

BEST OF THE PERL JOURNAL



Games, Diversions, and Perl Culture: Best of the Perl Journal

Jon Orwant

Editor

Linda Mui

Copyright © 2010

Portions of this book originally appeared in *The Perl Journal*, currently published by CMP, Inc. Printed in the United States of America.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly & Associates, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps. The association between the image of a flying

dragon and the topic of Perl games, diversions, and culture is a trademark of O'Reilly & Associates, Inc.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.



O'Reilly Media

Preface

This is the third of three "Best of The Perl Journal" O'Reilly books, containing the créme de la créme of the 247 articles published during *The Perl Journal* 's five-year existence as a standalone magazine. This particular book contains 47 articles about the leisure pursuits of Perl programmers. You won't find articles on web development or object-oriented programming here. This book is for relaxing and reveling in Perl culture—a mindset favoring programs that are weird, wacky, and hubristic.

This book is divided into seven sections:

Part I

This section contains six articles on the Perl culture, including an article by Larry Wall comparing computer languages to music, a "coffee-table" collection of the TPJ covers, an article on Perl style, two articles on home automation, and an analysis of the usefulness of the Usenet newsgroup comp.lang.perl.misc.

Part II

Many scientists gravitate toward Perl when they find that they can analyze their data more easily with Perl than other languages. In this section, you'll find articles on astronomy, genetic algorithms, bioinformatics, and scientific computing.

Part III

Perl was created by a linguist, and it shows; there is no better language for manipulating text, whether it's a simple task involving punctuation or full-fledged natural language processing. In this largest section of the book, 15 articles demonstrate a plethora of language-related tasks, from speech synthesis to "bots" that answer English queries to correcting typos and adapting your Perl programs for other languages.

Part IV

Most of this book is about leisurely pursuits, especially if your notion of leisure includes writing bots that converse well enough to be hit on. If it doesn't, this section has more traditional games, from an overview of all the games available on CPAN to a solitaire game. It has all of the Perl quiz shows as well, to help you test and increase your Perl knowledge.

Part V

Perl Poetry has been around since 1990, and has been published in the *Economist* and the *Guardian*. In addition to the Perl Poetry contest, this section includes an article on reporting error messages in verse and how to search for rhymes in Perl.

Part VI

This section has three articles on how Perl can help maintain a stable democracy: two on voting methods, and one on how to prevent nuclear accidents.

Part VII

Perl's flexibility lets you make your code look like readable computer programs, poetry, or modem line noise. TPJ began the Obfuscated Perl Contest, and in this section you'll find the winning entries from all five contests as well as a complete collection of the one-liners that I used to fill up excess space in the magazine.

Be aware that this book has 31 different authors. Each section, and the articles within them, are loosely ordered from general to specific, and also from most accessible to least. Since these spectra are not identical, it's not a strict progression. The book may be read straight through, or sampled at random. (In deference to the Perl motto, There's More Than One Way To Read It.)

Normally, O'Reilly likes their books to be written by one author, or just a few. Books that are collections of many independently-written chapters may get to press more quickly, but discordant tones, styles, and levels of exposition are jarring to the reader; worse, authors writing in parallel and under deadline rarely know what other contributors have covered, and therefore can't provide appropriate context.

That would indeed be a problem for this book had it been written in two months by 31 authors writing simultaneously. But in a sense, this book was written very carefully and methodically over six years.

Here's why. As editor of *The Perl Journal*, I had a difficult decision to make with every issue. TPJ was a grass-roots publication with no professional publishing experience behind it; I couldn't afford to take out full-color ads or launch huge direct-mail campaigns. So word of the magazine spread

slowly, and instead of a steady circulation, it started tiny (400 subscribers for issue #1) and grew by several hundred each issue until EarthWeb began producing the magazine with issue #13.

For every issue there were new subscribers, many of whom were new to Perl. Common sense dictated that I should include beginner articles in every issue. But I didn't like where that line of reasoning led. If I catered to the novices in every issue, far too many articles would be about beginner topics, crowding out the advanced material. And I'd have to find a way to cover the important material over and over, imparting a fresh spin every time. Steve Lidie's Perl/Tk column was a good example: it started with the basics and delved deeper with every article. Readers new to Perl/Tk who began with TPJ #15 didn't need to know about the intricacies of Perl/Tk menus covered in that issue. They wanted to know how to create a basic Perl/Tk application—covered way back in TPJ #1. But if I periodically "reset" topics and ran material already covered in past issues, I might alienate long-time subscribers

So I did something very unusual for a magazine: I made it easy (and cheap) for subscribers to get every single back issue when they subscribed, so they'd always have the introductory material. This meant that I had to keep reprinting back issues as I ran out. This is what business calls a Supply Chain Management problem. The solution: my basement.

A side-effect of this approach was that the articles hold well together: they tell a consistent "story" in a steady progression from TPJ #1 through TPJ #20, with little redundancy between them. TPJ was always a book—it just happened to be published in 20 quarterly installments.

There is another advantage to having a book with programs by 31 flavors of Perl expert: collectively, they constitute a good sampling of Perl "in the wild." Every author has his own preferences—whether it's use of the English pragma, prototyping their subroutines, embracing or eschewing object-oriented programming, or any of the other myriad ways in which Perl's expressivity is enjoyed. When you read a book by one author, you experience a single coherent (and hopefully good) style; when you read a book by dozens of experienced authors, you benefit from the diversity. It's an Olympic-size meme pool.

Naturally, there's some TPJ material that doesn't hold up well over time: modules become obsolete, features change, and news becomes history. Those articles didn't make the cut; the rest are in this book and the two companion books, *Computer Science & Perl Programming: Best of The Perl Journal* and *Web, Graphics, and Perl/Tk: Best of The Perl Journal*.

Enjoy!

Finding Perl Resources

Beginning with TPJ #10, I placed boxes at the top of most articles telling readers where they could find any resources mentioned in the article. Often, it ended up looking like this, because nearly everything in Perl is available on CPAN:

Perl	5.8	or	lat	er.		 	 	 •	 •	 .CPAN
Class	s::I\$	SA.				 	 			 .CPAN
Memo	ize.					 	 	 •		 .CPAN
Class	s::Mu	ulti	Lmet	hod	5.	 	 	 •		 .CPAN

The CPAN (Comprehensive Perl Archive Network) is a worldwide distributed repository of Perl modules, scripts, documentation, and Perl itself. You can find the CPAN site nearest you at http://cpan.org, and you can search CPAN at http://search.cpan.org. To find, say, the Class::Multimethods module, you can either search for "Multimethods" at http://search.cpan.org, or you can visit http://cpan.org and click on "Modules" and then "All Modules." Either way, you'll find a link for a *Class-Multimethods.tar.gz* file (which will include a version number in the filename). Download, unpack, build, and install the module as I describe in http://cpan.org/modules/INSTALL.html.

For information and code that isn't available on CPAN, there are "Reference" sections at the ends of some articles.

Conventions Used in This Book

The following conventions are used in this book:

Italic

Used for filenames, directory names, URLs, emphasis, and for the first use of a technical term.

Constant width

Used for code, command output, program names, functions, and email addresses.

Constant width bold

Used for user input and code emphasis.

Constant width italic

Used for code placeholders, e.g., open(ARGUMENTS).

Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly & Associates, Inc. 1005 Gravenstein Highway North Sebastopol, CA 95472 (800) 998-9938 (in the United States or Canada) (707) 829-0515 (international/local) (707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:

http://www.oreilly.com/catalog/tpj3

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For information about books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

http://www.oreilly.com

Acknowledgments

First, an obvious thanks to the 120 contributors for the three books in this series, and a special shout-out to the most prolific: Lincoln D. Stein, Mark Jason Dominus, Felix Gallo, Steve Lidie, Chris Nandor, Nathan Torkington, Sean M. Burke, and Jeffrey Friedl. Sean's articles, in particular, are well-represented in this book.

Next up are the people who helped with particular aspects of TPJ production. TPJ was mostly a one-man show, but I couldn't have done it without the help of Nathan Torkington, Alan Blount, David Blank-Edelman, Lisa Traffie, Ellen Klempner-Beguin, Mike Stok, Sara Ontiveros, and Eri Izawa.

Sitting in the third row are people whose actions at particular junctures in TPJ's existence helped increase the quality of the magazine and further its reach: Tim O'Reilly, Linda Walsh, Mark Brokering, Tom Christiansen, Jeff Dearth, the staff of Quantum Books in Cambridge, Lisa Sloan, Neil Bauman, Monica Lee, Cammie Hufnagel, and Sandy Aronson. Best wishes to the folks at CMP: Amber Ankerholz, Edwin Rothrock, Jon Erickson, and Peter Westerman.

Next, the folks at O'Reilly who helped this book happen: Hanna Dyer, Paula Ferguson, Sarmonica Jones, Linda Mui, Erik Ray, Betsy Waliszewski, Jane Ellin, Judy Hoer, Ellie Volckhausen, Sue Willing, and the late great Frank Willison.

People who helped out in small but crucial ways: David H. Adler, Tim Allwine, Elaine Ashton, Sheryl Avruch, Walter Bender, Pascal Chesnais, Damian Conway, Eamon Daly, Liza Daly, Chris DiBona, Diego Garcia, Carolyn Grantham, Jarkko Hietaniemi, Doug Koen, Uri Guttman, Dick Hardt, Phil Hughes, Mark Jacobsen, Lorrie LeJeune, Kevin Lenzo, LUCA, Tuomas J. Lukka, Paul Lussier, John Macdonald, Kate McDonnell, Chris Metcalfe, Andy Oram, Curtis Pew, Madeline Schnapp, Alex Shah, Adam Turoff, Sunil Vemuri, and Larry Wall.

Finally, a very special thanks to my wife, Robin, and my parents, Jack and Carol.

Chapter 1. Introduction

Programmers aren't usually associated with culture, except the sort that grows inside a fridge. But Perl is different; it's spawned an array of pastimes such as Obfuscated Perl and Perl Poetry that perplex some outsiders but seem perfectly natural to the renaissance hackers attracted to Perl. As Larry says in Chapter 2, Perl is an intentionally postmodern language, employing features of its ancestors with a sang-froid that encourages Perl programmers not to take their craft too seriously.

The seven sections of this book are a grab bag: 41 of the best articles from *The Perl Journal*, plus 6 extra articles compiled especially for this book. Together, they span the playful aspects of Perl (with a rather broad interpretation of "playful").

Each of the seven sections—culture, science, language, games and quizzes, poetry, politics, and obfuscated Perl—have their own introductions, so let's get on with it. First up is Part I, where you'll read about Perl's postmodernism, how to automate your household appliances, and other flavorful topics.

Speaking for the Best of TPJ authors, we hope you enjoy this collection, and that it inspires you not just to participate in these pastimes, but to create your own new ones.

Part I. Culture

In this part: Chapter 2 Chapter 3 Chapter 4 Chapter 5 Chapter 6 Chapter 7

In this section, six articles provide glimpses into the aesthetics of Perl. The articles touch on music, art, style, conversation, and the lifestyle of the lazy, impatient, and hubristic, in which appliances do the programmer's bidding.

We begin with the first article from the first issue of TPJ: an essay by Perl creator Larry Wall that compares programming languages to music. Two sentences from his article have always resonated with me:

In trying to make programming predictable, computer scientists have mostly succeeded in making it boring.

and:

LISP has all the visual appeal of oatmeal with fingernail clippings mixed in.

Personally, I like LISP, and agree with those who think that its Scheme dialect is ideal for teaching computer science. But reading Larry's sentiments made me realize why I defected from LISP to Perl: programming languages shouldn't make everything look the same. When all code looks identical, programming becomes a matter of rote instead of a creative act of literary expression. It is that creativity that gave Perl its culture, and is what gave rise to the topics covered throughout this book, from the Obfuscated Perl contest to error messages delivered in haiku.

Next, photographer Alan Blount chronicles the 20 TPJ covers. Alan's artwork sometimes sparked more reader mail than the magazine content. The lack of visuals inside the magazine made the external appearance of the magazine all the more important, and I'm indebted to Alan for all his work. As a software developer, Alan understands what catches the eye of hardcore coders like us, and as an artist he has the ability to render that visually. A rare combination.

Kurt Starsinic follows with his article on calculating the readability of Perl programs. As Kurt mentions, Microsoft Word uses a relatively simple algorithm to determine the readability of a document, but programs are tougher to analyze. Kurt's *Fathom* module makes clever use of the Perl compiler to perform the analysis.

The next two articles are on home automation: controlling appliances such as lights and fans from your Perl programs. Bruce Winter begins with a demonstration of his popular Perl-based MisterHouse system, and Bill Birthisel follows up with a look under the hood at the X10 protocol that makes it all happen. Clinton Pierce concludes the section with an analysis of the heavily trafficked comp. lang.perl.misc Usenet newsgroup, dispelling the myth that it's all heat and no light.

Chapter 2. Wherefore Art, Thou?

Larry Wall

I don't know whether a picture is really worth a thousand words (most pictures seem to be considerably larger these days), but when I give talks about Perl, I often put up a picture (Figure 2-1) showing where some of the ideas in Perl come from.



Figure 2-1. The origin of Perl

I usually make a joke about Linguistics not really being the opposite of Common Sense, and then proceed to talk a lot about both of them, with some Computer Science thrown in for good measure. But last December as I was giving a talk in Stockholm, someone asked me how Perl got its inspiration from

Art. I was stumped. I mumbled something semi-irrational (always appropriate when discussing Art) and went on to the rest of my talk.

But the question continued to bother me; or more specifically, it continued to bother my left brain. My right brain continued to be perfectly content with the purported connection. Unfortunately, it's also not at all forthcoming with the verbiage necessary to explain itself. Right brains tend to be like that. So let me see if my left brain can make something of it all.

Art is first of all based on the notion that there exist amoral decisions; that is, choices you can make either way, without feeling like you're being naughty or nice. So let's presume that the Artist has free will of some sort or another, and can therefore behave as your ordinary, everyday Creator.

Now, it's more or less immaterial whether your Artist creates because of a liking for Deluxe Designer Universes or merely because of a liking for caffeine. The simple fact is, we have Artists, and they do

Art. We just have to deal with it. We really do. You can make life miserable for the Artist, but the Artist has ways of getting revenge. (Of course, if you don't make an Artist miserable, they'll make themselves miserable, but that's a different story.)

We can further subdivide the Artists into those who enjoy getting their revenge by being *more* than properly miserable, and those who prefer to get their revenge by being *less* than properly miserable. Artists of the first sort will prefer to work in a more formal medium, one that inflicts extra pain on the Artist, such as composing sonnets, dancing ballet, or programming C++. Artists of the second sort tend to be much more fun-loving, free-wheeling, and undisciplined, whether the verb in question is composing, dancing, programming, or

slinging. (Especially slinging. There's nobody quite so joyful as a B.S. artist. I should know...)

There is, of course, a third category of Artist, the one who oscillates between the two extremes.

Perl was written first of all to let the Artist make amoral decisions. That's why the Perl slogan is "There's More Than One Way To Do It!" Perl doesn't really care whether you use cobalt blue or burnt umber in a particular spot in your painting. It's your choice—you're the Artist. You're responsible for the overall effect. Indeed, your boss will hold you responsible for the overall effect, so why should Perl?

But more than that, Perl is intended to be a medium for those who are tired of composing in a formal computer language, and want to write some "free verse" without arbitrary restrictions. Sure, from a motivational point of view, arbitrary restrictions are challenging to work with, but when's the last time you saw a gleeful COBOL programmer?

On the other hand, with Perl 5, we've made strides in making life wonderful for those Artists who oscillate. You can have your cake and eat it too. When you're in a manic mood, you can pour forth unstructured, unreadable (but expressive) code to your heart's content. Later on, when you are in a dour mood, you can put a -w and a use strict at the top of your script and greatly increase your level of discipline (read "pain"). Next, you can prototype your function definitions. While still in your somber state, you can go back and put whitespace in all your regular expressions and comment every last little bit as penance for your past indiscretions. You can restructure all your code into modules and unit test it in a jiffy because the Perl interpreter is so handy to invoke. Then as you swing back into a more carefree frame of mind, you can cheat by tweaking all those carefully encapsulated variables in all those painstakingly restructured modules. Ain't it the life.

Now, Linguistics may not be the opposite of Common Sense, but it's certainly the case that over the last twenty years or so, many Computer Scientists have come out in opposition to the Art of Programming. In trying to make programming predictable, they've mostly succeeded in making it boring. And in so doing, they've lost sight of the idea that programming is a human pursuit. They've designed languages intended more to keep the computer happy than to keep the programmer happy. Was any SQL programmer ever happy about having to declare a value to be varchar(255) ? Oops, now it's a key, and can't be longer than 60. Who comes up with these numbers?

Computer Scientists have also lost sight of the idea known to any Artist, that form and meaning are deeply interdependent. One of the ideas I keep stressing in the design of Perl is that things that *are* different should *look* different. The reason many people hate programming in Lisp is because everything looks the same. I've said it before, and I'll say it again: Lisp has all the visual appeal of oatmeal with fingernail clippings mixed in. (Other than that, it's quite a nice language.)

A large part of the design of Perl is driven by the dictates of visual psychology. That's why Perl lets you structure your code with the condition on the left or on the right, depending on which part you want to look important. That's why the large nested structures like while loops require an explicit beginning and end, while the small ones like list operators don't. That's why scalars start with \$, arrays with @, and

hashes with %. That's why file test operators look like -M, while numeric tests look like ==, and string tests look like eq. Perl is very much a What-You-See-Is-What-It-Does language. You can talk about readability all you like, but readability depends first and foremost on recognizability.

Music to My Ears

Like many computer geeks, much of my artistic training has been in music. Of all the arts, it most clearly makes a programmer/interpreter distinction, so perhaps it's natural for a musician to think about how interpreters work. But the interpreters for a computer language are located both in the computer and in the human brain. I don't always know what makes a computer sad (or happy), but I do have a pretty good idea what makes a person mad (or sappy). Er, sorry.

Anyway, when I was young, I was taught that music has progressed through four major periods:

Baroque,

Classical,

Romantic, and Modern. (The other so-called fine arts have also gone through these periods, though not necessarily at the same rate.) I always thought it rather curious that we called the current period Modern, since definitionally the idea of modernity seems to be a permanently latched-on state, bound to the cursor of time, so to speak. But that was because the word "modern" still meant something back then. This was, after all, the 1960s. Who could have guessed that Modern period would be followed by the Postmodern?

If you're willing to concede by now that the design of computer languages is an artistic medium of sorts (and

searches), then it's reasonable for us to ask ourselves whether programming languages have been progressing through the same sequence of artistic development. Certainly, people have occasionally claimed that Perl is "Baroque," to which my usual retort is, "Thanks, I like Bach too." But this is merest rhetoric (on both sides).

So what do we really mean when we talk about these periods? Let's start at the beginning, which is the Baroque period. Of course, it's not really the beginning. People were producing music long before they ever invented the bucket in which to carry the tune. But before and during the Baroque period, there was tremendous technological progress in both the production and publication of music. Composers and performers could make a name for themselves. Innovators were rewarded, but the forms of expression were heavily influenced both by cultural expectations and by available hardware. People were expected to improvise. What we got was more or less the Cambrian explosion of music.

Similarly, at the dawn of the computer era, there were new opportunities to innovate. The geniuses of that period improvised many forms of assembly language. To them, these languages all looked very different. But nowadays we tend to see all assembly language as the same, just as a lot of Baroque music seems the same to us, because the music tends to follow particular forms and sequences. Baroque music is structured like a weaving on a loom, and it's no accident that punch cards were invented to run looms before they were used to run computers.

It's easy to take a superior attitude toward these innovators, but this is unfair. We owe a great debt to these people. They invented the algorithms we use, even if the music does seem a bit limited at times. (Except for Bach, and Backus, of course.)

The Classical period was a time of standardization. Most of our modern instruments took their current form during this period, and this continued the trend of turning virtuosity into a marketable and portable commodity. Being able to program in FORTRAN was like being able to play the pianoforte. It was a skill you could use on someone else's machinery. Mozart could now go on tour.

The Romantic era was a time of seeing how far the classical forms could be stretched. And considerably stretched they were, in Beethoven and Mahler, as well as PL/1 and COBOL. The word "excessive" has been applied to all of them, as it will always be applied to anyone or anything that attempts to sling the entire universe around by any of its handles. But this is difficult at the best of times.

Finally, the typical overreaction took place, and we arrived in the Modern era, in which subtlety and minimalism were mandated, and antiquated cultural expectations were thrown over and thrown out. Reductionism and deconstructionism were the order of the day, from Bartók to Cage, and from Pascal to C. Music wasn't allowed to be tonal, and computer languages weren't allowed to do fancy I/O. All the gadgetry had to be visible and exposed. Everything had to look difficult, so we got stuck in the Turing Tarpit.

Of course, this is all oversimplified, and every language has aspects of each of these periods in it. And languages specialize in other ways: BASIC is like pop music. Tune into REXX for your easy listening classics. Tcl is fuzzy like jazz—you get to improvise a lot, and you're never quite sure who is interpreting what. Python is like MTV—it rocks, but it gets to be much of a sameness after half an hour or so.

Lisp is like church music down through the ages, adapting to whatever the popular culture is, from organ to electric guitar to synthesizer. That would make Scheme a kind of cult music, sung simply but with great fervor to an acoustic guitar.

C++ is like movie music, of titanic proportions, yet still culturally derivative by and large. Especially large. Sometimes it's hard to sit through the whole movie. And yet, as an intentionally Postmodern language, it's kinda fun, and gets the job done.

As for Java, using a subset of movie music, it's attempting to be the basis for every good feeling everywhere, the ground of all emotional being. Muzak. It's everywhere you want to be.

Shell programming is a 1950's jukebox—great if it has your song already.

And, of course, any language touched by ANSI starts to sound distinctly operatic.

So where does Perl fit in to this glorious mess? Like C++, Perl is a Postmodern language by design, unashamedly reconstructionist and derivative. Perl is neo-Baroque, neo-Classical, neo-Romantic, and even, in spots, neo-Modern.

What musical genre encompasses so much? Where can you find everything from Wagner to "Shave and a Haircut, Two Bits?" Where can you find multiple levels of abstraction, accessible to newbies and oldsters alike? What kind of music admits everything from harmonica to accordion to pipe organ? What music is object-oriented, in good one-to-one correspondence with the main action? What music is good for

programming in the small, but can be expanded to feature length as necessary? What music parodies everything in the world, yet leaves you feeling good about the world? What music is Perl?

Why, cartoon music, of course.

That's all folks!

Chapter 3. TPJ Cover Art: From Camels to Spam

Alan Blount

TPJ #1: The Camel

The Perl Journal A QUARTERLY NEWSLETTER DEVOTED TO THE PERL PROGRAMMING LANGUAGE VOLUME 1, ISSUE 1 \$5.00 SPRING 1006 LARRY WALLON THE PSYCHOLOGY OF PERL PIRL AND THE WEB USING X FROM PERL SENDING AND RECEIVING E-MAIL FROM YOUR PROGRAMS

Figure 3-1. The Fiesta 12" stuffed camel

Date: Fri, 1 Dec 1995 23:28:16 -0500 From: Jon Orwant <orwant@fahrenheit-451.media.mit.edu> To: blount@media.mit.edu
Subject: resume' stuffer

Wanna be Photo Editor for The Perl Journal? You'll get your name listed in the masthead, plus \$50/issue, plus expenses paid, plus a free subscription.

I'd use your fun fotos for all

cover pictures, and maybe for some inside stuff too. F'rinstance, here's what I was thinking for the premiere issue: a dorky looking camel (buy a stuffed camel and treat it like a product shot), shot portrait-style, like Time's "Man of the Year." (Sunglasses, cigarettes optional.) Think "WIRED".

I want striking, cool covers. You up for a little creativity? WHAT SAY YOU? Lemme put you on the payroll. You won't regret it. No sir.

As *The Perl Journal* 's Photo Editor, I photographed covers for 18 of the 20 standalone

TPJ magazines. I'm not a professional photographer. I've done occasional portrait and catalog work—beginning when Jon and I worked together at the MIT Media Lab—but I've never tried to make a living at it. (I'm a software developer by day, and my language tastes tend toward Python.) Nevertheless, when Jon told me he was thinking of starting up a Perl magazine and asked me to take care of the covers, I figured, "Why not?" The collaborations Jon and I had previously attempted, through classwork and extracurricular projects, had always proven entertaining.

The concept for the premiere issue wasn't much of a stretch. "Uhhhh, what visually represents Perl? My God! O'Reilly's camel! But ours will be stuffed...and fuzzy! And..." You get the idea.

The photo was shot on 35mm in a snowbank behind my Somerville, MA apartment on a cold January afternoon. I was hoping the snow would come out looking like the shifting desert sands, but due to production troubles and my poorly-conceived high-key shot, the final result came out pretty illegible, branding us as the bunch of amateurs we were. A second printing improved the contrast a bit, but the concept remained: a fuzzball camel standing out in the cold. I failed to do Jon's idea justice, but he kept me on nonetheless.

TPJ #2: The Pearl

Date: Sat, 6 Apr 1996 17:23:03 -0500 From: Jon Orwant <orwant@fahrenheit-451.media.mit.edu> To: blount@media.mit.edu Subject: Quantum

On my way back last night, pressed my nose up against the Quantum Books window to see if they put TPJ on the front rack, or the back rack (old issues and NeXTSTEP handbooks). Front rack: above Forbes, to the right of Byte, and diagonal from WiReD. At eye level. Rock on.



Figure 3-2. The Swarovski little clam

TPJ #2 provided strong competition with TPJ #1 for "most literal interpretation of Perl." Orwant had found a lovely crystal oyster somewhere, and I had just the macro lens for it.

I shot it on 35mm in full sunlight against black velvet on the roof deck at NetCentric Corporation (now deceased), my day job. I went for extreme contrast—we'd have no more indecipherable camels, thank you. While we didn't have a lot of concept going, I liked the resulting image, with the specular (the glint of light) placed just-so on the crystal pearl. I've read that pro glassware photographers will spend hours or days tweaking tabletop shots with tiny pieces of paper and tinfoil positioned off-camera to create reflections where they want them.

TPJ #3: RSA on Greenbar

Orwant had an article on steganography in this issue, about how to hide secret messages in plaintext. It was fitting, then, to hide a secret message on the cover: if you read alternating characters on the third line from the bottom, you can see *my* hidden message. (Remember, I'm a Python fan.) I only told Orwant after the issue went to press, but he seemed to take it OK.



Figure 3-3. RSA-in-3-lines-Perl, and the artist's conception of same. Dot matrix on greenbar, 1996

The image is a composite. I pinched a couple sheets of old greenbar paper from my weekend job teaching LEGO/Logo at the Boston Museum of Science, and used their antiquated dot-matrix printer to render the copy on plain white paper. I then photographed both on 35mm black-and-white negative, and passed the negs to our production people, who
composited the images, adding the green back in to the monochrome photo.

For the detail-obsessed, the line noise was generated by firing up Emacs on "RSA-in-three-lines-of-Perl" and scattering a bunch of Ctrl-t's around.

TPJ #4: Etch-a-Sketch.



Figure 3-4. The Perl interface to the GNU "Hello World" program enables the Radio Shack Armatron to render text on an Ohio Art Etch-a-Sketch

The first full-color

cover! With profits from his burgeoning Ph.D. stipend, Orwant came up with the big dollars needed to front for color covers, enabling the publication to look a bit less like "Maximum Rock-and-Roll" or "R2D2 Is An Indie Rocker" and more like a professional publication.

Orwant said that this was when people started taking the magazine seriously. The content was the same as previous issues, but only when the cover went glossy and full-color did people stop calling TPJ a newsletter and start calling it a magazine.

The photo was shot with a Burke and James 5 x 7" monorail with the 4 x 5" reducing back and a Komura lens in a Copal #1 shutter, on a NetCentric conference table. And yes, the sketch was faked. That's what photographers do. We fake everything.

TPJ #5: Commodities



Figure 3-5. LEGO guys speculate on commodities. Some lose shirt, some make millions. Perl makes the difference

Orwant had an article on "Futures Trading with Perl," so we decided to create our own futures pit in my apartment with

coffee beans, red beans, and rice.

A lot of readers liked this one, but I was never too big on it. I like "high concept" shots—this one seemed sort of muddled to me. It was shot in early spring 1997, on 4 x 5" Fuji Velvia.

And what to do with the leftovers? The coffee went quick, but in 2002 I still haven't eaten all the small red beans (Goya) or rice ("Bombay Basmati"). I made some ham and beans the other night with some of the Goyas. It came out pretty good, for five-year-old beans.

TPJ #6: Scrabble



Figure 3-6. Why Perl programmers make lousy Scrabble players

Orwant may have wanted this

cover to allude to the Obfuscated Perl Contest in the issue. Or perhaps to the articles on randomness or information retrieval. Either way, this was visually where I wanted to take TPJ. The shot was clean, used whitespace well, and the high resolution of the 4 x 5" transparency enlarged to a grainless image. More importantly, it had the comic tone that we were looking for. Readers liked it, and subscriptions were going up (perhaps not entirely due to the covers).

"Scrabble" was shot on 4×5 " on the floor of my apartment in Somerville. Orwant supplied the "nonstandard" tiles. He tells me that, fed into a Perl interpreter, this board actually executes. And he wonders why I like Python.

