.org that the CPAN-Search-Lite project is now on SourceForge, with source code available (<http://sourceforge.net/projects/cpan-search/>). “This doesn’t have all the great features of search.cpan.org,” he wrote, “but it may be useful to those interested in setting up a minimal searchable CPAN database—it can even be used on a box without a local CPAN mirror (albeit without the capabilities of extracting module and package docs).”

Perl and Inline Octave Code

I use Perl to manage files, and Octave (a high-level, numerical-computation language) to crunch numbers. Recently, I worked on a project that generated enormous data files, which needed to be processed and then analyzed—a perfect task for my two favorite languages. Since I’d just heard a mighty cool talk on the *Inline* module, it seemed clear to me that I needed to write *Inline::Octave*. Unlike some other *Inline* languages such as C or Java, Octave runs as an interpreted environment and does not natively support sockets or other interprocess communication facilities. The choices were, therefore:

- Modify Octave.
- Control it from Perl by typing into the interpreter and reading the output.

I chose the latter approach to allow the technique to work with unmodified Octave.

Perl allows controlling the STDIN, STDOUT, and STDERR of a process using the *IPC::Open3* module. However, it contains a number of warnings; for example, “This is very dangerous, as you may block forever.” Suitably forewarned, I set out on this hazardous enterprise, learning a number of tricks, which I will describe in this article.

Example

Is the global temperature rising? Lets suppose we’ve decided that we don’t trust those pundits, and we’d like to calculate for ourselves whether the earth is getting warmer. However, for some crazy reason, we do trust random stuff published on the Internet, so we type “Daily Temperature Data” into Google. The third link, “Temperature Data Archive,” looks promising and takes us to <http://www.engr.udayton.edu/weather/>. Looking further, we find a link with the title, “All sites in single file (about 4.5 Megabytes),” and download it.

Andy is an assistant professor at the School of Information Technology and Engineering at the University of Ottawa in Canada. He can be reached at adler@site.uottowa.ca

```
$ wget ftp://ftp.engr.udayton.edu/jkisssock/gsod/allsites.zip
```

The zip file has a simple structure; data from each temperature measurement site is in a separate archive file, and each file is structured into lines with whitespace-separated numbers for the day, month, year, and temperature. Clearly, this is a job for Perl. The script I’ll describe here, *temp-analyse.pl*, takes the name of the file as a parameter and prints the calculated data.

Perl Code

Here’s the beginning of *temp-analyse.pl*:

```
01 use Time::Local;
02 my ($city, @time, @temp, %rates);
03
04 open F, "unzip -c $ARGV[0] |" or die "Can't open $ARGV[0]: $!";
05 while (<F>) {
06     if (/^\s+ (\d+) \s+ (\d+) \s+ (\d+) \s+ ([^\d.]+)/x) {
07         next if $4 == -99;           # code for no data available
08         push @time, timegm( 0, 0, 12, # sec, min, hour (assume noon)
09                             $2, $1-1, $3-1900); # mday, month (0-11), year
10         push @temp, ($4-32)/1.8;     # convert to degree C
11     }
12     if (/^\s+ inflating: \s+ (\w+) \.txt/x) {
13         process(\@time, \@temp) if $city;
14         $city= $1; @time= (); @temp= ();
15     }
16 }
17 process(\@time, \@temp); #last one
18 close F;
19
20 printf "Average change is %1.4f ± %1.4f (°C/year) for %d cities\n",
21        calc_stats()->as_list;
```

We need to convert from time in days, months, and years to a linear unit of time, such as seconds since the epoch. Perl has lots of time modules for advanced processing, but *Time::Local* is included with Perl, does the job, and is easy to use. We include this

module in line 1 and use it in line 8. Perl also provides a great file-open function to allow reading from a pipe (line 4), and we use the command-line `unzip -c filename` to feed each file in turn to Perl.

The command `unzip -c` will loop through the contents of the archive, and for each file, output “inflating: filename” followed by the file contents. We parse the file contents (whitespace-separated numbers) on line 6 using a regular expression. The last number is the temperature and can include a minus sign and a decimal point,

In Perl, we are able to easily glue code in various languages together thanks to the infrastructure provided by Brian Ingerson's Inline module

so we use the pattern `([\\-\\d\\.]+)` to match any number of these characters. The web site indicates that a temperature of `-99` indicates missing data. We first test for that condition (line 7) and skip the processing if it is found. Otherwise, if the data are good, they are stored into the lists `@time` and `@temp`.

We detect that a new filename in the archive is being decompressed by matching the text “*inflating:*” (line 12). This means that the data for the previous file is complete, and can be processed by the subroutine `process` (line 13). The lists are then reset (line 14), and we return to processing the contents of the new archive file. When all data are received, we process the data in the last file (line 17) and close the file (not strictly necessary). Then, in lines 20–21, we call a subroutine to calculate the statistics and print them for the user.

Up to this point, Perl is unquestionably the right choice of language. Its ability to integrate with other tools via a pipe and to parse text with a minimum of painful syntax has allowed us to accomplish this part very quickly. Unfortunately, the mathematical analysis of this data is less easy with Perl. Of course, a suite of mathematical modules does exist for Perl (PDL), but the most common language for this sort of analysis is Matlab. Luckily, the open source GNU Octave software (<http://www.octave.org/>) is mostly compatible with Matlab and can be readily integrated with Perl.

Octave Code

Since this is an article about Perl, the calculations will be described briefly. Figure 1 illustrates some of the difficulties in estimating the trend line from the data. The graph shows temperature (degrees C) versus time for the years 1995–2003 for my hometown (Ottawa, which happens to have one of the largest annual temperature ranges). The raw data (red), shows a very large annual temperature cycle, much larger than the effect we’re trying to measure (blue). In order to estimate the trend, we need to remove the components of the data that are in phase with the year (green). The best fit line is then calculated for the data (blue), and the slope of the line in degrees/year is the temperature trend.

The code shown below implements the functions `process` and `calc_stats`. `time` and `temp` are vectors corresponding to the Perl lists `@time` and `@temp`. Lines 25–31 calculate the component of

the signal in phase with the first three harmonics of the year frequency. The value 365.2422 is the number of days in a year (see <http://www.straightdope.com/mailbag/mleapyr.html>). In lines 32–34, the year harmonics are removed from the signal (to calculate `temp_clean`), the best fit line is calculated, and the subroutine return value `TperYr` is calculated. Finally, in lines 36–37, the slope is stored in the vector `sites` that, because of the keyword `global`, may be accessed by the subroutine `calc_stats`. This last subroutine calculates the data mean and standard error, and the number of sites:

```
24 function TperYr= process( time, temp );
25     time_step= [0;diff(time)/2] + [diff(time)/2;0];
26     year_freq= 1/( 365.2422*24*60*60 );
27     harmonics= 2*pi*(1:3)*year_freq;
28     year_osc= [ sin(time * harmonics), cos(time * harmonics) ] ...
29               .* (time_step * ones(1,2*length(harmonics)));
30     component= (temp' * year_osc) ./ sumsq( year_osc );
31
32     temp_clean= temp - year_osc * component'; # remove year harmonics
33     fit = polyfit( time, temp_clean, 1);      # fit to line
34     TperYr = fit(1) / year_freq;              # convert to deg/year
35
36     global sites=[]; static site_no=1;
37     sites( site_no++ ) = TperYr;             # store city data
38 endfunction
39
40 function stat_data= calc_stats()             # mean, std err, number
41     global sites;                            # load city data
42     n_sites= length(sites);
43     stat_data= [mean(sites), std(sites)/sqrt(n_sites), n_sites];
44 endfunction
```

Inline

In Perl, we are able to easily glue code in various languages together thanks to the infrastructure provided by Brian Ingerson's *Inline* module. The *Inline::Octave* module, which I wrote, allows us to glue these functions together with two lines of Perl:

```
23 use Inline Octave => q{
24-43 # Octave code (above)
45 } ;
```

Finally, with the code assembled together, we get:

```
$ perl temp-analyse.pl allsites.zip
Average change is 0.0585 ± 0.0096 (°C/year) for 324 cities
```

Our results agree with the pundits; there is a warming trend. Be careful, however; this analysis looks at a small slice of time,

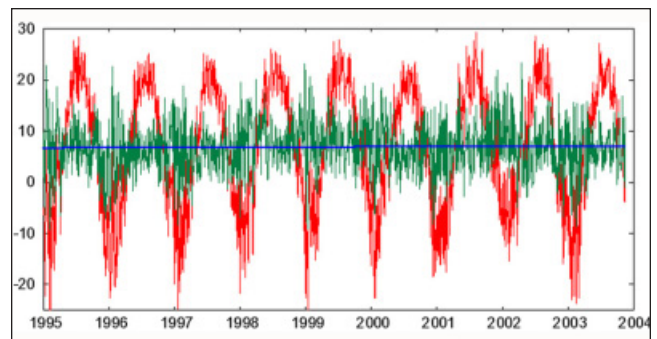


Figure 1: Temperature (red) (degrees C) vs. time (years) in Ottawa. Components in phase with the year are removed (green), and the best fit line calculated (blue).

is not mathematically rigorous, and, most importantly, says nothing about the underlying causes.

A Closer View of the Glue

The *Inline* infrastructure allows the details of the glue to be hidden. An *Inline* language may be compiled and linked to Perl, such as C, or may be interpreted, such as Python, Java, or Octave. One wrinkle in this classification is that Java, being a hybrid interpreted/compiled language, can also be linked to Perl. Both *Inline::Python* and *Inline::Java* set up a socket connection for interprocess communication. Unfortunately, stock Octave does not have sockets, so the best way to control it seemed to be to type commands into it from Perl, and read back the output. (After *Inline::Octave* was developed, Paul Kienzie developed a similar

[Perl's] ability to integrate with other tools via a pipe, and to parse text with the minimum of painful syntax, has allowed us to accomplish this part very quickly

Tcl/Tk glue to Octave, and decided to approach it by building a socket library for Octave. For details, see <http://www.arxiv.org/pdf/physics/0211037>)

In order to do this, Perl provides the *IPC::Open3* module to control the STDIN, STDOUT, and STDERR of a process. However, the documentation says: “If you try to read from the child’s STDOUT writer and their STDERR writer, you’ll have problems with blocking.” Well, I did try, and I did have problems; however, in solving these problems, I learned a number of tricks that I describe here. These techniques are appropriate for any interpreter and should work for shells and for command-line utilities. The core requirement is that the interpreter tool is understood well enough that it can’t give unexpected output.

In order to illustrate the techniques used to do this kind of interprocess communication, I’ve built a small module, *Example.pm*, which illustrates the IPC functionality:

```
01 package Example;
02 use strict;
03 use Carp;
04
05 use IO::Handle;
06 my $Oerr= new IO::Handle;
07
08 use IPC::Open3;
09 open3( my $Oin, my $Oout, $Oerr, "octave -qH");
10 eval{ setpriority 0,0, (getpriority 0,0)+4; } #lower priority slightly
11
12 use IO::Select;
13 my $select = IO::Select->new($Oerr, $Oout);
```

The *Carp* module (line 3) is used in order to allow errors and warnings to appear from the point of view of the calling subrou-

time. Prior to calling *open3*, it is important to ensure that we have a valid STDERR (*\$Oerr*); otherwise, error output will be sent to STDOUT. This is accomplished in lines 4–5, using the *IO::Handle* module. The *octave* process (line 9) is started with appropriate command-line flags, and handles to STDIN, STDOUT, and STDERR are created. *open3* will die if the process can’t be started, which is just fine for this example. I’ve found that decreasing the priority of the parent process (Perl) slightly (line 10) can make this kind of communication more efficient. The idea is that the interpreter has more of a chance to fill its output buffer, and Perl makes fewer (but larger) input reads. The priority control functions are not available on some systems (e.g., Windows) and will cause a fatal error. The solution is to wrap line 10 in “eval {}”.

In lines 12–13, an *IO::Select* object is created with the STDOUT and STDERR filehandles. *IO::Select* is a (very convenient) wrapper around select that allows multiple filehandles to be waited on simultaneously. Without this function, one could make a blocking read on one handle, but if the writing process wrote to the other, it could result in deadlock. I’ve quite often seen programmers who try to solve this type of problem with polled I/O; nonblocking reads are alternately made on each filehandle. This is an inefficient, unreliable, and generally bad solution. The *select* approach allows the OS kernel to do the job (perhaps using hardware interrupts or other facilities unavailable to user-space software). Surprisingly, *IO::Select* works fine on windows (tested on 2000+), even while the underlying C implementation only works for sockets and not filehandles in win32.

Once the process has been started with *open3*, we want to be able to call a subroutine *interpret*, which will call the interpreter with the supplied string and return its output:

```
14 sub interpret {
15     my $cmd= shift;
16     my $marker= "-9Ahv87uhBa8l_80ng,zU9-"; # random string
17     my $marker_len= length($marker)+1;
18
19     print $Oin "\n\n$cmd\ndisp('$marker');fflush(stdout);\n";
20
21     my $input;
22     while ( 1 ) {
23         for my $fh ( $select->can_read() ) {
24             if ($fh eq $Oerr) {
25                 process_errors();
26             } else {
27                 sysread $fh, (my $line), 16384;
28                 $input.= $line;
29             }
30         }
31         last if substr( $input, -$marker_len, -1) eq $marker;
32     }
33
34     # leave process blocked doing something, or it can't handle CTRL-C
35     print $Oin "\n\nfread(stdin,1);\n";
36     return substr($input, 0, -$marker_len );
37 }
```

In order to avoid deadlock, we need to clearly separate the phases when we write to or read from the interpreter. To be able to detect the end of the output from the interpreter, we define a marker (*\$marker*, lines 16–17) with a random string that is, with any luck, very unlikely to appear in the output by itself. Then (line 19), the command *\$cmd* is sent to the interpreter STDIN (*\$Oin*). To the command, we prepend newlines (explained later), and append a command (*disp*) to print the marker after the completion of the instruction.

In lines 21–32, we enter an infinite loop, waiting for output from the interpreter and looking for the marker string. On line 23,

we use the *select* function to wait on input from the interpreter on either STDERR and STDIN. If STDERR input is received (lines 24–25), we do special processing in the subroutine *process_errors*; otherwise, STDIO is read and appended to *\$input* (lines 27–28). The function *sysread* does nonbuffered I/O on a filehandle. It is required in this instance—using *\$line= <\$fh>* results in a deadlock.

The input is tested for the *marker* string (line 31) and, if found, the loop is terminated and the interpreter output (without the marker) is returned at line 36. One final trick is to leave the interpreter blocked reading on STDIN. This is the reason for the prepended newlines in line 19; the *fread* instruction needs some “throw away” input to consume. This blocking read is required to allow a clean shutdown of Perl and Octave when the user enters Ctrl-C. There seems to be an issue with GNU *readline* (which is used by Octave and many other interpreters, including bash). If the child interpreter receives Ctrl-C and the interpreter is not blocked, *readline* seems to panic and prevent a clean shutdown. This is a hack, but it may be useful in other scenarios as well. Another possible approach would be to trap Ctrl-C (using a custom *\$SIG{INT}* subroutine), then send appropriate commands to the interpreter.

The interpreter sends errors and warnings to its STDERR. If we detect this, the subroutine *process_errors* is called:

```
39 sub process_errors
40 {
41     my $select= IO::Select->new( $Oerr );
42
43     my $input;
44     while ( $select->can_read(0.1) ) {
45         sysread $Oerr, (my $line), 1024;
46         last unless $line;
47         $input.= $line;
48     }
49
50     croak "$input (in octave code)" if $input =~ /error:/;
51     carp  "$input (in octave code)" if $input;
52 }
53
54 1;
```

The concept behind *process_errors* is that warnings and errors will be short bursts of text written to STDERR. We are thus able to temporarily stop reading from STDIN and focus exclusively on STDERR until the error text is complete. The end of the stream of error text is defined to be 0.1s with nothing read on the filehandle. An *IO::Select* object is created with only the STDERR filehandle (line 41). This allows us to test for activity with a timeout with the *can_read* method (line 44). After the error text is read, we need to decide whether it constitutes a warning or an error; for this simple example, we simply test for the string “error:” (line 50). By using the *carp* and *croak* functions, warnings and errors are generated, which will appear as a “warn or die” within the code that calls the *interpret* subroutine. Normally, Perl will print *warn* text to STDERR and continue, but will stop execution for *die*. Of course (being Perl), this behavior can be altered. *die* can be intercepted by wrapping the commands in *eval*:

```
eval {
    # commands
}; if ( $? ) {
    # error handler
}
```

warn can be intercepted by defining a handler for the *__WARN__* pseudosignal. Within a block, we use *local* to override

\$SIG{__WARN__} to a handler subroutine of our choice. See *perl-var(1)* for the gory details:

```
{ local $SIG {__WARN__} = sub {
    my $warning = shift ;
    # warning handler
};
# commands
}
```

In order to illustrate the functioning of *Example.pm*, the following examples call *Example::interpret* with a string that is evaluated with:

```
1) no errors ("1/2");
2) a warning ("1/0"), and
3) a syntax error ("1/**")

01 $ perl -MExample -e'print Example::interpret("1/2")'
    ans = 0.50000
02 $ perl -MExample -e'print Example::interpret("1/0")'
    warning: division by zero (in octave code) at -e line 1
    ans = Inf
03 $ perl -MExample -e'print Example::interpret("1/**")'
    parse error:
    >>> 1/*
        ^ (in octave code) at -e line 1
```

In terms of performance, this technique seems to be mostly limited by the speeds of the interpreters, and the loops and memory allocation for the I/O. Measurements on a PII 266 MHz (running Linux Mandrake 9.1) show that data can be transferred at about 0.8 MB/sec, while Perl can write to */dev/null* at about 50 MB/sec.

Thus, we’ve seen that Perl makes it quite easy to integrate an interpreter into a Perl script by controlling its input and output filehandles. It is, of course, quite true that there are several traps to avoid, and mistakes typically result in process deadlock. On the other hand, when done carefully, this capability can be very useful. We see that, once again, Perl merits the distinction as “the duct-tape of the Internet.”

TPJ



Catching Cheats With the Perl Compiler

Laziness, impatience, hubris. Perl users have been raised to believe that these are the virtues of a good programmer, but they have a dark side. They are also the character flaws of the cheat and plagiarist:

Laziness: I can't be bothered learning how to program in this language.

Impatience: If I copy off my friend, then I'll be able to do stuff I actually enjoy doing sooner.

Hubris: I won't get caught.

The issue of plagiarism doesn't often come up in the world of Perl, perhaps because of the Perl community's commitment to open source and giving credit where it's due. But it's a different story in the introductory Perl programming course that I teach at Monash University. Here, the assignments I set for my students must be the students' own work, and students who copy others' work without giving credit are considered to be cheating the system. Transgressors are punished, swiftly and mercilessly.

At least they would be if a tool existed for comparing Perl programs with each other. There are plenty of tools for comparing C and Java and other languages, but I couldn't locate any for Perl. Ironically, a package my university uses to compare C code, called "Moss," uses Perl, but doesn't compare Perl source code itself.

Perhaps this absence of a comparison tool is partly due to the aforementioned lack of need, but it surely must also be because Perl is a notoriously difficult language to parse. Simple substring comparison isn't good for detecting similarities in code because people change indentation, comments and variable names. To properly get a picture of what the program is doing, it's necessary to parse the source.

Only *perl* can Parse Perl

There are two choices when it comes to parsing Perl. The first option is to write a Perl grammar in whatever yacc-like notation you prefer, and generate a parser that accepts that grammar. While this is easy in C, it's close to impossible in Perl because of the language's syntactic idiosyncrasies. However, this may not be too great a handicap since typical Perl programs, as written by neophytes, don't use such features; it may be possible to parse a decent subset of Perl using off-the-shelf tools such as *Parse::RecDescent*. A big advantage of this approach is that whitespace, comments, and other nontokens that the Perl parser ignores could be examined, too, for hints of common source-code ancestry.

The second solution is to make *perl* (the executable) parse Perl (the language), something that, by definition, it will always get right. There are two ways: The first is to use Perl's *-Dx* command-

line option, which spits out an ugly syntax dump of a program, and parses the output into some other form. A few years ago, this would have been the only option. But with the introduction of the *B::** suite of compiler back ends, there is a better choice: Create a new subclass of *B* that picks salient features in the source code's parse tree, and pipe the program through it. Unfortunately, some features of the source, such as whitespace, will be lost because the Perl tokenizer strips these before the parse tree is built.

I think that a robust solution to the code-similarity problem needs to use some of each of the aforementioned two approaches. For a quick-and-dirty solution, however, I opted to make use of the Perl compiler and wrote a module called *B::Fingerprint*, which turns a program into a reasonably short and descriptive string, which, in turn, can be analyzed using more traditional string-comparison tools.

The M.O. of a Plagiarist

Plagiarists typically start with a working, completed piece of code written by someone else, and either try to work it into their own broken code or scrap their own code and spend the rest of their time trying to make the original code look different. Because they don't have a great deal of confidence in the language, they tend to make small, incremental changes to the code and hope the program still works (as a rule, plagiarists aren't terribly good at testing code).

The most common transformations are:

- Rewriting the comments;
- Indenting the code differently;
- Changing variable names, and
- Reordering subroutines in the program.

Somewhat rarer changes include changing *if* to *unless* and reordering a bunch of independent initialization statements.

Any technique that compares programs for evidence of copying should try to downplay the effects of these transformations and look at the program's deeper structure, which will probably be left untouched.

B::Fingerprint

As its prefix suggests, *B::Fingerprint* is a compiler back end. Back ends are modules that can examine or manipulate the opcode tree of a Perl program, and usually finish up printing something interesting about the program. Perhaps the most well known is *B::Deparse*, which emits a human- (and perl-) readable rendition of Perl code. That something like *B::Deparse* can even exist means that there is a great deal of information available in the opcode tree for *B::Fingerprint* to examine.

Some back ends (such as *B::Deparse*) are interested in the tiny details that make up a piece of code. Others, like *B::Showlex* (which identifies the lexical variables that a subroutine uses), are interested

Debbie teaches Perl and assembly at Monash University in Australia. She can be reached at debbiep@csse.monash.edu.au.

in only one part of the code. *B::Fingerprint*, on the other hand, needs to give a broad overview of all of the code, so that similarities between two programs will engender similar fingerprints. In this case, a fingerprint is a long string that characterizes the program.

To understand how to detect when programs come from the same source, you have to use an almost forensic technique. You have the scene of the crime (the programs) but nothing else. The rest you have to assemble yourself from the evidence. So it helps to understand what usually happens to a piece of code when someone tries to cover their tracks. *B::Fingerprint* manages to work because it completely ignores the things that a plagiarist usually changes. For instance, *B::Fingerprint* doesn't care about variable names at all; all it knows is that a scalar was used *here* in the code. Even if you change all the scalar variable names in a program to *\$fish*, the fingerprint will be unchanged.

From a technical viewpoint, *B::Fingerprint* walks the opcode tree of the program, printing a symbol for each tree node it sees. Perl opcodes come in about a dozen different kinds; for instance, there are binary operators that correspond to two-argument Perl operations like addition, and list operators that appear anywhere a sequence of operators needs to be evaluated in some order, such as in a Perl list or a sequence of Perl statements. Each opcode type that *B::Fingerprint* sees produces a different character of output in the fingerprint. Some operator types have child nodes; these are always printed as suffixes, between braces.

Here's the fingerprint for *B::Fingerprint* itself:

```
perl -MO=Fingerprint B/Fingerprint.pm
```

In RCS, these numbers constitute the delta from *A* to *B*, and are all that is actually stored in the revision control directory. For my purposes, all I care about is that it took three block moves to create a string of length 8. This ratio is a good indicator of how much of *B* came from *A*: the lower the ratio, the more similar the code is.

Probably the only salient feature you could pick out easily is the string of 24 “\$” characters, corresponding to the list of initializers for the `%opclass` variable.

Comparing Fingerprints

Creating the fingerprints of programs is only half of the problem. It's still necessary to compare two fingerprints to see how similar they are (hence, how similar the original programs are). Doing this well turns out to be surprisingly difficult.

One metric that can be used to establish how similar programs are is to take one fingerprint, and find out how many changes need to be made to it to arrive at the other program's fingerprint. This isn't always symmetrical (so, for instance, program *A* can be 80 percent the same as program *B*, but *B* may be only 65 percent the same as *A*), but it's capable of ranking similar pairs of fingerprints above dissimilar ones.

Ideally, the comparison algorithm should be able to distinguish small changes from large changes. However, “small” and “large” don’t necessarily relate to lines of code affected. For instance, changing the order of subroutines in a file is a trivial modification, even though several hundred lines may have been relocated. Wrapping an *if* condition around a block is a more significant change to a program, though it may result in only a small change to its fingerprint.