TPJ #7: Spiderball



Figure 3-7. Here it comes

The centerpiece is a Microsoft creepyball that one of Orwant's friends scored at a tradeshow. It's menacing the "pearl," a Brian Dubé silicone juggling ball. Orwant adds, "This was around the time that many people in the Perl community were worried about Microsoft embracing and extending Perl through ActiveState."

This was the first shot made in my new loft. Tech details: $4 \times 5''$ Fuji Velvia, Speedotron strobe through a softbox, shot against white seamless.

We were trying to elicit a sense of foreboding. Did it work? Note that this was the first and only image that "violates the title," just like Springsteen's head in front of "Rolling Stone," and most O'Reilly animal covers.

TPJ #8: The Coffee Cup Fiasco



Figure 3-8. Good to the last drop

Around this time, every computer magazine had an issue on Java, and each had the same boring

cover: a coffee cup. For the TPJ issue on Java, I wanted an alternative interpretation. This one generated some fan mail.

```
Date: Thu, 18 Dec 1997 22:06:46 -0600
Subject: cover "art"
From: [a subscriber]
To: letters@tpj.com
Hi TPJ,
I love TPJ, but I want to be one of the
first to tell you that the cover
of Issue #8 is pretty revolting. Inside
good; outside revolting. Please
try harder!
Thank you.
Date: Mon, 22 Dec 1997 08:09:22 -0700
From: [another subscriber]
To: staff@
tpj.com
Subject: Issue #8
Cover
I don't know what point, if any, you were
trying to make with the
cover of the Winter 1997 (Issue #8)
edition, but I found it to be
absolutely disgusting. I would hope you
would have better taste
than that.
From: [yet another]
To: letters@tpj.com
Subject: Comments on Winter 97 issue TPJ
I think your magazine is good. But the
Cover was in bad taste. A
```

cigarette butt floating in coffee is sickening. I covered it with my own art.

More comments:

Even though I might not read all the articles, I appreciate the variety offered. Mv main reason for providing this feedback is to express my intense dislike of the coffee cup cover photo. I am usual guite careful with how Т treat my periodicals since I use them for reference after having read them. But the coffee cup grossed me so much that I ripped the cover off before I stared reading the magazine.

Thanks for an otherwise informative and enjoyable magazine.

Also:

I began to pick up the magazine from the top back rack where the Barnes and Noble I go to keeps it. As my fingers closed around it I thought, "Good! It seems thicker this issue!". As I began to lift it I saw the reference to Java and I thought "This looks good!".

Then I saw the cover. Retching violently I dropped the disgusting image as quickly as I could. I cannot believe that the person who allowed that cover to go out has anything to do with computers. Isn't the hacker

aversion to the insanely stupid and disgusting form of self-abuse called smoking known over there? I will not buy this issue of the Journal which is sad as the contents this questionaire based on look fascinating but I will not suffer to gaze upon that cover ever again. Thank God I have not yet subscribed! Stupid! Stupid!

I went all the way out to E.P. Levine's (a pro photo shop) on Boston's waterfront to pick up the Formica-looking cardboard tabletop. So sorry.

TPJ #9: Drummer/Coder Wanted



Figure 3-9. Funktion (opening for \$_, The Scalar Formerly Known As ARG)

Back in college, I made a lot of posters for dorm parties, student-produced cable shows, and the like. Orwant and I thought it'd be entertaining to cross the DIY punk stolen photocopies poster aesthetic with corporate HR. The Rapture background added that religious element so pervasive in programming.

This was shot on 35mm, on McGrath Highway at Twin City Plaza in Somerville. The Band:

Sigue Sigue Sputnik. The computer: TRS-80 Model 100. The "Rapture" poster below has since been painted over, although you can still see them plastered throughout Boston.

TPJ #10: The Underwood Typewriter



Figure 3-10. The only machine that Perl doesn't run on

This was our finest hour. I picked up the typewriter at an antiques store in Colorado, and shot on $4 \times 5''$ on white

seamless. Orwant supplied the copy, except for the "using 1 for l" gag—that was mine.

Once I installed gcc, the rest of the configuration went fine.

TPJ got a lot of requests for making this

cover into a poster or screensaver, but Orwant's taste for Perl novelties began and ended with his Magnetic Perl Poetry Kits.

TPJ #11: The Conspiracy

This is the first of several that I produced entirely, including the heading and all the overlaid text, which explains why it looks so crappy. Rather than buying barcode software, Orwant wrote a barcode generator in Perl and generated a tiff. I cut the entire thing together in Photoshop, The Gimp, and PowerPoint. Yes, PowerPoint. Several

TPJ covers were laid out in PowerPoint. What, pay for Illustrator?



Figure 3-11. -X File: The true value is out there

I didn't have much of a concrete idea for this issue. I was trying to put an X-Files spin on the Cult of Larry, or something. The film reel in the shot is 1970s 8mm pr0n.

That's Larry at the lower left, and a Luxo lamp to his right. The printouts are a collection of Larry's quotes. We were back to shooting on 35mm. Jon bought me a Nikon Coolscan, which cut several days off of production time for each

cover.

TPJ #12: The Atari Perl Cartridge

Do you think we fooled anyone with the Perl

game nestled in the stack of

cartridges on the left? It took several hours of tweaking the image with The Gimp to get the look right. Yes, that's an original Wico Bat Handle stick in the background. And my old Sears cartridges from circa 1980. "Target Fun" rocks.



Figure 3-12. Game over

Shot on 35mm with a Nikon FM2.

TPJ #13: Dance Remixes

Another Gimp extravaganza. Boy do I suck at layout. We started with an ActiveState Perl CD-ROM, and my antiquated Sony Discman

Orwant and I were pretty conscious of falling into the trap of doing too many "Look—Perl in an unexpected place!" covers. After the Armatron, Scrabble, typewriter, Atari cartridge, and finally the CD, the concept had pretty much run out of steam. It was time to move on. Ripping on Microsoft is always good for a few laughs....



Figure 3-13. D.J. Larry Dogg and the Porterz bring their unique retro-millennial page-thrashing funktions to a stereo near you. Shout-out to the Mixmasta Gimp 1.0

TPJ #14: Outlook Not So Good

Did anyone get this? Anyone? [Tap tap] Is this thing on? Outlook could either be construed to mean the Microsoft product of the same name, or the sale of TPJ to EarthWeb. The best art allows the viewer to interpret the art in his or her own way.

One improvement from the

EarthWeb buyout is that we got Real Designers to help with the

cover. That's why the title isn't grainy and aliased.

It's a real pain to make a

Magic 8 ball look good on film. Do you show the hand shaking the ball? Do you want the black plastic to shine? And if you're not careful, every time you move the ball in the shot, you have to re-shake it to make "Outlook not so good" appear again. I guess that's why professionals have photo assistants. "Assistant, go shake the 8 ball."

Shot on 35mm, with a Nikon FM2.



Figure 3-14. The spirits care not for Microsoft's mailer

TPJ #15: Braille blocks

I had nothing to do with this one. Didn't even collect a check.

This issue contained an article on

Braille (Chapter 17), and author Sean M. Burke got his friend David

Ondrik to take a photo of the Braille tiles (made by Tack-Tiles, Inc. at http://www.tack-tiles.com/).

The text spelled out is "<capital>The <capital>Perl <number>5 <com>pil<er> r<ea><ch><ed> l<in>e <number>850 <and> di<ed><period>."



Figure 3-15. Alphabet blocks in Braille

TPJ #16: e. e. cummings' Gravestone

Oh, the irony. The famous poet, known for his penchant for lowercase letters, memorialized in all caps. This was for a special issue on poetry in Perl, containing Chapter 37, Chapter 38, and Chapter 39.

Orwant and I spent a couple of hours searching Forest Hills cemetery for the

headstone on a beautiful fall day. It's too bad we didn't bring an industrial blow-dryer to clear the rainwater off the stone. Note the precisely positioned leaf at the lower left—that's Orwant's brilliant

art direction.

We visited the cemetery on October 14, 1999, which (unknown to us) happened to be e.e.'s birthday. Eerie.



Figure 3-16. e.e. cummings was one of the most innovative contemporary poets, known for using distorted punctuation and syntax to convey subtle shades of meaning. His name was most often spelled in all lowercase, and the theme of birth pervades his poems

You just don't see a lot of lowercase on headstones these days. I wonder if, with the current SMS/IM aesthetic, we'll start getting "hr li3x william denny, aka b1tr00tr, once l33t, now ded. peece out beeatch."

TPJ #17: Napster

This was back when Napster was just starting to take off, and Lincoln Stein had an article inside on automating Napster searches with Perl. I would have loved to have shot this cover for some record industry magazine.

I'm indebted to Jon Dakss for most of the vinyl. That's my own sad 1990s CD collection.



Figure 3-17. One Boston college dorm saved money recently by buying a cheap PC instead of a stereo and downloading MP3s instead of buying CDs. Pictured: vinyl LPs, CDs, and disk drives

TPJ #18: Spam

To: Jon Orwant <orwant@tpj.com> From: Alan

Blount <blount@alum.mit.edu>
Subject: Re:

cover foto

It was a throwaway shot at the end of the roll. Shutter speed was too slow to handhold the focus well. Hence the blur on the spam.

But that's not your story. It's blurry because it's Art. We're looking to set a mood here. A sort of drunk mood, after a party, coming back to the apartment, and there's spam in your mail, which you can't quite recognize at first because of your blurred vision.



Figure 3-18. Spam visits Apartment 12B

Simon

Cozens had an article in this issue on filtering mail with his Mail::Audit module, so we decided on an homage to SPAM for the

cover. Hormel has this to say about use of their delicious meat product to mean

Unsolicited Commercial Email:

We do not object to use of this slang term to describe UCE, although we do object to the use of our product image in association with that term. Also, if the term is to be used, it should be used in all lowercase letters to distinguish it from our trademark SPAM, which should be used with all uppercase letters.

This slang term does not affect the strength of our trademark SPAM. In a Federal District Court case involving the famous trademark STAR WARS owned by LucasFilms, the Court ruled that the slang term used to refer to the Strategic Defense Initiative did not weaken the trademark and the Court refused to stop its use as a slang term. Other examples of famous trademarks having a different slang meaning include MOUSE, to describe something MICKEY as unsophisticated; TEFLON, used to describe President Reagan; and CADILLAC, used to denote something as being high quality.

This was shot in a New York City co-op on 35mm, using existing light.

TPJ #19: Monopoly Money



Figure 3-19. Monopoly money

There were two articles on Perl's finance modules in this issue, so I trotted out my old Monopoly game and scanned in some money, whitewashing the resulting image a bit so you'd still be able to read the text overlaid on top. No camera
involved—just my new flatbed scanner. I love the high-res detail.

As much as I like this

cover, it's unfortunate that we didn't find some way to allude to the "DeCSS in English" article, which shows the output of a program that uses Perl to convert C to English, applied to the DVD decryption software.

TPJ #20: WAP

A digital photo by Michael Davis. The last Earthweb issue. The last Blount issue. And the first human—Dan Brian—on TPJ!



Figure 3-20. Dan Brian demonstrates the promise of WAP

It's not that I don't like portraiture. We had never attempted it before for two reasons. First, Perl geeks don't photograph well. It's not that they're only-a-mother-could-love-them ugly, but we're not exactly in the beauty industry here. The second reason is that while there are certainly many Perl luminaries who merited a cover photo, the logistics of getting to the same physical location were more than we could manage. Most of the covers were shot sometime between midnight and 4 a.m. the night before (or a few days after) deadline—not the best time to ask someone to pose for the camera.

But I really dig Dan out in the snow. The fog on his glasses makes the shot.

With TPJ #20 and the sale to CMP, my TPJ tenure ended. I had a great time.

Chapter 4. Perl Style

Kurt Starsinic

What is good coding practice? What is readable code?

For some programmers, these questions lead to heated arguments. In the relatively young field of programming, it's natural that generally accepted rules of style and usage haven't yet emerged. Fortunately, our colleagues in the more mature field of philology (the study of language as used in literature) have set examples that we can follow. In this article, I'll describe Fathom, a module that grades the readability of Perl programs.

Background

You may have experience with the grammar check feature of some word processors, which finds likely spelling, grammar, and usage errors in your documents. These tools can be quite useful, particularly for people who don't do much writing, or for people who haven't had much writing instruction.

As a programmer who works mostly in teams, often training new or junior programmers during time-critical projects, I want automated ways to encourage compliance with team coding standards. I know that such tools can (and do) work for business writing, but I've been unable to find a tool that would do the job for business coding. I did some investigation to see if any of the available grammar checkers could be adapted for use with program code.

Existing Measures

There are many well-known measures of readability in literature. You may have heard Flesch-Kincaid, FOG, SMOG, Bormuth, or other readability or grade level tests; Microsoft Word uses three

Flesch-Kincaid tools to evaluate style. These tests generally look at the average number of syllables per word and the average number of words per sentence, then report a single number which indicates either the grade level (1-12) or

of

readability (usually 1–100) of the document. As an example, the Flesch-Kincaid formula for determining the grade level of a document is:

```
((average sentence length in words) *
(0.39)
+ ((average syllables per word) * 11.8)
- 15.59
```

Unfortunately, these measures don't map well onto code; for example, how many syllables are there in ++ or $\{$ or \$? Is select easier to read than gethostbyname?

Once I realized that I wouldn't be able to simply run one of the prose-readability tests on my code and get meaningful results, I began to study the design and function of those tests. Then, I constructed a working model for code readability.

The Basic Units

After thinking about tools like Flesch-Kincaid, and discussing the idea of a readability tool with colleagues, I came up with a basic model for a

code readability metric. I decided to measure the number of tokens

per expression, the number of

expressions per statement, and the number of statements per subroutine. Some sample tokens:

```
++

$foo::bar
;
{

&&

any keyword
```

Some sample expressions:

```
0.2
($a + 6)
wantarray ? @a : 0
```

And some sample statements:

```
$a = $foo::bar * 7;
$x++;
```

The Tool

Given the basic model I've described, I wrote a module,

Fathom, that grades the

readability of a

Perl program. It rates on an open-ended scale, where 1 indicates a trivial program, 5 indicates "mature" code, 7 indicates very sophisticated code, and anything over 7 is Very Hairy. I established the following norms for mature code:

3 tokens per expression6 expressions per statement20 statements per subroutine

From this, I came up with the following formula:

```
code complexity
=
  (( average expression length in tokens)
* 0.55)
+ (( average statement length in
expressions) * 0.28)
+ (( average subroutine length in
statements) * 0.08)
```

If you plug the norms (3, 6, 20) into this formula, you'll see that ideal mature

code actually gets a score of 4.93; that's because I rounded all the multipliers to two decimal digits, to keep things simple.

Usage

First, you'll need to install

Fathom. You can find it on CPAN, under authors/id/K/KS/KSTAR.

After installing Fathom, you can invoke it as follows:

perl -MO=Fathom filename

The output looks like this:

315 tokens
97 expressions
17 statements
1 subroutine
readability is 4.74 (easier than the norm)

Why This Should Be Hard To Do

Perl is an unusual programming language, in that it has *dynamic syntax*; that is, any programmer can write code that extends or changes the syntax

of Perl. Consider the following code:

```
use Mystery;
if (mystery /1/ . . .
```

You can't parse this without knowing about Mystery.pm! Let's consider two different versions of Mystery.pm.

Version 1:

```
package Mystery;
sub main::mystery { return 5; }
1;
```

Version 2:

```
package Mystery;
sub main::mystery() { return 5; }
1;
```

These two packages are almost trivially different. They both define one function, named mystery, which returns the value 5. However, the second version uses a prototype. In the first case, our program parses as:

```
if (mystery( the results of matching the regular expression /1/\ldots
```

In the second case, it parses as:

```
if (mystery() divided by 1 divided by ...
```

By the time you've written a program that can successfully parse every possible case, you've rewritten Perl!

The Perl Compiler to the Rescue

Fortunately, the

Perl compiler gives us access to the pertinent guts of Perl, allowing us to calculate the tokens and expressions directly; see the

Fathom source

code for details. Without the compiler, this project would have been prohibitively difficult.

Here are some examples of Fathom evaluations:

```
Benchmark.pm
27 tokens
7 expressions
5 statements
1 subroutine
readability is 2.91 (very readable)
Apache::AdBlocker
47 tokens
13 expressions
6 statements
1 subroutine
readability is 3.08 (readable)
CGI/Carp.pm
66 tokens
22 expressions
11 statements
1 subroutine
readability is 3.09 (readable)
```

```
per15.005/eg/travesty
259 tokens
96 expressions
33 statements
1 subroutine
readability is 4.94 (easier than the norm)
s2p
2588 tokens
826 expressions
384 statements
11 subroutines
readability is 5.12 (mature)
CGI.pm
521 tokens
180 expressions
54 statements
1 subroutine
readability is 6.85 (complex)
DBI.pm
835 tokens
252 expressions
58 statements
1 subroutine
readability is 7.68 (very difficult)
diagnostics.pm
767 tokens
272 expressions
96 statements
1 subroutine
readability is 10.02 (obfuscated)
```

Future Directions

I intend to continue to refine

Fathom in several ways: by tweaking its basic formula to produce more accurate grades, by considering the placement and length of comments and pods, by having it identify problematic

code sections, and by having it make specific suggestions for improvement.

There are also some problems I hope to address in the near future: Fathom doesn't see

code that executes at compile time, such as code in BEGIN blocks or use statements, and sometimes it counts implicit tokens, such as \$ in a foreach statement. These limitations probably won't make much statistical difference in a medium-to-large program, but they could give wildly strange grades to one-liners and other short hacks.

Fathom also opens the door to a whole suite of companion tools: a program that checks variable names against a site-wide naming policy; a tool, much like C's indent, to normalize the indentation of

Perl code; and likely several more tools, based on experience and feedback. Some of these are already being developed by others.

Perl's extraordinary architecture makes it possible to produce very powerful companion tools without having to re-invent the wheel. Fathom was developed with a relatively small amount of original code—it simply hooks into the pre-existing Perl internal data structures to do its job. Similarly, the Perl debugger uses built-in features of Perl, plus a minimal amount of black magic, to provide a full-featured debugging environment for your Perl programs.

In most other languages, writing a tool like Fathom would force you to start from scratch, since some of the best tools for other languages (e.g., gdb, indent, and cxref for C) are based on code that is completely independent from the compilers or interpreters that they complement. In the case of languages that are still undergoing refinement (such as C++), maintenance of these tools can be a nightmare. However, Fathom will continue to work *even if Perl's syntax changes*, because it's hooked into the Perl compiler itself!

I hope that you're so intrigued by Fathom that you'll want to refine it, rewrite it, or develop new tools in a similar vein. Try this at home, kids!

Acknowledgments

Fathom would not have been possible without Malcolm Beattie's outstanding work on the Perl compiler. Stephen McCamant's

B::Deparse module was tremendously helpful in demonstrating how to write a compiler backend. And, of course, I couldn't have done any of this without such a rich language as Perl.

Chapter 5. Home Automation with MisterHouse

Bruce Winter

When most people think of Home Automation (HA), they think of "The Jetsons," where every appliance in the house has a mind of its own and occasionally does something you want it to do. The classic image of a HA geek is a man who lives in his easy chair, controlling every aspect of his life with voice commands and buttons like you see in Figures Figure 5-1 and Figure 5-2.



Figure 5-1. MisterHouse has several Web interfaces. This one is designed for easy use with the touch screen of small Internet appliances



Figure 5-2. This part of the web interface allows you to toggle, brighten, or dim X10 modules

While this may actually be the ultimate dream of a few couch potatoes, this is not what most people want. To many people, a night alone in a

smart house would be worse than a night at the Bates Motel. [1] [2]

This article shows you my take on HA, used for practical, everyday chores that even your technophobe spouse or relative might appreciate. You can do it with the computer you have today. You don't have to be Bill Gates to afford it. And you can do it all with your favorite programing language!

HA! Perl?

So why does Perl make an excellent choice for an HA programming language? Let me count the ways:

Concise and object-oriented

HA is object-oriented programming in its most literal sense, allowing Perl's concise object syntax to shine. What could be simpler and more intuitive than a event like this:

set \$christmas lights ON;

Behind the scenes, we build a generic serial port object, and then easily build objects for more specialized serially-controlled devices that inherit the generic serial port's methods.

Good with strings

Every good

smart house should be able to read and write. And boy, can Perl read and write!

As an example, consider the David Letterman Top 10 list. Sometimes very funny, but always past my bedtime. With a few lines of Perl, we can retrieve the previous night's list from the Web and convert it into a speakable string that can then be served with the morning breakfast:

```
# Get the HTML and translate to text
my $html = get 'http://www.cbs.com/
```

```
latenight/lateshow/top_ten/';
my $text = HTML::FormatText->new(lm =>
0, rm => 150)->
format(HTML::TreeBuilder->new()->parse($html));
$text =~ s/^.+?the Top Ten/The Top Ten/s;
# Delete text before the list
$text =~ s/(.+\n *1\..+?)\n.+/$1\n/s;
# Delete text after the list
$text =~ s/([^\.\?\!])\n/$1\.\n/g;
# Add a period at end of lines.
speak $text;
```

The mighty eval

Rather than stopping and restarting the entire program from scratch each time we want to change something, we can use eval to quickly load in just the code that has changed.

This is especially useful with a complex HA program, where small changes to event programming can be frequent and you don't want your house offline for very long.

Free and ubiquitous

Not only is Perl free, but it also runs everywhere. The HA community is currently split between Windows (lots of fun software/peripherals) and Unix (reliable, which is kind of important for HA). Using Perl, we can easily switch between platforms when it makes sense.

^[1] [Include possibly apocryphal story about Bill Gates' first night in his new NT-controlled mansion, where he couldn't turn off the TV?—Editor 1]

^[2] [We can't verify that, so we'll just mention it in a footnote and say that we were unable to confirm the story. Hey, it works for political candidates...—Editor 2]

MisterHouse

Now that you're convinced that Perl is good (admittedly not a tough sell for TPJ readers), how can you use it to control your house? With MisterHouse! Cue the infomercial music....

In 1999 I began a Perl program called MisterHouse. We now have an active group of over 500 users and 100 contributors who have helped shape it into a powerful HA program with the following features:

- It executes actions based on voice input, time of day, file data, serial port data, and socket data.
- It has an optional Tk interface, pictured in Figure 5-3, with menus for all objects and voice commands.
- It has a

web interface that mirrors the Tk interface to allow control and feedback from any browser, either on a local intranet or on the Internet. Check ours out at http://misterhouse.net:8080.

- On Windows systems, it uses OLE calls to Microsoft VR (Voice Recognition) TTS (Text To Speech) engines.
- On Unix systems, it can use the Festival TTS engine.
- On Linux systems, it can use the IBM ViaVoice VR and TTS engines.

- Using the SerialPort module, it can read and write to any serial device, on either Windows or Unix.
- It has modules for reading and writing to the X10 ActiveHome (CM11) and Firecracker (CM17) interfaces. See Chapter 6 for more details.
- It also has modules for reading and writing digital, analog, and X10 signals using other HA interfaces from JDS, HomeVision, and Weeder Technologies.
- Other hardware interfaces we have code for include IRMan (for receiving IR signals),

LCDproc (for writing to various LCDs and reading keypad input),

ham radio modems (for tracking vehicles with GPS units), and modem code for caller ID and paging.

To Mislas House version 2.1 PiD: 44	4233						200 K
the Connect Greek							Deall 0 0 0 0 0 0 - Hall
Art Learn II GP, purity + gplann: Pari gplann: Pari gplann: Pari gplann: Dat gplann: Dat GP, purity + gplann: Dat GP, purity + GP, p	taran taraf Soto So Arnon Jamilari T Arnon Jamilari T Arnon Jamilari T	4 74 1 4 7 4 7 4		Desarral	Lan		227/09 14280 15 percent. The car is investigation of worst at 29 mph 8.8 mile 2. 227/09 14280 25 memorily "the car is investigation of COUCY-1 227/09 14280 25 memorily the same 7.00 PM 227/09 14280 25 memorily two and two percent of the car 227/09 14290 25 memory two availed like to be not very senset. Furthaga yee would be 227/09 14290 25 memory two availed like to be not very senset. Furthaga yees would be 227/09 14290 25 memory two availed like to be not very senset. Furthaga yees would be 227/09 14290 25 memory two availed like to be not very senset. Furthaga yees would be 227/09 14290 25 memory two availed like to be not well be interest. For percent device to 227/09 14290 25 memory two availed like to be not well be interest. For percent device to 227/09 14290 25 memory two availed like.
Ling Man Immittation	Street, Sonay	[in	Th passes	C Office		Clice says: csigh) Just great, another hum as the internet who's got a 'good' q 02/27/00 12:54:15 normal: Web access hum core by com	
	Entrop Test	P Bana	Mate		02227/00 13:50:45 normal: It is 74 degrees 36 percent inside. 47 degrees 34 4 02227/00 13:50:42 normal: The winod is guiding at 10 MPH from the west. Aver		
	E throughout P through			-		- 87	With the second a memory of the second a memory.
	Heat Telan	F4				1 2 3	how 1. Channel 4, WCCD. Perfect Munler, Perfect Team. From 8:00 PM 68 183
	Hast Tang	C 10	0.64	C 81	0.0040	17 201	100/27/00 13:50:01 normal. The laundry liner has been lucked off.
	PeoTer	C 0+	C 01				2)27/00 13:49:45 earmal: Samian taday is at 6:50 AM, subset is at 5:55 PM.
binarter rest man rate first on had to first	Date Section 1	free and				12	2/27/00 13:49:05 normal: Notice, Nick has been on the computer for 8.0 hours
viegher how mail rain hain on had in the	Internet Longs	T area	5.4			- 23	+
bringher from multi sat have on half in the	MP3 Seatt	- mail	attines	Eut.		- 8 P	2/27/08 13:48:35:1
Weather from much can have no fast in the	Phone Tananta	W 8				12	K PLAY
Visible Non-rack untern en hate Ne Visible Non-rack undersen ein hate Ne Visible Non-rack undersen ein hate ist Visible Non-rack under ist Fast anticas kyr at Tast anticas kyr at Tast anticas kyr at Tast anticas kyr at Tast anticas kyr at	Thuche: 2/25 6 2399 1201 106 00 CMT 16.00 213 15.00					6	R PLAYFILE Kimp/Bruce samp/BRUCE SONGS ALLPLS
	Change 188 -2.50/12.201 (87-16.25/-6.676.8132.10/-6.176.						2/27/98 13:48:15:1
	Test 1 (a.m.)		Teta	ME, MYR	а 5 Смх	OK DELETE 02/27/08 13:48:10 Winama Billeri stadiot Dours assess all Daubhan Shree	
	Next out:					100	2/27/80 13.48.15: Faund window wisamp: 2682
	APPE Speak	Core	(E) tanks	C MS		CHE	at 02/27/00 13:48:15: Waiting for winamp
	APRIL Part	C now	Class	F 5P5	C Mt	1.45	227/00 13.48.15 Venting for whitep
Tait seturation of 40	TV menth	-	TV Arres			0	2)27/08 13:48:87: Updating rep3 database for k/mp3 [dm/d/winamp/Scoor Brj
Test selecan lgit elli	- UR Mude	C Ande	C AAM	CDI		2	2)27/08 13:47:51: Shoulcast data: playing+1, time+1443 minutes, status+24
1	And and Address of Talance	11.40 am				15	2/77/06 13.46.53 set Scartala family OFF
23.48.00 diarted	A Mana Tan	1.6.00	C 620 m	1 idea	1.20	10	2)27/08 13:46:48: set Scortain_rack OFF
	Tailout folia 1		I Banta Vit	475		100	2)27/00 13:46:37: set Scortain_nick OFF
	What are fourt 12 / 18 from the mosth used					6	0/27/00 13:46:58: W08555N-9 is traveling north at 55 mph 23.5 miles usu
1	Rain Becaul.	Statut: 0.0	/ 1.1 Bars	m. 171		30	2/27/00 13:45:34: Oops, 1 tell asleep for 7 seconds
						- 22.	

Figure 5-3. This Perl/Tk interface to MisterHouse has the same functions as the web interface. Commands are on the left, Tk widgets in the middle, and the speech and print logs are on the right

It has Internet hooks for reading and writing email, HTTP, or FTP files.

MisterHouse consists of one main Perl program called mh (no relation to the mail reading program) and a bunch of library, data, and user code files. The basic flow of mh is:

Setup

Load modules, read configuration files, open serial and socket ports. This step takes about 15 seconds on a 100-MHz Pentium and consumes about 15 MB of memory.

Read and eval user code

The user code is parsed so that all code except global variables and object declarations are put into a loop subroutine. This is then passed to eval.

Run the Loop

Each pass through the loop checks for voice commands, keyboard input, and socket/serial/file data. Then the user code loop is evaluated and global variables and objects are set for the next pass of the loop.

Using a sleep parameter, you can control how often the loop is executed. With a 100 millisecond sleep period, mh

runs about nine passes per second on a 100- MHz Pentium, using about 15% of the CPU.

The perl2exe program from http://demobuilder.com is used to create optional mh binaries for Windows and Linux, so that you can run mh even if you don't have a recent version of Perl installed, or are missing some required modules.