The algorithm I settled on is Walter Tichy's string-to-string block-move algorithm, used in his RCS source-code revision con-

The program *compare* (See Listing 1) accepts a number of file names as arguments and constructs fingerprints for each of them

B::Fingerprint and *compare* are available for download at <http://www.csse.monash.edu.au/~debbiep/perl/compare/>.

References

L. Allison, “Suffix Trees,” <http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/Suffix/> (contains an explanation of Ukkonen’s algorithm and pseudocode, which I copied with permission).

D. Lopresti and A. Tomkins, “Block Edit Models for Approximate String Matching,” *Theoretical Computer Science* (1997), vol. 181, no. 1, pages 159–179.

Moss (Measure of Software Similarity): <http://www.cs.berkeley.edu/~aiken/moss.html>

W.F. Tichy, “The String-to-String Correction Problem with Block Moves,” *ACM Transactions on Computer Systems* (1984), vol. 2, no. 4, pages 309–321.

W.F. Tichy, “RCS: A System for Version Control,” *Software—Practice and Experience* (1991), vol. 15, no. 7, pages 637–654.

E. Ukkonen, “On-line Construction of Suffix Trees,” *Algorithmica* (1995), vol. 14, no. 3, pages 249–260.

TPJ

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

Listing 1

```
#!/usr/bin/perl -w
```

```
#
# compare: compare N Perl programs with each other.
#
# usage:
#   compare [-n max] file ...
# where
#   max is the maximum number of pairs of similar programs to report.
#

use strict;

use Getopt::Std;

our %opts;
getopts("n:", \%opts);

# How many cases to report?
our $topcases = $opts{"n"};

# Suffix-tree-building code, adapted from
# http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/Suffix/ based on
# E. Ukkonen's linear-time suffix tree creation algorithm. Used with
# permission.
{
    my $infinity = 999999; # Just has to be longer than any string passed in.

    sub buildTree
    {
        my $fp = shift;

        # Build root state node.
        my $rootState = { };
        my $bottomState = { };
        my ($sState, $k, $i);

        for ($i = 0; $i < length $fp; $i++)
        {
            addTransition($fp, $bottomState, $i, $i, $rootState);
        }

        $rootState->{sLink} = $bottomState;
        $sState = $rootState;
        $k = 0;

        # Add each character to the suffix tree.
        for ($i = 0; $i < length $fp; $i++)
        {
            ($sState, $k) = update($rootState, $fp, $sState, $k, $i);
            ($sState, $k) = canonicalize($fp, $sState, $k, $i);
        }

        return $rootState;
    }

    sub update
    {
        my ($rootState, $fp, $sState, $k, $i) = @_;

        my ($oldRootState) = $rootState;
        my ($endPoint, $rState) = testAndSplit($fp, $sState, $k, $i-1,
        substr($fp, $i, 1));

        while (!$endPoint)
        {
            addTransition($fp, $rState, $i, $infinity, { });
        }
    }
}
```

```
    if ($oldRootState != $rootState) { $oldRootState->{sLink} = $rState;
    }

    $oldRootState = $rState;
    ($sState, $k) = canonicalize($fp, $sState->{sLink}, $k, $i-1);
    ($endPoint, $rState) = testAndSplit($fp, $sState, $k, $i-1,
    substr($fp, $i, 1));
    }

    if ($oldRootState != $rootState) { $oldRootState->{sLink} = $sState; }

    return ($sState, $k);
}

sub canonicalize
{
    my ($fp, $sState, $k, $p) = @_;

    if ($p < $k)
    {
        return ($sState, $k);
    }

    my ($k1, $p1, $sState1) = @{$sState->{substr($fp, $k, 1)}};

    while ($p1 - $k1 <= $p - $k)
    {
        $k += $p1 - $k1 + 1;
        $sState = $sState1;
        if ($k <= $p)
        {
            ($k1, $p1, $sState1) = @{$sState->{substr($fp, $k, 1)}};
        }
    }
    return ($sState, $k);
}

sub testAndSplit
{
    my ($fp, $sState, $k, $p, $t) = @_;

    if ($k <= $p)
    {
        my ($k1, $p1, $sState1) = @{$sState->{substr($fp, $k, 1)}};

        if ($t eq substr($fp, $k1 + $p - $k + 1, 1))
        {
            return (1, $sState);
        }
        else
        {
            my $rState = { };
            addTransition($fp, $sState, $k1, $k1 + $p - $k, $rState);
            addTransition($fp, $rState, $k1 + $p - $k + 1, $p1, $sState1);
            return (0, $rState);
        }
    }
    else
    {
        return (exists $sState->{$t}, $sState);
    }
}

sub addTransition
{
    my ($fp, $thisState, $left, $right, $thatState) = @_;

    $thisState->{substr($fp, $left, 1)} = [$left, $right, $thatState];
}

$| = 1;
```

```

# Perl executable.
our $perl = $^X;

# All fingerprints, keyed by filename.
our %fp;

# Suffix trees of all fingerprints, keyed by filename.
our %tree;

# Stop comparing fingerprints after this many blocks.
our $ceiling;

# Get all fingerprints.
foreach my $filename (@ARGV)
{
    # This is OK as long as characters in name of $file are safe.
    my $fingerprint = `$perl -MO=Fingerprint $filename`;
    if (! $?)
    {
        # Remember this file's fingerprint.
        $fp{$filename} = $fingerprint;
        # Insert the fingerprint into the suffix tree forest.
        $tree{$filename} = buildTree($fingerprint);
    }
}

# Now compare each pair of fingerprints.
my @result;
my $count = 0;
foreach my $file1 (keys %fp)
{
    foreach my $file2 (keys %fp)
    {
        next if $file1 eq $file2;

        my $length1 = length $fp{$file1};
        my $length2 = length $fp{$file2};

        # Progress meter.
        print int ($count++ / ((keys %fp) * (keys %fp)) * 100), "% complete\r"
        if -t STDOUT;

        # Do we have a maximum number of cases to report?
        if (defined $topcases && @result >= $topcases)
        {
            $ceiling = $result[-1]{ratio} * $length2;
        }
        else
        {
            undef $ceiling;
        }

        # Compare the files.
        my $blocks = compare($file1, $file2);

        push @result, (
            file1 => $file1,
            length1 => $length1,
            file2 => $file2,
            length2 => $length2,
            blocks => $blocks,
            ratio => $blocks / $length2,
        );

        # Ripple down new element in @result to keep it sorted.
        # If keeping only the top N cases, this is quicker than
        # sorting afterwards.
        if (defined $topcases)
        {
            my $pos;
            my $new = $result[-1];
            # Insertion sort algorithm.
            for ($pos = @result - 2; $pos >= 0; $pos--)
            {
                if ($new->{ratio} < $result[$pos]->{ratio})
                {
                    # Ripple up an element.
                    $result[$pos+1] = $result[$pos];
                }
                else
                {
                    # Found the right place.
                    last;
                }
            }
            # Insert the new item at its place.
            $result[$pos+1] = $new;

            # Lose the (now) last element?
            if (@result > $topcases) { pop @result; }
        }
    }
}

```

```

    }
}

# If collecting all cases, sort so that more similar code is near start
# of list.
if (!defined $topcases)
{
    @result = sort {$a->{ratio} <=> $b->{ratio}} @result;
}

# Present results.
foreach my $result (@result)
{
    print
        $result->{ratio}, " ", $result->{blocks},
        "\n", $result->{length2}, ": ",
        $result->{file1}, " => ", $result->{file2},
        "\n";
}

sub compare
{
    my ($file1, $file2) = @_;

    # We're trying to reconstruct fingerprint $fp2 from $fp1, so need
    # suffix tree from $fp1.
    my $tree1 = $tree{$file1};
    my $fp1 = $fp{$file1};
    my $fp2 = $fp{$file2};

    # Number of blocks counted so far.
    my $blocks = 0;

    my $pos2 = 0;
    # Keep going while there's any of fingerprint 2 to do.
    BLOCK: while ($pos2 < length $fp2)
    {
        # Find a path through $tree1 that matches the part of $fp2
        # we're up to.
        if (!exists $tree1->{substr($fp2, $pos2, 1)})
        {
            # This character doesn't exist at all in $tree1. Next block.
            $pos2++;
            next BLOCK;
        }

        # There's an entry in the suffix tree.
        for (my $state = $tree1->{substr($fp2, $pos2, 1)}; # Start at root.
            defined $state; # Stop if finished a leaf node.
            $state = $state->[2]{substr($fp2, $pos2, 1)}) # Next node.
        {
            # Walk through characters in this state, comparing with $fp2.
            for (my $count = 0; $count <= $state->[1] - $state->[0]; $count++)
            {
                # Are there any more characters, and if so, do they match?
                if ($state->[0] + $count < length $fp1 &&
                    $pos2 < length $fp2 &&
                    substr($fp1, $state->[0] + $count, 1)
                    eq substr($fp2, $pos2, 1))
                {
                    # Got a match, move on to the next character.
                    $pos2++;
                }
                else
                {
                    # Characters don't match; this is the end of a block.
                    next BLOCK;
                }
            }

            # Finished this state, and it all matched. Go do the next one.
        }
    }
    continue
    {
        # Count the blocks as we go.
        $blocks++;
        if (defined $ceiling && $blocks > $ceiling)
        {
            # Exceeded the ceiling, return.
            last;
        }
    }

    return $blocks;
}

```

TPJ

Grokking Web Archives

I am in the middle of the desert in Iraq, but even here, in the modern army, I get to use the Internet sometimes; and with portable diesel generators, I have power to run my laptop. I wonder what this used to be like before Microsoft Word, upon which our military operations seem utterly dependent. Even with all of our fancy equipment, surfing the Web is a challenge both in the limited bandwidth and available time.

When I get a chance to get on the Internet, I have to use a government-approved computer, which means I have to use whatever they happen to have and it always is some form of Windows. I plan my computer time carefully and work my way through my list as quickly as I can so I can do as much as possible in my limited time—there is always a line for the computers. I do not stop to read long web pages, for instance. I save them to a floppy disk, then read them later. If I didn't, I would use up all of my time just trying to get through the first page of Damien Conway's Exegesis 6.

Since all of these computers, no matter which part of Iraq I may be in, run Windows, I have to use Internet Explorer, which allows me to save web pages as Text, HTML, Complete Web Page, or a Web Archive. Unfamiliar with these choices, I saved a bunch of pages as Web Archives. I expected it to create a bunch of files just like Mozilla does, but instead, I got a single file. This turned out to be a good thing since I had an easier time copying one file than a file and a directory, so I did not have to think about it much. However, Internet Explorer on my Powerbook did not know what to do with it, although it thought about it for a long time before giving up.

Perl to the rescue! I looked at the file in BBEdit—I initially reached for `HTTP::Response` to handle this, but its interface does not have anything for breaking apart a multipart message. I could use it to munge the web archive, then extract the HTTP message body so I could split it up to get the various parts, then create `HTTP::Response` objects out of that, but I did not want to do that much work. In this case, a module is more cost than benefit.

I need to do three things to be able to read the web archives. First, I need to extract each part. Next, I need to decode the data to put them in their natural states; and once I have all of the pieces, I need to ensure that the browser finds them all. That last part is the trickiest—I need to make sure the original HTML text tells

my browser to look for image files locally instead of at their original location.

To do that, I need to know a little about HTTP messages. They come in two parts—a header and a message body—and those are separated by a blank line. In this case, since it comes from Windows, a blank line is a line containing only the string `\x0D\x0A`, the literal bit pattern for a carriage-return line feed. I cannot rely on the `\r` and `\n` representations because the Mac has an odd notion about what those are—the perlport manual page explains it. As long as I use the literal representation, I am okay. Two of these in a row (one for the previous line end and one for the blank line show me where the header ends).

I also know that a multipart HTTP message has a boundary string. This string signals the start of a new part, and the particular string shows up in the main header as part of the Content-Type header:

```
From: <Saved by Microsoft Internet Explorer 5>
Subject: Example Web Page
Date: Mon, 27 Oct 2003 11:58:12 +0300
Content-Type: multipart/related;
    boundary="---=_NextPart_000_0000_01C39C81.9921A8A0";
    type="text/html"
```

The header for the entire web archive gives me the boundary string `---=_NextPart_000_0000_01C39C81.9921A8A0`, which looks complicated but could be simpler. Perl will not care though, so it does not matter.