Objects in the Home

The best way to explain how mh uses objects and methods is with an example. This code allows for voice control of a light that is plugged into an X10 lamp module:

```
$hall_light = new X10_Item 'A1';
$v_hall_light = new
Voice_Cmd 'hall light [on,off]';
set $hall_light $state if $state = said
$v hall light;
```

The A1 string is the X10 identifier we chose for the lamp module. Each state in the Voice_Cmd enumeration creates a separate voice command. When one of those commands is recognized, the Voice_Cmd said method returns the relevant state.

The set...if...said line is put in the loop subroutine, so it is evaluated on every mh loop. If someone says "hall light on," then said will return on, and the X10_Item set method is called to turn the light on. (See Chapter 6 for more details on how the X10 protocol is implemented.)

Voice_Cmd is just one of the mh objects. Here is a list of some of the others:

File_Item

Reads, writes, and monitors data from files.

Process_Item

Creates, kills, and monitors background processes.

Serial_Item

Reads, writes, and monitors serial port devices.

Socket_Item

Creates client and server sockets.

Group

Allows for groups of objects, for easier control.

Timer

Starts, stops, and monitors timers.

Talking and Listening

The holy grail for the would-be

smart house is reliable voice input (Voice Recognition or VR) and understandable voice output (

Text To Speech or

TTS). Although this has progressed a long way in the last few years, it is not quite to the point where you can tap your Star Trek communicator badge and have a meaningful conversation about where to point your house's phasers to quiet your neighbor's dog.

You can, however, put on a microphone headset, or stand in front of a desktop microphone in a fairly quiet room, and have your house recognize specific, pre-programmed phrases. This mode of VR, called *Command and Control*, is much more reliable than dictation, in which any arbitrary text can be spoken. Dictation requires you to train your VR engines to recognize your voice; Command and Control does not.

The mh Voice_Cmd object has two types of phrase enumeration. As shown earlier, text enclosed in [] characters is used to control what is returned with the said method. To allow for variations on phrasing, you can also enclose text within { } characters. For example, this statement:

```
$v_lights = new Voice_Cmd '{turn the,}
{living,dining} room fan [on,off]';
```

would create these recognizable phrases:

```
turn the living room fan on/off
turn the dining room fan on/off
```

living room fan on/off dining room fan on/off

There are currently two VR engines that mh can use. On Windows, Microsoft has an OLE-controllable engine, and on Linux,

IBM has made ViaVoice binaries available. There are three TTS engines that mh can use: IBM's Outloud on Linux, Microsoft's TTS engine for Windows, and the

Festival TTS engine, available on a variety of platforms.

All of these engines are freely downloadable (locations are available in the mh install instructions), and you can mix and match the engine of the day. mh interacts with the ViaVoice and Festival engines through socket-based servers, so we can run mh on a different machine and a different OS. For example, I run mh on Windows because I prefer the TTS engine (it uses pre-digitized voices to give an amazingly human-like voice), but I run the VR engine on a Linux box because of the improved accuracy of the IBM ViaVoice engine.

A Smart TV Guide

I promised you some practical, don't-scare-the-spouse-away type of examples. Here is an example of how you can have your house announce when a favorite TV show is starting:

```
$favorite tv shows
mv
"voyager, dilbert, family guy";
$f tv file =
                     new File Item
"$config parms{data dir}/tv infol.txt";
if (
time cron '0,30 18-22 * * *') {
  run qq[
get tv info -times $Hour:$Minute -keys
"$favorite tv shows" ];
set watch $f tv file 'favorites now';
}
if (
changed $f tv file) {
 my @data =
read all $f tv file;
  speak "Notice, a favorite show is just
starting. @data";
}
```

The File_Item object has set_watch, changed, and read_all methods, used to monitor and read file data. The

time_cron function is used to test against a cron-formatted time string (in this case, every half hour between six and ten in the evening).

The run function creates a separate process to run the get_tv_info program, so that the main mh loop does not pause while get_tv_info runs. On Windows, run calls Win32::Process. On Unix, fork is used.

get_tv_info is a Perl program bundled with the mh distribution. It queries data from a DB_File database created by another mh event that parses TV programming info from web pages. If you have an IR output device, you can also use an mh event to automatically record shows.

Whole House CallerID

Tired of being your children's phone secretary? With this code, you can monitor incoming calls using your caller ID modem:

```
$phone modem = new Serial Item
'ATE1V1X4&C1&D2S0=0+VCID=1', 'init',
'serial3';
set $phone modem 'init' if $Startup;
# Look for modem caller ID data like this:
#
    NMBR = 507 - 123 - 4567
#
    NAME =
WINTER BRUCE
my $modem data;
if (my $data = said $phone modem) {
    $modem data .= $data;
    if (\$data =~ /NAME =/) {
           my ($caller, $number, $name) =
Caller ID::make speakable($modem data);
           speak(rooms => 'all', text =>
$caller);
logit dbm("$config parms{data dir}/phone/
callerid.dbm",
                    $cid number, "$Time Now
$Date Now $Year name=$cid name");
        undef $modem data;
    }
}
```

The if \$Startup check sends the modem an initialization string to put it into caller ID mode. The Serial_Item said method returns a string each time a record is received from the modem. Once we have the NAME record, we use the Caller_ID::make_speakable function to convert the data to into a string like: "Phone call from Larry Wall from California." It can also use an optional rule to return a phonetic version of the name (TTS engines don't do well with names like Bret Favre or Galileo Galilei), a digitized audio file, so you can have the caller announce their call using their own voice! A snapshot of the MisterHouse phone log is shown in Figure 5-4.

12 display, calls: Phone Log Display	50 R			
Inceming calls (#D entries)	Butgoing calls (DAV entries)			
C Sort by name C Sort by number C Sort by date	C Sort by name C Sort by number @ Sort by date			
Som 82/27/00 13:17:86 The humper family Som 82/27/00 12:17:55 Firston and and and and and and and and and an	1 See 01/22/100 31 50 115 The Numper Family 1 1 See 01/22/100 31 50 115 The Numper Family 1 1 See 01/22/100 31 50 115 The Numper Family 1 1 See 01/22/100 31 50 115 The Numper Family 1 1 See 01/22/100 31 50 115 The Numper Family 1 1 See 01/22/100 31 50 115 The Numper Family 1 5 State 02/22/100 31 50 115 The Numper Family 5 5 State 02/22/100 31 50 115 The Numper Family 5 5 State 02/22/100 31 50 115 The Numper Family 5 5 State 02/22/100 31 50 115 The Numper Family 5 5 State 02/22/100 31 50 116 The Numper Family 5 5 State 02/22/100 31 50 116 Family 51 80 11 5 5 State 02/22/100 31 40 116 Family 51 80 11 5 7 State 02/22/100 31 40 116 Family 51 80 11 5 7 State 02/22/100 31 40 116 Family 51 80 11 5 7 <td< td=""></td<>			
Database of callers (386 entries)	List of log files			
Search1	incoming files Outgoing files			
F Sort by name (Sort by number (Sort by # ca	115			
Last-Der 24 1999 calls- 1 WY BING Last-Last 22 2000 calls- 5 WY BING Da Last-Last 2 2000 calls- 5 WY BING Da Last-Last 2 2000 calls- 5 WY BING DA Last-May 2 1999 calls- 1 WY BING DAN Last-May 2 Nove Calls- 1 WY BING DAN Last-Last 2 WY BING Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG Calls- 2 WY BING JOINT A Last-Last 4 WY BIG CALLS- 2 WY BING JOINT A Last-Last 4 WY BIG CALLS- 2 WY BING JOINT A Last-Last 4 WY BIG CALLS- 2 WY BING JOINT A Last-Last 4 WY BIG CALLS- 2 WY BING JOINT A Last-Last 4 WY BIG CALLS- 2 WY BING JOINT A Last-Last 4 WY BIG CALLS- 2 WY BING JOINT A Last-Last 4 WY BIG CALLS- 2 WY BING JOINT A Last-Last 4 WY BIG CALLS- 2 WY BING JOINT A Last-Last 4 WY BIG CALLS- 2 WY BING JOINT A Last-Last 4 WY BIG CALLS- 2 WY BING JOINT A Last-Last 4 WY BIG CALLS- 2 WY BING JOINT A Last-Last 4 WY BIG CALLS- 2 WY BING JOINT A Last-Last 4 WY BIG CALLS- 2 WY BING JOINT A Last-Last 4 WY BING JOINT A Last 4 WY BING JOINT	Caller(d, 1999, 12, log phase, 1999, 12, log caller(d, 1999, 11, log phase, 1999, 11, log caller(d, 1999, 10, log phase, 1999, 10, log caller(d, 1999, 00, log phase, 1999, 00, log caller(d, 1999, 01, log phase, 1999, 01, log caller(d, 1999, 01, log phase, 1999, 01, log			

Figure 5-4. With a Caller ID unit, MisterHouse can log all incoming and outgoing phone calls. They can then be viewed or searched with a web interface or the Perl/Tk interface shown here (home phone numbers blurred intentionally)

If you really want to get fancy, you can wire up a relay-controlled

PA system, and modify a pa_stub subroutine so you can use the rooms option of the speak function to direct the speech to specific rooms.

Finally, the logit_dbm function logs the name and number into a DBM file so we can query it to find names and numbers of callers.

Squeaky Stairs

Is your house too new to have any of those endearing squeaky stair steps that are the bane of teenage children trying to sneak out at night? This code monitors a wireless X10 motion sensor to play back simulated digitized squeaks:

```
$timer_stair_movement = new Timer();
$movement_sensor = new
Serial_Item('XA2AJ', 'stair');
if (state_now $movement_sensor eq 'stair'
and inactive $timer_stair_movement) {
   set $timer_stair_movement 60;
   play(file => 'stairs_creak*.wav');
}
```

The Timer object is used so that we have no more than one squeak per minute, which prevents squeak-seeking kids from doing jumping jacks in front of your sensors. The state_now Serial_Item method returns stair when the motion sensor (set to code A2) is triggered and sends string A2AJ. The play function will randomly play one of the *stairs_creak*.wav* files.
You Have Mail

This code periodically checks your email account and summarizes incoming messages:

```
$p_
get_email = new Process_Item 'get_email';
start $p_get_email if $New_Minute and
!($Minute % 10) and
&
net_connect_check;
if (
done_now $p_get_email) {
    my $text = file_read
"$config_parms{data_dir}/get_email.txt";
    speak $text if $text;
&scan_subjects("$config_parms{data_dir}/
get_email.scan");
}
```

The start method of the Process_Item object runs the get_email program every 10 minutes so long as the net_connect_check function verifies that you're online. The Perl get_email program reads the headers of new email (for accounts specified in a configuration file, get_email.scan) and creates a summary in the file get_email.txt. The done_now method returns true when the get email process is finished.

get_email also creates a *get_email.scan* file that has the first few lines of each email. The scan_subjects function then checks that file for remote, password-protected commands, so you can control your house from your email-enabled cell phone!

There are other MisterHouse programs:

Vehicle Tracking

Shows how you can use a ham radio to track vehicle positions.

Event Reminders

Shows how you can program event reminders, directly and using the Outlook interface.

Time and Temperature With an Attitude

Shows how to sync up with Internet clocks and weather data.

Say What?

To give you a better feel for types of things our house says, here is a list of things I copied out of a recent speech log file:

- Turning furnace heat on after 23 hours at 67.6 degrees.
- Turning furnace heat off after 20 minutes at 69.1 degrees.
- Notice, the sun is bright at 32 percent, and it is cold outside at 24 degrees, so I am opening the curtains at 8:07 a.m.
- The van is traveling north at 58 mph 0.8 miles west of Wal-Mart.
- The car is parked at Friedell School.
- Email account laurel has 4 new email messages from The S F gals.
- Email account nick has 1 new email message from accountmanager.
- Notice, Nick has been on the computer for 2.1 hours today.
- The front door has been left open.
- Phone call from Mal and Beth Winter from Montana.
- Notice, there is 1 favorite show starting now: "Dilbert" on channel 8.

- 8:58 p.m. VCR recording will be started in 2 minutes for "South Park" on channel 47.
- Notice, there were 668 web hits from 74 clients in the last day.

Here is an example of a VR

session. To keep it from accidentally responding to human-to-human conversation, we use special phrases to activate and deactivate it:

```
Human: Hey MisterHouse.
mh: What's up?
Human: Christmas lights on.
mh: I heard christmas lights on.
Human: What is the outside temperature.
mh: The outside temperature is 26
degrees.
Human: Go to sleep.
mh: Later.
```

If you have an HTTP-capable MP3 player, you can hear what our house sounds like and optionally have a conversation with a heavily-modified Eliza ChatBot at http://misterhouse.net:81/ mh/web/speak.

Give Your House a Brain

Not ready to take the plunge yet? Consider this: For \$50 you can order from

x10.com the send-only X10

FireCracker kit (described in Chapter 6) or the more flexible send and receive

ActiveHome kit. In either case, all you need is an open serial port and a lamp or appliance you want to control. There are no wires to mess with, so you can even use it in a rented apartment or dorm room.

If you don't need or want the complexity and overhead of mh, you can use the X10 Perl modules from CPAN and write your own interface. Even if you don't need X10 control, you might still find the voice, time, and Internet functions of mh of some use. For example, my twin sons each have a bare-bones version of mh running on the computers in their rooms (they call it MisterRoom), just so they can have remote and timed control of their MP3 players (pictured in Figure 5-5).

Category Item	s Groups Widg	gets Other About SI	eech Log IV	Login	Search:	
Appliances GPS_tracking HVAC Informational Internet Lights Miss Miss Miss Miss Miss Miss Other Phone Test Timed_Events Timers TV Voice Weather	Musie Status Autho MP3 Search: MP3 Ge		Status <u>Author</u> MP3 Genu	ized re:	rock	
	monitor_shoutc	ast 🔗 Check the she	outcast player		02/27/00 13:00:16	
	in the second	mā Set Nicks mp	3 player to			
	mp3_control	Play	Set Nicks mp3 player to			
	mp3_control	Bet the house	mp3 player to	Play Stop Pause Next Song Previous Song		
	mp3_playlist	Set house mp	b player to play!			
	mp3_playlist	Load The mp3 database • Random Song				
	pa_control	nusic of		Toggle Shuffle Toggle Repeat		
	pa_control	speakers 💌			02/25/00 13:52:52 off	
	monitor_shoutcast			02/25/00 22:58:34 Connect		
	Refresh Recently Spoken Test 0 0.02/27/00 13:07:10 normal: Web access from ne mediaone net 0 0.02/27/00 13:02:32 normal: It is 73 degrees 37 percent inside, 47 degrees 34 degree windchill outside. 0 0.02/27/00 13:02:23 normal: bruce has 1 new message from 3on Orwant, 0 0.02/27/00 13:02:02 normal: 33 web hits from 8 clients in the last hour 0 0.02/27/00 13:01:48 normal: To a dog his owner is Napoleon; hence their popularity. 0					
10				E C	Wai Local intranet	Ē

Figure 5-5. All voice commands are listed by category. The Music category allows you to remotely control various MP3 players and to build and search a database of MP3 songs

You can find the mh documentation and installation instructions at the web site http://misterhouse.sf.net.

Chapter 6. Home Automation: The X10 Nitty-Gritty

Bill Birthisel

Consider small computers. No, not notebooks, subnotebooks, or even "embedded" single board controllers. Blackberry, Psion, Palm—still way too big. Game consoles, microwave oven keypads, clock radios? Getting closer. TV remotes, computer keyboards, and talking toys are at the top end of this class. The real foot soldier of home automation is the *microcontroller*—a tiny computer optimized to do a few things well. *Tiny* is the key word here—

microcontrollers need to be small, cheap, and able to run on low power.

In this article, we will introduce some of the nitty-gritty details of

working with home automation. We'll talk about two microcontroller-based boxes you can connect to your computer: one simple (the

CM17) and one complex (the

CM11). Each sends messages to devices that plug into your house power and control individual items (one lamp or appliance per device). We'll start with the CM17 and use it to describe the commands, protocols, and timing issues common to most home control installations. Then we'll look at the CM11 and the extra complications associated with receiving commands. Finally, we'll look at some of the issues a real home automation application needs to consider. Along the way, we'll examine some the Perl modules you can use to manipulate the CM17 and CM11. The examples will run on your computer even if you don't have any of the hardware, although you won't be able to "watch der blinkin' lights." In practice, the modules are commonly called from within a user environment such as MisterHouse, which Bruce Winter described in Chapter 5.

The Perl slogan, "There's More Than One Way To Do It," also applies to home automation. There are lots of vendors, lots of products, multiple protocols, and a variety of command sets. You can build kits, purchase individual components, or even buy complete packages. Some of the products are even *compatible*. I've selected specific models for this article because the CPAN modules supporting them are the most mature, but don't consider that a product endorsement. Modules for other types and other vendor protocols are in development and may be available by the time you read this. Check the ControlX10:: namespace at your favorite CPAN site.

The term "X10"

is a bit ambiguous; in this article, it refers to a mechanism for sending data messages via house wiring (

power line modulation). There is also a company named X10, Inc., that manufactures and sells the

CM11 and

CM17 controllers.

The CM17 (Figure 6-1) is a little box about twice the size of your thumb that sends on and off commands via radio to household appliances. It's sold under the trade name "FireCracker,"

and it plugs into a serial port (the little 9-pin connector on most PC hardware). It draws its power from the port—no batteries required.



Figure 6-1. The FireCracker

Simple Output Commands: The CM17

There are at least four separate communication transport systems found in home automation: radio (used by the CM17), power line (CM11 and others using the X10 protocol), infrared (TV remote controls and line of sight devices), and direct-wire (CEBus, CAN, BACnet, and other slimmed down industrial I/O protocols). The CM17 transmits radio signals to another box that powers on or off an appliance plugged into it and retransmits commands via X10 power line modulation to other device controllers. Inside each box, a microcontroller waits to decode and implement any commands sent its way. A box that converts wireless signals to X10 power line commands is shown in

wireless signals to X10 power line commands is shown in Figure 6-2.

Microcontrollers are designed to manage a small amount of rigidly defined I/O and to talk exclusively to other

microcontrollers and computers. "Spartan" is a polite description for the user interface; it's tough to implement a shell, much less a modern GUI, in 256 bytes of RAM. You program microcontrollers by toggling individual bits or sending a handful of encoded bytes to a serial port. Each microcontroller implementation is different; fortunately, the gory details can be collected in Perl modules, so most users will never have to know them.

The ControlX10::CM17 and ControlX10::CM11 Modules

The ControlX10::CM17 and ControlX10::CM11 modules are collections of functions that make it possible for you to manipulate CM17 and CM11 boxes from your Perl program. We'll see a number of examples throughout this article. The first is so simple it requires no special hardware at all—we'll simply pretend we have the necessary hardware and ask a CM17 to toggle a lamp on and off twice. (To actually turn something on and off, we'd need a serial port, a CM17, and a radio receiver that talks the same RF protocol.)



Figure 6-2. An X10 wireless link module, which takes wireless signals from a computer and converts them to X10 commands sent over household wiring

The program below, cm17_no_hardware.pl, imports SerialStub.pl (available on the web page for this book at http://www.oreilly.com/catalog/tpj3), which creates an object that behaves like a serial port with a CM17 box

plugged into it. We never talk to the actual box, but the CM17 module doesn't care—it finds the methods it wants, so it's happy.

```
#!/usr/bin/perl -w
require 'SerialStub.pl'; # Emulate port
hardware
package main;
use
ControlX10::CM17;
use strict;
my $serial object = SerialStub->new (); #
Creates object
print "Turning address A1 ON\n";
send cm17($serial object, 'A1J');
print "Turning address A1 OFF\n\n";
send cm17($serial object, 'A1K');
print
        "Repeat
                   same
                          toggle with
debugging:\n";
$ControlX10::CM17::DEBUG = 1;
print "Turning address A1 ON\n";
send cm17($serial object, 'A1J');
print "Turning address A1 OFF\n";
send cm17($serial object, 'A1K');
```

The program's output will be something like this:

```
| Turning address A1 ON
| Turning address A1 OFF
|
| Repeat same toggle with debugging:
```

```
| Turning address A1 ON
L
CM17: SerialStub=HASH(0x804a9c4) house=A
code=1J
L
CM17:
                 Sending:
done
| Turning address A1 OFF
| CM17: SerialStub=HASH(0x804a9c4) house=A
code=1K
CM17:
                      Sending:
11010101101010100110000000010000010101101
done
```

The main part of the example turns a device on and off twice. It creates an object and passes it to the send_cm17 function. We also turn on verbose output during the second toggle sequence to see how the command is translated.

What's in a Command?

The microcontroller in the CM17 uses only three of the wires in the serial port: a ground and two signals called RTS and DTR. At least one of the signals must be on at all times to provide power to the microcontroller; both are on during idle periods between commands. The command is a series of 40 pulses (a 0 bit is RTS pulsing off, and a 1 bit is a DTR pulsing off). The CM17: Sending: lines above show typical 40-bit output strings.

The send_cm17 function takes a serial object and a command string as arguments, and calls a strange selection of serial object methods. It doesn't use either the read or write lines of the serial port—it just passes them through so something else (a terminal, or device like a

CM11) can use the port simultaneously (as long as it doesn't use DTR or RTS, and as long as the operating system permits port sharing—some devices, like a serial mouse, need exclusive access).

The command_string usually contains three parts: the *house code*, the *unit code*, and the *operation*. There's one house code per house (out of sixteen possible: A..P), one unit code per appliance (1..9, A..G), and seven possible operations for the CM17, as shown in Table 6-1.

Table 6-1. CM17 operations

Operation	Function	
хJ	Unit On (requires unit code x)	
xK	Unit Off (requires unit code x)	
L	Brighten Last Light Programmed 14%	
М	Dim Last Light Programmed 14%	
N	All Lights Off	
0	All Lights On	
Р	All Units Off	

Not all receivers support operations L, M, N, and O.

This division into sixteen house (major) and sixteen unit (minor) parts is a common feature of the protocols. I'm not worried about interference here in rural Wisconsin, but the sixteen house settings mean that in a crowded apartment building neighbors are less likely to be toggling your living room lights.

Our second example, cm17_bit_toggle.pl, will toggle a real lamp or appliance on and off if you have a CM17, a transceiver, and a device controller set to address A1. An X10 lamp module is shown

in Figure 6-3.



Figure 6-3. An X10 lamp module

To run cm17_bit_toggle.pl, you'll need a serial port module installed on your system: either

Win32::SerialPort (for Windows) or

Device::SerialPort (for Unix or Mac OS X). On some operating systems, you will also need permission to open the port and access the hardware. See the documentation bundled with the

SerialPort modules for specific platform details. cm17_bit_toggle_pl will run without a CM17, but the results will be like those cm17_no_hardware.pl. (The module can't determine itself whether a CM17 is present.)

```
#!/usr/bin/perl -w
# cm17_bit_toggle.pl
# USAGE: perl cm17_bit_toggle.pl [ PORT ]
# PORT defaults to COM1 on Win32 and /dev/
ttyS0 on linux
# any command line parameter overrides the
default
# you'll need to specify which port to use
on other OS
require 'start_port.pl'; #
Initialization for real ports
```

```
use ControlX10::CM17;
use strict;
my $serial object = open port (@ARGV);
# Returns newly created object
# for example, PORT is just passed through
if specified
print "Turning address A1 ON\n";
send cm17($serial object, 'A1J');
print "Turning address A1 OFF\n\n";
send cm17($serial object, 'A1K');
print
        "Repeat
                 same toggle with
debugging:\n";
$ControlX10::
CM17::DEBUG = 1:
sleep 1;
print "Turning address A1 ON\n";
send cm17($serial object, 'A1J');
sleep 1;
print "Turning address A1 OFF\n";
send cm17($serial object, 'A1K');
```

cm17_bit_toggle.pl is almost exactly the same as cm17_no_hardware.pl, and the output on your screen will be identical. The important differences are hidden in the require. All the initialization and operating system details are provided by start_port.pl, shown below:

```
# USAGE: require 'start_port.pl';
# $port_object = open_port ($port);
#
# $port defaults to "COM1" on Win32 and
"/dev/ttyS0" otherwise
# You'll need to specify which port to use
```

```
on any OS other than linux
use vars qw($OS win);
# We start with some cross-platform black
magic
BEGIN { # Decide which
module to use based on the operating system
  | = 1;
  SOS win = (S^{O} = ~/win/i) ? 1 : 0;
  if ($OS win) { eval "use Win32::
SerialPort 0.19" }
              { eval "use
  else
Device::SerialPort 0.07" }
 die "$@\n" if $@;
}
use strict;
# open port() takes a port name as
parameter, and provides
# plausible defaults if none specified
(Win32 and linux)
sub open port {
    my $port = shift;
    my $serial port;
    if ($OS win) {
        $port = "COM1" unless ($port);
                        $serial port =
Win32::SerialPort->new ($port);
    } else {
             $port = "/dev/ttyS0" unless
($port);
                        $serial port =
```

```
Device::SerialPort->new ($port);
            print "\n=== Bypassing ioctls
===\n\n" unless $serial port->can ioctl;
    }
     die "Can't open serial port $port:
$^E\n" unless ($serial port);
    $serial port->handshake("none");
          CM17 doesn't care about other
       #
parameters unless the pass-through
    # port is used. The
CM11 doesn't need ioctls--but it does have
to
           set the traditional serial
         #
parameters like baud, parity, stop bits,
etc.
     $serial port->error msg(1);
                                          #
Use built-in error messages
    $serial port->user msg(0);
    $serial port->databits(8);
    $serial port->baudrate(4800);
    $serial port->parity("none");
    $serial port->stopbits(1);
    $serial port->dtr active(1);
      $serial port->write settings || die
"Could not set up port\n";
    return $serial port;
}
1;
```

The BEGIN block at the start of start_port.pl is deceptively simple. Win32::

SerialPort translates the primitive elements of the Windows serial driver interface into more useful methods for your Perl program. Similarly, Device::SerialPort provides portability across POSIX-compliant operating systems.

```
start_port.pl also checks that the OS-specific system
calls (ioctls) needed to run the
```

CM17 are available for your platform. If they aren't, you won't be able to control real hardware, but the examples will still run and print what they are doing. You'll see a === Bypassing ioctls === message if the module can't issue the necessary hardware pulses.

Let's see another example of the ControlX10::CM17 module. Here's slow speed.pl:

```
#!/usr/bin/perl -w
# USAGE: perl slow speed.pl [ PORT ]
# same PORT defaults as cm17 bit toggle.pl
require 'start port.pl';
                                         #
Initialization for real ports
use ControlX10::CM17;
use strict;
my $serial object = open port (@ARGV);
print "Turning address A1 ON\n";
send cm17($serial object, 'A1J');
print "Turning address A2 ON\n";
send cm17($serial object, 'A2J');
print "Turning address A3 ON\n";
send cm17($serial object,
                           'A3J');
print "Turning address A4 ON\n";
send cm17($serial object,
                           'A4J');
print "Turning address A5 ON\n";
send cm17($serial object, 'A5J');
```

print "Turning all house A OFF\n"; send_cm17(\$serial_object, 'AP'); \$serial object->close || die "\nclose

problem with port\n";

129

Timing Issues

If you run slow_speed.pl, you'll see output like this after as long as ten seconds:

```
Turning address A1 ON
Turning address A2 ON
Turning address A3 ON
Turning address A4 ON
Turning address A5 ON
Turning all house A OFF
```

Why is it so slow? The answer might seem glib: it doesn't need to be any faster. I mentioned earlier that the transceiver echoes commands to other devices via power line modulation. The X10 power line protocol transmits one bit per power line cycle and requires 47 cycles for a typical message—around a second. If you have the appropriate hardware, you can run the following experiment: set a lamp module to the same address as a transceiver with a built-in mechanical relay. When you send an on or off command to that unit, you'll hear the relay make an audible click about a second before the lamp responds. Since

microcontrollers have small command buffers, you don't want to have too many operations pending. Running the

CM17 module at about the same rate as the power line transmission reduces the amount of queueing and buffering needed.

all_together.pl is like slow_speed.pl, but groups the units into related house addresses, letting a single pair of commands simultaneously alter up to 16 appliances:

```
#!/usr/bin/perl -w
  # FILE: all together.pl
  # USAGE: perl all together.pl [ PORT ]
  # same PORT defaults as cm17 bit toggle.pl
  require 'start port.pl'; # Initialization
  for real ports
  use ControlX10::CM17;
  use strict;
  my $serial object = open port (@ARGV);
  print "address A All Lamps ON\n";
  send cm17($serial object, 'AO');
  print "address A All OFF\n";
  send cm17($serial object, 'AP');
  $serial object->close || die "\nclose
  problem with port\n";
all together.pl runs much faster
                                          than
slow speed.pl.
```

Bidirectional I/O with the CM11

Before we look at the issues involved in designing a home automation application, let's look at another type of hardware controller. The CM11 talks to a computer via a serial port, and communicates with unit modules only via household power lines. It also monitors the power lines for commands it *didn't* send, such as from motion detectors or RF transceivers, and reports those to the computer. It includes a built-in clock and even permits *cron*-like macro programming ("turn A3 ON at 09:00"). The

CM11 interface is so unlike the

CM17 that both devices can share the same serial port. Like cm17_no_hardware.pl, this program (cm11_no_hardware.pl) should run on all systems regardless of hardware:

```
#!/usr/bin/perl -w
# FILE:
cml1_no_hardware.pl
# USAGE: perl cml1_no_hardware.pl;
require 'SerialStub.pl'; # Emulate port
hardware talking to CM11
package main;
use ControlX10::CM11;
use strict;
my $serial_object = SerialStub->new ();
# creates and initializes object
```

```
my $no block = 1; # Return immediately
unless data waiting
read cm11($serial object, $no block);
# A bit toggle like that in the earlier
CM17 examples
print "----\n\n";
print "Sending A1 ON\n";
send cm11($serial object, 'A1');
send cm11($serial object,
                          'AJ');
print "Sending A1 OFF\n";
send cm11($serial object,
                          'A1');
send cm11($serial object, 'AK');
# Emulate a "data waiting to be read"
message from the CM11
print "----\n\n";
my \$incoming = chr(0x5a);
$serial object->fakeinput($incoming);
if (read cm11($serial object, $no block)) {
                          $datain
                   my
                                      =
receive cm11($serial object);
    print "Received $datain\n" if (defined
$datain);
}
print "Sending A2 OFF\n";
send cm11($serial object, 'A2');
send_cm11($serial object, 'AK');
```

A CM11 command consists of two parts: one or more addresses (a house/unit combination, like A2) and one or more operations (a house/operation combination, like AK). This is actually closer to the command signal format sent on the power lines than the 40-bit string we saw in the CM17.

The send cmll

method looks much like the send_cm17 method used in earlier examples, but it does a lot more work. It computes a checksum, validates the transmission, handles retries, and checks for incoming data. The CM11 can also initiate communication, sending a single character "read me" request when it has data waiting to be processed.

Let's look at one final example that assumes we have the hardware we need to manipulate our appliances. The result is follow_leader.pl. After initializing, the program loops, waiting for activity on unit A1. If it sees any, it sets A2 to the same state. This is a somewhat contrived example, but it illustrates some important elements of an automation control loop—in part because it spends most of its time scanning for inputs of interest. (I used the *\$block* option to get a loop timing of about a second instead of an explicit sleep 1.)

```
#!/usr/bin/perl -w
# USAGE: perl follow_leader.pl [ PORT ]
# same PORT defaults as cm17_bit_toggle.pl
# output A2 changes when the CM11 reports
a change in A1
require 'start_port.pl';  #
Initialization for real ports
use ControlX10::CM11;
use strict;
my $serial_object = open_port (@ARGV);
my $DUMMY = 0;  # Set this to true if
running without a CM11
```

```
my $reps = 30;
my \ block = 0;
my $a2 state = 'AK';
                             # OFF
my $a2 new = 'AK';
                             # OFF
while (\$reps-- > 0) {
                                    # Loop
continuously for 30 seconds
    print ".";
    if (read cm11($serial object, $block))
{
        # poll CM11 for "data waiting"
                              $datain =
                         my
receive cm11($serial object);
        if (defined $datain) {
            print "\nReceived $datain\n";
              $a2 new = 'AJ' if $datain =~
/A1AJ/;
              $a2 new = 'AK' if $datain =~
/A1AK/;
        }
    }
     $a2 new = 'AJ' if $DUMMY && $reps ==
15;
    # A2 follows detected changes in A1
    if ($a2 state ne $a2 new) {
        if ($a2 new eq 'AK') {
            print "\nSending A2 OFF\n";
            $a2 new = $a2 state = 'AK';
        } else {
            print "\nSending A2 ON\n";
            $a2 new = $a2 state = 'AJ';
        }
        send
cm11($serial object, 'A2') unless $DUMMY;
                 send cm11($serial object,
```

```
$a2_state) unless $DUMMY;
    }
print "\n";
$serial_object->close || die "\nclose
problem with port\n";
undef $serial_object;
```

If you run follow_leader.pl without a CM11, set \$DUMMY = 1; and you'll see a result like this:

```
| .....
| Sending A2 ON
| .....
```

The example simulates receiving an A1 ON halfway through, and prints a dot roughly every second to indicate cycles through the loop. With a CM11 and an external RF command unit, you'll see output like the following:

```
| .....
| Received A1AJ
|
| Sending A2 ON
| .....
| Received A1AK
|
| Sending A2 OFF
| .....
| Received A1AJ
|
| Sending A2 ON
| .....
| Received A1AK
|
| Sending A2 OFF
| ....
```

A Few More Considerations

So what other issues are involved in creating a real home automation program? In Chapter 5, Bruce Winter discusses some of the user interface issues and one approach to managing them. I'll concentrate on the issues that arise in a read/write hardware setup and those associated with multiple users as well. Most of these comments apply to both the CM11 and CM17 (and to other computer interfaces as well):

- 1. It takes a long time to talk to household appliances—so long that if you're going to be having some appliances depend on what others are doing, or if you're going to have a spiffy user interface that monitors many appliances, you need an event queue. This lets your computer perform other tasks between attempts at communication.
- 2. A real application needs to know what is turned on and what is turned off. We can keep track of what we've done to each unit, or if CM11-style two-way communication is available we can send an "are you on" status query. However, this is slow, so we want to maintain a local model of the appliances whenever possible.
- 3. If we wish to allow commands from multiple sources (say, from a web page, command line, and program loop), or from multiple users, we need to resolve issues such as who "owns" the port.
- 4. Some means of override needs to be provided in case the state data becomes stale or invalid. When the computer

is rebooted (or the program stopped and restarted), we won't know if an appliance is on or off unless we saved the previous state.

- 5. When many people have access to the same set of appliances, you might want to restrict access for certain user/appliance combinations. In this case, authentication may be required.
- 6. We have touched only lightly on error handling and event logging. The device controllers can detect many kinds of errors, including collisions on the power line and invalid commands. The CM11.pm module uses checksums to validate communications with the CM11 box, and the box itself can detect some errors and retry unsuccessful transmissions. The CM17 is a one-way device, so its error handling is more limited.

Bruce Winter has addressed most of these, and many more, in his excellent MisterHouse program described in Chapter 5. It's well worth a bit of study before you reinvent the wheel.

Chapter 7. A Day in the Life of comp.lang.perl.misc

Clinton Pierce

The fate of

Usenet has been bantered about quite a bit lately. Slashdot posters have proclaimed the death of Usenet (http://slashdot.org/article.pl?sid=00/02/04/2224201), and even a Washington Post article claims that Usenet may have outlived its usefulness (http://search.washingtonpost.com/wp-srv/WPlate/2000-02/04/0661-020400-idx.html). They both seem to suggest that Usenet

newsgroups will be replaced by special-purpose web-based discussion forums, like Slashdot. Furthermore, flamewars and a barrage of CGI questions in

comp.lang.perl.misc have had many in the Perl community wondering about the viability of comp.lang.perl.misc, and how long it can last. Figure 7-1 shows a sample posting from the newsgroup.



Figure 7-1. A question and answer on comp.lang.perl.misc

Myself, I'm a big fan of the newsgroup. I learned a lot of Perl from it, and more importantly I learned how to answer questions for new Perl programmers. I gained a lot of experience by simply lurking, and then posting, to the group. What follows is an in-depth analysis of the articles posted to comp.lang.perl.misc, all nicely trimmed and categorized—and a little speculation on what makes the group tick.

A Little History

The

Usenet group comp.lang.perl.misc was created in May 1995, deprecating the older newsgroup comp.lang.perl. The charter describes it as a newsgroup for "discussion of issues of all sorts relating to Perl," which is apt: the topics in the group range from discussion of Perl's guts to CGI programming.

The group is high-traffic, often receiving over 200 posts a day. The posters are new Perl programmers, experts, and the occasional visits by people who were looking for other groups. The "experts" can be people who are still learning Perl and happen to know enough to answer the question at hand, up to gurus who speak at conferences, write books, and hack on Perl itself.

There are six Perl newsgroups in all:

comp.lang.perl.misc

General discussion about Perl.

comp.lang.perl.moderated

Like c.l.p.misc, but higher quality.

comp.lang.perl.tk

Discussion of the Perl/Tk graphical toolkit.

comp.lang.perl.modules

Discussion of particular Perl modules.

alt.perl

Perl discussion for those who don't know about c.l.p.misc.

comp.lang.perl.announce

Certain messages selected by the newsgroup owners.

How the "Day in the Life" Was Done

Before diving into the statistics, a brief description of how they were gathered. I analyzed all the threads with messages posted on Groundhog Day, 2000. Why February 2? Because I decided to write this article on the 4th, and my local newsfeed had several days of articles remaining in the spool. This meant that I didn't have to use Deja News (now absorbed into Google) to retrieve articles. I wasn't trying to find (or avoid) flame wars; it was just a convenient day.

Any thread that was open on February 2 was considered for the statistics. According to my newsfeed, exactly 250 articles were posted on that day. Once messages posted on other days that belonged to February 2 threads were included, there were 432 articles that needed to be read, sorted, graded, and judged.

For reading the articles, I wrote a small Perl/Tk application to pop up each article one at a time along with a clickable scorecard (shown in Figure 7-2). The headers (except for the Subject: line) and the signatures were removed while I scored to help prevent bias. After years of reading c.l.p.misc, I was sometimes able to determine a poster just by his writing style, but I tried not to let that affect my grading.





Each article was judged on its own merits. The quality of the questions (except for determining whether the poster's subject line was appropriate) was ignored. If a poster flamed a good question, or counter-flamed, it was still considered a flame. Sarcastic, flame-baited answers were considered both answers and
flames—articles could fall under multiple headings—the same message could be "off-topic," "flame," and an "answer."

The Raw Statistics

There were 109 questions posted to the group in that period, of which 30 included some code. This means that the person asking the question (and later on, the person answering) included some attempt at writing the code and was seeking correction, syntax help, advice, or debugging.

17 questions were simply off-topic, having no Perl content whatsoever. The topic for c.l.p.misc is "issues of all sorts relating to Perl," so anything involving Perl is fair game. Many people in c.l.p.misc feel that CGI questions are off-topic for the group, and I do as well-except when it's about CGI.pm module usage, or Perl CGI programs that aren't working. To me, this is just the latest trend in computing. It used to be client/server applications, and next it distributed computing. Perl adapts to might be its environment so well, it's natural that c.l.p.misc's posts would drift toward current trends. And it's not just CGI. Half the off-topic posts were about SQL or databases of one kind or another, having nothing to do with Perl (or its modules) whatsoever.

How many of those 109 questions were answered? All but four of them were eventually answered with code, suggestions to move along to another group, documentation pointers, or other helpful information. This is a response rate of over 96%. Some of them took a few days to be answered, and I stopped checking after four days to see if the remaining four would be answered. Of the 109 questions, only 60 had what I'd consider to be good subject lines, where the body of the message was adequately described. (I was being quite generous.) At this point, I'd like to kick out a few soapboxes, and point out that of those 49 bad subject lines, only one was changed by the responder.

On the subject of subject lines, the articles broke down like this for on-topic articles:

209 Perl language usage, bugs, questions, and answers
29 Windows-specific questions
20 Web questions
18 Module usage questions
14 Questions about programs Perl interacts with (sendmail, shell questions, ...)
12 Books

And like this for off-topic articles:

18 Web articles (CGI, HTTP, or web server issues)
10 Database (SQL language, database server issues, ...)
9 Questions about running programs other than Perl
8 Windows, Unix, or OS conversations unrelated to Perl
4 JavaScript postings

There were 30 articles that were neither questions nor answers, just casual on-topic conversation about Perl.

The Day's Weather Report

Overall, things in the newsgroup were calm that day. 178 answers were posted to the group, and a surprising 70 of them included code written by the responder.

Flames were moderate that day as well. Only 14 articles were categorized as true flames: content-free, without any merit whatsoever and existing only to say, "NO, THAT'S WRONG!" Surprisingly (or not, depending on your view) almost all eleven flames were from channel regulars.

Finishing on a good note, there were sixteen messages that were Thank You's. Not simply "TIA" (thanks in advance), but honest thanks for information passed along. They acknowledged the advice sent, seemed to benefit from it, and were polite enough to respond accordingly. Who says civility is dead?

The group c.l.p.misc has a blessing almost unique to Usenet

newsgroups: there's almost no spam. Only three articles out of all 432 could be considered advertising, and two were on-topic. The two on-topic posts were a Call for Papers for the O'Reilly Perl Conference in Monterrey, and a job listing for a Perl programmer (which should have been posted elsewhere). The off-topic posting, way off the mark, was a broadly-crossposted Call for Papers for a USENIX Windows NT Symposium. Shame on USENIX; they should know better.

Netiquette Nits

Abusive crossposting was almost non-existent. Most of the crossposts were between

comp.lang.perl.misc and any one of alt.perl, comp.lang.perl.modules, or the obsolete comp.lang.perl. 70 articles were crossposted total. Only 5 articles were crossposted to three or more groups, and none (except the aforementioned USENIX spam) to more than 5.

Despite all of the moaning about munging email addresses when comp.lang.perl.moderated was being created, addresses are almost never hidden in comp.lang.perl.misc. By visual inspection, there were 418 unique email addresses over that period. Twenty of them were obviously forged or undeliverable. (I didn't actually send mail to the 418 addresses to verify they were real, but I did read all of the relevant articles and only counted those that were clearly munged for spam protection.)

The other netiquette gripe is about trimming quoted posts and submitting

Jeopardy-style messages—where the quoted article appears after the response, so that reading top-to-bottom the answer appears first and then the question. Only 248 of the 314 articles with any quoting had the quoted material trimmed down at all. Only one person who employed Jeopardy-style quoting trimmed the quoted material.

Of the 66 articles that weren't trimmed at all (but contained quoted material), 37 were Jeopardy-style and 29 were traditional (quoted material, then response). This seems to

support the notion that Jeopardy-style posters do not trim at all, and make up the majority of those posters who don't trim.

And what about the FAQs? Of the 92 on-topic questions, only 15 were on the FAQ list (the "core" FAQ, the Win32 FAQ, or the Idiot's Guide). Five of the remaining questions were questions about the FAQ—all of them because the FAQ sometimes presents simple and complex answers to the same question without clear explanation as to which should be used. In one respondent's words, "The FAQ tries to be too cute sometimes." This is not a new complaint. The 72 remaining questions could not clearly be answered by any of the FAQs. What I unfortunately did not analyze were the proportion that could have been answered by reading the documentation. (That is, questions that were clearly about well-documented features and did not need further explanation by someone more experienced in the field.)

The Regulars

Who answers all of these questions? The group has a core of regulars, to be sure. These folks have been reading c.l.p.misc for years and still find time to reply to questions and to monitor the group. Tom

Phoenix, Randal

Schwartz, and Larry

Rossler, to name a few. These guys were posting when I was just learning Perl and they're still going strong.

Of course, there are some who have dropped out. Larry Wall no longer posts to

comp.lang.perl.misc, and recently Tom Christiansen has dropped out as well. The high volume of posts, the number of novices, and the frequency of FAQs contributes to the dropout rate. The comp.lang.perl.moderated group and the perl5-porters mailing list remain refuges for those who want to hear these voices.

In addition to the hardcore old-timers are those who answer a question here and there. As the core gets whittled down, they find themselves taking on more and more questions.

Why do people post? A quick survey of the regulars appearing in Greg

Bacon's comp.lang.perl.misc statistics turned up a few reasons. Foremost seems to be that they're doing this as a learning experience for themselves. As Jonathon

Stowe put it, "Every program I write now—I treat it as if it were going to be exposed to the criticism of c.l.p.m. and so apply those coding standards to everything I write."

There is also a wish not to see Perl misrepresented, and genuine desire to help others. The last reason common to all the regulars who answered was to participate in the Perl community in some small way.

The Future of comp.lang.perl.misc

Where does the group go from here? I asked the regulars if they were satisfied with the state of comp.lang.perl.misc, and almost universally they answered no. The answers were not unexpected:

- Posters need to read more documentation before posting.
- Posters need to lurk, or search newsgroup archives before posting.
- Posters need to stay on topic.

For comparison's sake, I showed the messages to other Usenet old-timers and the reaction was, "Well, these apply to *any* group on Usenet..."

Some of the regulars have taken fanatical measures to ensure that the group stays on track, and this lends a reputation (deserved or not) to c.l.p.misc that it's a newbie-hostile group. However, like any other static institution bombarded with change and youth, it has been changing and flexing to suit its environment. When the newbies wear down the old regulars, new regulars step in and take up the slack. I believe the group is doing just fine, and isn't getting any worse or better over the years. A quick survey of February 2, 1998 and February 2, 1996 shows that the group hasn't changed much. Some of the regulars have gone, others have arrived. The topics have changed (more DBI and web, less networking and system administration), but the basic themes haven't.

It would be nice if newbies lurked more, and read the documentation before posting. But even so, the group itself

seems to be responding well and shaping itself to the trends it encounters.

comp.lang.perl.misc should continue to be a useful resource for a long time to come.

Author's Note: Since this article was written, followup research shows that the types of articles posted to comp.lang.perl.misc remains the same, but with small changes: scattered throughout the postings are mentions of Perl 6, the "regulars" I mentioned have rotated out, and the number of CGI questions has dropped off slightly. The most noticeable difference, though, is the volume: on any given day in 2002, there are only half as many active threads as there were on the same day in 2000. Perhaps Usenet is being replaced by other information sources such as specialized web sites (Perl Monks, http://learn.perl.org) or mailing lists, or people are finally reading the documentation.

Part II. Science

In this part:
Chapter 8
Chapter 9
Chapter 10
Chapter 11

In this section, four articles demonstrate some scientific applications of Perl, with two hobbyist-oriented topics followed by two professional topics. John Redford begins with an article about how he designed and built a sundial for his backyard, using Perl to give it accuracy down to the minute. Brad Murray and Ken Williams follow with a tutorial on genetic algorithms in Perl, showing how to breed functions that incrementally evolve toward a desired solution.

One of the most popular articles in TPJ's history was Lincoln Stein's Chapter 10, which chronicled how he used our favorite language to glue together disparate data formats originating in genome laboratories scattered around the globe. Astronomers Karl Glazebrook and Frossie Economou conclude the chapter with a description of the Perl Data Language, an extension to Perl optimized for manipulating large data sets such as high-resolution pictures.

Chapter 8. Perl-fect Sundials

John Redford

This article describes a Perl 4 program for making sundials that can depict the current time to about a minute's accuracy. Typical

sundials are only accurate to about half an hour, but the Perl sundials I'll describe correct for the latitude and longitude, and even for the eccentricity of the earth's orbit.

To refresh your memory, sundials point to the current time with a shadow. In Figure 8-1 you can see the tip of the shadow of the *gnomon*—the upright part of a sundial—falling on a marking that indicates the time.



Figure 8-1. A sundial with a gnomon

For our sundials, the gnomon's shadow will fall on a plate marked with lines of "constant time." The plate markings are generated in Adobe PostScript by the Perl program, and can then be plotted and transferred to a brass sheet or some other weatherproof surface. Figure 8-2 shows such a sundial. Because the sun occupies any particular position in the sky exactly twice per year, two dials are needed: one for the days between the

winter solstice and the

summer solstice (December 21 to June 21), and one for the other half of the year. The first dial is used in the winter and spring, when the sun is descending to its lowest angle at the winter solstice. The second dial is used in the summer and fall, when the sun is rising toward its highest angle at the summer solstice.



Figure 8-2. A Perl sundial, with accuracy down to the minute

Before describing how to generate such dials, I'll provide a little

background.

Sundials are just about the oldest functioning machines known. Dials have been found in Egypt dating from the reign of Thutmose III in around 1500 BCE, and mention of them also appears in Chinese writings from around that time. The Greeks and Romans used them extensively, although their hour lines were of equal spacing and therefore do not represent intervals of equal time.

One of the most famous dials of antiquity, the

Tower of the Winds, still stands in Athens. It's an octagonal tower about fifteen meters high and five across, built around 100 CE. It has a sundial inscribed on each of its eight faces, and is surmounted by a frieze showing figures representing the eight winds. It uses iron rods for

gnomons, which clearly have not been out in the rain for two thousand years, but the dials themselves could date back that far. This is one of the only Roman devices still working (besides perhaps a few

aqueducts).

The oldest dial in the U.S. is said to be in Salem, Massachusetts, dating from 1650. Massachusetts can also claim the favorite poet of gnomonists,

John Greenleaf

Whittier, who wrote in 1881:

With warning hand I mark time's rapid flight,

From life's glad morning to its solemn night.

Yet through the dear God's love I also show

There's light above me by the shade below.

This appears as a motto on nineteenth century dials all over the country, and manages to pack a quadruple metaphor into only four lines.

Sundials are compared to a prophet, to the course of life from birth to death, to the promise of heaven, and finally to its contrast with death. Sundials are traditionally associated with mortality, of course, which may be why they appealed to the Puritan strain in New England. It may also be why they are so commonly used as garden ornaments; they act as a *memento mori* in the midst of all the greenery. This association is especially evident in a giant dial at a Vietnam War memorial in Kentucky. An entire plaza is marked off as a sundial, with a giant gnomon at one end. The names of all the local men who died in the war are inscribed on the plaza, and the tip of the gnomon's shadow brushes across each name on the day of the man's death.

Most dials, though, are simply public sculpture. The Royal Observatory at Greenwich, England, has a striking dial with a broad cylindrical plate below two arching dolphin tails acting as

gnomons. The Museum of Science in Worcester, Massachusetts, has a dial where the observer acts as the gnomon. You reach up and put your finger at the appropriate place on a crossbar, and your finger's shadow marks the time. The Mount Laguna Observatory dial projects a shadow on a cylindrical map of the world, which shows not only the time, but the place in the world where the sun is directly overhead. Baltimore has a dial with seventeen facets, showing the time in Baltimore, Tokyo, Jerusalem, Sitka, Pitcairn, San Francisco, Cape Cod, Rio Jancito, London, Fernando Po, Cape Town, and Calcutta; it was built by the man who supplied the granite for the Library of Congress.

Building Your Own

So how can you actually draw these dials? The mathematics is actually quite straightforward. The key is being able to express the position of the sun as a function of date and time. Three kinds of motion have to be taken into account: the rotation of the earth during the day, the orbiting of the earth around the sun, and the slight speedup and slowdown caused by the

eccentricity of the earth's orbit.

First we pick a coordinate system. Imagine that you are standing at the North Pole. Almost straight above your head is Polaris, the North Star. On the

vernal equinox (March 21), one of the two days when day and night are of exactly equal length, the sun skims exactly along the horizon for 24 hours. Call the "up" direction the Z axis, making the X and Y axes horizontal. Let the X axis point to where the sun sits on the horizon at six o'clock in the evening on the vernal equinox. With X and Z fixed, Y then points to where the sun is at noon. Our coordinate system is shown in Figure 8-3.

Now the position of the sun can be represented as a vector in X, Y, and Z. At 18:00 on the equinox, the sun is directly along the X axis, so the

sun vector S is (1, 0, 0). In Perl we write this as:

@S = (1, 0, 0); # at 18:00 on equinox

At 12:00, the sun is on the Y axis:

@S = (0, 1, 0); # at 12:00 on equinox

The sun vector rotates clockwise around the Z axis according to the time of day, going 360 degrees in 24 hours. In polar coordinates, the angle between the sun and the X axis is called the *right ascension*, and the angle between the sun and the horizontal plane is called the *declination*. However,

Cartesian coordinates are easier to deal with, as we'll see later on.



Figure 8-3. Sun position at different times and dates at the North Pole

So let's define a function, RotZ, that rotates a three-element vector @V by an angle a in radians:

Now we can express the sun vector on the equinox as a function of the time in hours, \$h, as:

```
@S = &RotZ (($h - 18) * $twopi/24, (1, 0,
0));
```

This takes a unit vector (1, 0, 0) at 18:00, and rotates it by the number of hours after 18:00.

However, the sun is not fixed compared to the background stars—it moves all the way around the sky in the course of a year. Of course, it's the earth that's actually moving around the sun, but for our purposes the situation is symmetric, so we can pretend the sun is moving. The sun moves along a path called the *ecliptic*, which is tilted at an angle of about 23.5 degrees to the horizon at the North Pole. At its highest point on the summer solstice (June 21), the sun will be 23.5 degrees above the horizon all "day" long, and at its lowest point on the winter solstice (December 21), it'll be 23.5 degrees below. To work this motion into

the sun vector, an initial vector can be rotated around Z by a date angle, and then the result rotated by the ecliptic angle around X:

Note that the date angle will be zero on the vernal equinox.

One other correction needs to be made. The earth does not move at equal speed along its orbit for the whole year. It moves fastest when the ellipticity of its orbit brings it closest to the

sun, and slowest when it is farthest away. This causes the sun to be a few minutes ahead of or behind where it would be if the orbit were a perfect circle. The point of closest approach is called *perihelion*, and occurs on December 31. The most distant point is called *aphelion* and occurs on July 1. The dates of perihelion and aphelion change slowly over time, as does the

eccentricity of the earth's orbit. (The combination of these two effects produces cycles hundreds of thousands of years long. These are called *Milankovitch cycles*, and are thought to have enough influence on the earth's climate to have been a cause

of the Ice Ages.) A first order approximation of the effect of the eccentricity can be expressed by:

where *\$eccentricity* is 0.0169, the ratio of the distance between the ellipse's foci and its major axis. Circles have an eccentricity of 0, and straight lines have an eccentricity of 1. *\$Aphelion* is the number of days from January 1 to July 1.

The above equations can be combined to give a vector for the sun at any time and day at the North Pole. Since few readers live there, however, we must adjust for other latitudes and longitudes. The latitude can be handled by another rotation, this time around the X axis:

```
#latitude angle in radians
$l_angle = (90 - $latitude) * $twopi/360;
@S = &RotX ($l angle, @S);
```

The longitude is a little more complicated, since clock time matches sun time only in the center of time zones. Each time zone is 15 degrees of longitude wide (360 degrees of longitude divided by 24 time zones), and is centered on every fifteenth degree. Thus Greenwich Standard Time is plus or minus 7.5 degrees of longitude zero, Eastern Standard Time is plus or minus 7.5 degrees of longitude 75 degrees, and so on. The initial hour time must therefore be corrected for the distance from the central longitude:

```
$h_angle += ($longitude -
$CentralLongitude) * $twopi * 15 / 360;
```

The central

longitudes for Eastern, Central, Mountain, and Pacific Standard Time are 75, 90, 105, and 120 degrees respectively. For the UK the central longitude is zero (since the British came up with this scheme they got to put themselves in the center), for Japan it's -135 (longitude east of Greenwich is negative), and for eastern Australia it's -150. (One should also correct by an hour if Daylight Savings Time is in effect. Unfortunately, DST goes into effect on different days in different countries.)

After all these transformations, we finally have a vector for the sun at a particular day, time, and place. We now need to find the position

of the tip of the gnomon on the sundial plate. This is where the Cartesian representation of the vector becomes convenient. Figure 8-4 shows how the

tip vector T is the sum of the gnomon vector G, minus the sun vector scaled by a multiplier m. Since the gnomon and the tip vectors are at right angles, their dot product is zero.



Figure 8-4. Vector diagram of sun, gnomon, and shadow

In vector notation this can be written:

T = G - mS

and:

 $G \bullet T = 0$

To solve for *m*, we can perform a dot product on both sides:

$$G \bullet T = G \bullet (G - mS)$$

=> 0 = G • G - m(G • S)
=> m = |G| / Sz

If *G* is defined to be the local unit Z axis (i.e., the gnomon sticks straight up and is of unit length), then G = (0,0,1), and in Perl we can write:

```
QT = (-\$S[0] / \$S[2], -\$S[1] / \$S[2], 0);
```

And that's it! We can now position any kind of mark anywhere on the dial starting with the date and time. We can mark exact dates and times, or draw segments between points for lines of constant time or date. The full code for finding the sun vector is in the routine GetSunV in the code listing, and the tip vector is in GetTipV.

In Figure 8-5, the hyperbolas are lines of constant date. That is, they track the

gnomon shadow on the days of the summer solstice and winter solstice. The S-shaped lines are of constant time, where the thick lines mark the hours, and the dashed lines are at ten minute intervals. The little round mark shows the position

of the gnomon, and the bar above it shows how long it should be. To use the dial, just install a gnomon of the correct length, lay the dial exactly horizontally in a sunny place, and rotate it until the time read matches a clock's time. The time can be matched down to the minute. The precision is limited by the fuzziness of the tip of the shadow. Since the sun isn't a point source, the shadow tip isn't perfectly sharp.



Figure 8-5. PostScript image generated by sundial.pl

The above math can even be extended to handle dials that are not horizontal. Take the

sun vector and rotate it around Z to handle dials that are not aligned north-south, then rotate it around X by whatever angle it is tilted at. This allows one to create dials on surfaces other than simple planes: cubes, icosahedrons, buckyballs, and so on.

To see more examples of what people have done with sundials, check out the web page of the British Sundial Society at http://www.sundials.co.uk. They have pointers to pictures and descriptions of dials all over the world. The U.S. and Canada have the North American Sundial Society (NASS) at http://www.shadow.net/~bobt/nass/nass.htm. They publish a quarterly journal called *The Compendium*, a compendium being "a single instrument incorporating a variety of dial types and ancillary tools."

The full listing of my program can be found on the web site for this book: http://www.oreilly.com/catalog/tpj3. The entire program is a few hundred lines long, but you've seen the key lines above. (Like most software, 10% of the code does 90% of the work.) Still, those few lines are the distilled result of thousands of years of mathematics and astronomy. What used to require the career of an expert mathematician now takes a dozen lines of Perl. Modern techniques like vector algebra, and modern machinery like Perl and PostScript, can bring a very old instrument to life again.