After the header comes the blank line, then the message body. The message body contains all of the parts of the web archive and each part has the boundary string in front of it. The data after the boundary string is another HTTP message, so I have to extract the data the same way I just did for the web archive message body:

```
---=_NextPart_000_0000_01C39C81.9921A8A0
Content-Type: text/html;
    charset="Windows-1252"
Content-Transfer-Encoding: quoted-printable
Content-Location: http://www.example.com/index.html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML><HEAD><TITLE>Example Web Page</TITLE>
<META http-equiv=3DContent-Type content=3D"text/html"; =

---=_NextPart_000_0000_01C39C81.9921A8A0
Content-Type: image/gif
Content-Transfer-Encoding: base64
```

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.

I can see the parts I want and I can see how I need to process them. The HTML text has special characters turned into their hexadecimal representations preceded by an equal sign. For instance, the equal sign itself becomes the literal sequence `=3D`, and HTML text has a lot of equal signs. Additionally, each line ends with a bare equal sign. Somewhere, this format must have solved a problem (or, at least, I hope it did). The images are much easier to process since I just base64-decode them.

As I took all of this research and turned it into a program, I went through two major designs: Just dump all of the files into the current directory, then, to mirror the “Web Page Complete,” I create a directory for the image and supporting files. The first program was too sloppy—it created a lot of files in the current directory—so in this article, I only show the second program (see Listing 1).

Lines 1–2 start my program in the usual manner, with warnings on and the *strict* pragma in effect so I do not get sloppy.

On line 4 I pull in the *MIME::Base64* module since I will use its *decode_base64()* function to process the image data. That is as complicated as that will get.

On line 6, I start a *do {}* block to grab file data. The *do {}* block creates a scope so I can temporarily affect special variables and limit the lifetime of some temporary variables. I set *\$/*, the input record separator (the fancy way of saying line ending) to *undef* so I can slurp in the file in one shot. I open the file I specify in the first command-line argument, *\$ARGV[0]*, and make sure I was able to do that, *die()*-ing otherwise. Finally, I read a line—the entire file because *\$/* is *undef*—and since that is the last evaluated statement, it becomes the return value of the *do {}* block, which I assign to *\$archive*.

On line 12, I split *\$archive*, the web archive HTTP message, into its header and message body and assign those to *\$header* and *\$multipart*, respectively. My *split()* regular expression uses the literal values of the line endings to avoid portability problems between Windows and my Mac, which are pernicious in this case. I also use *split()*’s optional third argument to specify that I want to end up only with two parts. I know other blank lines exist in the data, and I do not want to *split()* on those because I cannot ensure that they come after an HTTP header. They could be in the middle of HTML text, for instance.

On line 14, I extract the boundary string from the header with a simple regular expression. A match in list context returns, as a list, the parts of the regular expression I remembered in parentheses. I create the list context by putting the assignment list in parentheses. Once I have that string in *\$boundary*, I use it in another *split()*, on line 16, to get each part of the web archive. Each part becomes an element of the array *@parts*. In that *split()* regular expression, I start with the *\Q* sequence to quote any special characters in the boundary string so Perl does not interpret them as regular expression metacharacters. The full stop, “.”, shows up in the boundary string, for instance.

I have to play with *@parts* before I go on. The first element is going to be the overall web archive header since that comes before the first boundary string, and the overall message body ends with the boundary string, so *split()* creates an empty element at the end of the array. I *shift()* off the first element and *pop()* off the last so *@parts* only contains the elements I need to recreate the web page.

On line 19, I call my own *parse* routine on the first element of *@parts*, which I *shift()* off the array. I expect that to be the HTML text of the page. The *parse()* routine starts on line 45 and is really just a wrapper for other routines. Its argument is the HTTP message for the part I am processing. First, *parse()* calls *÷* to split the message into the header and message body. Since I use

the ampersand in front of *÷*, the *divide()* routine can see the argument list, *@_*, of *parse()*. I do not have to pass any arguments to *divide()* because Perl does it for me. Read more about this in the *perlsb* manual page.

Once *divide()* returns the header and the message body, I use my *location()* routine, on line 47, to extract the file name from the Content-Location header, and I use the *encoding()* routine, on line 48, to get the value of the Content-Transfer-Encoding header. I return those two items and the body for a total of three items as the return values for *parse()*. On line 19, I save those to their own variables.

I want to save the image files in a directory that has a similar name as the HTML text file, so I extract the file name portion

*When I get a chance to
get on the Internet, I have
to use a government-approved
computer, which means I have
to use whatever they happen
to have and it always is some
form of Windows*

(sans extension, that is) from *\$location* and assign that to *\$directory*. Again, a match in list context returns a list of the things remembered in parentheses. My intended directory name ends up in *\$directory* and, in line 21, I give it a default value if, for some reason, the match failed. On line 22 I create the directory. If that fails, I catch it later, on line 37, when I try to write a file in that directory.

On line 24, I use my own *unquote()* routine on line 50 to undo the quoted-printable encoding of the HTML text. In the substitution on line 53, I use the substitution’s *e* flag and a little bit of Perl code to determine what I want to use as replacement text. I take the hexadecimal representation, *3D*, for instance, and turn that into the correct character. I use Perl’s *hex()* function to ensure Perl turns the string *3D* into the correct number, then Perl’s *chr()* function to take that number and return the ASCII character.

On line 25, I munge the HTML *IMG* tags to ensure the URLs point to the directory where I will store all of the images. I use the images’ original file names but munge their paths.

On line 27, once I am done processing the HTML text, I save it to a file.

The rest of the script processes the remaining parts. For my purposes, those parts are images that are base64-encoded, but you may run into other things and have to add a bit of programming.

On line 31, I start a *foreach* loop that will go through each of the remaining parts. First, as I did before, I use the *parse()* routine to get the information about the part. On line 35, I use the *decode_base64()* function from *MIME::Base64* to turn the body into its original data if the encoding is Base64. On line 37, I open a file in the directory that I created earlier and save the file there.

Now, when I look in the current working directory, I should have an HTML text file and a directory with the supporting files. When I view the HTML file in my browser, the page looks just like it did when I was on the network.

101 Perl Articles!



From the pages of
The Perl Journal,
Dr. Dobbs'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 look-
ing for. Delivered as HTML files in a
ZIP archive or CD-ROM image, down-
load *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!

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

Listing 1

```
1  #!/usr/bin/perl -w
2  use strict;
3
4  use MIME::Base64;
5
6  my $archive = do {
7      local $/;
8      open my $fh, $ARGV[0] or die "Could not open $ARGV[0]: $!\n";
9      <$fh>;
10 };
11
12 my( $header, $multipart ) = split /(?:\x0D\x0A){2}/, $archive, 2;
13
14 my( $boundary ) = $header =~ m/boundary="(.*?)"/;
15
16 my @parts = split /\Q$boundary/, $multipart;
17 shift @parts; pop @parts;
18
19 my( $location, $encoding, $body ) = parse( shift @parts );
20 my( $directory ) = $location =~ m/(.*)\./;
21 $directory = "$location.d" unless $directory;
22 mkdir $directory;
23
24 unquote( $body ) if $encoding eq 'quoted-printable';
25 $body =~ s/<img(.*)src="(.*?)"/<img$1src="$directory/$2"|isg;
26
27 open my $fh, "> $location" or die "Could not open $location: $!\n";
28 print $fh $body;
29 close $fh;
30
31 foreach my $part ( @parts )
32 {
33     my( $location, $encoding, $body ) = parse( $part );
34
35     $body = decode_base64( $body ) if $encoding eq 'base64';
36
37     open my $fh, "> $directory/$location"
38         or die "Could not open $location: $!\n";
39     print $fh $body;
40 }
41
42
43 sub divide { ( split /(?:\x0D\x0A){2}/, $_[0], 2 ) }
44
45 sub parse { my( $h, $b ) = &divide; ( location( $h ), encoding( $h ), $b ) }
46
47 sub location { ( $_[0] =~ m|Content-Location:.*?(\\S+)|i ) }
48 sub encoding { ( $_[0] =~ m|Content-Transfer-Encoding:(s*(\\S+)|i ) ) }
49
50 sub unquote
51 {
52     $_[0] =~ s/=\\x0D\\x0A//g;
53     $_[0] =~ s<=([0-9A-F]{2})>{
54         chr hex $1
55     }ge;
56 }
```

TPJ



Bayesian Analysis For RSS Reading

Simon Cozens

I am very good at distracting myself from getting things done. One of my best displacement activities used to be flicking through various web news sites: the BBC news, Slashdot, use.perl.org, linuxtoday.com, and that's before we get started on the comics.

Occasionally, I have spurts of enthusiasm and try to work out ways to curb my distractions in order to free up time for more interesting forms of procrastination. Instead of having to cycle through all these pages to see if there was any new news or a new edition of the comic, wouldn't it be a lot more efficient if these news sites told me when there was something new for me to read? Then I could spend the equivalent time that I'd save playing chess or something...

Thankfully, other people have obviously had the same question, as one of the big trends in online publishing over the past two or three years has been the surge in syndication of web news sites via Remote Site Syndication(RSS) and RDF. RSS is an XML format that gives the headlines and some of the content of stories on a site in a machine-readable way. This can be read by an RSS aggregator like NetNewsWire (<http://ranchero.com/netnewswire/>), which periodically grabs the RSS feeds of all the sites you visit often. By parsing the feeds, it can work out if there are any articles you haven't seen yet and give you a summary of the new news.

This was a good start, but it wasn't enough. So, we'll first have a look at how we can manipulate RSS in Perl, and then we can find out how I managed to cut down my procrastination time even more!

XML::RSS

The first handy tool to deal with RSS is the obviously named `XML::RSS` module. It can be used both to create new RSS files

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 pumpking. Simon can be reached at simon-cozens.org.

and to parse existing feeds and turn them into Perl data structures. It's this latter use that I'll discuss here.

For instance, to get a list of the current news stories on the Perl news site use.perl.org, we start by downloading its RSS feed:

```
use LWP::Simple;
my $xml = get("http://use.perl.org/perl-news-short.rdf");
```

Then we turn this into an `XML::RSS` object:

```
use XML::RSS;
my $rss = XML::RSS->new;
$rss->parse($xml);
```

And now we have a load of information about the feed. For instance, the title and link:

```
print $rss->{title}, ": ", $rss->{link}, "\n";
```

An RSS feed contains a list of items, which we can retrieve. Each item returns a hash reference:

```
for my $item (@{$rss->{items}}) {
    print $item->{title}, ": ", $item->{link}, "\n";
}
```

And sometimes, the content providers put a short description or the first few paragraphs of the news item in the description tag of an item:

```
for my $item (@{$rss->{items}}) {
    print $item->{title}, ": ", $item->{link}, "\n";
    print $item->{description}, "\n\n";
}
```

However, in order to create a nice news aggregator, we still have to do a lot of work ourselves: downloading the feeds, parsing them, working out which articles have already been seen,

and so on. This isn't the Perl way. Hasn't someone else done this for us?

Timesink

Regular readers will know that I find it hard to get through a column without mentioning at least one of the two most interesting modules in the Perl world for me: *Class::DBI* and the Template Toolkit; well, this time, I get to mention both.

Richard Clamp also got the RSS bug around the same time as me, and rather than be dependent on an application running on

*Instead of having to trudge
through a small number of web
pages in search of new news, I
now have to trudge through a
large number of news feeds in
search of interesting news*

one particular machine, he created a web-based aggregator that could keep track of what he had and hadn't read. His project, Timesink, uses *XML::RSS* to download and parse the feeds, *Class::DBI* to store the stories in a database, and the Template Toolkit to display the rendered stories to the user. It supports multiple users subscribed to a variety of different feeds, and is remarkably small and well-engineered.

On the other hand, Timesink is still very much in development and isn't very well documented yet, but I managed to get it set up and running on one of my computers. It provided me with the platform I needed to get the next stage of my procrastination optimization under way.

You see, the problem is that with the advent of RSS, it's very easy for me to subscribe to a large number of news sources, many of which will only be of marginal interest to me but occasionally throw up something worthwhile. Instead of having to trudge through a small number of web pages in search of new news, I now have to trudge through a large number of news feeds in search of interesting news.

I first got the computer to decide for me what was new; my thought was that the obvious way to proceed was to get it to decide what's interesting. My dream aggregator would read my news for me, and only present me with the news I want to read.

Based on Bayes

How can a computer tell me what's interesting? By watching what I tell it is interesting, working out the features of an "interesting" article, and then weighing up whether a given news article would be best categorized as "interesting" or "boring." This turns the question into a machine learning problem, and machine learning has been worked on by far greater minds than mine.