Chapter 9. Genetic Algorithms

Brad Murray

Ken Williams

In most programming tasks, we humans carefully state the problem, code an algorithm to solve the problem, and turn the computer loose to run the algorithm. Then we hope the correct solution pops out, and if it doesn't we send an irate bug report to the people who developed our programming language.

Genetic algorithms (

GAs) offer a different approach: we still carefully state the problem, but we let the computer find a suitable algorithm and apply that algorithm to the problem to produce a solution. Generating algorithms programmatically usually means working with code as data, which has traditionally left it more in the realm of LISP and related languages. But since Perl can generate and evaluate code on the fly, it's capable of handling generalized GAs.

The core principle behind a GA is that of

evolution. You start with a set of

random *organisms*, each of which is a program. You then run each of these programs and determine their *fitness*, which is the degree to which they succeed at the required task. Once the

fitness of each is determined, you jump through some hoops to remove bad entries (natural selection), randomly permute some remaining entries (mutation), and intermingle other entries (

hot algorithmic sex).

After repeating this process through several generations, the population will begin to converge on a solution. Often, it is not quite the solution you were expecting, and therein lies a lot of the fun in building a GA engine.

The Genetic Code

The

genetic code we will use to describe our organisms has many parallels to genetic code found in real plants and animals. The processes we set in motion when simulating evolution often look like biological processes, too. However, there is also a huge number of differences between real and simulated evolution. Draw analogies at your own risk.

The GA task we'll set before ourselves in this article is to find algebraic expressions that can generate a certain predetermined set of data points when applied to a given set of input data—otherwise known as "curve fitting."

We'll call these algebraic expressions our *organisms*, and we'll represent them as simple binary

syntax trees composed of *functions* and *terminals*. Each function has two branches representing its arguments, and each argument can either be another function or a terminal. Terminals are dead-end leaf nodes and are usually constants or one of the input parameters in the problem. For instance, in an organism represented by the algorithm

add (multiply (2, x), 5), we have the functions add and multiply, and the terminals 2, x, and 5.

Each problem requires some twiddling by the

Genetic Engineer. You need to determine a set of functions and terminals that will be part of your organisms' makeup. You might think of these as roughly equivalent to the various base pairs in DNA, or perhaps to various short sequences of base pairs. For the example we'll show here, we're trying to fit a function to the data:

```
-1407, -931, -577, -327, -163, -67, -21,
-7, -7, -3, 23, 89, 213, 413, 707, 1113,
1649
```

which just happens to be the output of $3x^3 + 2x^2 - x - 7$ applied to each integer on the interval [-8,8]. To hit this (and we'll assume that we have no idea what the solution should look like), we'll use a function set that includes:

```
sin(x,y)
log(x,y)
mul(x,y)
add(x,y)
```

We know that the real solution only needs the last two; the first two are there just to confuse the GA. We picked sin(x, y) because some regions on the graph look like they might be sinusoidal. We picked log(x, y) just to drive up the computing time and make things interesting.

We know that sin(x, y) and log(x, y) look odd, since sin and log only take one argument apiece. But by our definition, each function in our syntax tree has two branches. In the case of unary functions, we simply throw the second argument away when computing, but it is valuable to keep it in the data structure since a mutation may change a sin to an add and suddenly make the second argument interesting. As the second argument could be a whole tree of functions on its own, this could get very interesting indeed.

So given these functions, we build an array of functions as strings. Remember we said that we needed code as data? Here it is, to be eval'ed later:

```
my @functions = (
    # Format is 'function <pretty version>:
<actual code>'
    'function add($a,$b): ($a+$b)',
    'function mul($a,$b): ($a*$b)',
    'function sin($a,$b): sin($a)',
    'function log($a,$b): log( $a ? abs($a)
: 1 )',
);
```

Notice that our log function is protected from zero or negative values. Any function that has sensitive arguments needs to be appropriately protected, since we don't want to waste computing cycles on organisms that are just going to blow up when we evaluate them. In general, you want to make sure that all your functions can handle any data thrown at them—if they can't, they're going to explode as their complexity grows.

Next we need terminals. For this exercise we have two kinds: x (the parameter that we will vary over the interval of interest; i.e., the input to these algebraic expressions) and a constant integer between -9 and 9. We specify these as subroutine references:

```
sub { int(rand(20) -
10) } ); # Some random number
```

which return either the name of the parameter ("x") or the random number.

Now, what do our organisms *look* like? Well, as we said, they're syntax trees that looks something like Figure 9-1.



Figure 9-1. A syntax tree organism

This

corresponds

to

mul (add (sin (x, 7), -5), mul (x, x)), which in turn reduces to $(sin (x) - 5) x^2$), which is, of course, wrong. Let's see if evolution can help.

Assembling an Organism

We'll represent each organism as a Perl object. If making little

genetic organisms isn't a good opportunity to use object-oriented programming, we don't know what is.

Now, it's one thing to know what a bridge looks like, but designing and building a bridge is something else altogether. We think it's safe to say the same is true of organisms (unless you—ahem—go about it in the "natural" way). We'll build ours

recursively, with some sanity checks.

First we check against a maximum depth value and plug in only terminals past this point. This keeps the tree from getting too crazy at the outset (it will get crazy enough later). If we are inside the maximum depth, we randomly select a function or terminal, with an arbitrary 2:1 weight toward terminals. If we selected a function, then we call the organism's _buildTree method again to get two more nodes to use as input to the function. And so on.

This builds a hash of nodes for us, each of which has three components: $tree{contents}$, which contains either a terminal value (a constant or x in this case) or a function, and $tree{'a'}$ and $tree{'b'}$, which are references to other nodes. If the content is a terminal, left and right pointers are not generated.

Survival of the Fittest

Just generating random organisms is not enough. We need to rank them according to their

fitness, so we can decide which to cull. We also need to determine

fitness so that we know when to stop: unlike real

evolution, we are trying to hit a fixed target. Evolution, however, is a feedback mechanism, and is therefore designed to hit a moving target. This means that once we reach a perfect fit for the data, the algorithm will keep trying new things even though the current fit is perfect. This will result in the program oscillating around the correct answer, which doesn't help. If you are trying to find an algorithm to hit a moving target, you still need to know the fitness at each generation, though you will probably have to do some statistical work on your results in order to find the mean success rate over the changing target.

We calculate the fitness by averaging the unsigned difference between each fixed data point and the corresponding result of the organisms' function (its phenotype). Thus fitness is a non-negative number, and a fitness of zero indicates a perfect organism. To calculate the

output of the syntax tree, we have a function called fitness:

```
sub fitness { # Determine the fitness of
an organism in this crazy world
    my ($org, @target) = @_;
    my $sumdiff = 0;
```

```
foreach (0..$#target) {
    $sumdiff += abs($org->
evaluate({'x'=>$_}) - $target[$_]);
    }
    return $sumdiff/@target;
}
```

fitness repeatedly calls the organism's evaluate method at points on the interval that interest us. Think of the evaluate method as the organism's whole reason for existence; if the organism were a leech, the method would be called suck_blood. If it were a gerbil, the method would be called freak_out. The method simply applies the embedded behavior of the organism to the given input data. In this case, we evaluate a single-variable algebraic expression for a given number.

The evaluate method is just a simple front end that calls Perl's eval on the result of the organism's expression method.

```
sub evaluate {  # Find the value of the
organism on the input data
  my $self = shift;
  my $params = shift;
  return $self->code->($params->{x});
}
sub code {  # Turn the syntax tree into a
Perl coderef
  my $self = shift;
  # Check the cache
    return $self->{code} if defined
$self->{code};
```

```
my $expr = $self->
as string;
  return self -> \{code\} = eval "sub \{my \ x
= shift; return $expr}";
}
sub as string {  # Turn the syntax tree
into a readable expression
 my $self = shift;
 my $tree = shift || $self->{tree};
  # Check the cache
    return $self->{string} if defined
$self->{string};
    local $ = $tree->{contents}; # A
shortcut for the current node
  if ( s/^function [^:]*:\s*(.*)/$1/ ) { #
Extract the format picture
                            s/\ ([a-zA-Z]+)/
$self->as string($tree->{$1})/ge;
  }
    $self->{string} = $ if $tree eq
$self->{tree}; # A nasty trick
 return $ ;
}
```

Since as_string works on a recursive data structure, it's natural that it's a recursive method. If the current node represents a function, we scan the function description and pull out the Perl expression that implements that function. Then we replace any embedded variables (a or b) with the

Perl expression their syntax tree represents. If the current node is a terminal, we leave it alone.

We happen to particularly like the way the as_string method combines its recursion and

caching techniques.

Building the Perl code

from the syntax tree is a pretty intensive process, and we don't want to do it over and over again for the same organism, so we cache our results. We only want to put the result in the cache when we're done with the work; that is, when we exit from the topmost call to this recursive subroutine. We detect whether that's the case by comparing the node we're currently working on to the topmost node in \$self 's tree. If they're the same, we cache and finish. The trick is to compare these two references as strings. When a reference is used in a string context, it contains a representation of the reference's memory address, so if the two references evaluate to the same string, they're the same reference. Sneaky. Most of the time this type of caching requires a wrapper function to handle the caching and a recursive function to do the real work, but we're getting away with a single function.

So now that we know how to evaluate the success or failure of each organism, we need to do something about it.

Sex and Mutation

It won't be enough to just throw out the bottom half of the list and generate a new set of random organisms from scratch. This would be akin to trying to make a watch from a bucket of parts by shaking the bucket until a watch spontaneously assembles, or trying to cook a pot of soup by repeatedly throwing away the worst-tasting parts and adding random objects from the world. Evolution doesn't work that way, and we don't have enough time to wait for pure randomness to cough up the algebra we want.

Our strategy is to rank the organisms by fitness and perform three operations on the list:

- 1. Cull some bad organisms from the bottom of the list.
- 2. Mutate some percentage of the remainder by randomly changing a node on the syntax tree.
- 3. Mate some individuals with some others to produce offspring with similar attributes.

Culling is simple: we just unshift the same number of organisms that we are going to add by mating. In the case of our example, we have set the parameter \$mates to 5, so we remove that many individuals. In point of fact what we actually do is mate and pop five times. Same thing.

Mutating is also pretty straightforward: we grab an organism at random and mutate it with its mutate method:
This is a little deceptive. What it does is generate a list of the nodes in the syntax tree with _treeIndex, and then pick a random one and substitute a new randomized branch. This mutation mechanism is pretty drastic, so we don't do it often. The likelihood of improving an organism is very small, though it is important to keep some random element of change happening within the population to keep it from settling at some local maximum that fits the function well, but not as well as we want.

The clear_cache method simply clears the cached information we built in the expression method, since it's no longer valid. The index generator looks like so:

```
# Generates a list of all the nodes in
this tree. These are
# references to the stuff in the object
itself, so changes to the
# elements of this list will change the
object.
sub _treeIndex {
    my $self = shift;
    my $tree = shift || $self->{tree};
    my @sofar = @;
```

```
# Dump the content nodes into a list
if ($tree->{contents} =~ /^function /) {
   return(@sofar,
        $tree,
        $self->_treeIndex($tree->{'a'},
@sofar),
        $self->_treeIndex($tree->{'b'},
@sofar)
        );
   } else {
      return(@sofar, $tree);
   }
}
```

And naturally it is recursive as well. Maybe this article should have been about recursion instead, but we were confident that you already knew all about recursion.

```
Finally, we take the top n
```

organisms and

mate them with a random other organism from the list. Each mating involves taking a tree index of each partner, selecting a random point in each, and grafting the end of one list on to the beginning of the other. This is known as a *crossover permutation*, which is similar to the

genetic scrambling that occurs in sexual reproduction.

The mate method also uses _treeIndex:

```
sub mate {
  my ($self, $partner) = @_;
  my $self_clone = bless { tree => dclone
$self->{tree} }, ref($self);
```

```
#Get part of a node from $partner and
stick it somewhere in $self_clone
```

This is not so different from the

mutation, except that we know that each chunk of the new tree previously belonged to some relatively successful individual, so the chances are higher that the result will also be fit than if we did a random mutation. Note that we use dclone from the Storable module to create deep copies of complex structures rather than write our own cloning routine. This is because we are lazy.

The Terrifying Results

When all of this is plugged into a framework that iterates over generations, tests results, and dumps the results on screen, we discover that there is an awful lot of incomprehensible output. This output needs to be examined by hand to determine what's going on, although I'm sure that someone out there is willing to donate routines to assist in the analysis. One run we did for 1,000 generations came up with the following results.

By generation 50 the GA had stumbled on a 12-node equation that simplified to $3x^3$, which gave it a mighty good fit over the complete interval. This is somewhat deceptive, as the method we use for testing fitness is inordinately happy with fitting the ends where the numbers are huge. A better fitness test would weight the targets evenly so that the fit in the middle would be better. Still, this gave us a substantially complete fit and showed that the GA could establish the order of the target polynomial very rapidly.

By generation 200 the GA had found the second term—the results reduced to $3x^3 + 2x^2$. Again, given the fitness calculation it seemed unlikely that it would discover any further refinements of interest as the "value" of such refinements was so small. This outlines the importance of selecting a fitness function that reflects your needs! Our fitness function basically said that we are interested primarily in matching the broad strokes of the curve without regard for the details, which is what we got.

By termination the GA had started fiddling with sines and natural logs in order to better fit the middle region of the graph. This dead end caused it to actually diverge from the real curve outside of the tested interval, while improving the fit inside—while it better met our stated criteria, it was actually taking us further afield. Again, it shows how important it is to tell computers what you want rather than make them guess.

Overall, the convergence to our stated criteria was very rapid and effective, as we had very solid fitness levels and the order of the polynomial correctly established by generation 50.

Other Applications

This GA engine could easily be extended to do different work. Simply fitting functions to data is cute, but by altering the function set, the terminal set, and the evaluation process, the GA can be used to generate behavioral algorithms for robots, control system solutions, and so on. Not that it's going to be easy: your evaluation routines will become very complex for non-mathematical projects, but it's still feasible. Examples from John

Koza's

Genetic Programming include:

Emulate food foraging behavior in ants

The terminal set is MoveRandomly, MoveToNest, PickUp, and DropPheromone. The function contains set custom routines IfCarryingFood, IfFoodHere, MoveToAdjacentFoodElse, MoveToAdjacentPheromoneElse, and Progn (which executes a list of instructions—we think we can just drop it and eval everything, but we haven't tried. You tell us).

It's not instantly clear how to do this, but it should be possible. We look forward to hearing the results of your own experiments on the Perl-AI mailing list!

Find the Fourier Series for a given periodic function

The terminal set is just a constant from -10.000 to 10.000.

The function set contains XSIN, XCOS, +, -, *, and %. XSIN and XCOS are two-argument functions taking the coefficient and the harmonic (truncated to an integer). There are plenty more, including ones that develop neural nets and other sophisticated decision-making or controlling algorithms. We recommend you pick up Koza's book to investigate further.

Going Further

As research in

genetic programming has advanced, some new mechanisms have been considered. One of the more interesting ones is "islanding." This involves running more than one population in isolation from others, and occasionally migrating some small number of the fittest individuals from one "island" to another.

When you first run your GA, you will probably notice that the speed of convergence and the quality of the result depends quite heavily on the initial set of organisms and the nature of the first few mutations and matings. This is probably a flaw in our engine somewhere, but if it is fundamental to

GAs, then islanding provides a way to occasionally introduce "ideas" to each system that are both good and novel. In theory this should increase the fitness of each population.

You have probably also noticed that we have built the GA so that it can be distributed—because the algebraic expressions are implemented as strings to be evaluated rather than as references to subroutines, expensive runs can be designed so that the organisms are serialized and shipped to separate processors for the fitness runs. We haven't done this yet, so we're looking forward to either your results or your donations of hardware to facilitate the project. A farm of Sparcs would be fine.

Other Fitness Functions

For the purposes of this article, we thought it best to use a relatively naïve fitness calculation (just add the error at all the data points), rather than spend time talking about how to come up with a better function. If you experiment with various other methods for calculating the fitness, you may achieve better results or faster convergence. In particular, you might add the squares of the calculated error rather than just the absolute values of the error. Let us know how it works out!

The full program described in this article, genetic.pl, is available on the web page for this book: http://www.oreilly.com/catalog/tpj3.

Resources

- Genetic Programming, John R. Koza, MIT Press.
- The Perl-AI mailing list: perl-ai@perl.org.
- Illinois Genetic Algorithms Lab (IlliGAl): http://gal4.ge.uiuc.edu/.
- Perl's Storable module: http://www.perl.com/CPAN.

Chapter 10. How Perl Saved the Human Genome Project

Lincoln D. Stein

Author's Note: It is now six years since I wrote this article, and though much has changed, a surprising amount has remained the same. The human genome was successfully sequenced about a year ago, thanks in no small part to thousands of Perl scripts large and small, and the human genome project has now spawned genome sequencing projects for such organisms as the mouse, the chicken, the cow, the mosquito, the honeybee, the chimp, and—believe it or not—the duck-billed platypus.

The BoulderIO system described in the body of the text has long since been supplanted by a powerful and flexible body of code called BioPerl (http://www.bioperl.org), the collective work of dozens of committed programmers and biologists.

Perl remains the savior of the genome project now more than ever. Just a few weeks ago I found myself sitting in an auditorium listening to Jim Mullikin of the Wellcome Trust Sanger Institute describe how he had solved a problem that was once thought insurmountable: to assemble an entire genome (the mouse, in this case) in a single shot, without the tedious experimental mapping and subcloning that was previously thought to be critical to make the problem soluble. His genome assembly software, named Phusion, is a pipeline of Perl scripts wrapped around a nugget of high-performance *C* code. As Jim put it, "Perl and 70 gigabytes of main memory is all you need!"

DATE: February 1996

LOCATION: Cambridge, England, in the conference room of the largest

DNA sequencing center in Europe.

OCCASION: A high-level meeting between the computer scientists of this center and the largest DNA sequencing center in the United States.

THE PROBLEM: Although the two centers use almost identical laboratory techniques, almost identical databases, and almost identical data analysis tools, they still can't interchange data or meaningfully compare results.

THE SOLUTION: Perl.

The human genome project was inaugurated in the early 1990s as an ambitious international effort to determine the complete DNA sequence of human beings and several experimental animals. The justification for this undertaking is both scientific and medical: by understanding the genetic makeup of an organism in excruciating detail, it's hoped that we'll be better able to understand how organisms develop from single eggs into complex multicellular beings, how food is metabolized and transformed into the constituents of the body,

how the nervous system assembles itself into a smoothly functioning ensemble. From a medical perspective, the wealth of knowledge that will come from knowing the complete DNA sequence will greatly accelerate the process of finding the causes of (and potential cures for) human diseases.

Six years after its birth, the

genome project was ahead of schedule. Detailed maps of

the human and all the experimental animals had been completed (

mapping out the DNA using a series of landmarks is an obligatory first step before determining the complete DNA sequence). The sequence of the smallest model organism, yeast, was nearly completed, and the sequence of the next smallest, a tiny soil-dwelling worm, wasn't far behind. Large scale sequencing efforts for human DNA were soon to be in full swing.

The scale of the human

DNA sequencing project is enough to send your average Unix system administrator running for cover. From the information-handling point of view, DNA is a very long string consisting of the four letters G, A, T, and C (the letters are abbreviations for the four chemical units forming the "rungs" of the DNA double helix ladder). The goal of the project is to determine the order of letters in the string. The size of the string is impressive, but not particularly mind-boggling: 3×109 letters long, or some three gigabytes of

storage space if you use one byte to store each letter with no compression techniques.

Three gigabytes is substantial, but certainly manageable by today's standards. Unfortunately, this is only what's required to store *finished* data. The storage needed to determine this sequence for *experimental* data is far vaster. The essential problem is that

sequencing technology was limited to reading stretches of at most 500 contiguous letters. In order to determine sequences longer than that, the DNA must be sequenced as small overlapping fragments called "reads"

and the jigsaw puzzle reassembled by algorithms that look for areas where the sequences match. Because the DNA sequence is nonrandom (similar but not-entirely-identical motifs appear many times throughout the genome), and because DNA sequencing technology is noisy and error-prone, one ends up having to sequence each region of DNA five to ten times in order to reliably assemble the reads into the true sequence.

This increases the amount of data by an order of magnitude. On top of this is all the associated information that goes along with laboratory work: who performed the experiment, when it was performed, the section of the genome that was sequenced, the identity and version of the software used to assemble the sequence, comments anyone wants to attach to the experiment, and so forth. In addition, one generally wants to store the raw output from the machine that performs the sequencing. Each 500 letters of sequence generates a data file 20–30 kilobytes in length!

That's not the whole of it. It's not enough just to determine the sequence of the DNA. Within the sequence are functional areas scattered among long stretches of nonfunctional areas. There are genes, control regions, structural regions, and even a few viruses that got entangled in human DNA long ago and persist as fossilized remnants. Because the genes and control regions are responsible for health and disease, one wants to identify and mark them as the

DNA sequence is assembled. These annotations generate yet more data.

Altogether, people estimate that some one to ten terabytes of information will need to be stored in order to see the human genome project to its conclusion.

Bioinformatics and Perl

So what's

Perl got to do with it? From the beginning, researchers realized that informatics would have to play a large role in the genome project. An informatics core formed an integral part of every genome center that was created. The mission of these cores was two-fold: to provide computer support and databasing services for their affiliated laboratories, and to develop data analysis and management software for use by the genome community as a whole.

The initial results of the informatics groups' efforts were mixed. Things were slightly better on the laboratory management side of the coin. Some groups attempted to build large monolithic systems on top of complex relational databases; they were thwarted time and again by the highly dynamic nature of biological research. By the time a system that could deal with the ins and outs of a complex laboratory protocol had been designed, implemented, and debugged, the protocol had been superseded by new technology and the software engineers had to go back to the drawing board.

Most groups, however, learned to build modular, loosely-coupled systems whose parts could be swapped in and out without retooling the whole system. In my group, for example, we discovered that many data analysis tasks involve a sequence of semi-independent steps. Consider the steps that one may want to perform on a bit of DNA that has just been sequenced.

First, there's a basic quality check on the sequence: is it long enough? Are the number of ambiguous letters below the maximum limit? Then there's the "vector check."

For technical reasons, the human DNA must be passed through a bacterium before it can be sequenced (this is the process of cloning). Not infrequently, the human DNA gets lost somewhere in the process and the sequence that's read consists entirely of the bacterial vector. The vector check ensures that only human DNA gets into the database. Next, there's a check for repetitive sequences. Human DNA is full of repetitive elements that make fitting the sequencing jigsaw puzzle together challenging. The repetitive sequence check tries to match the new sequence against a library of known repetitive elements. A penultimate step is to attempt to match the new sequence against other sequences in a large community database of

DNA sequences. Often a match at this point will provide a clue to the function of the new DNA sequence. After performing all these checks, the sequence along with the information that's been gathered about it along the way is loaded into the local laboratory database.

The process of passing a

DNA sequence through these independent analytic steps looks kind of like a pipeline, and it didn't take us long to realize that a

Unix pipe could handle the job. We developed a simple Perl-based

data exchange format called *boulderio* that allowed loosely coupled programs to add information to a

pipe-based I/O stream. BoulderIO is based on tag/value pairs. A

Perl module makes it easy for programs to reach into the input stream, pull out only the tags they're interested in, do something with them, and drop new tags into output the stream. Any tags that the program isn't interested in are just passed through to standard output so that other programs in the pipeline can get to them.

Using this type of scheme, the process of analyzing a new DNA sequence looks something like this (this is not exactly the set of scripts that we use, but it's close enough):

A file containing the new DNA sequence is processed by a Perl script named name_sequence.pl, which has one job: giving the sequence a new unique name and putting it into BoulderIO format. The output looks like this:

```
NAME=L26P93.2
SEQUENCE=GATTTCAGAGTCCCAGATTTCCCCCAGGGGGTTTCCAGAGAGCC
```

The output from name_sequence.pl is next passed to the quality checking program, which looks for the SEQUENCE tag, runs the quality checking algorithm, and writes its conclusion to the data stream. The data stream now looks like this:

```
NAME=L26P93.2
SEQUENCE=GATTTCAGAGTCCCAGATTTCCCCCAGGGGGTTTCCAGAGAGCCC
QUALITY_CHECK=OK
```

Now the data stream enters the vector checker. It pulls the SEQUENCE tag out of the stream and runs the vector checking algorithm. The data stream now looks like this:

```
NAME=L26P93.2
SEQUENCE=GATTTCAGAGTCCCAGATTTCCCCCAGGGGGTTTCCAGAGAGCCC
QUALITY_CHECK=OK
VECTOR_CHECK=OK
VECTOR_START=10
VECTOR_LENGTH=300
```

This continues down the pipeline, until at last the load_lab_database.pl script collates all the data, makes some final conclusions about whether the sequence is suitable for further use, and enters all the results into the laboratory database.

One of the nice features of the BoulderIO format is that multiple sequence records can be processed sequentially in the same Unix pipeline. An = sign marks the end of one record and the beginning of the next:

```
NAME=L26P93.2
SEQUENCE=GATTTCAGAGTCCCAGATTTCCCCCAGGGGGTTTCCAGAGAGCCC
=
NAME=L26P93.3
SEQUENCE=CCCCTAGAGAGAGAGAGAGCCGAGTTCAAAGTCAAAACCCATTCTCC
=
```

There's also a way to create subrecords within records, allowing for structured data types.

Here's an example of a script that processes the BoulderIO format. It uses an object-oriented style, in which records are pulled out of the input stream, modified, and dropped back in:

```
use Boulder::Stream;
$stream = new Boulder::Stream;
while
($record=$stream->read_record('NAME','SEQUENCE'))
{
    $name = $record->get('NAME');
    $sequence = $record->get('SEQUENCE');
    ...continue processing...
    $record->add(QUALITY_CHECK => "OK");
    $stream->write_record($record);
```

}

The interesting thing is that multiple

informatics groups independently converged on solutions that were similar to BoulderIO. For example, several groups involved in the worm

sequencing

project began using a data exchange format called .*ace*. Although this format was initially designed as the data dump and reload format for the

ACE database (which is tailored to biological data), it happens to use a tag/value format that's very similar to BoulderIO. Soon *.ace* files were being processed by

Perl script pipelines and loaded into the ACE database at the very last step.

Other Uses for Perl

Perl found uses in other aspects of

laboratory management. For example, many centers (including my own) used Web-based interfaces for displaying the status of projects and allowing researchers to take actions. Perl scripts are the perfect engine for Web CGI scripts. Similarly, Perl scripts run email database query servers, supervise cron jobs, prepare nightly reports summarizing laboratory activity, create instruction files to control robots, and handle almost every other information management task that a busy genome center needs.

So as far as laboratory management went, the informatics cores were reasonably successful.

Systems integration, however, was not so rosy.

The problem will be familiar to anyone who has worked in a large, loosely organized software project. Despite best intentions, the project begins to drift. Programmers go off to work on ideas that interest them, modules that need to interface with one another are designed independently, and the same problems get solved several times in different, mutually incompatible ways. When the time comes to put all the parts together, nothing works.

This is what happened in the

genome project. Despite the fact that everyone was working on the same problems, no two groups took exactly the same approach. Programs to solve a given problem were written and rewritten multiple times. While a given piece of software wasn't guaranteed to work better than its counterpart developed elsewhere, you could always count on it to sport its own idiosyncratic user interface and data format. A typical example is the central algorithm that assembles thousands of short DNA reads into an ordered set of overlaps. At last count there were at least six different programs in widespread use, and no two of them use the same data input or output formats.

This lack of interchangeability presents a terrible dilemma for the

genome centers. Without interchangeability, an informatics group is locked into using the software that it developed in-house. If another genome center develops a better software tool to attack the same problem, a tremendous effort is required by the first center to adopt that tool.

The long-range solution to this problem is to come up with uniform

data interchange standards that genome software must adhere to. This would allow common modules to be swapped in and out easily. However, standards require time to agree on, and while the various groups are involved in discussion and negotiation, there is still an urgent need to adapt existing software to the immediate needs of the genome centers.

Here is where

Perl again came to the rescue. The Cambridge summit meeting that introduced this article was called in part to deal with the data interchange problem. Despite the fact that the two groups involved were close collaborators and superficially seemed to be using the same tools to solve the same problems, on closer inspection nothing they were doing was exactly the same. The four main software components in DNA sequencing projects are:

- A trace editor to analyze, display, and allow biologists to edit the short DNA read chromatograms from sequencing machines.
- A read assembler, to find overlaps between the reads and assemble them together into long contiguous sections.
- An assembly editor, to view the assemblies and make changes in places where the assembler went wrong.
- A database to keep track of it all.

Over the course of a few years, the two groups had developed suites of software that worked well in their hands. Following the familiar genome center model, some of the components were developed in-house while others were imported from outside. Perl was used as the glue to fit these pieces together. Between each pair of interacting modules were one or more Perl scripts responsible for massaging the output of one module into the expected input for another.

When the time came to interchange data, however, the two groups hit a snag. Between them they were now using two trace editors, three assemblers, two

assembly editors, and (thankfully) just one database. If two Perl scripts were required for each pair of components (one for each direction), one would need as many as 62 different scripts to handle all the possible interconversion tasks. Every time the input or output format of one of these modules changed, 14 scripts might need to be examined and fixed.

The two groups decided to adopt a common data exchange format known as

CAF (an acronym whose exact meaning was forgotten during the course of the meeting). CAF would contain a superset of the data that each of the analysis and editing tools needed. For each module, two Perl scripts would be responsible for converting from CAF into whatever format Module A expects ("CAF2ModuleA") and converting Module A's output back into CAF ("ModuleA2CAF"). This simplified the programming and maintenance task considerably. Now there were only 16 Perl scripts to write, and when a module changed only two scripts would need to be examined.

This episode is not unique. Perl has been the solution of choice for

genome centers whenever they need to exchange data, or to retrofit one center's software module to work with another center's system.

So Perl has become the software mainstay for computation within genome centers as well as the glue that binds them together. Although genome informatics groups are constantly tinkering with other high-level languages such as Python, Tcl, and Java, nothing comes close to Perl's popularity.

How has Perl achieved this remarkable position? Several factors are responsible:

 Perl is remarkably good for slicing, dicing, twisting, wringing, smoothing, summarizing, and otherwise mangling text. Although the biological sciences do involve a good deal of numeric analysis now, most of the primary data is still text: clone names, annotations, comments, bibliographic references. Even DNA sequences are text-like. Interconverting

incompatible data formats is a matter of text mangling combined with some creative guesswork. Perl's powerful regular expression matching and string manipulation operators simplify this job in a way unequalled by any other modern language.

- Perl is forgiving. Biological data is often incomplete, fields can be missing, or a field that is expected to be present once occurs several times (because, for example, an experiment was run in duplicate), or the data was entered by hand and doesn't quite fit the expected format. Perl doesn't particularly mind if a value is empty or contains odd characters. Regular expressions can be written to pick up and correct a variety of common errors in data entry. Of course this flexibility can be also be a curse. I talk more about the problems with Perl below.
- Perl is component-oriented. Perl encourages people to write their software in small modules, either using Perl library modules or with the classic Unix tool-oriented approach. External programs can easily be incorporated into a Perl script using a pipe, system call, or socket. The dynamic loader introduced with Perl 5 allows people to extend the Perl language with C routines or make entire compiled libraries available for the Perl interpreter. (The world's collected wisdom about biological data has been gathered into a bundle of modules called "BioPerl.")
- Perl programs are easy to write and fast to develop. The interpreter doesn't require you to declare all your function prototypes and data types in advance, new variables spring into existence as needed, calls to undefined functions only cause an error when the function is needed. The debugger works well with

Emacs and allows a comfortable interactive style of development.

• Perl is a good

prototyping language. Because Perl is quick and dirty, it often makes sense to prototype new algorithms in Perl before moving them to a fast compiled language. Sometimes it turns out that Perl is fast enough so that the algorithm doesn't have to be ported; more frequently one can write a small core of the algorithm in C, compile it as a dynamically loaded module or external executable, and leave the rest of the application in Perl. For an example of a complex

genome mapping application implemented in this way, see http://www.genome.wi.mit.edu/ftp/pub/software/ RHMAPPER/.

• Perl is a good language for CGI scripting, and is growing in importance as more labs turn to the Web for publishing their data.

Problems with Perl

My experience in using Perl in a genome center environment has been extremely favorable overall. However, I find that Perl has its problems too. A relaxed programming style can lead to many errors that more uptight languages would catch. For example, Perl lets you use a variable before it has been assigned—a useful feature when that's what you intend, but a disaster when you've simply mistyped a variable name. Similarly, it's easy to forget to declare a variable local to a subroutine, so that a global variable is modified instead.

If one uses the -w switch religiously and turns on the use strict vars pragma, Perl catches these problems (and others). However, there are more subtle gotchas in the language that are not so easy to fix. A major one is Perl's lack of

type checking. Strings, floats, and integers all interchange easily. While this greatly speeds up development, it can cause major headaches. Consider a typical genome center Perl script responsible for recording the information of short named subsequences within a larger

DNA sequence. When the script was written, the data format was expected to consist of tab-delimited fields: a string followed by two integers representing the name, starting position, and length of a DNA subsequence within a larger sequence. An easy way to parse this would to split into a list like this:

```
($name, $start, $length) = split("\t");
```

Later on in this script, some arithmetic is performed with the two integer values, and the result is written either to a database or STDOUT for further processing.

Then one day the input file format changes without warning. Someone bumps the field count up by one by sticking a comment field between the name and the first integer. Now the unknowing script assigns a string to a variable that's expected to be numeric, and silently discards the last field on the line. Rather than crashing or returning an error code, the script merrily performs integer arithmetic on a string, assuming a value of zero for the string (unless it happens to start with a digit). Although the calculation is meaningless, the output may look perfectly good, and the error may not be caught until well downstream in the processing.

Nevertheless, when the genome project was foundering in a sea of incompatible data formats, rapidly-changing techniques, and monolithic data analysis programs that were already antiquated on the day of their release, Perl saved the day. It's not perfect, but Perl fills the needs of the genome centers remarkably well, and is usually the first tool we turn to when we have a problem to solve.

Chapter 11. PDL: The Perl Data Language

Karl Glazebrook

Frossie Economou

Since this article was originally written, PDL has evolved considerably and adopted a large modular structure. The PDL functions now number in the hundreds, preventing us from listing them all. However, the principles behind PDL are unchanged, and this article remains a good introduction to them. It has been slightly updated to reflect the latest PDL version.

Extolling the virtues of Perl and its many uses to the readers of TPJ is preaching to the converted. Nevertheless, there is one fundamental area in computing where Perl has been conspicuously absent:

number crunching and data analysis.

Tarred by the same brush as other scripting languages, Perl (which in fact is semi-compiled), is perceived as too slow and memory-devouring for heavy numerical computations because it doesn't lend itself to storing and retrieving zillions of numbers quickly. This has been a source of great frustration to the authors, both enthusiastic Perl (ab)users who resent being forced to use more primitive environments for their

astronomical data analysis. Perl's potential for manipulating numerical data sets speedily and elegantly via an extension was obvious. Hence PDL, the

Perl Data Language, was born. PDL is a Perl extension, so you get the convenience of programming in Perl with the speed of compiled C.

PDL introduces a new data structure: the "pdl numerical array," often referred to as a "piddle." (This unfortunate nickname has led to some rather dubious puns in the source code.) Anyway, a piddle is a special object that can contain a large block of efficiently-stored numbers for manipulation with normal mathematical expressions. For example, if a is a piddle containing a 3 x 4 x 6 chunk of data, then the Perl statement b = sin(a) will do exactly what you think: set b equal to a but with every value replaced by its sine. Easy—and because each operation is implemented via compiled C code, it's nearly as fast as a hand-crafted C program.

The peridi Shell

PDL can be used normally from a script—simply use PDL. But it also has a

shell interface for interactive

data analysis and prototyping. Here we'll play with the PDL shell, called peridl, which we'll invoke from the command line. (This article assumes you have PDL-2.0 or later, PGPLOT-2.0 (which itself requires the pgplot graphics library), and

Perl 5.003 or later. If you also have the right versions of the Term::ReadLine and Term::ReadKey modules, the perldl

shell allows interactive command line editing.)

% perldl

The peridl shell behaves like Perl's debugger. For instance, we can assign values to variables and print them with p:

```
perldl> $b = 2
perldl> p $b/2
1
perldl> p $b/3
0.6666666666666667
perldl>
```

Since PDL is really about

matrices, let's create a 2 x 3 matrix and multiply it by itself:

```
perldl> $a = pdl [5,3,4], [6,4,3]
perldl> print $a
[
    [5 3 4]
    [6 4 3]
]
perldl> $b = $a * $a
perldl> print $b
[
    [25 9 16]
    [36 16 9]
]
```

But to have true fun with PDL, we'll first need some data. Luckily the PDL distribution comes with a picture of the sky stored in

FITS, the standard format for

astronomical data. PDL also supplies rfits, a function that reads FITS files and returns a piddle containing the data. So let's read in our image and plot it:

```
perldl> $a = rfits "PDL1.11/m51.fits"
IO loaded
BITPIX = 16, size = 65536 pixels
Reading 131072 bytes
BSCALE = 1.00000000000 && BZERO =
0.000000000000000
```

Now we have data—and we didn't have to spend three nights freezing up a mountain to get it. What do we know about it? That it is 16-bit with 65,536 elements. But is it 65536 x 1 or 256×256 or even $16 \times 16 \times 16 \times 16?$

perldl> **p dims \$a** 256 256

Not surprisingly (after all, it's a picture of the sky) we have a square two-dimensional image: 256 x 256. dims is a PDL function that returns the dimensions of a piddle. But what about the data values?

```
perldl> stats $a
Mean = 104.193572998047, RMS =
67.425420896103, Median = 88
Min = 24, Max = 500
```

stats is a

PDL function that returns statistical information about a piddle. We can even print some of it now—Jon might get upset if we displayed 65,536 numbers, so let's go for the bottom left corner instead:

```
perldl> p sec($a, 0, 3, 252, 255)
[
   [50 51 54 53]
   [50 50 53 54]
   [51 52 53 52]
```

```
[54 53 54 51]
]
```

sec returns a section of a piddle; the above statement displays the rectangle between elements (0,252) and (3,255). Additional dimensions are handled seamlessly: we just pass the extra coordinate values as arguments. (There are more compact ways of specifying such slices.)

Perhaps you're getting restless at this point. Let's abandon the function calls and jump to the cool stuff.

```
perldl> use PDL::Graphics::PGPLOT
perldl> imag $a
Loaded PGPLOT
Displaying 256 x 256 image from 24 to 500
...
```

This pops up a window displaying Figure 11-1.



Figure 11-1. The Messier 51 spiral galaxy

My god, Dave, it's full of stars! And so it should be—this is in fact an image of Messier 51, a

spiral galaxy similar to our own but at a distance of 200,000,000,000,000,000 miles, give or take a few billion. That's too far for us to invade, but we can at least humiliate it, as shown in Figure 11-2.

```
perldl> imag sin(0.05 * $a)
Displaying 256 x 256 image from
-0.999990224838257 to 0.999992072582245 ...
```



Figure 11-2. The Messier 51 galaxy, humiliated

Since we're exploring cosmology, let's create something out of nothing:

perldl> \$r = rvals zeroes 20,20

As you can see,

PDL functions can be chained together just like

Perl functions. Two PDL functions are cascaded here: rvals and zeroes. First, zeroes creates a piddle full of zeroes—in this case, a 20 x 20 matrix with every element zero. (There's also a ones function.)

Then rvals fills that piddle with values representing the distance of each element from the center.

perldl> \$g = exp(-(\$r/6)**2)/108.08
perldl> imag \$g

which displays Figure 11-3.

Alert readers will note that the exp function was used to generate a two-dimensional

Gaussian. The less mathematically inclined will say it looks like a blob. Let's inflict a bit more punishment on Messier 51 by *convolving* it with our newly-created Gaussian filter. This enables us to simulate what we would see if we were observing through very bad viewing conditions, such as a (possibly drunken) haze (Figure 11-4).

```
perldl> use PDL::ImageND
perldl> $b = convolve $a,$g
perldl> imag $b
```

You might want to know that this operation takes 0.5 seconds on a 600 Mhz iBook with PDL. Doing this with a 2D array in normal (non-PDL) Perl takes 25 times longer and uses 11 times as much memory. Is that cool or what?

Figure 11-5 shows an unsharp masked image, often used in astronomy to emphasize sharp features against a bright background, such as stars in a galaxy, the giant luminous gas

clouds we call HII regions, or foreground objects such as UFOs (er, weather balloons).

```
perldl> imag $a-$b
```



Figure 11-3. A two-dimensional Gaussian blob



Figure 11-4. Convolving a Gaussian with an image to blur the image
Listing of a Few PDL Functions

Starred items (e.g., log10*) act as mutators: when you say log10(inplace(a)), every element of a is replaced with its logarithm base 10.

Table 11-1 shows some of the PDL functions defined in PDL::Core.



Figure 11-5. The FireCracker

Table 11-1. PDL::Core functions

Function	Meaning
+ - * / > < >= <= << >> & ^ == != += -= *= /= %= **= <<= >>= &= = ^= <=> ** % ! ++ ``" atan2* sqrt* sin* cos* log* exp* abs*	Array operators/ functions (same as Perl and C but they act element by element)
x	Matrix multiplication
~	Matrix transpose
byte short ushort long float double convert	Type conversions
pdl	Create/copy a piddle
topdl	Coerce to piddle if scalar
howbig	Size of piddle datatype in bytes
nelem	Number of elements

Function	Meaning
dims	Return list of dimensions
inplace	Perform operation in place
list	Convert piddle to list, e.g., for (list \$x) { }
listindices	Return list of index values (1D)
log10*	Take log base 10
min max sum	Min/max/sum of piddle
zeroes ones	Create zero/ one-filled piddle
sequence	Create sequence-filled piddle

Function	Meaning
reshape	Reshape the dimensions of a piddle
sec	Subsection of a piddle
set	Setting values of data or subsection
at	Return pixel value at (x, y, z,)
axisvals* xvals* yvals* zvals*	Fill piddle with axis values
rvals	Fill piddle with distance from its center
callext	Call external C code in dynamically loadable object
convolve	Convolve image with

Function	Meaning
	kernel (real space)
hist	Histogram of data
stats	Return mean and standard deviation
transpose	Matrix transpose
qsort*	Quick sort piddle
median	Median of piddle
oddmedian	Lower odd median of piddle

Some functions from other PDL modules are shown in Table 11-2.

Table 11-2. Key functions in modules other than PDL::Core

Module	Function	Meaning
PDL::Primitive	fibonacci*	Compute Fibonacci series
PDL::Image2D	cc8compt*	Connected 8-component labelling
PDL::Io::Misc	rfits	Read a FITS format file
	wfits	Write a FITS format file
	rcols	Read columns in a text file into piddles
	rgrep	Read regexp matches into piddles
PDL::Graphics::PGPLOT	imag	Display an image
(PGPLOT graphics)	ctab	Load an image color table
	line	Plot vector as connected points
	points	Plot vector as points
	errb	Plot error bars

Module	Function	Meaning
	cont	Display image as contour map
	bin	Plot vector as histogram, e.g., bin hist \$data
	hi2d	Plot image as 2D histogram
	poly	Draw a polygon
	vect	Display two images as a vector field
	hold	Hold current plot window range, e.g., for overlays
	release	Autoscale new plot window for each command
	rel	Synonym for "release"
	env	Define a plot window, put on "hold"
PDL::Math	asin*	Inverse sine
	sinh*	Hyperbolic sine

Module	Function	Meaning
	bessj0*	Standard Bessel function
PDL::Fit::Polynomial	fitpoly1d	Fit polynomial to 1D data

Where Are We Now?

As of May 2002, the latest version of PDL is 2.3.2, supporting 3D graphic manipulation using OpenGL, an interface to the SLATEC numerical library, many other graphics and maths libraries, access to the TIFF, GIF, PostScript, and other formats supported by PBM+. PDL also features virtual slicing, easy C extensions, and fast implicit looping.

Anyone wishing to discuss the rather technical issues surrounding PDL development is welcome to join the perldl mailing list at perldl-request@jach.hawaii.edu or the porters list at pdl-porters-request@jach.hawaii.edu. Send your questions to the former and your complaints to the latter. Finally, the obligatory URL: http://pdl.perl.org. The location of the web site is shown in Figure 11-6.



Figure 11-6. Location of PDL web server

Part III. Language

In this part:
Chapter 12
Chapter 13
Chapter 14
Chapter 15
Chapter 16
Chapter 17
Chapter 18
Chapter 19
Chapter 20
Chapter 21
Chapter 22
Chapter 23
Chapter 24
Chapter 25
Chapter 26

In this section, Perl demonstrates what makes it the language of choice for manipulating language, with fifteen articles covering everything from state-of-the-art research in natural language processing and speech synthesis to practical problems like formatting text and matching names.

Natural language processing—getting a computer to understand human language—is one of those fields that seems easy at first but is actually fraught with difficulties. NLP textbooks often demonstrate the perversity of English with sentences like "Colorless green ideas sleep furiously," which is grammatical but nonsensical; "The horse raced past the barn left," which seems ungrammatical but isn't; and "Time flies like an arrow," which is perfectly good English but has four competing interpretations.

The section begins with two articles about programs that converse: John Nolan's article on a bot that dispenses psychiatric advice, and Kevin Lenzo's article on the purl bot, which helps out Perl novices on Internet Relay Chat. The ever-prodigious Kevin follows up with another of the research areas that he pursues at Carnegie Mellon: open source speech synthesis in Perl.Next, Prof.Damian Conway shows you how to format text automatically with Text::Autoformat, which manipulates the indentation, quoting, bulleting, and margins of text.

Linguist Sean Burke has six articles in this book—more than anyone else—all of them about language in one form or another. The next two articles, on music and Braille, demonstrate "little languages" constructed for a specific purpose.

NLP hacker Dan Brian follows with two articles on using Perl to give your computer programs an understanding of English. The first article is about his Lingua::Wordnet module, which gives your programs the ability to use relationships between words—to know which are synonyms and antonyms, which are subsets and supersets, and so on. Dan's second article is about the Lingua::LinkParser module, which provides a Perl interface to the most popular natural language system available. Prof.Khurshid Ahmad and Duncan White follow with an article on using morphology—the structure of words—to begin with a word (e.g., "compute") and generate related words from it ("computes," "computing,"

Next up is Brian Lalonde, who dissects the tricky problem of matching variations on human names. This is not as easy as it sounds: "Bill Gates" is the same person as "William Gates III", and someone named "Peggy" can also go by "Margaret." Simple regular expressions won't suffice; you need a little intelligence to reliably match names.

Sean Burke returns to help you ready your programs for the 5.7 billion people who don't speak English as a first language, with articles on localization and internationalization. He follows with an article on simulating typos, comparing the standard QWERTY keyboard to the Dvorak keyboard, with a brief excursion into Dutch, Italian, and Tibetan. Dave Cross continues with an article on how to correct typos in subroutine names. Even if you don't mistype subroutine names frequently, every Perl coder should be aware of the uses to intercept nonexistent AUTOLOAD trick he subroutines. Finally, Tuomas Lukka concludes the article with a description of how he learned Japanese via his program, which automatically translates Japanese into English as he surfs the Web.

Chapter 12. Chatbot::Eliza

John Nolan

Of all the

chatterbots—programs that converse with humans—Eliza is the most famous. The original Eliza was written by Professor Joseph

Weizenbaum of MIT and described in the *Communications of the ACM* in 1967 (Vol. 10, No. 8). This program is older than I am, and yet remains fascinating to this day. It's one of the all-time classic programs in computer science. Eliza pretends to be a Rogerian psychiatrist; whatever the human says, it replies—usually with a question—in an attempt to encourage the patient to elaborate.

The

Eliza algorithm has been cloned dozens of times in all kinds of programming languages, including Fortran, Basic, Pascal, C, Java, and JavaScript. The first Eliza was written in a Lisp-like language called MAD-Slip, way back in pre-Unix days. (Eliza is named after Eliza Doolittle, the cockney-speaking woman taught to speak proper English in G.B. Shaw's book *Pygmalion*.)

While taking a course in Natural Language Processing, I was surprised to find that much of the research in the field still uses Lisp. Lisp is a fine language, but Perl can do anything Lisp can do, and Perl source code is much easier to read. I searched the Web for Eliza clones, but I couldn't find any written in Perl. So I wrote one. The

Chatbot::Eliza module is a faithful clone of Weizenbaum's Eliza algorithm. It encapsulates Eliza's behavior within an object.

You can install Chatbot::Eliza just like any other Perl module. Once installed, this little bit of code is all you need to start an interactive Eliza session:

```
use Chatbot::Eliza;
$mybot = new Chatbot::Eliza;
$mybot->command interface;
```

Let's see what this looks like. If you install the Chatbot::Eliza module from CPAN, save the three lines of code above to a file, and then execute it, here's what the output looks like:

```
Eliza: Please tell me what's been bothering you.
you:
```

This is an

interactive session; type your reply to

Eliza after the you: prompt. Here's a sample run:

```
Eliza:
        Please
                 tell
                             what's
                                      been
                        me
bothering you.
you: Well, it's my boss.
Eliza: Your boss?
you: Yes. He's not very friendly.
Eliza: I see.
you: He is always upset.
Eliza: When?
       Always. Especially when things are
vou:
hectic.
Eliza: Can you think of a specific example?
      Well, yesterday he said he was unhappy
vou:
```

```
with all of us.
Eliza: That is interesting. Please
continue.
you: He says I am too lazy.
Eliza: Is it because you are too lazy that
you came to me?
you: I'm not lazy.
Eliza: Do you believe it is normal to be
not lazy?
```

You can set a few parameters of your Eliza object, such as its name, or a

configuration file for it to read:

```
$myotherbot = new
Chatbot::Eliza "Brian", "myscript.txt";
$myotherbot->command interface;
```

In this way, you can customize what the chatterbot says by providing your own configuration file. This consists of a list of

keywords,

decomposition rules, and

reassemble rules. If you don't like Eliza's default rules, you can write your own.

For instance, the following lines in *myscript.txt* would have Eliza (or Brian, as we've called it above) begin with one of two otherworldly greetings, chosen at random:

```
initial: Greetings Earthling!
initial: Take me to your leader!
```

Chatbot::Eliza contains a default configuration file with default greetings, salutations, quit -equivalents, and rules for determining how Eliza should converse. If you want to

watch Eliza think, you can turn on the debugging output before you launch your session:

```
$mybot->debug(1);
$mybot->command interface;
```

Let's look at the session we saw earlier but with debugging turned on:

```
vou:
       He is always upset.
       1> always
                : * : O He is always upset
               --> When ?
Eliza: When?
you: Always. Especially when things are
hectic.
        1> always
                : * : 0 Always
                 --> Can you think of a
specific example?
Eliza:
       Can you think of a specific
example?
vou:
         Well, yesterday he said he was
unhappy with all of us.
        2 > was
               : * was i *
                : * i was *
                : * was you *
        -1> xnone
                : * : 0 xnone
                --> That is interesting.
Please continue.
Eliza:
          That is interesting. Please
continue.
```

you: He says I am too lazy.

How It Works

The

Eliza algorithm is actually relatively straightforward. It consists of three steps:

- Search the input string for a keyword.
- If we find a keyword, use the list of "decomposition rules" for that keyword, and pattern-match the input string against each rule.
- If the input string matches any of the decomposition rules, then randomly select one of the "reassemble rules" for that decomposition rule, and use it to construct the reply.

As we saw above, Eliza read the input string He says I am too lazy and found the keyword i. It ran through its entire list of keywords, but i is the only one that matched. The keywords are ranked; if more than one keyword matches, it picks the most salient.

Next, it applied all the decomposition rules for the keyword i (e.g., * i was *, * i am * @happy *, and so on) to see if any matched. One rule did: * i am *. Using this rule, we isolate parts of the input string around i am: the two phrases He says and too lazy.

Next we randomly select a reassemble rule: Is it because you are (2) that you came to me. We use this rule to construct the reply. We replace (2) with the text that matched the second asterisk in the decomposition rule—in our example, the string too lazy. Finally, Eliza replies with, Is it because you are too lazy that you came to me?

The

Eliza algorithm has pre- and post-processing steps as well. These handle the transformation of words like I and you; you can read the documentation embedded in the module to learn more.

You can also access all of the module's internal functions from your program. For example, using the transform method, you can feed a string to Eliza and fetch its response:

```
$string = "I'm sad.";
$response = $mybot->transform( $string );
```

The Eliza bot is an object, and its configuration data is encapsulated, which means that you can instantiate other Eliza bots, each with their own distinct configurations.

Here, we create two bots and make them talk to one another:

```
use
Chatbot::Eliza
my ($harry, $sally, $he says, $she says);
$sally = new Chatbot::Eliza "Sally";
$harry = new Chatbot::Eliza "Harry";
$he says = "I am sad.";
srand( time ^ ($$ + ($$ << 15)) );</pre>
                                          #
Seed the random number generator.
print $harry->name, ": $he says \n";
while (1) {
         $she says = $sally->transform(
$he says );
    print $sally->name, ": $she says \n";
         $he says = $harry->transform(
$she says );
    print $harry->name, ": $he says \n";
}
```

Two bots conversing produces interesting results only if we have clever scripts. Below, we see sample output from the program, with both bots using the default Eliza script:

```
Harry: I am sad.
Sally: I'm sure it's not pleasant to be
sad.
Harry: Do you think coming here will help
you not to be sad?
Sally: You're not really talking about me
-- are you?
Harry: Perhaps I am in your fantasies.
Sally: What makes you think of my
```

```
fantasies?
Harry: Your fantasies?
Sally: Are you worried about someone
else's fantasies?
Harry: Would you prefer if I weren't
worried about someone else's fantasies?
```

In general, the default Eliza script does not produce any sensible conversation when interacting with itself. (In fairness, people who talk to themselves often don't make much sense either.)

One of the reasons the original

Eliza was so successful was Weizenbaum's clever rules and limited domain; the role of a mock Rogerian psychotherapist doesn't require much intelligence, and a relatively simple algorithm can pull it off. But as Weizenbaum found, even this relatively simple program fooled naive users into believing that it understood what they were saying. Scarier still, some people actually felt more comfortable talking to Eliza than to a flesh-and-blood psychiatrist.

Here's an

interactive session, this time with a human deliberately trying to expose

Eliza's weaknesses:

```
Eliza: Please tell me what's been
bothering you.
you: Why are you asking me that?
Eliza: Do you sometimes think I am asking
you that?
you: What time is it?
Eliza: Why do you ask?
you: You don't understand.
Eliza: We were discussing you -- not me.
you: Who are you?
```

Eliza: What answer would please you most? you: I have asdwet in the asdgasdg. Eliza: Do you say you have asdwet in the asdgasdg for some special reason? my am we are is in the always my you: brother Eliza: Your brother? you: You your in I my me mine. Eliza: Your you mine? I have a gun. Give me your money! you: Eliza: Can you elaborate on that? vou: Do you remember the Kennedy assassination? Eliza: Did you think I would forget the Kennedy assassination? you: Stop asking questions. Eliza: That is interesting. Please continue.

What Now?

The CPAN includes many modules that allow a script to interact easily with resources on the Internet. For example, it is straightforward to write scripts that combine the Chatbot::Eliza module with the Net::IRC module or the CGI module. (The CPAN distribution of Chatbot::Eliza includes a sample CGI script—a Web-based Eliza.)

References

The CMU Artificial Intelligence Repository

http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/classics/0.html

Hayden's Java Eliza

http://chayden.net/eliza/Eliza.html

Goerlich's Java Eliza

http://philly.cyberloft.com/bgoerlic/eliza.htm

The white paper on the Loebner Prize competition

http://www.lazytd.com/lti/pub/aaai94.html

ftp://ftp.nl.cs.cmu.edu/usr/mlm/ftp/pubs/aaai94.ps.Z

Julia, an Eliza-like chatterbot that roams on TinyMUDs

http://www.lazytd.com/lti/julia/

The CYC Project

http://www.cyc.com

BotSpot

http://www.botspot.com

UMBC AgentWeb

http://www.cs.umbc.edu/agents/

The Simon Laven page ("Chatterbot Central")

http://www.simonlaven.com/

Chapter 13. Infobots and Purl

Kevin Lenzo

```
<JUM> Whenever I set it to not autoload
images with
          Netscape 2.01, the whole program
locks up.
        Anyone know why?
<Irving> no
   <JUM> Does anyone know where I can get
Netscape 2.0????
   <url> i think netscape 2.0 is at
                 ftp://archive.netscape.com/
archive/index.html
   <JUM> I am forever grateful, Url.
    <JUM> Url: Are you running ver 2.01
with success?
   <url> jum: bugger all, i dunno
   <JUM> OK.
   <JUM> Thanks, Url
   <url> de nada, jum
```

You've probably heard about the Turing test, the Loebner prize, or other contests that measure how much a program can act like a human. Chapter 12 discussed Chatbot::Eliza, a module that behaves like a Rogerian therapist. Instead of asking how intelligent a program can be, let's explore the usefulness of impersonating a human. Enter the *infobot*: an autonomous program that converses with users.

Infobots are an ongoing experiment in how we can interact as communities with a common institutional memory. If that sounds too grandiose, think of it as a study in interactive graffiti.

IRC

The

infobots first appeared on the

EFNet (Eris-Free Net) Internet Relay Chat (IRC) in June of 1995. On IRC, people talk to one another (typically, to entire groups of people) in *channels*, each devoted to a particular topic. When a user creates a channel, he or she becomes a channel operator, which gives them powers over other users.

Some channels are popular; the #macintosh channel has about 50 users regardless of the time of day. The #perl channel has 74 users as I write this. EFNet is the largest noncommercial chat network, with about 40,000 users from around the world at any given moment.

At Carnegie Mellon University, we have operated *irc.cs.cmu.edu* as an

EFNet leaf node since 1996. I've been able to develop the infobots because I was the administrator of the machine; in general, EFNet doesn't appreciate bots, since they are often used for abuse. For instance, bots have been used to spam users with advertisements for porn sites or to take control of other people's

channels.