There are a number of text categorization modules on CPAN, many of them better than the one I ended up using. A technique called Bayesian analysis has been in vogue for text categorization in the past year or two, particularly for spam filtering, but to be honest, it's not really that good. There's a module called *AI::Categorizer* that implements a whole bunch of better approaches.

On the other hand, one of the advantages of Bayesian analysis being in vogue is that it has been implemented a lot, and this means that it's likely that one of its implementations has a very simple interface. *Algorithm::NaiveBayes* was the module I picked in the end, and it does indeed have an easy-to-use interface.

The analyzer in *Algorithm::NaiveBayes* takes a number of "attributes" and corresponding "weights," and associates them with a set of "labels." Typically, for a text document, the attributes are the words of the document, and the weight is the number of times each word occurs. The labels are the categories under which we want to file the document.

For instance, suppose we have the news article:

```
The YAPC::Taipei website is now open and calls for registration.  
Welcome to Taipei and join the party with us!
```

We would split that up into words, count the occurrence of the relevant words, and produce a hash like this:

```
my $news = (  
    now => 1, taipei => 2, join => 1, party => 1, us => 1,  
    registration => 1, website => 1, welcome => 1, open => 1,  
    calls => 1, yapc => 1  
);
```

Then, we can add this article to our categorizer's set of known documents:

```
$categoriser->add_instance( attributes => $news,  
                           labels    => [qw( perl conference  
                                             interesting )] );
```

When we've added a load of documents, the categorizer has a good idea of what kind of words make up Perl stories, what kind of words suggest that the document is about current affairs, what kind of stories I would categorize as "interesting," and so on. We then ask it to do some sums, and can find out what it thinks about a new document:

```
$categoriser->train;  
my $probs = $categoriser->predict( $new_news );
```

This returns a hash mapping each category to the probability that the document fits into that category. So, once we have our categorizer trained with a few hand-categorized documents, we can say:

```
if ( $categoriser->predict( $new_news )->{interesting} > 0.5 ) {  
    # Probably interesting  
}
```

Hooray!

Of course, you may be asking how we turned those documents into hashes of words anyway, and what I meant when I said we only look at the "relevant" words.

Well, turning a string into a hash isn't too difficult:

```
my %hash;  
$hash{$_}++ for split /\s+/, $string;
```

This doesn't do too badly, but it does have some problems. It copes badly with punctuation, for starters, it passes on things that are not words, like numbers, and it treats differently cased words ("the" versus "The") separately. We can try refining it a little:

```
$hash{$_}++ for map lc, split /\W+/, $string;
```

Or, we could try a fundamentally better approach; the *Lingua::EN::Splitter* module was designed for precisely this kind of task and knows to normalize case, be careful of punctuation, and so on. We can ask its *words* function to return just the words in a string and leave the implementation details to someone else:

```
$hash++ for @{words($string)};
```

That's neater; however, it still has two problems. First, there are a lot of words in English text that are pretty much redundant. They don't carry any semantic content, they're just there to move

One of the advantages of Bayesian analysis being in vogue is that it has been implemented a lot, and this means that it's likely that one of its implementations has a very simple interface

the story along, as it were—words like “as,” “it,” and “were.” These words, called “stopwords,” don't help us know whether or not something is about Perl or about Java, whether a text is interesting or not, so we should get rid of them.

Of course, we don't want to collect a list of such stopwords ourselves. That's a job for the author of the *Lingua::EN::StopWords* module, which exports the *%StopWords* hash. This allows us to grep out the stopwords from our list, like so:

```
$hash++ for grep (!$StopWords{$_}) @{words($string)};
```

The second problem is that, for an *XML::RSS* item, we should give more weight to words in the title of a news item than in the description because the title is supposed to be a concise summary of what the piece is about. Hence, we come up with two functions like this:

```
sub invert_item {
    my $item = shift;
    my $hash;
    invert_string($item->{title}, 2, \%hash);
    invert_string($item->{description}, 1, \%hash);
    return \%hash;
}

sub invert_string {
    my ($string, $weight, $hash) = @_;
    $hash->{$_} += $weight for
        grep { !$StopWords{$_} }
            @{words($string)};
}
```

This gives title words twice as much weight as the words in the body and returns a hash we can feed to the categorizer. We're only interested in two categories, “interesting” and “boring,” so that makes it easier for us.

Now, we have enough parts of the equation to put together our automatic newsreading tool: A way to mark a document as interesting or boring, and a way to predict whether a new document is likely to be interesting.

Automatic Newsreading

How do we fit this into Timesink? The first modification is to provide the user the interface for categorizing a story manually. User interface stuff lives in the templates, so we'll begin by editing the templates.

After the description of the story, we add two links that the user can use to categorize the story, and we add a display showing the percentage of “interest,” using a method we'll fill in soon:

```
[% IF item.published %]
    <small>Published: [% date.format(item.published) %]</small>
[% END %]
+ [% IF user %]
+   <p align="right">
+     [ <a href="?feed=[%feed.id %]&item=[% item.id %]&class=interesting">
+       Interesting? </a> ]
+     [ <a href="?feed=[%feed.id %]&item=[% item.id %]&class=boring">
+       Boring? </a> ]
+   ]
+   ( [% user.interest(item) * 100 | format("%.1d") %] %)
+ </p>
+ [% END %]
```

Second, if these links are clicked, we need to do something—hand off the ID of the story and the ID of the user to a routine that adds a new document to an *Algorithm::NaiveBayes* categorizer. Thankfully, the *Template::Plugin::CGI* helper makes it really easy for us to get the data back out of the query string:

```
IF user;
    feeds = user.feeds;
+   IF cgi.param('class');
+     SET dumm = user.mark_item(cgi.param('item'), cgi.param('class'));
+   END;
ELSE;
```

Remember that the class is going to be interesting or boring depending on which link the user clicked. Finally, from the template point of view, we need to change the order in which the stories are displayed so that the most interesting stories float to the top:

```
<h2><a href="/>Timesink:</a> <a href="[% feed.link %]" target="_new">[%
feed.name %]</a></h2>
-[% FOREACH item = feed.items.nsort('seq') %]
+ [%
+   IF user && feed;
+     SET items = user.sort_interest(feed.items);
+   ELSE;
+     SET items = feed.items.nsort('seq');
+   END;
+   FOREACH item = items %]
```

Now we have a few methods that we need to write: the *mark_item* method, which calls *add_element* given a user and a feed ID; *interest*, which trains the categorizer and computes the probability that an article is interesting; and the *sort_interest* function, which sorts the articles (unseen articles by interest, then seen articles by interest).

These methods have to be on the *user* object (*Timesink::DBI::Subscriber*) because what's considered interesting depends on who's reading the page. However, we immediately have a problem.

Where do we acquire the *Algorithm::NaiveBayes* categorizer that we're going to use? This somehow has to be persistent because we want the user to be able to go away and come back to the web site tomorrow and still have the system know what makes an interesting article, and it needs to work on a per-user basis.

We also need to save away the categorizer after we add a new document to the corpus, and restore it again when we need to make a decision about a new piece of news. The *Algorithm::NaiveBayes* object is very simple, and we can use the core *Storable* module to serialize and reload it:

```
my %bayes_cache;

sub _bayes_file { $config->bayes_toks."/".$shift->id.".nb"; }

sub get_bayes {
    my ($self) = @_;
    my $user = $self->id;
    return $bayes_cache{$user} if $bayes_cache{$user};

    my $nb;
    eval { $nb = retrieve($self->_bayes_file) } if -e $self->_bayes_file;
    $nb ||= $bayes_cache{$user} = Algorithm::NaiveBayes->new(purge => 0);
}
```

_bayes_file turns the user's ID into a file name and *get_bayes* uses *Storable::retrieve* to get the object from that file, unless we already have an up-to-date analyzer hanging around in the *%bayes_cache* hash.

Similarly, when we're finished with the object, we use *Storable::store* to put the analyzer back to file:

```
sub done_with_bayes {
    my $self = shift;
    my $nb = shift;
    store($nb, $self->_bayes_file);
}
```

Now, filling in the missing methods becomes nice and simple. For instance, marking an item as interesting or uninteresting is:

```
sub mark_item {
    my ($self, $item_id, $class) = @_;
    return unless my $item = Timesink::DBI::Item->retrieve($item_id);
    my $nb = $self->get_bayes;
    $nb->add_instance( attributes => $self->invert_item($item),
                      label      => $class );
    $self->done_with_bayes($nb);
}
```

While interest is a thin wrapper around the routine we saw earlier:

```
sub interest {
    my ($self, $item) = @_;
    eval { $self->get_bayes->predict(
        attributes => $self->invert_item($item)
    )-> {interesting};
    };
}
```

Finally, to sort the items efficiently, we use the “Orcish Maneuver” to cache the things we're looking up:

```
sub sort_interest {
    my ($self, $items) = @_;
    my @items = @$items;
    my $nb = $self->get_bayes;
    $nb->train;
```

```
my (%seen, %interest);
my @sorted = sort {
    ($seen{$a} ||= $self->is_seen($a)) <=>
    ($seen{$b} ||= $self->is_seen($b))
    ||
    ($interest{$b} ||= $self->interest($b)) <=>
    ($interest{$a} ||= $self->interest($a))
} @items;
$self->done_with_bayes($nb);
return \@sorted;
}
```

(Note: The “Orcish Maneuver” is a term coined by Joseph Hall and Randal Schwartz in their book *Effective Perl Programming* [Addison-Wesley, 1997, ISBN 0201419750] and is so named as a pun on the phrase “or cache.”)

The complicated *sort* block will first find articles that haven't been seen before, then will order articles by interest from highest to lowest.

As you click to signify that you find a piece interesting, it floats to the top while less interesting articles sink to the bottom. Over time, my automatic newsreader learns what I like to read and gets better at suggesting stories for me, saving me even more time and allowing me to find new and interesting ways to avoid getting on with writing *The Perl Journal* articles...

[Ed: We'd like to point out that since Simon's procrastination has now become highly efficient, we'll be asking him for his column one week early next month.]

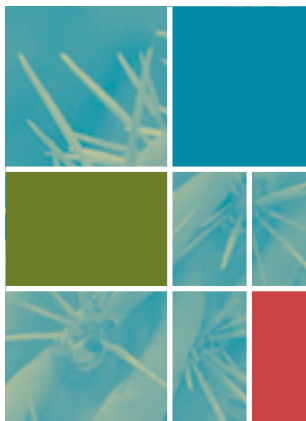
TPJ

**Fame & Fortune
Await You!**

**Become a
TPJ
author!**

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

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



A Better *Data::Dumper*

Randal Schwartz

A few years ago, I stared quite heavily at the source to the core module *Data::Dumper*—enough to make my eyes hurt. I was trying to “reverse engineer” the output, so that I could write an un-dumper that would evaluate the resulting string of Perl code and get the original values back, without unleashing the full Perl expression evaluator. I succeeded in that, although the results were far too slow to be useful in a practical sense.

However, while I was staring at *Data::Dumper*’s guts, I noticed that there seemed to be no provision for noticing that a scalar reference was a reference to a scalar that existed as the value of another array or hash element and, thus, it dumped those values incorrectly. For example:

```
use Data::Dumper;

$Data::Dumper::Purity = 1; # try your hardest

my @values = qw(zero one two three);
my $ref_to_element = \$values[1];
my $all = [$ref_to_element, \@values];
print Dumper($all);
```

which results in:

```
$VAR1 = [
  \one',
  [
    'zero',
    ${$VAR1->[0]},
    'two',
    'three'
  ]
];
```

The problem is that *\$VAR1->[0]* is a reference to one copy of “one,” while *\$VAR->[1]->[1]* is a different copy of “one,” so changing one won’t change the other. The link between the two elements has been severed.

Randal is a coauthor of Programming Perl, Learning Perl, Learning Perl for Win32 Systems, and Effective Perl Programming, as well as a founding board member of the Perl Mongers (perl.org). Randal can be reached at merlyn@stonehenge.com.

I immediately reported the bug to the Perl developers, but this three-year-old bug has not yet been fixed. Rather than simply raise the issue again, I decided it was time to whip out the coding palette and provide some reference code that can do references correctly, especially since the problem also seems to exist in the YAML library and the Perl debugger’s *x* function as well. (Only *Storable* seemed to do the right thing—good for them.)

And I’ll have to say it was a fun exercise, which I bring to you as Listing 1. Because the listing is rather long, I’ll focus on some of the key points rather than my usual rambling style.