Resources for IRC and bots are listed in the sidebar Using IRC and Bots.

Using IRC and Bots

To experience these bots in action, sign onto IRC and check them out. You'll need an IRC client, which you can download for free from the sites below. To join the Perl channel, type /join #perl.

More information about EFNet and IRC

http://www.irchelp.org/

Infobot home pages

http://www.cs.cmu.edu/~lenzo/infobot.html http://www.infobot.org

and

The IRC Help home page

http://www.irchelp.org/

The Eggdrop home page

http://www.valuserve.com/~robey/eggdrop/

The Computer bot

http://www.networks.org/irc/computer.html

The UN bot

http://networks.org/irc/

Bots and Infobots

One popular type of beneficial bot is the Eggdrop, developed by Robey

Pointer. Eggdrops are designed for channel protection; they can be linked together to monitor people or programs that spam users. Unfortunately, they have also been subverted for the exact behavior they were meant to guard against. (Lest you get the wrong idea, most of IRC is simple chatting. It's not populated with teenage

net.punks talking about techno-warfare, but there are vandals in every community.)

More benign bots exist as well. Some bots run interactive games, like chaosbot on #chaos or robbot on #riskybus. On #chaos, people join one of two teams and compete to guess items in a Top 10 list; on #riskybus, the users wager with fake money in a Jeopardy-like game. Even the IRC server administrators use bots to monitor connections and activity.

Infobots are different. They exist to collect information and answer questions. I have been running infobots for over two years now, and they have continued to evolve with the help of their communities. The bots are missed when they're away—when mine crash and don't respawn, I'll get an immediate "Where's the bot?" when I sign on, even before anyone says hello.

So What?

Why are

infobots so popular? Well, they converse in natural language, they serve as a community memory, and they learn. The initial motivation for

infobots came from #macintosh, where the same tired questions were asked again and again. We realized that if the answers were recorded, being helpful would be less of a chore, because we wouldn't have to repeat ourselves all the time. Even if no one on the channel knew the answer, the infobot could reply. This is an act of hubris and laziness, and thus well-suited to Perl.

There are other bots on

EFNet: aack on #unixhelp, which has a fixed set of about 150 facts, and Computer and UN, which provide canned answers to questions. They don't learn anything over time, nor do they take advantage of Perl's wonderful text processing. Let's look at some examples of people using one of my bots: url (pronounced "earl"):

The question here is a permutation of Where is X?, or, more simply, X?, to which url responds with a sentence of the form X is Y. The surface forms of the replies

(somebody said...) are chosen at random to make url seem less mechanical.

My bots don't learn in a deep Artificial Intelligence sense, but they do get more helpful over time. url, like a sponge, soaks up information that it hears on the channel.

Are You Spongeworthy?

Yes. Everyone is. Whereas other bots are strictly information stores loaded by a priesthood of factoid keepers, mine are egalitarian. Here's a demonstration of purl, the infobot that lives on the EFNet Perl channel, #perl, 24 hours a day:

```
<juice> i am not very happy
<juice> me?
<purl> you are, like, not very happy
```

purl has soaked up quite a few

factoids, as has her older brother url. (It's interesting to note that people often refer to purl as "she," while url is almost invariably a "he.") She can learn from any declarative statement including a verb of being (is, are, am, and so on). url is more constrained, and requires that the part of the sentence after the verb contain a recognizable URL. The http://isn't required; both url and purl infer it when necessary.

Here's a short exchange from #perl:

On #macintosh:

```
url is also on the
```

EFNet

channels #macdev, #linuxos, #avara, and #distributed. There is also a version of url running on the Undernet; their databases were originally synchronized, but they are now learning and interacting independently. (The dates shown are when the current processes began, not when the

bots began learning.) A relative of url and purl, called rurl, hangs out on #robogeeks. It's less an information repository than a personification of bad attitude.

Sometimes people think they're interacting with an actual human being; the excerpt at the beginning of this article is real. I've seen people come into a channel, ask a series of questions and get pointers to good information, and then thank the bot and say, "Thank God someone was listening!" It's even been hit on.

You Can't Do That in Public!

This brings us to the question of what goes into the interaction. Why does it sometimes fool people, and what keeps it from making inappropriate comments? Admittedly, it picks up many irrelevant things and occasionally spits out a useless comment, but it doesn't happen often, and when it does it merely adds to the charm.

People on #macintosh endured the early development cycle with great aplomb, providing a lot of feedback. I get feedback from several channels now, but it's still mainly #macintosh and #perl that supply the most feedback. Here's an example of a discussion about how the bot should behave on #macintosh. (Just to clarify, I go by the nickname oznoid, or oz for short.)

```
<tonyola> oz - a suggestion - url should
only accept queries
         beginning with "url,...."
<golgo13> tonyola: No.
         * Zebe gives bizzy a botsnack
  <br/>
<bizzy> i hate having to type "url"
  <bizzv> :)
 <durkin> oznoid, do you plan to make his
database of facts
           available. i mean, i think his
large knowledge is
                one of the most coveted
features..
  <barry> give him a preferences window :)
<golgo13> Half of url's usefulness is
answering questions from
          people who don't know to address
```

him. <barry> url, preferences..then you set what va like :) <tonyola> but url keeps breaking in when he's not wanted otherwise <Fatale> is that really that much of an inconvenience, compared to his usefulness? < elle> url is allowed to interrupt anytime :) <Fatale> i often wonder why it's such a problem, when you have free access to all that information <bizzv> heh <golgo13> It is ridiculous for for someone to come in and say "Where is stuffit expander?" and then for a regular to go "url: Stuffit expander?" <durkin> url, nslookup fall.back.host.net <url> fall.back.host.net is 205.166.250.10

Here are a few design decisions I've made:

- IRC lets you identify privileged users, but my bots don't use the information. There are no special people, and everyone has equal power over the bots. This makes it public property, and part of the community.
- If the bots aren't directly addressed, and they don't know anything about the question, they keep quiet. This is how users in large channels often act, except that humans sometimes make ad hominem attacks at new users out of boredom. My

infobots are a bit more polite.

• If the bots are addressed, they always reply, even when they don't know the answer.

• The bots support interaction through private messages, as well as on the public channel. This means people can get information or teach the bot without annoying the channel. When you send a personal message to the bot, it always replies:

• The bots can tell other people about things. For instance, "tell oznoid about X" would cause it to send oznoid a private message, something like "nick wants you to know: X is Y."

```
<longhair_> purl, tell johnpc about fish
      <purl> okay, longhair_, i did.
```

• The bots can volunteer information about things even if there is no obvious question—as long as the key phrase is long enough. That is, there is a minimum length utterance for which it will always blurt out an answer.

```
<humblepie> the perl journal
<purl> it has been said that the
```
Anyone can ask the

bots to leave the channel at any time, so that they can be banished when they're not wanted. The request must be made in public; a private message is too sneaky. When they leave, they say goodbye to the person in public just before leaving. They can be messaged to rejoin by anyone.

```
<fimmtiu>
purl, leave #perl
<purl> goodbye, fimmtiu.
*** purl has left channel
#perl
```

• The bots are generally polite, replying to hello, thanks, and a few other things with random responses, to avoid seeming mechanical.

```
<fimmtiu> Hey, purl.
<purl> what's up, fimmtiu
<PerlJam> hi purl
<purl> niihau, PerlJam
```

("Niihau" means "hi" in Mandarin.) The bots won't understand all greetings, however:

```
<Murr> yo purl-boy
<fimmtiu> What's up, purl?
<purl> up is, like, what's not down
or the direction
parallel and opposite the
vector sum of all
gravitational forces at a
given point
```

- The bots answer help requests by private message only. This can be extended to other functions (like internic and nslookup) at the option of the bot master.
- Users can change a bot's definitions with a comment like this: No, purl, Definitions can be supplemented with X is also ... or removed with Forget The bot will confirm the action when possible.

```
<oznoid> x?
  <purl> x is, like, my friend
<oznoid> no, purl, x is y
  <purl> okay, oznoid.
<oznoid> x is also z
  <purl> okay, oznoid.
<oznoid> x?
  <purl> rumour has it x is y or z
<PerlJam> dammit purl, listen to oznoid
when he's talking to you!
<PerlJam> ;-)
  <oznoid> purl, forget x
      <purl> oznoid: I forgot x
```

What Are You Thinking?

When someone enters a channel and asks a question, the bot will answer if it understands the question and knows something apropos. It parses the question with rules like these:

```
$finalOuestionMark
                            =
                                        (s/
                      # Get the "?"
s*(?+$//);
s/^\s*does
                   any(one|body)
know\s+//i;
                          # Remove polite
pre-query
if
       (s/^(who|what|where) \s+//i)
                # We've got a guestion
{
     $questionWord = $1;
}
if
    (s/^{(is|are)}) + //i or
s/(s+(is|are))/(i)  { # Get the verb
     verb = $1;
}
```

The bot then checks to see if it recognizes the remainder of the sentence. It does this by reducing the sentence to a *canonical form*: a standardized representation for the data that retains only the important

information. In this case, that means no determiners (a, an, or the), and a few other simplifications. This covers queries such as:

```
Where is foo?
Does anyone know where foo is?
```

```
What are foo
Foo?
```

The

infobots actually have a long list of verbs to check; the code shown above is for demonstration purposes only. This example shows only portions of the text being eliminated; in reality, the substitutions are used later to canonicalize the sentence for more advanced processing.

It was easier to do things this way than to have url parse a full grammar specified in Backus-Naur Format (BNF). One reason is that people define new words on the fly, so it's not possible to specify the complete grammar in advance without using a /.*/ placeholder to allow for new words. Furthermore, people don't speak in well-formed sentences very often, especially

on IRC. There are even textual equivalents of filled pauses (um, ah) that are quite communicative, but difficult to parse because they can appear anywhere and can be spelled in unusual ways.

Eating fragments of the input and storing them in variables allows the

infobots to build up the contents of a *semantic frame* (a data structure containing attribute-value pairs) by eliminating the known fragments and judging the significance of the remainder. The approach is quite similar to *Phoenix grammars* or *semantic phrase parsers* developed by Wayne

Ward and others, which consume chunks of the input to build up a frame. Things like this have been used quite successfully for parsing in speech recognition tasks such as the ATIS (Airline Travel Information System) systems developed at CMU, MIT, and elsewhere; they continue to be a part of our work at CMU.

At a high level, the processing goes like this:

Is the sentence obviously a question?

If it is, url canonicalizes the question and replies if it has an answer. Otherwise, url replies that it doesn't know, but only if it was directly addressed.

Questions also can include mathematical expressions for evaluation or requests for

information from third parties.

If it isn't, and if url recognizes the key (the left-hand-side X of a statement such as X is Y, or the right hand side of a question like Where is X?) in the database, it volunteers the factoid from the database if the key is long enough (e.g., more than 6 characters) or if url was directly addressed.

Is the sentence an explicitly defined functional form?

My bots understand commands like nslookup (which converts computer names to IP addresses) or internic (which performs an Internic "whois" query). If the user's utterance is one of these commands, url executes the external command and displays the result.

Is the sentence parseable as a statement (X is Y)?

If it is, url checks to see whether X is already in the database. In that event, it looks to see whether the user was aware of that—if he said, No, url, X is Y or X is also Y. In that case, url replies ok,

<name_of_user>. Otherwise, it says, ...but X is Y... if it was addressed directly. This prevents the bots from paying attention to O-dogg below:

```
<orwant> bless is a function that tells
a reference that it's now an object.
<O-dogg> bless is cool.
```

If the sentence isn't parseable as a statement, url admits confusion—but only if it was directly addressed.

As you can see, url can act very differently depending on whether it was addressed directly. That happens when a user calls url by name, or sends a private message. It replies in kind:

```
<paraquat>
purl, do you have a life?
<purl> i haven't a clue, paraquat
```

A slightly more involved interaction:

You can see that it's not always as fluid as one would like, but it gets better through use. When an infobot annoys the channel, I hear about it pretty quickly.

Idiot Savant

The

infobots grow every day, and some of them have over 40,000 factoids. They have to be able to access their

information quickly, and checking every item in the database against the input would take too long. The

infobots use tie and DB_File, so that the database appears as a pair of large hashes but actually reside on disk, and can be looked up without requiring lots of RAM. Of course, that means whatever you're looking for has to match the hash key exactly. That's why the input must be canonicalized, stripped of irrelevant utterances like hmm..., a, an, and the, among others. This canonicalization is performed both when the item is stored and when the bot tries to answer a query. This doesn't always work quite the way you'd like, and so you occasionally get Rainmanesque responses: aaaah... i know something about X... aaah... That would be Y, yeah.

Since the entire database resides on disk, and not in memory, the run-time memory requirements are reasonable—on the order of 5.5 megabytes for url, the largest bot. Because no explicit database search is required, the bot's responses are nearly instantaneous, which is important for the real-time communication

on IRC. Currently, we use one database for singular entries and another for plurals, so it's quite easy to maintain subject-verb agreement even when the query has no verb. I'm merging them into a single, unified database in which each entry has an attribute for each case.

A Sense of Play

url has an elder brother named hocus, who lacks the frontend filters that restrict knowledge to URLs. hocus soaked up anything that looked declarative—anything at all. When the first

infobots appeared without this restraint, people quickly became annoyed by the frivolous responses. purl is actually quite hocus-like, but with more stop expressions, which reject some queries and prune others.

Once people realize the bot is a bot, they often have a little fun with it, stuffing it with questionable

information. The bots aren't shy about expressing their newfound ideas in public, sometimes to the annoyance of channel curmudgeons.

Eventually, those who find the bot annoying at first usually begin to appreciate its usefulness. Still, the play continues, sometimes sparked by a random bot comment and leading to the definition of a few humorous factoids.

```
<purl> okay, hindeur.
<q[merlyn]> there
<q[merlyn]> like that
<q[merlyn]> purl is also a miss take!
  <hindeur> this bot is fun!
     <purl> okay, g[merlyn].
<q[merlyn]> purl?
     <purl> yes, q[merlyn]?
<q[merlyn]>
purl, purl?
     <purl> it has been said that i am the
bot that's got
             the info that's hot and a lot
that's not and all
            of that rot or a knut or buggy
sometimes. or a
            miss take!
     <Taer> sorry. wife stole camel 2ed.
            * nickm gets funky with purl
<q[merlyn]> purl is also funky
```

...time passes...

What? We Ordered No Pizzas!

Bots are a public resource, and as such are susceptible to vandalism. Some people try to break the bots. That's not a bad thing if they're just being playful and testing the limits, but sometimes it's an act of malice. For instance, since any user can change or delete entries, we occasionally find useful nuggets of information replaced by phrases like Microsoft sucks or Apple sucks. At least the damage is visible, and rapidly fixed by the channel.

At one point, I allowed people to get a random entry from the database. Soon someone was steadily lobotomizing the bot, by retrieving random entries and having the bot forget them, one by one. Since I maintain logs of the bot's interactions, I was able to back out all the changes from that user, but the experience was unpleasant.

The math handling abilities of the bot use eval, which can be extremely dangerous. Early experience made me check the input very carefully before evaluating it—this is why it won't handle all expressions. Even the Safe module isn't adequate protection; the documentation enumerates some of the side effects, such as infinite loops and memory hogging, that might occur from certain inputs. I simply reject anything outside of a very limited subset of functions. Even with this precaution, users found and manipulated a bug in the system libraries that made the bot crash.

The

channels get quite defensive of the bots when they see people trying to vandalize them, or even when people are merely rude. This is prevalent on #macintosh, where url has been a fixture for a couple of years.

Future Directions

The infobot code is undergoing substantial revisions; for one thing, it is being modularized. Since there are now versions that work standalone on the desktop, with IRC, or with Zephyr (another messaging system), the infobot code is being decoupled from the communications protocol. Net::IRC, a module designed specifically for manipulating the Internet Relay Chat protocol, will be integrated into the infobot code. Some of us have also been talking about connecting several bots together, expanding the types of statements and questions, implementing per-channel "personalities," and connecting networks of infobots-each an expert on its own topic. We have settled on the Bazaar model rather than the Cathedral model of software development, as described in Eric S. Raymond's "The Cathedral and the Bazaar."

We want as many people getting the source as possible, making interesting modifications to it, and giving it back to the community. Just like Perl itself.

Where to Get It

The code is a mess right now. Fortunately we have a group of people and a mailing list, and we're redesigning it from scratch. By the time this article is published, we are hoping you won't have to use the collection of barnacles that comprise the current infobot. The source has been available for some time now, warts and all—I had to get over my desire to wait until it was perfect. It's far from perfect now, but it has been improving quickly since the public release. To get on the mailing list, send me mail at lenzo@cs.cmu.edu.

Acknowledgments

People who have been running infobots have been a great resource, particularly Adam T.

Lindley (abstract on IRC), who has done extensive work with script and set up the web site at http://www.infobot.org. Patrick

Cole (wildfire on IRC) helped replace the IRC client code and the user profiling system. Dennis

Taylor (fimmtiu), who is also involved with Net::IRC, was also a great source of ideas and inspiration.

All my work is due, in one way or another, to a love of Perl itself. Perl's philosophy appeals to me more than any other programming medium. Thanks to the p5p mailing list and all of the people working on Perl. We owe ya, bigtime.

References

Nolan, John. Chapter 12. TPJ #9, pp. 16–18.

Raymond, Eric S. "The Cathedral and the Bazaar." http://earthspace.net/~esr/writings/cathedral-bazaar/ cathedral-bazaar.html.

Seneff, Stephanie. "Robust parsing for spoken language systems." Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, 1992, Vol. 1, pp. 189–192.

Ward, Wayne. "Understanding spontaneous speech: the Phoenix system." *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 1991, Vol. 1, pp. 365–367.

Chapter 14. Speech Synthesis

Kevin A. Lenzo

Talking computers are ubiquitous in science fiction; machines speak so often in the movies that we think nothing of it. From an alien robot poised to destroy the Earth unless given a speech key ("Klaatu barada nikto" in *The Day the Earth Stood Still* and echoed in *Army of Darkness*) to the terrifyingly calm HAL 9000 in 2001: A Space Odyssey, machines have communicated with people through speech. The computer in "Star Trek" has spoken and understood speech since the earliest episodes. Speech sounds easy, because it's natural to us—but it's not natural for computers.

Let's ignore the problem of getting computers to think of things worth saying, and consider only turning word sequences into speech. Let's ignore prosody, too. Prosody—the intonation, rhythm, and timing of speech—is important to how we interpret what's said, as well as how we feel about it, but it can't be given proper care in the span of this article; partly because it's almost completely unsolved. The input to our system is stark and minimal, as is the output—there are no lingering pauses or dynamics, no irony or sarcasm. (Not on purpose, anyway.)

If we are given plain text as input, how should it be spoken? How do we make a transducer that accepts text and output audio? In this article, we'll walk through a series of Perl speech synthesizers. Many of the ideas here apply to both natural and artificial languages, and are recurrent themes in the synthesis work at Carnegie Mellon University, where I spend my days (and nights). The code in this article is available on the web site for this book (http://www.oreilly.com/catalog/tpj3) and at http://www.cs.cmu.edu/~lenzo/tpj/synthesis_examples.html. Audio samples of each method are included.

Pre-Recorded Sentences

As a first attempt, we could use

pre-recorded speech for *everything* we might want our computer to say. It wouldn't be very flexible, but it would sound great. For every possible input, we'd need a recording. Your phone company probably does this—"What city, please?," or "What listing?"

Spokesvoices like James Earl Jones are recorded as expertly as possible to show off the sound quality, and greet you with phrases like "Welcome to Bell Atlantic." This sort of thing is pretty easy to do—just play the right sound file at the right time:

```
while (<>) {
    say $
speech{$_};
}
```

%speech is a hash of speech samples, perhaps tied to a database using tie or dbmopen. say is a subroutine that knows about sending audio to something that can play it, and \$ is, of course, Perl's default scalar variable. As limited as this might seem, it is the perfect solution for applications that require only a small vocabulary—in games like WarCraft, for instance, where the characters have a few randomized responses they utter as you order them about—"Yes, Master?" or "Of course, Master." If you need any more flexibility, though, you'll be stuck—and if you ever need to record new material, you have to be careful to emulate the same recording conditions.

Lexical Synthesis in One s///

Let's look some variations of the expression s/(\$text)/ speech \$1/eg (which was the original title of this article as it appeared in TPJ):

```
while (<>) {
    s/($text)/$speech{$1}/g;
    say $_;
}
```

Here's guide to this code snippet:

- \$text is a regular expression that accepts words
- %speech is a hash of audio waveforms, one per word
- \$1 holds each match of \$text (within the parentheses)
- say is a subroutine that sends a waveform to an audio device.

The s///g replaces all of the words in \$text with hash entries from %speech; this *tokenizes* the

input, and turns each token into an audio waveform. More formally, *\$text* is an expression that matches words in the language defined by *%speech*. Whole words are replaced with their audio entries. Words not in the language have no entry in *%speech*, and therefore produce no output. So, for our purposes, the expression in *\$text* can be quite simple:

 $t = ' \ s + | \ s + '$

 $\s +$ matches one or more whitespace characters, and $\S +$ matches one or more non-whitespace characters. Clearly, punctuation is a part of the word in this tokenization. For example, if the previous sentence were the input text, "Clearly," and "tokenization" would become tokens, as would all the other words; if the sentence you're reading now were the input, we'd even have quoted words as tokens. If we remove punctuation, and fold the uppercase letters to lowercase so that "Clearly" and "clearly" are the same word, we can generalize our tokenization a bit. With a few changes we can eliminate some punctuation, in much the same way whitespace was separated from the words. For instance, we could do this:

\$text = '[.!?"]|\S+|\s+'

which turns some of the punctuation into separate tokens. We can convert to lowercase with the lc operator:

```
while (<>) {
    s/($text)/say $
speech{lc($1)}/eg;
}
```

The sample is spoken immediately, rather than explicitly passed to say after the entire line is processed. The /e on the end of the s/// tells Perl to evaluate say speechlc(s1) as a full-fledged Perl expression.

This can work nicely for a small, constrained vocabulary, but you need a sample for every word ever to be spoken. The result sounds unnatural, because the words are spoken without regard to their context in the sentence. For instance, words at the end of an utterance are often spoken more slowly, and with a final drop in pitch. This technique doesn't handle that.

The Out-of-Vocabulary Problem: Synthesis in One s///e

How do we handle

"out-of-vocabulary" words? If we wanted our system to speak a word for which there is no presampled audio, we could break it down into smaller parts: *phonemes*. Just like our written words are composed of an alphabet of letters, our spoken words are composed of an alphabet of phonemes. So we'd need to make a mapping from a sequence of text words to phonemes, and ultimately to audio waveforms.

Now, the regex *\$text* remains the same, but we change the substitution:

```
while (<>) {
    s/($text)/map { say $speech{$_}} }
text2phones lc($1)/eg;
}
```

The %speech hash now contains phonemes. More precisely, it contains *phones*, which are concrete,

context-dependent realizations of abstract phonemes. For instance, the phoneme /T/ can be spoken as the phone [D] in words like "butter," or as [T] if you pronounce it in isolation (such as at a spelling bee). The audio depends on more than just the phoneme; it also depends on the setting and context.

A couple of points: Notice that the language is still a "regular" grammar—the sort that can be recognized by a vanilla regular expression. (Perl's so-called "regular expressions"

are more powerful than we need, because they can handle irregular grammars as well.) (A regular grammar deals with patterns of symbols composed with only concatenation ("and"), disjunction ("or"), and closure (zero or more iterations of a pattern). Perl regular expressions allow things like look-ahead assertions, which makes them able to parse context-sensitive languages, and means that Perl's regular expressions aren't regular at all.)

Text-to-Phoneme Conversion

What magic lies hidden in text2phones? English spelling encodes a great deal of history, but the pronunciations of words are not always obvious—for instance, words such as "night," "knight," "ought," and "cough" don't sound like they have a "gh" in them; well, at least not the "g" as in "great" or the "h" as in "history." Languages such as Spanish and Greek have closer relationships between their written and spoken forms; Chinese, on the other hand, has no relationship whatsoever.

There are a number of possible strategies for tackling this problem. Many

speech synthesizers use a two-tiered method—first, check if it's in a small dictionary of pronunciations, and, if not, resort to a

text-to-phoneme conversion strategy. Any word pronounced properly by the rules is removed from the dictionary, so as to keep the total conversion package reasonably small. *Festival*, a free, open source synthesizer from the University of Edinburgh, uses this strategy, as does my own Perl-based synthesis, *phonebox*.

Looking things up in a dictionary is good work when you can get it, but you'll always run into out-of-vocabulary words for which there's no predefined pronunciation. Even ignoring the fact that human languages change all the time by adding words and changing usages, the number of entries will get quite large if every word is included. Building generic text-to-phoneme transducers is an interesting problem that requires generalization to unforeseen data; *decision trees*, borrowed from the discipline of machine learning, work reasonably well. A decision tree is an organization of nodes in which each node represents a decision; you can download Perl code to build dictionary decision trees from http://www.cs.cmu.edu/~lenzo/t2p and from the web site for this book. The technique is described there, and it includes the letter-to-phoneme converter as well as some sample data.

At CMU we train

decision trees for text-to-phoneme conversion using pronouncing dictionaries such as the CMU dictionary, the Oxford Advanced Learner's dictionary, the

Moby lexicon, or any number of others. Given a set of words and their pronunciations, a set of

alignments between the letters and phones is produced, generating a mapping from letters to phonemes. For example, here are some possible alignments of the letters in the word "algebraically" with a string of phonemes:

algebraically AELJHAHBREYIHK_L_IY

There are some tricky issues in generating good alignments; for this example alone there are 78 possible alignments. This number comes from C(13,11]—pronounced "thirteen choose eleven"—which is the number of possible ways you can choose eleven items from a set of thirteen. In this case, there are thirteen letters and eleven phonemes, so there are eleven slots that get phonemes, and two that don't. The ones that don't get a phoneme end up aligning with a null. C(n,k) is equal to n!/(k!(n-k)!), where n! is n factorial, or $n \cdot n-1 \cdot n-2 \cdot \dots \cdot 1$.) Good alignments are critical for getting acceptable results from any learning method.

NetTalk, a neural network for speech output that is successful despite relatively poor performance of its

text-to-phoneme rules, uses this sort of encoding of letters in a window of neighboring letters. Methods for generating these alignments are discussed in [Black,

Lenzo, and Pagel] and [Pagel, Lenzo, and Black].

Once the alignments have been generated, they are exploded into *feature vectors*, one per letter, that include some context: three letters on each side. Each feature (neighboring letter, in this case) is given a name; "L" for the letter itself, "L1" for one to the left, "R1" for one to the right, and so on. The word "absurdities," for instance, ends up producing eleven feature vectors, as shown in Table 14-1.

L3	L2	L1	L	R1	R2	R3	Phoneme	
-	-	-	a	b	s	u	АН	
-	-	a	b	s	u	r	В	
-	a	b	s	u	r	d	S	
a	b	s	u	r	d	i	ER	
b	s	u	r	d	i	t	_	
s	u	r	d	i	t	i	D	

Table 14-1. The eleven feature vectors of "absurdities"

L3	L2	L1	L	R1	R2	R3	Phoneme	
u	r	d	i	t	i	e	АН	
r	d	i	t	i	e	s	Т	
d	i	t	i	e	s	-	IY	
i	t	i	e	s	-	-	_	
t	i	e	s	-	-	-	Z	

These vectors are taken together to build a

decision tree that tests the features and produces an output phoneme (or no phoneme at all, denoted by the underscore in the fifth and tenth vectors). The result is an example of a *finite state transducer*, the core of several

speech synthesis systems, such as the Lucent synthesizer. Here's one subtree trained from the CMU dictionary:

```
if ($feature{'L'} eq 'C') {
    if ($feature{'R1'} eq 'A') {
        if ($feature{'L1'} eq 'A') {
            if ($feature{'L2'} eq 'F') {
                return 'S';
            }
        if ($feature{'L2'} eq 'R') {
            if ($feature{'L2'} eq 'R') {
                if ($feature{'L3'} eq 'U') {
                    return 'S';
                }
                return 'K';
        }
        return 'K';
```

}

This snippet implies that the letter C (tested in the order L, R1, L1, L2), should become the S phoneme (as in the Americanized "façade") if it has the context shown in Table 14-2.

Feature	L3	L2	L1	L	R1	R2	R3	Phoneme
Letter	?	F	A	С	А	?	?	S
Testing order	-	4	3	1	2	-	-	

Table 14-2. The ç in façade

If the second letter to the left were an R instead of an F, the subtree would output the S phoneme if there's an R to the left of it ("Curaçao"), and the K phoneme otherwise. You can see that these

trees actually represent a great deal of the dictionary directly.

Once the text has been converted into a phoneme sequence using these letter contexts, each phoneme can be immediately replaced with its audio waveform, but only if we ignore the context of the phoneme.

More Context: Two Substitutions

You'll quickly find that simply stitching together phoneme sequences

sounds pretty bad; examples of this are on both the book's web site and my own. Part of the reason is that the actual *sounds* of a language depend upon the neighboring sounds—they're

context-dependent. Just as the letter transducer uses context to decide which phoneme to speak, the phone depends upon the neighboring phonemes. We need the phonemes to know who their neighbors are, but as long as we have just one s/// that does the whole job, we can't do that.