The goal is simple: Write an *uneval* routine, such that the sequence of:

```
use Data::Stringer qw(uneval);
my $string = uneval(@some_list);
my @new_list = eval $string;
```

results in *@new_list* being a deep copy of *@some_list*, even if the list contains scalars, references to arrays, references to hashes, and blessed references of those. And, of course, references to the thing must not result in the thing being copied, but being referenced instead. For example, the above data gets dumped as:

```
use Data::Stringer;
my @values = qw(zero one two three);
my $ref_to_element = \$values[1];
my $all = [$ref_to_element, \@values];
print uneval $all;
```

which results in the string of:

```
my (@X806f84, @X810114, @X8133a4);
@X806f84 = ('zero', 'one', 'two', 'three');
@X810114 = (00, \@X806f84);
@X8133a4 = (\@X810114);
$X810114[0] = \@X806f84[1];
@X8133a4;
```

Although this string isn’t quite as pretty as the *Data::Dumper* version, it’s more accurate. Notice the next-to-last line, which forces the first element of the result array to be a reference to the

second element of the nested array. That's the crucial piece missing in the *Data::Dumper* version.

The dumping strategy is rather simple minded and broken into two main passes. On the first pass, we walk the supplied list of values, recursively, creating a symbol table *%stab*, declared in line 17. This is accomplished with a queue of values to be processed in line 31. The *%stab* hash will end up being populated with three kinds of entries. Scalars have a key of *\$X* followed by a hex address of the actual symbol table address (as returned by stringifying a reference to that item). Similarly, arrays and hashes have *@X* and *%X* followed by the hex address, respectively.

Both arrays and hashes hold the reference to the value as the value in the *%stab* hash. The scalars are a bit different: Their value is a one- or three-element array ref. The first element is a reference to the original scalar value. The second and third elements are populated when we find a scalar with that address as a value within an array or hash that we're scanning. The second element is a name like the keys of *%stab* (and should map to an entry when pass 1 is complete), and the third element is the array index or hash key. This is the missing piece in *Data::Dumper* and friends: the record of where a scalar might live if not as a separate symbol table location.

The recursion comes about from the core of *pass_1_item*, defined in lines 35–68. Each item to be dumped is a reference to a scalar, array, or hash. Line 38 constructs the appropriate *%stab* key using the *ref_to_label* routine. This routine is defined down in lines 202–218 and uses *overload::StrVal* to ensure that we can extract an unoverloaded string value for the reference even if the class has a stringification overload method. *\$id* is the hex address, usually beginning in 0x. Line 208 converts this string into a suitable identifier component. Lines 209–217 sort out the core type (not considering whether or not the reference is blessed), and return back a variable name of the appropriate type to hold the value.

Back up in *pass_1_item*, we check this string again (line 41) to see its native type. If it's a scalar, line 42 stores the value (possibly autovivifying an array ref: Thank you Perl!), and moves on. If the scalar value is also a reference, then we need to dump the referenced scalar, so the reference is pushed onto the working *@queue* (line 43).

For an array or hash, things get a bit more complicated because we must keep track of any elements in case they are referenced from somewhere else. The code is similar. First, store the reference into *%stab* (lines 45 or 54), then walk through the values (beginning in lines 46 and 55). For each element, we take its address and create a *%stab* entry, noting the containing data structure and key or index used to access the value (lines 49 and 58). And, if the element is a reference to somewhere, we also add it to the work queue (lines 50 and 59).

Speaking of the work queue, we have to allow for the possibility of mutually recursive and self-recursive data structures:

```
my @one = qw(won one);
my @two = qw(two too to);
push @one, \@two;
push @two, \@one;
my $string = uneval(@one);
```

As we're scanning *@one*, we'll need to follow the reference to *@two* at the end. But when we get to the end of *@two*, we don't want to scan over *@one* again. Line 39 handles the duplicate scanning rejection by simply refusing to scan any particular scalar, array, or hash more than once.

The first item dumped is the input parameters. Because the input parameters need to be dumped as the output, we retain the *%stab* key in line 30 being returned from the first invocation of

pass_1_item at line 66. This particular array name will be the designated output array as well.

Once pass 1 is complete, every scalar, array, and hash that belongs to the dumped set has been identified and copied to the virtual symbol table. To dump the data, we merely need to walk this virtual symbol table. The *pass_2_routine* (lines 74–85) manages the process. The steps can be seen as: declaring the variables, initializing those variables (except for deferred entries), handling the deferred items, blessing any blessed references, and then evaluating the designated top-level array as a result.

First, the declarations are dumped, using *pass_2_declarations* defined in lines 118–126. A single *my* construct encloses all scalar, array, and hash names, except for those scalars that exist as elements of another array or hash.

Then the bulk of the work comes out of the initialization phase, starting with *pass_2_initializations* defined in lines 128–133. Key-value pairs from *%stab* are passed in to *pass_2_initialization*, which is defined starting in line 142. If it's a scalar (line 146), it's a simple assignment, unless the value was an element of a larger data structure, in which case, it simply disappears.

The value for any scalar (variable or element) might be a reference to an element of an array or hash, however, and this is where *pass_2_value* comes in to help. Looking back to the definition (starting in line 87), we see that references to scalars are handled specially. If the reference to a scalar is an element of an array or hash already seen, then line 95 will have a three-element list, setting *\$place* and *\$index* to the actual scalar's location. In that case, we can't provide a scalar value for this initialization. Instead, we add a *@deferred* element, which does the initialization after all other initializations are complete, and return a 00 value instead. This double-0 value is just a 0, but gives an indication to me staring at the output that this value will be replaced during the deferred stage, just as we saw in the example earlier.

Array and hash initialization works similarly in lines 152–165, except that we have to keep track of which element we are looking at in case the deferred initialization needs to reference an element of a larger structure (as in the previous example).

Once the core initializations are complete, we go back to dump out the deferred initializations (if any). This patches up all the values that were dumped as "00" during the initialization subpass, to point at the elements of arrays and elements of hashes as needed. Then, it's time for a blessing or two, perhaps. Lines 135–140 call *pass_2_blessing* for each *%stab* entry, defined in lines 173–188. If it's a scalar, we need to get the actual element out of the array ref, noting its location for the proper blessing if it's also an array or hash element.

Lines 182–187 determine the proper blessed class, getting around any issues with an overloaded stringification once again. And if the value is blessed, the proper blessing is generated in line 184.

The only thing left to describe is *quote_scalar*, which generates a nice printable representation of a scalar. The *undef* value is a simple *undef* return. Otherwise, if the value is safe as a number, the number form is preferred. Otherwise, a single-quoted string is conjured up. I seem to recall that there are numbers that do not stringify well, but I couldn't figure out how to construct one in time for this article deadline. But the *die* check at the end provides protection in that case, anyway.

So, there it is. A better *Data::Dumper* that handles references to arrays and references to hashes. Of course, the real *Data::Dumper* has a lot more bells and whistles, so I hope that the authors of data-dumping routines will use this code as a model, rather than hoping that I will eventually replace their code. Until next time, enjoy!

Listing 1

```
=1= package Data::Stringer;
=2=
=3= use 5.006;
=4= use strict;
=5= use warnings;
=6=
=7= require Exporter;
=8=
=9= our @ISA = qw(Exporter);
=10=
=11= our @EXPORT = qw(uneval);
=12=
=13= our $VERSION = '0.01';
=14=
=15= require overload;
=16=
=17= my %stab;
=18=
=19= ## $stab{'\x0x123456'} = \thevalue
=20= ## $stab{'\x0x123456'} = \thevalue
=21= ## $stab{'\x0x123456'} = [\thevalue]
=22= ## $stab{'\x0x123456'} = [\thevalue, $aggregate, $index] # for
                                     # elements
=23=
=24= BEGIN {
=25=
=26=     my @queue;
=27=
=28=     sub uneval {
=29=         %stab = @queue = ();
=30=         my $label = pass_1_item(@_);          # prime the pump
=31=         pass_1_item(shift @queue) while @queue; # drain the pump
=32=         return pass_2($label);                # dump the result
=33=     }
=34=
=35=     sub pass_1_item {
=36=         my $ref = shift;
=37=
=38=         my $label = ref_to_label($ref);
=39=         return $label if $stab{$label}; # already seen
=40=
=41=         if ($label =~ /\^$/ ) { # scalar
=42=             $stab{$label}[0] = $ref;
=43=             push @queue, $$ref if ref $$ref;
=44=         } elsif ($label =~ /\^@/ ) { # array
=45=             $stab{$label} = $ref;
=46=             for my $index (0..$#$ref) {
=47=                 for ($ref->[$index]) { # carefully creating alias, not
                                     # copy
=48=                     my $thislabel = ref_to_label(\$_);
=49=                     $stab{$thislabel} = [\$_, $label, $index];
=50=                     push @queue, $_ if ref $_;
=51=                 }
=52=             }
=53=         } elsif ($label =~ /\^%/ ) { # hash
=54=             $stab{$label} = $ref;
=55=             for my $key (keys %$ref) {
=56=                 for ($ref->{$key}) { # carefully creating alias, not
                                     # copy
=57=                     my $thislabel = ref_to_label(\$_);
=58=                     $stab{$thislabel} = [\$_, $label, $key];
=59=                     push @queue, $_ if ref $_;
=60=                 }
=61=             }
=62=         } else {
=63=             die "Cannot process $label yet";
=64=         }
=65=
=66=         return $label;
=67=     }
=68= }
=69=
=70= BEGIN {
=71=
=72=     my @deferred;
=73=
=74=     sub pass_2 {
=75=         my $result_label = shift;
=76=
=77=         @deferred = ();
=78=         return join("",
=79=             pass_2_declarations(),
=80=             pass_2_initializations(),
=81=             map("$_\n", @deferred),
=82=             pass_2_blessings(),
=83=             "$result_label;\n",
=84=
=85=
=86=
=87=         sub pass_2_value {
=88=             my $value = shift;
=89=             my $set_place = shift;
=90=             my $set_index = shift;
=91=
=92=             if (ref $value) {
=93=                 my $label = ref_to_label($value);
=94=                 if ($label =~ /\^$/ ) { # it is a scalar, so it might be an
                                     # element
=95=                     (my ($value, $place, $index) = @{$stab{$label}}) >= 1 or
=96=                     die;
=97=                     if ($place) {
=98=                         if ($place =~ /\^[0%]/ ) {
=99=                             push(@deferred,
=100=                                element_of($set_place, $set_index) . " = \" .
=101=                                element_of($place, $index) . "\";
=102=                             return "00"; # placeholder for a deferred
                                     # action
=103=                         } else {
=104=                             die "dunno place $place";
=105=                         }
=106=                     } else {
=107=                         return "\\$label"; # no place in particular
=108=                     }
=109=                 } else {
=110=                     return "\\$label";
=111=                 }
=112=             } else {
=113=                 return quote_scalar($value);
=114=             }
=115=         }
=116=
=117=         sub pass_2_declarations {
=118=             return join("",
=119=                 "my (",
=120=                 join(" ",
=121=                     grep {
=122=                         /\^[0%]/ or /\^$/ and not $stab{$_}[1]
=123=                     } keys %stab),
=124=                 ");\n");
=125=
=126=         }
=127=
=128=         sub pass_2_initializations {
=129=             return join("",
=130=                 map(pass_2_initialization($_, $stab{$_}),
=131=                     sort keys %stab),
=132=                 );
=133=
=134=         }
=135=
=136=         sub pass_2_blessings {
=137=             return join("",
=138=                 map(pass_2_blessing($_, $stab{$_}),
=139=                     sort keys %stab),
=140=                 );
=141=
=142=         }
=143=
=144=         sub pass_2_initialization {
=145=             my $label = shift;
=146=             my $value = shift;
=147=
=148=             if ($label =~ /\^$/ ) { # scalar
=149=                 if (@$value > 1) { # it's an element:
=150=                     return "";
=151=                 } else {
=152=                     return "$label = ".pass_2_value(${$value->[0]}).";";
=153=                 }
=154=             } elsif ($label =~ /\^@/ ) { # array
=155=                 return "$label = (".join(" ",
=156=                     map {
=157=                         pass_2_value($value->[$_], $label,
=158=                                     $_);
=159=                     } 0..$#$value,
=160=                     ").");";
=161=             } elsif ($label =~ /\^%/ ) { # hash
=162=                 return "$label = (".join(" ",
=163=                     map {
=164=                         pass_2_value($_) .
=165=                         " => " .
=166=                         pass_2_value($value->{$_},
=167=                                     $label, $_);
=168=                     } keys %$value,
=169=                     ").");";
=170=             } else {
=171=                 die "Cannot process $label yet";
=172=             }
=173=         }
=174=     }
=175= }
```

```

=169=
=170= }
=171=
=172=
=173= sub pass_2_blessing {
=174=     my $label = shift;
=175=     my $value = shift;
=176=
=177=     ## get to the proper location of an element for scalars
=178=     if ($label =~ /^\\$/ ) {
=179=         $label = element_of($value->[1], $value->[2]) if @$value > 1;
=180=         $value = $value->[0];
=181=     }
=182=     my ($package) = overload::StrVal($value) =~ /^(.*)=;/;
=183=     if (defined $package) {         # it's blessed
=184=         return "bless \\$label, \".quote_scalar($package), \";\n";
=185=     } else {
=186=         return "";
=187=     }
=188= }
=189=
=190= sub element_of {
=191=     my $label = shift;
=192=     my $index = shift;
=193=     if ($label =~ s/^\\@/\\$/ ) {
=194=         return "$label\\[\".quote_scalar($index).\\]\";
=195=     } elsif ($label =~ s/^%/\\$/ ) {
=196=         return "$label\\{\".quote_scalar($index).\\}\";
=197=     } else {
=198=         die "Cannot take element_of($label, $index)";
=199=     }
=200= }
=201=
=202= sub ref_to_label {
=203=     my $ref = shift;
=204=
=205=     ## eventually do something with $realpack
=206=     my ($realpack, $realtype, $id) =
=207=         (overload::StrVal($ref) =~ /^(?:\\(?:.*)\\)?(?:\\^\\(?:\\*)\\)\\$/ )
=208=         or die;
=209=
=210=     s/^0x/X/ or s/^/X/ for $id;
=211=     if ($realtype eq "SCALAR" or $realtype eq "REF") {
=212=         return "\\$id";
=213=     } elsif ($realtype eq "ARRAY") {

```

```

=212=         return "\\$id";
=213=     } elsif ($realtype eq "HASH") {
=214=         return "%$id";
=215=     } else {
=216=         die "dunno $ref => $realpack $realtype $id";
=217=     }
=218= }
=219=
=220= sub quote_scalar {
=221=     local $_ = shift;
=222=     if (!defined($_)) {
=223=         return "undef";
=224=     }
=225=     {
=226=         no warnings;
=227=         if ($_ + 0 eq $_) {             # safe as a number...
=228=             return $_;
=229=         }
=230=         if ("$_" == $_) {             # safe as a string...
=231=             s/([\\\'\\\"])/\\$1/g;
=232=             return '\\\' . $_ . '\\\'';
=233=         }
=234=     }
=235=     die "$_ is not safe as either a number or a string";
=236= }
=237=
=238= 1;

```

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/>



Perl Template Toolkit

Jack J. Woehr

Perl Template Toolkit coauthor Andy Wardley is the developer of the Template Toolkit, a Perl library providing for embedding textual substitution and live Perl code in text documents and text fragments. The template idea has been seen before in Perl, in a variety of CPAN offerings. The Template Toolkit emerged as an admixture and distillation of extant ideas into a framework with a variety of capabilities, ranging from those equivalent to m4 macro preprocessing, to inline code execution and database access facilities equivalent to those provided in Java by Java Server Pages or by the attractive-but-foundering-in-the-marketplace NET.DATA from IBM.

From a computer-science viewpoint, what we have in Perl templates is one of those simple ideas that can be folded over on itself fractally, lavishly, and almost infinitely. Furthermore, the ease of Template Toolkit development in Perl compares favorably with the numbing weight and complexity of real-world Java JSP development. The template idea, though applicable to a wide range of tasks, is, like much of Perl itself, aimed primarily at the generation of web pages. *Perl Template Toolkit* starts from generation of static documents, works its way through dynamic web pages, plugins, database access, XML, and even discusses modifying the Template Toolkit itself.

Perl Template Toolkit is a really good book. The Template Toolkit is one of those ideas whose time has come, the authors are as expert as expert can be, and the book itself is a quality production. I can find few faults—one certainly is sanguinity about the complexity of what is being conveyed. The audience for *Perl Template Toolkit* is proclaimed to be:

“...anyone building and maintaining web sites or other complex content systems. No prior knowledge of Perl, the Template Toolkit, or HTML is required to apply the basic techniques taught in this book. Some of the more advanced topics require some degree of familiarity with Perl...”

This assertion seems a bit over-optimistic on the part of the authors or the editor. Certainly the concept of templates is accessible to anyone who has used any form of macro substitution in the preparation of text documents, such as the mail merge feature of popular word processors. But Perl—deep, profound Perl—permeates *Perl Template Toolkit*. Even the initial section on “Installing the Template Toolkit” goes on for pages, offers suggestions for the make, and proffers various URLs, interestingly omitting to mention the simplest and most reliable method of installing and configuring CPAN offerings for use:

```
cpan: install Template
```

Sometimes, experts find an aspect of their work so elementary they are convinced it will be simple to any reader, forgetting that

Jack J. Woehr is an independent consultant and team mentor practicing in Colorado. His website is <http://www.softwoehr.com>.

Perl Template Toolkit
Darren Chamberlain, David Cross,
and Andy Wardley
O'Reilly and Associates, 2004
592 pp., \$39.95
ISBN 0-596-00476-1

merely the peculiar use of nonalphanumeric characters in the most basic Perl syntax can stun the nonexpert like a deer being caught in the headlights. You do want to know Perl to buy this book, but if you are on a team, you can certainly lend this book to a document preparer who is not a Perl programmer, while expecting to be called upon to mentor that individual. Setting up a development chain and production system leveraging the Template Toolkit is definitely not end-user work, though the authors are correct that end users with little or no programming experience can be expected to maintain many kinds of templates themselves, once the basic templates for the project have been written or at least exemplified.

Another grimly humorous aspect of the whole pursuit is that if you wish to run the examples that involve an Apache web server (really the meat of the book), you must take advantage of the extensive information contained in Perl Template Toolkit for configuring the *Apache::Template* module under *mod_perl*, which means that you may find you need to read another O'Reilly offering, *Practical mod_perl*, if you haven't already figured out how to adequately configure *mod_perl*. When I got right down to it, I discovered that being adequately configured on my home Solaris/Linux/BSD boxes in order to riffle through the code in this book was nontrivial, though quite enjoyable and [re]educational (web server configuration being one of those things you put out of mind when not dealing directly with it on a day-by-day basis). I found myself updating my Apache version and building or rebuilding bunches of Perl modules, including *mod_perl* as a minimum requirement to review this book.

Lastly, I find that the downloadable source code to accompany the book is indifferently presented, just sort of glommed onto HTML pages with none of the care or stylishness that distinguishes the rest of the literary effort. This can always be cleaned up over time, of course, on the source web site; one hopes the authors will make the post-production effort to generate a somewhat cleverer source presentation for their readers.

The book's web site (<http://www.oreilly.com/catalog/perl/tt/index.html>) offers the usual summary, table of contents, sample chapter, example code, etc. The Template Toolkit also has its own website at <http://template-toolkit.org/>.

TPJ

Source Code Appendix

Deborah Pickett “Catching Cheats with the Perl Compiler”

Listing 1

```
#!/usr/bin/perl -w

#
# compare: compare N Perl programs with each other.
#
# usage:
#   compare [-n max] file ...
# where
#   max is the maximum number of pairs of similar programs to report.
#

use strict;

use Getopt::Std;

our %opts;
getopts("n:", \%opts);

# How many cases to report?
our $stopcases = $opts{"n"};

# Suffix-tree-building code, adapted from
# http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/Suffix/ based on
# E. Ukkonen's linear-time suffix tree creation algorithm. Used with
# permission.
{
    my $infinity = 999999; # Just has to be longer than any string passed in.

    sub buildTree
    {
        my $fp = shift;

        # Build root state node.
        my $rootState = { };
        my $bottomState = { };
        my ($sState, $k, $i);

        for ($i = 0; $i < length $fp; $i++)
        {
            addTransition($fp, $bottomState, $i, $i, $rootState);
        }

        $rootState->{sLink} = $bottomState;
        $sState = $rootState;
        $k = 0;

        # Add each character to the suffix tree.
        for ($i = 0; $i < length $fp; $i++)
        {
            ($sState, $k) = update($rootState, $fp, $sState, $k, $i);
            ($sState, $k) = canonicalize($fp, $sState, $k, $i);
        }

        return $rootState;
    }

    sub update
    {
        my ($rootState, $fp, $sState, $k, $i) = @_;

        my ($oldRootState) = $rootState;
        my ($endPoint, $rState) = testAndSplit($fp, $sState, $k, $i-1, substr($fp, $i, 1));

        while (!$endPoint)
        {
            addTransition($fp, $rState, $i, $infinity, { });

            if ($oldRootState != $rootState) { $oldRootState->{sLink} = $rState; }

            $oldRootState = $rState;
            ($sState, $k) = canonicalize($fp, $sState->{sLink}, $k, $i-1);
            ($endPoint, $rState) = testAndSplit($fp, $sState, $k, $i-1, substr($fp, $i, 1));
        }

        if ($oldRootState != $rootState) { $oldRootState->{sLink} = $sState; }

        return ($sState, $k);
    }

    sub canonicalize
    {

```

```

my ($fp, $sState, $k, $p) = @_;

if ($p < $k)
{
    return ($sState, $k);
}

my ($k1, $p1, $sState1) = @{$sState->{substr($fp, $k, 1)}};

while ($p1 - $k1 <= $p - $k)
{
    $k += $p1 - $k1 + 1;
    $sState = $sState1;
    if ($k <= $p)
    {
        ($k1, $p1, $sState1) = @{$sState->{substr($fp, $k, 1)}};
    }
}
return ($sState, $k);
}

sub testAndSplit
{
    my ($fp, $sState, $k, $p, $t) = @_;

    if ($k <= $p)
    {
        my ($k1, $p1, $sState1) = @{$sState->{substr($fp, $k, 1)}};

        if ($t eq substr($fp, $k1 + $p - $k + 1, 1))
        {
            return (1, $sState);
        }
        else
        {
            my $rState = { };
            addTransition($fp, $sState, $k1, $k1 + $p - $k, $rState);
            addTransition($fp, $rState, $k1 + $p - $k + 1, $p1, $sState1);
            return (0, $rState);
        }
    }
    else
    {
        return (exists $sState->{$t}, $sState);
    }
}

sub addTransition
{
    my ($fp, $thisState, $left, $right, $thatState) = @_;

    $thisState->{substr($fp, $left, 1)} = [$left, $right, $thatState];
}

$| = 1;

# Perl executable.
our $perl = $^X;

# All fingerprints, keyed by filename.
our %fp;

# Suffix trees of all fingerprints, keyed by filename.
our %tree;

# Stop comparing fingerprints after this many blocks.
our $ceiling;

# Get all fingerprints.
foreach my $filename (@ARGV)
{
    # This is OK as long as characters in name of $file are safe.
    my $fingerprint = `perl -MO=Fingerprint $filename`;
    if (! $?)
    {
        # Remember this file's fingerprint.
        $fp{$filename} = $fingerprint;
        # Insert the fingerprint into the suffix tree forest.
        $tree{$filename} = buildTree($fingerprint);
    }
}

# Now compare each pair of fingerprints.
my @result;
my $count = 0;
foreach my $file1 (keys %fp)
{
    foreach my $file2 (keys %fp)

```

```

{
    next if $file1 eq $file2;

    my $length1 = length $fp{$file1};
    my $length2 = length $fp{$file2};

    # Progress meter.
    print int ($count++ / ((keys %fp) * (keys %fp)) * 100), "% complete\r" if -t STDOUT;

    # Do we have a maximum number of cases to report?
    if (defined $topcases && @result >= $topcases)
    {
        $ceiling = $result[-1]{ratio} * $length2;
    }
    else
    {
        undef $ceiling;
    }

    # Compare the files.
    my $blocks = compare($file1, $file2);

    push @result, (
        file1 => $file1,
        length1 => $length1,
        file2 => $file2,
        length2 => $length2,
        blocks => $blocks,
        ratio => $blocks / $length2,
    );

    # Ripple down new element in @result to keep it sorted.
    # If keeping only the top N cases, this is quicker than
    # sorting afterwards.
    if (defined $topcases)
    {
        my $pos;
        my $new = $result[-1];
        # Insertion sort algorithm.
        for ($pos = @result - 2; $pos >= 0; $pos--)
        {
            if ($new->{ratio} < $result[$pos]->{ratio})
            {
                # Ripple up an element.
                $result[$pos+1] = $result[$pos];
            }
            else
            {
                # Found the right place.
                last;
            }
        }
        # Insert the new item at its place.
        $result[$pos+1] = $new;

        # Lose the (now) last element?
        if (@result > $topcases) { pop @result; }
    }
}

# If collecting all cases, sort so that more similar code is near start
# of list.
if (!defined $topcases)
{
    @result = sort {$a->{ratio} <=> $b->{ratio}} @result;
}

# Present results.
foreach my $result (@result)
{
    print
        $result->{ratio}, " ", $result->{blocks},
        "/", $result->{length2}, ": ",
        $result->{file1}, " => ", $result->{file2},
        "\n";
}

sub compare
{
    my ($file1, $file2) = @_;

    # We're trying to reconstruct fingerprint $fp2 from $fp1, so need
    # suffix tree from $fp1.
    my $treel = $tree{$file1};
    my $fp1 = $fp{$file1};
    my $fp2 = $fp{$file2};

```

```

# Number of blocks counted so far.
my $blocks = 0;

my $pos2 = 0;
# Keep going while there's any of fingerprint 2 to do.
BLOCK: while ($pos2 < length $fp2)
{
    # Find a path through $treel that matches the part of $fp2
    # we're up to.
    if (!exists $treel->{substr($fp2, $pos2, 1)})
    {
        # This character doesn't exist at all in $treel. Next block.
        $pos2++;
        next BLOCK;
    }

    # There's an entry in the suffix tree.
    for (my $state = $treel->{substr($fp2, $pos2, 1)}; # Start at root.
        defined $state; # Stop if finished a leaf node.
        $state = $state->[2]{substr($fp2, $pos2, 1)} # Next node.
    {
        # Walk through characters in this state, comparing with $fp2.
        for (my $count = 0; $count <= $state->[1] - $state->[0]; $count++)
        {
            # Are there any more characters, and if so, do they match?
            if ($state->[0] + $count < length $fp1 &&
                $pos2 < length $fp2 &&
                substr($fp1, $state->[0] + $count, 1)
                    eq substr($fp2, $pos2, 1))
            {
                # Got a match, move on to the next character.
                $pos2++;
            }
            else
            {
                # Characters don't match; this is the end of a block.
                next BLOCK;
            }
        }
        # Finished this state, and it all matched. Go do the next one.
    }
}
continue
{
    # Count the blocks as we go.
    $blocks++;
    if (defined $ceiling && $blocks > $ceiling)
    {
        # Exceeded the ceiling, return.
        last;
    }
}

return $blocks;
}

```

brian d foy “Grokking Web Archives”

Listing 1

```

1  #!/usr/bin/perl -w
2  use strict;
3
4  use MIME::Base64;
5
6  my $archive = do {
7      local $/;
8      open my $fh, $ARGV[0] or die "Could not open $ARGV[0]: $!\n";
9      <$fh>;
10 };
11
12 my( $header, $multipart ) = split /(?:\x0D\x0A){2}/, $archive, 2;
13
14 my( $boundary ) = $header =~ m/boundary="(.*?)"/;
15
16 my @parts = split /\Q$boundary/, $multipart;
17 shift @parts; pop @parts;
18
19 my( $location, $encoding, $body ) = parse( shift @parts );
20 my( $directory ) = $location =~ m/(.*)\./;
21 $directory = "$location.d" unless $directory;
22 mkdir $directory;
23
24 unquote( $body ) if $encoding eq 'quoted-printable';
25 $body =~ s/<img(?:.*?)src="[^"]*/(.*?)"/<img$1src="$directory/$2"|isg;
26
27 open my $fh, "> $location" or die "Could not open $location: $!";
28 print $fh $body;
29 close $fh;

```



```

30
31 foreach my $part ( @parts )
32 {
33     my( $location, $encoding, $body ) = parse( $part );
34
35     $body = decode_base64( $body ) if $encoding eq 'base64';
36
37     open my $fh, "> $directory/$location"
38         or die "Could not open $location: $!";
39     print $fh $body;
40 }
41
42
43 sub divide    { ( split /(?:\x0D\x0A){2}/, $_[0], 2 )      }
44
45 sub parse     { my( $h, $b ) = &divide; ( location( $h ), encoding( $h ), $b ) }
46
47 sub location { ( $_[0] =~ m|Content-Location:.*?(\\S+)|i )    }
48 sub encoding { ( $_[0] =~ m|Content-Transfer-Encoding:\\s*(\\S+)|i ) }
49
50 sub unquote
51 {
52     $_[0] =~ s/=\\x0D\\x0A//g;
53     $_[0] =~ s<=([0-9A-F]{2})>{
54         chr hex $1
55     }ge;
56 }

```

Randal Schwartz “A Better *Data::Dumper*”

Listing 1

```

=1=    package Data::Stringer;
=2=
=3=    use 5.006;
=4=    use strict;
=5=    use warnings;
=6=
=7=    require Exporter;
=8=
=9=    our @ISA = qw(Exporter);
=10=
=11=    our @EXPORT = qw(uneval);
=12=
=13=    our $VERSION = '0.01';
=14=
=15=    require overload;
=16=
=17=    my %stab;
=18=
=19=    ## $stab{'%x0x123456'} = \@thevalue
=20=    ## $stab{'%x0x123456'} = \%thevalue
=21=    ## $stab{'$x0x123456'} = [\\$thevalue]
=22=    ## $stab{'$x0x123456'} = [\\$thevalue, $aggregate, $index] # for elements
=23=
=24=    BEGIN {
=25=
=26=        my @queue;
=27=
=28=        sub uneval {
=29=            %stab = @queue = ();
=30=            my $label = pass_1_item(\\@_);      # prime the pump
=31=            pass_1_item(shift @queue) while @queue; # drain the pump
=32=            return pass_2($label);              # dump the result
=33=        }
=34=
=35=        sub pass_1_item {
=36=            my $ref = shift;
=37=
=38=            my $label = ref_to_label($ref);
=39=            return $label if $stab{$label}; # already seen
=40=
=41=            if ($label =~ /^\\$/) {          # scalar
=42=                $stab{$label}[0] = $ref;
=43=                push @queue, $$ref if ref $$ref;
=44=            } elsif ($label =~ /^\\$/ ) { # array
=45=                $stab{$label} = $ref;
=46=                for my $index (0..$#$ref) {
=47=                    for ($ref->[$index]) { # carefully creating alias, not copy
=48=                        my $thislabel = ref_to_label(\\$_);
=49=                        $stab{$thislabel} = [\\$_, $label, $index];
=50=                        push @queue, $_ if ref $_;
=51=                    }
=52=                }
=53=            } elsif ($label =~ /^%/ ) { # hash
=54=                $stab{$label} = $ref;
=55=                for my $key (keys $$ref) {
=56=                    for ($ref->{$key}) { # carefully creating alias, not copy

```

```

57=         my $thislabel = ref_to_label(\$$_);
58=         $stab{$thislabel} = [\$_, $label, $key];
59=         push @queue, $_ if ref $_;
60=     }
61= }
62= } else {
63=     die "Cannot process $label yet";
64= }
65=
66=     return $label;
67= }
68= }
69=
70= BEGIN {
71=
72=     my @deferred;
73=
74=     sub pass_2 {
75=         my $result_label = shift;
76=
77=         @deferred = ();
78=         return join("",
79=             pass_2_declarations(),
80=             pass_2_initializations(),
81=             map("$_\n", @deferred),
82=             pass_2_blessings(),
83=             "$result_label\n",
84=             );
85=     }
86=
87=     sub pass_2_value {
88=         my $value = shift;
89=         my $set_place = shift;
90=         my $set_index = shift;
91=
92=         if (ref $value) {
93=             my $label = ref_to_label($value);
94=             if ($label =~ /^$/) { # it is a scalar, so it might be an element
95=                 (my ($value, $place, $index) = @{$stab{$label}}) >= 1 or die;
96=                 if ($place) {
97=                     if ($place =~ /^[%]/) {
98=                         push(@deferred,
99=                             element_of($set_place, $set_index) . " = \" .
100=                             element_of($place, $index) . "\";");
101=                         return "00"; # placeholder for a deferred action
102=                     } else {
103=                         die "dunno place $place";
104=                     }
105=                 } else {
106=                     return "\\$label"; # no place in particular
107=                 }
108=             } else {
109=                 return "\\$label";
110=             }
111=         } else {
112=             return quote_scalar($value);
113=         }
114=     }
115= }
116=
117=
118= sub pass_2_declarations {
119=     return join("",
120=         "my (",
121=         join(" ",
122=             grep {
123=                 /\^[%]/ or /\^$/ and not $stab{$_}[1]
124=             } keys %stab),
125=         ");\n");
126= }
127=
128= sub pass_2_initializations {
129=     return join("",
130=         map(pass_2_initialization($_, $stab{$_}),
131=             sort keys %stab),
132=         );
133= }
134=
135= sub pass_2_blessings {
136=     return join("",
137=         map(pass_2_blessing($_, $stab{$_}),
138=             sort keys %stab),
139=         );
140= }
141=
142= sub pass_2_initialization {
143=     my $label = shift;
144=     my $value = shift;
145=

```

```

146= if ($label =~ /^$/) { # scalar
147=     if (@$value > 1) { # it's an element:
148=         return "";
149=     } else {
150=         return "$label = ".pass_2_value(${$value->[0]}).";\\n";
151=     }
152= } elsif ($label =~ /^@/) { # array
153=     return "$label = (.join(\" \", \"
154=         map {
155=             pass_2_value($value->[$_], $label, $_);
156=             } 0..#$value,
157=         ).");\\n";
158= } elsif ($label =~ /^%/) { # hash
159=     return "$label = (.join(\" \", \"
160=         map {
161=             pass_2_value($_) .
162=                 \" => \" .
163=                 pass_2_value($value->{$_}, $label, $_);
164=             } keys %$value,
165=         ).");\\n";
166= } else {
167=     die "Cannot process $label yet";
168= }
169= }
170=
171=
172=
173= sub pass_2_blessing {
174=     my $label = shift;
175=     my $value = shift;
176=
177=     ## get to the proper location of an element for scalars
178=     if ($label =~ /^$/ ) {
179=         $label = element_of($value->[1], $value->[2]) if @$value > 1;
180=         $value = $value->[0];
181=     }
182=     my ($package) = overload::StrVal($value) =~ /^(.*)=/;
183=     if (defined $package) { # it's blessed
184=         return "bless \\$label, \".quote_scalar($package), \";\\n";
185=     } else {
186=         return "";
187=     }
188= }
189=
190= sub element_of {
191=     my $label = shift;
192=     my $index = shift;
193=     if ($label =~ s/^@/$/) {
194=         return "$label\\[".quote_scalar($index)."]\"";
195=     } elsif ($label =~ s/^%/$/) {
196=         return "$label\\{\".quote_scalar($index)."}\"";
197=     } else {
198=         die "Cannot take element_of($label, $index)";
199=     }
200= }
201=
202= sub ref_to_label {
203=     my $ref = shift;
204=
205=     ## eventually do something with $realpack
206=     my ($realpack, $realtypes, $id) =
207=         (overload::StrVal($ref) =~ /^(?:\\.*)?(?![^=]*\\\\(\\\\|\\\\\\\\)*\\\\/) or die;
208=         s/^0x/X/ or s/^X/ for $id;
209=         if ($realtypes eq "SCALAR" or $realtypes eq "REF") {
210=             return "\\$id";
211=         } elsif ($realtypes eq "ARRAY") {
212=             return "\\@id";
213=         } elsif ($realtypes eq "HASH") {
214=             return "%$id";
215=         } else {
216=             die "dunno $ref => $realpack $realtypes $id";
217=         }
218=     }
219=
220= sub quote_scalar {
221=     local $_ = shift;
222=     if (!defined($_)) {
223=         return "undef";
224=     }
225=     {
226=         no warnings;
227=         if ($_ + 0 eq $_) { # safe as a number...
228=             return $_;
229=         }
230=         if ("$_" == $_) { # safe as a string...
231=             s/[\\'\\`]/\\\\$1/g;
232=             return '\\\' ' . $_ . '\\\'';
233=         }
234=     }

```

```
=235=     die "$_ is not safe as either a number or a string";  
=236=   }  
=237=  
=238=   1;
```

TPJ