For an example of why context is necessary, consider the words "pat" and "spat." In "pat," the p is *aspirated*—it's spoken with a puff of air. In "spat," there is no aspiration. Also, when an R comes after a vowel, it colors the vowel in a way that's different from an R that's before a vowel; L does the same thing, revealing the two kinds of [L] in the word "lateral."

Capturing the context and using it to pick the sample can be accomplished easily with Perl's irregular expressions:

```
s/($text)/phones lc($1)/eg;
s/(?=(^|$phoneme))\s*($phoneme)\s*(?=($phoneme|$))/
say $
speech{"$2($1,$3)"}/eg;
```

Here, the (?=) items are assertions that must be satisfied but aren't considered part of the matching text.

The first s/// turns a word sequence into a phoneme sequence, and the second speaks a phoneme in the context of its left and right neighbors; it captures some of the coarticulation effects that smear sounds into one another. This would be fine if you had every possible entry for every possible phoneme in context listed in <code>%speech</code>. That number can be quite large—for instance, if the size of the phonemic inventory is, say, 51 sounds, that gives a total of 132,651 (51 cubed) entries in the table. It's difficult to get complete coverage of these units from a single talker, let alone with consistent quality, so we turn that last hash table lookup into a subroutine. A phoneme in context like this is often called a *triphone*, meaning one phoneme with its left and right neighbors fixed.

So the hash of samples, <code>%speech</code>, needs to contain some entries for the "backed-off"

triphones, such as P(R), which is the phoneme P with an R on the right and anything on the left. It will also need P alone, in case neither context matches.

This is effectively an implementation of *optimality theory*—it finds the best non-empty subset given a partially ordered set of constraints. The "unwrapping" of the single phoneme into itself plus context is another example of the two-level conversion we used in the letter-to-phoneme rules: a finite context-sensitive language is expressed with an equivalent regular grammar, which lets us parse in a particularly easy way—with finite state machines. With these context-sensitive rewrite rules, we have a triphone synthesizer. It doesn't consider prosody, and it performs no analysis beyond one level of neighboring context, but it works.

What Else?

Even with everything described here, there's much still missing intonation, stress, voice quality: everything that really makes a voice sound like an individual. A complete speech synthesis system is more complex than what you've seen here, but the basic techniques here can be applied to any system.

Now that speech-interactive systems are actually out there, on the phone and on the desktop, we're seeing just how unsolved many of these synthesis problems really are. They don't sound very good; there's not a lot of individuality in the voices; they certainly don't sound interested in what they're doing; but the work goes on. I like to think we're making progress—the Earth does not stand still. Gort! Klaatu barada nikto.

References

The Festival Speech Synthesis System. http://www.cstr.ed.ac.uk/projects/festival.html.

Phonebox text-to-speech synthesis. http://www.cs.cmu.edu/ ~lenzo/phonebox/.

t2p text-to-phoneme converter. http://www.cs.cmu.edu/~lenzo/t2p/.

CMU Dictionary. http://www.speech.cs.cmu.edu/cgi-bin/ cmudict.

Oxford Advanced Learner's Dictionary. http://www.speech.cs.cmu.edu/comp.speech/Section1/ Lexical/cuvolad-dict.html.

Moby Lexicon. http://www.dcs.shef.ac.uk/research/ilash/ Moby/.

NetTalk data. http://www.boltz.cs.cmu.edu/benchmarks/ nettalk.html.

Black, Lenzo, and Pagel, "Issues in Building General Letter to Sound Rules," for the *1998 ESCA Speech Synthesis Workshop*, Jenolan Caves, Blue Mountains, Australia. http://www.cs.cmu.edu/~lenzo/areas/papers/ESCA98/ ESCA98_lts.ps.

Friedl, Jeffrey. *Mastering Regular Expressions*. O'Reilly & Associates, Inc., 1997.

Pagel, Lenzo, and Black. "Letter to Sound Rules for Accented Lexicon Compression", for the 1998 International

Conference on Spoken Language Processing, Sydney, Australia. http://www.cs.cmu.edu/~lenzo/areas/papers/ ICSLP98/ICSLP98 lts.ps.

Sejnowski, T.J., and Rosenberg, C.R. *Parallel networks that learn to pronounce English text*. In Complex Systems, 1, 145-168. 1987.

Sproat, Richard, editor. *Multilingual Text-to-Speech Synthesis, the Bell Labs Approach.* Kluwer Academic Publishers, 1998.
Chapter 15. Lazy Text Formatting

Damian Conway

Don't you just *hate* getting an email that's been for matted

for the wrong number of columns? It's an unprovoked ass ault

on your poor visual cortex. And it's a thoughtless insult, to o.

It screams: "Hey, you aren't even worth the eight keystr okes

it would take me to correctly set my editor's autowrap!" > And, of course, it only gets worse when quoted email is involved. > Even when someone tries to do the right thing, they just end > up frying more of your neurons as you attempt to untangle > the mess that most text formatters make of the standard > quoting conventions. It's *no* fun trying to separate the meaning > from the massage.

What the world needs is a

text reformatter that looks at the contents—and context—of the ASCII it's munging, and then Does The Right Thing automagically.

Text::Autoformat

And that's exactly what the Text::Autoformat module gives you. Specifically, it provides a subroutine named autoformat that wraps text to fixed

margins. However, unlike other text

wrapping modules (such as Text::Wrap, Text::Correct, or Text::Reflow), autoformat reformats its input by analyzing the text's structure: identifying and rearranging independent paragraphs by looking for visual gaps, list bullets, changes in quoting, centering, and underlining.

If you're happy to live with autoformat's reasonable defaults, then

reformatting a single paragraph (taking it from STDIN and printing it to STDOUT) is no more complicated than this:

use
Text::Autoformat;
autoformat;

The

default width of the reformatted

text is from column 1 to column 72, but it's very easy to change that (and a plethora of other defaults) by giving autoformat the appropriate options:

```
autoformat {left=>8, right=>64};
```

Or the equivalent, but often more convenient, alternative:

```
autoformat {left=>8, width=>57};
```

If autoformat's first argument isn't a hash reference, that argument is stringified and used as the

text to be formatted. For example:

```
autoformat $msg_
text;
```

Likewise, if it's called in a non-void (scalar or list) context, autoformat returns the formatted text, rather than printing it to STDOUT.

Normally, autoformat only reformats the first paragraph it encounters, and leaves the remainder of the text unaltered. This behavior seems odd initially, until you realize that the single most common use of autoformat is in the following one-liner:

```
perl -MText::Autoformat -e'autoformat'
```

and that the obvious thing to do with this one-liner is to map it onto a convenient keystroke in your text editor, thereby providing intelligent, single-key,

paragraph-at-a-time reformatting. For example, if you're a viuser, you might add this to your .exrc file:

map f !G perl -MText::Autoformat -eautoformat

That is: map the f key to grab every line from the current editing position to the end of the file and filter it through Perl. Then, to provide that filter, the Text::Autoformat module is loaded and autoformat is called.

If autoformat's default were to reformat everything it was sent, then you'd have to write:

```
map f !} perl -MText::Autoformat
-eautoformat
```

and you'd be stuck with vi's *much* less sophisticated understanding on what constitutes a paragraph. More on that shortly.

Of course, the real power of the module is best seen when it operates on multiple paragraphs simultaneously. To convince autoformat to do that—to reflow every paragraph you send it—you need to ask explicitly, with another option:

```
autoformat { all=>1 };
```

Which leads to the obvious "just-fix-it-all-up-for-me-would-ya" editor macro:

```
map F !Gperl -MText::
Autoformat -eautoformat{all=>1}
```

Paragraphs

The autoformat subroutine gives the illusion of understanding the structure of an input

text because it has a series of very good heuristics (i.e., guesses) for locating and separating paragraphs.

Most

text formatters-and many

text editors—define a paragraph to be a sequence of characters terminated by two or more consecutive newlines. Indeed, this is Perl's notion of a paragraph (which you can grab with a single readline by setting the \$/ variable to an empty string, as described in the *perlvar* documentation).

That's very annoying, because it doesn't cope with how real people write paragraphed

text. Real people leave spaces and tabs on "empty" lines. Real people (and many web browsers) bunch up lists of bulleted and numbered points with no whitespace at all between them. Real people quote email messages, which transforms formerly empty lines into non-empty $\n\t>\n$ sequences.

Because real people do such things, autoformat understands all these notions of a paragraph. Even when they're all used at once. Even when they're used inside one another (for example, quoting a list of bulleted points).

Quoting

One of Text::Autoformat's most useful

paragraphing heuristics is that any sequence of lines beginning with standard "quoter" characters is a single piece of

quoted text, in which the quoters should be preserved and only the

text to the right of them reflowed.

The standard quoters that autoformat recognizes are nested combinations of the characters:

! # % = | : >

Angle brackets can also be preceded by alphabetic characters. So, for example, autoformat would take a series of paragraphs like this:

```
> ! > calling map in a void context is the
sign
> ! > of a sick mind
> !
> ! I don't see why.
> Me either, I regularly do it and I'm
still
> quite sane. I often split in a void
context
> too, but there's a bug in Perl that
seems to
> cause that to mess up $_[0], $_[1], etc.
> ! > Sigh. Have you bothered to read the
man
> ! > page on split??? Yes, I know I wrote
```

```
this
> ! > before that reply: it's a miracle.
```

and reformat them like so:

```
> ! > calling map in a void context is
> ! > the sign of a sick mind
> !
> ! I don't see why.
> Me either, I regularly do it and I'm
> still quite sane. I often split in a
> void context too, but there's a bug
> in Perl that seems to cause that to
> mess up $_[0], $_[1], etc.
> ! > Sigh. Have you bothered to read
> ! > the man page on split??? Yes, I
> ! > know I wrote this before that
> ! > reply: it's a miracle.
```

That's the whole point. By understanding the structural conventions of typical

plaintext, autoformat can reflow it logically, rather than physically.

Lists

Often plaintext will include lists that are either bulleted with punctuation characters, simply

numbered (i.e., 1., 2., 3., etc.), or hierarchically numbered (1, 1.1, 1.2, 1.3, 2, 2.1., etc.) Whether or not it is physically separated from each of its neighbors, each bulleted item is implicitly a separate paragraph and needs to be formatted individually, with the appropriate indentation.

autoformat takes care of that

renumbering, and can also detect unordered bullets (the characters *, ., +, and –), special markers that ought to be outdented (such as NB: and p.s.), Arabic and Roman numerals, single alphabetic letters, and

hierarchical combinations of these (for example, 2.a(ix)).

Besides

adjusting the left margin so that the marker is outdented from the paragraph

text, autoformat renumbers each numbered point sequentially (using the first number as its starting point). For example, given the following

text:

```
You're wrong for the following reasons:

1. I'm right.

1.a. I'm *always* right

1. Even if you were right, you

have the order

wrong.

1.x. You suggested:

> D. Analyze the
```

problem carefully > C. Design the algorithm appropriately > A. Code solution systematically > E. Test thoroughly > B. Ship eventually 1.n. The proper sequence is: A. Code solution expediently B. Ship immediately E. Test sporadically (charge user for maintenance) F. Release "upgrade" periodically (charge user again) autoformat {all = 1}> produces: You're wrong for the following reasons: 1. I'm right. 1.a. I'm *always* right 2. Even if you were right, you have the order wrong. 2.a. You suggested: > D. Analyze the problem carefully > C. Design the algorithm > appropriately > A. Code solution systematically > E. Test thoroughly > B. Ship eventually 2.b. The proper sequence is: A. Code solution expediently

```
B. Ship immediately
C. Test sporadically
(charge user
for maintenance)
D. Release "upgrade"
periodically
(charge user
again)
```

Notice that autoformat got the

hierarchical ordering correct, and that it *didn't* renumber the quoted list, even though it reflowed the

text within the quoted section. That makes sense, since renumbering the quoted list might change its meaning in a way that

reformatting wouldn't.

The $\verb"autoformat"$ subroutine also handles renumbering of lists marked with

Roman numerals. For example, the list:

```
Examples of the five declensions are:

i. terra, terra, terram, terrae,

terra

v. modus, mode, modum, modi,

modo, modo

x. nomen, nomen, nomen, nominis,

nomini,

nomine

ix. portus, portus, portum,

portus, portui,

portu

mmmclxiv. dies, dies, diem, diei,

diei, die
```

would be reformatted thus:

Examples of the five declensions are:

- i. terra, terra, terram, terrae, terrae, terra
- ii. modus, mode, modum, modi, modo, modo
- - iv. portus, portus, portum, portus, portui, portu v. dies, dies, diem, diei, diei,
 - v. dies, dies, diem, diei, diei, die

autoformat is even smart enough to right-justify the numbers, so as to align the paragraph bodies cleanly.

Of course automatically handling

lists of letters and

lists of Roman numerals presents an interesting challenge. A list such as:

```
I. Put cat in box.M. Close lid.P. Activate Geiger counter.
```

should obviously be reordered as I...J...K, whereas:

```
I. Put cat in box.
M. Close lid.
XLI. Activate Geiger counter.
```

should clearly become I...II...III.

But what about:

```
I. Put cat in box.M. Close lid.L. Activate Geiger counter.
```

The autoformat subroutine resolves this ambiguity by always interpreting a list with alphabetic bullets as being English letters, unless the full list contains only valid Roman numerals, and at least one of those numerals is two or more characters long. So the final example above would become I...J...K—as you might have expected.

Quotations

Literary quotations present a different challenge from quoted email. A typical formatter would re-render the following quotation:

```
"We are all of us in the gutter, but
some of us
are looking at the stars"
-- Oscar
Wilde
English playwright
```

like so:

"We are all of us in the gutter, but some of us are looking at the stars" -- Oscar Wilde English playwright

But autoformat recognizes the quotation structure and preserves both indentation and attribution:

```
"We are all of us in the gutter,
but some of us are looking
at the stars"
-- Oscar Wilde
English
```

playwright

It even outdents the leading quotation mark nicely.

Widows

Did you notice that in the previous example, autoformat broke the second line earlier than it needed to? It did that because, if the full margin width had been used, the formatting would have left the last line oddly short:

```
"We are all of us in the gutter,
but some of us are looking at the
stars"
-- Oscar Wilde
English
```

```
playwright
```

Typographical misdemeanors of this type (known as *widows*) are heavily frowned upon in typesetting circles. They look ugly in

plaintext too, so autoformat avoids them with a kind of Dickensian artful dodge: stealing extra words from earlier lines in a paragraph, to provide the widowed word with adequate company.

The heuristic used is that final lines must be at least ten characters long. If the last line is too short, the paragraph's right margin is reduced by one column, and the paragraph is reformatted. This process iterates until either the last line exceeds nine characters or the

margins have been narrowed by 10% of their original separation. In the latter case, the reformatter gives up and just uses its original

formatting.

Justification and Sentencing

The autoformat subroutine can also take an option that tells it how the reformatted

text should be justified. For example:

```
autoformat {justify => 'right'};
```

The alternative values for this option are: 'left' (the default), 'right', 'centre' (or 'center'), and 'full'.

Full justification is interesting in a fixed-width medium like plaintext because it usually results in uneven spacing between words. Typically,

text formatters provide for this by distributing the extra spaces into the first available gaps of each line:

```
R3> Now is the Winter of our
discontent made
R3> glorious Summer by this son of York.
And all
R3> the clouds that lour'd upon our
house In
R3> the deep bosom of the ocean buried.
```

This produces an odd visual effect, so autoformat reverses the strategy and inserts extra spaces at the end of lines (which most readers find less disconcerting):

```
R3> Now is the Winter of our
discontent made
R3> glorious Summer by this son of York.
And all
R3> the clouds that lour'd upon our
house In
R3> the deep bosom of the ocean buried.
```

Even if explicit

centering is not specified via the {justify =>
'centre'} option, autoformat will automatically
detect centered

paragraphs and preserve their justification. It does this by examining each line of the paragraph and asking itself: "If this line were part of a centered paragraph, where would the midpoint have been?"

By making the same estimate for every line in the paragraph, and then comparing the estimates, autoformat can deduce whether all the lines are centered with respect to the same axis of symmetry (with an allowance of plus or minus 1 to cater for the inevitable integer rounding). If a common axis of symmetry is detected, autoformat assumes that the lines are supposed to remain centered, and automatically switches on center-justification for that paragraph.

You can also optionally perform

case conversions on the

```
text being processed, using the case => option. The alternatives are 'upper', 'lower', 'title', and 'highlight'.
```

Title casing capitalizes the first letter of each word:

```
The Strange And Gruesome Case Of The Tab-indented Python.
```

and

highlight casing does the same, except that it ignores trivial words:

The Strange and Gruesome Case of the Tab-indented Python.

A fifth alternative is {case => 'sentence'}. This mode attempts to produce correctly-cased sentences: first letter in uppercase, subsequent words in lowercase (unless that word is originally in mixed case). For example, the paragraph:

```
POVERTY, MISERY, FRIENDLESSNESS, ETC. are
ever
the lot of the VisualBasic hacker. 'tis an
immutable law of Nature! Whom the GODS
would
DESTROY, they FIRST force to code Word
MACROS.
```

under {case => 'sentence' } becomes:

Poverty, misery, friendlessness, etc. are ever the lot of the VisualBasic hacker. 'Tis an immutable law of Nature! Whom the gods would destroy, they first force to code Word macros.

Note that autoformat is clever enough to recognize that the period in abbreviations such as "etc." is not a sentence terminator, and that the first capitalizable letter of "'tis" is the "t," and that words like "VisualBasic" and "Nature" should retain their existing capitalizations.

Future Features

There is an endless list of other smart things

Text::Autoformat could be extended to do. Here's a short preview of some coming attractions:

Columns

A

future release of

Text::

Autoformat will recognize columns within a paragraph and allow the user to independently control their layout and justification, even under margin adjustments. For example, given:

Name	Mark	Comment
====	====	======
Pat	99	Unusually high
score.	Suspect?	
Kim	72	Solid performance
Leslie	51	Just scraped
through	this time	

you'll be able to call:

```
autoformat {justify => ['left',
'centre', 'left'], width => [undef,
undef, 20]};
```

and produce:

Name	Mark	Comment
====	====	======
Pat	99	Unusually high
		score. Suspect?

Kim	72	Solid performance
Leslie	51	Just scraped
through		
		this time

Transliteration

autoformat will eventually provide smart 8-to-7 bit transliteration (the way the Text::StripHigh module does now), so that text like:

```
¥ This exampleÕs © Erwin Schridinger
N1/442(±1) Un≠ertaint" Stra§e, -stland.
```

could be transformed into this:

```
* This example's (c) Erwin Schroedinger,
     No42(+/-1) Uncertainte' Strasse,
Ostland.
```

Mail headers

autoformat was originally developed as a lazy way to clean up incoming and outgoing email. It does that exceptionally well, so long as you keep it away from the headers. Sendmail doesn't take kindly to autoformat's misguided efforts with them:

```
To: Jon Orwant
<orwant@oreilly.com> From:
damian@
conway.org Subject: Re:
When's the next meeting of the
Secret Perl Cabal? References:
```

```
<200011100411.PAA17166@indy05-
.csse.monash.edu.au>
```

A future version of the module will detect mail headers and either leave them alone or wrap them intelligently.

Mark-up

Another irritation is that autoformat blindly attempts to reformat HTML, pod, Perl code, and many other things it should just ignore. The very next release of

Text::Autoformat will have a

"leave-it-the-hell-alone" option that causes autoformat to disregard any (non-bulleted) text that is indented. Later versions may also be able to automatically diagnose marked-up sections of text—and perhaps code examples—and just magically skip them.

Configurability

Currently, the list of abbreviations and "stop words" that autoformat knows about is fixed, as are the set of quoter characters, and list bullets. This should obviously be user-configurable, and will be in a forthcoming release.

Meanwhile, despite these niggles, Text::Autoformat does a remarkably good job at what it was designed for: making ASCII text reformatting as easy as (in)humanly possible.

So you no longer have *any* excuse for sending email that slops over the margin.

Chapter 16. Perl and MIDI: Simple Languages, Easy Music

Sean M. Burke

What Music is Perl?

—Larry Wall, Chapter 2

MIDI (Musical Instrument Digital Interface) is a standard for representing music as a series of notes, rather than as raw audio data. MIDI is to raw audio (e.g., *.au, .wav*, or *.aiff* files) as vector graphics (PostScript) are to bitmaps (e.g., *.png* or *.gif* files).

With MIDI, you can make music without actually having to perform waveform synthesis, just like PostScript lets you draw circles without having to worry about trigonometry. For the purposes of this article, MIDI represents music as a series of events—where each event is basically "turn a note on or off." These events happen at certain times, on a certain channel (what others call a "voice" or "track"), with a certain note number, at a certain volume. Most any computer with a sound card can play MIDI files.

When I first started reading the MIDI internals, MIDI seemed like an ideal format for composing music in Perl, so I set out to make routines for encoding to and decoding from the MIDI binary format. In August 1998, I uploaded to CPAN my first release of the imaginatively named "MIDI-Perl," a mostly object-oriented interface to these encoding and decoding routines. However, as I stood back from the pile of code, I realized that while I had created a fine object model for representing MIDI files, the most basic data structure, the MIDI event, was no more suited to musical composition than raw PostScript is for composing architectural blueprints.

Hard Things Possible

Imagine that you need to examine a series of addresses and save any located in the U.S. to a file. You might do that with a program like this:

```
open US, ">us.txt" or die "Can't write to
us.txt!";
while (<DATA>) {
    next if /^\s*\#/s;
    chomp;
    my ($address, $url, $country) = split(/
\s+/, $, 3);
    next unless $country =~ /^U\.?S\.?/i;
    print US "$address: $url\n";
}
 END
# List of addresses, in the format:
# Address
                   URL
Country
foo@bar.org http://bar.org
staff@tpj.com http://tpj.com
                                        U.S.
                                        USA
# We'll want to ignore this one:
pati@lebar.fr http://www.lebar.fr
France
```

This program basically implements a language; after its _______ END_____ are some "instructions" in that language. Now, the

language is simple: lines are ignored if they start with optional whitespace and then a # sign. If a line can be split into three non-null fields, it's considered an instruction to save the first two fields, but only if the last field begins with US or U.S. or us. This language doesn't have anything like what we'd expect in a real programming language—it has no flow-control structures, no variables, no ability to define procedures. But it is a language in the sense that it has a syntax (an order things have to be in to make sense) and semantics (what the elements in the language mean).

Now, the semantics here are quite restricted, so it's a language only in the same sense that a simple markup language like HTML is a language, or the way that the notation for recording chess moves is a language, or the way that common music notation is a language. None of these are languages you could write a program to calculate 5×12 in, but each has its purpose.

None of these languages makes hard things possible. For example, if you wanted to represent the behavior "save the output of an address if today is Tuesday," you'd be out of luck. You could change the program so that it would work, but then it would be a different language.

Now, there are two ways to make a language more flexible. You can make a *novel* language, or an *extensional* language.

And so I found myself in the not-entirely-anticipated position of designing and implementing a language that would provide an interface for musical composition, a language that would use

MIDI for output but be several levels higher in abstraction.

Now, it's the job of language designers to construct their languages so that they will—to quote Larry Wall—"make easy things easy, and hard things possible." What's implied is that there are two domains that a programming language bridges: the way we think of a problem, and the way we code up the program that solves the problem. Making "hard things possible" is a matter of making the language open-ended enough that one can do just about anything with it, while making "easy things easy" is a matter of making sure that simple ideas can be coded concisely.

In this article I'll first explain my ideas about types of simple languages, and then from that I'll show how these ideas led me to design the

MIDI::Simple language the way I did. My goal is not so much to document

MIDI::Simple, but instead to illustrate how specific problems in language design led me to make MIDI::Simple the way it is. Readers whose interests do not include both language theory and music can skip sections as they wish.

Approach 1: A Novel Language

A Novel Language is what we have already: a language independent of whatever you implement it in. Here, we've implemented our language in Perl, but none of Perl's power is available to it. Now, if you want the language to do more, you could write it so that it has flow control, like Perl's if statements and while loops. If you've never done anything like this before, it seems quite daunting—and it is! (Well, not *too* daunting, or else there wouldn't be so many programming languages around. Think of all the early-80s BASICs written in machine language and fitting in a few kilobytes of ROM.)

Now, this is not to say that implementing just any high-level language in Perl is trivial—not by a long shot—but implementing a language with simple syntax and simple semantics, like a LOGO dialect, or even a simple Lisp dialect (as in Abelson & Sussman 1996), is quite doable. Or, you could create an

extensional language.

Approach 2: An Extensional Language

An Extensional Language is what I call a language that is really an extension of Perl, provided as subroutines in a Perl library, module, or class. Now, it might strike you as arrogant to say that Perl plus five subroutines from require "my great lib.pl" constitutes a new language. First off, I didn't say it was an entirely new language. Second, it does what you want, and that's probably all you're interested in, right? The advantage to using an Extensional Language is that you're still in Perl, so you get to use all of its features—variables, flow control. and The SO on disadvantage is that you have to work within the confines of Perl syntax. However, Perl syntax is so free-form that this is not a problem.

There's a parallel here in the creation of jargons: Chemists, when they needed a language to talk about chemical structures with precision, could have decided to make up a whole new rich Novel Language, with verb tenses and noun compounding and all the goodies we expect in a natural language. But that would mean figuring out the best way to implement predicate argument structure and pronoun binding and phonology and all the other cruft of natural language—the same way that making a Java virtual machine in Perl means having to implement exception handling.

Instead, chemists implemented chemical names as an Extensional Language based on English. Now, "boron trifluoride" isn't exactly core English, but it's more English

than it is Thai or Klingon; and, I admit, when the chemical "extension" to English starts spawning forms like "dichloro-1,2-ethane" and

"2-chloro-4-ethylamino-6-isopropylamino-1,3,5-triazine," it begins to leave every natural

language behind. But when a chemist says "boron trifluoride is an inorganic gas," you understand that *something* is an inorganic gas, even if you can't picture what a boron trifluoride molecule looks like. Similarly, if you're reading Perl code and you see &funk_it, or \$this->thingify, you probably can't guess what funk_it or thingify do, but at least you can tell they're subroutines.

Easy Things Easy

Consider these

MIDI events, expressed as they would be sent to the MIDI encoder in MIDI::Event:

```
# event, wait-time, channel, note number,
volume
['note_on', 0, 1, 25, 96],
['note_off', 96, 1, 25, 0],
['note_on', 0, 1, 29, 96],
['note_off', 96, 1, 29, 0],
['note_on', 0, 1, 27, 96],
['note_off', 96, 1, 27, 0],
['note_on', 0, 1, 20, 96],
['note_off', 192, 1, 20, 0],
```

...and so on, for a total of thirty-three such lines. I won't explain the exact details of this format, as it's not relevant to the rest of this article, but obviously requiring a composer to write notes like this is not making easy things easy. A more intuitive formulation of notes would involve these qualities:

- Duration: quarter note, half note, and so on
- Pitch: a note-letter and octave number, such as A5
- Volume: either a number or one of the common abbreviations from sheet music, such as mf for medium-loud
- Channel number; for MIDI, a number between 0 and 15

In my early attempt at a music language (implemented as a Novel Language with an interpreter written in Perl), I ended

up with a language that looked like this. Each note occupies a line of its own and specifies the channel, volume, duration, and pitch in order:

```
note c1 f qn Cs2
# Cs2 = C sharp, octave 2
# qn = quarter note
note c1 f qn F2
note c1 f qn Ds2
note c1 f hn Gs1
# yup, hn = half note
note c1 f qn Cs2
note c1 f qn Ds2
# ...and so on...
```

Incidentally, this tune I'm trying to represent is the Westminster Chimes (Figure 16-1, graphic courtesy of Matthew A. Siegler), better known as what fancy clocks play when they chime the hour. See Sturdy 1998.



Figure 16-1. The Westminster Chimes

Now, what makes these lines of note-making code different from typical lines of code in a high-level programming language is redundancy. Duration, pitch, volume, and channel often stay the same from one event to the next. So, in the name of brevity, I decided that each note's properties should be inherited from the previous note. And so, in the new language thus defined, the above code can be rephrased as:

```
note c1 f qn Cs2
note F2
note Ds2
note hn Gs1
```

```
note qn Cs2
note Ds2
# ...and so on...
```

I was on the way to a workable language. But then, when I wanted to expand it, I arrived at the same question raised in the first example of this article—do I continue with my Novel Language, adding primitive flow control, functions, variables, and data structures? Or do I implement an

Extensional Language? I had two considerations.

First, I wanted my language to allow *algorithmic composition*—

composition where the notes are partially determined by the output of algorithms developed by the composer. Composers like Xenakis use algorithms that involve some fairly complex algebra. If I wanted my simple Novel Language to do anything like that, I'd need to implement algebraic capabilities for that language. I'd also need to provide mathematical functions like sines and logarithms, and data structures plus the functions to manipulate them. That sounded like a lot of work.

Second, both kinds of

music languages exist—very simple languages, notably the *abc* language (Walshaw 1998); and composition

extensions to high-level languages, notably the

Forth-extension HMSL (Burk 1998). But, I asked myself, which kind of language would be most useful to the world? The *abc* language seemed well designed, implemented, and supported by various ports and utilities. So implementing a simple Novel Language would be reinventing the wheel. As for Extensional Languages, HMSL seemed the richest—I was even willing to learn Forth to deal with it! However, the

implementation of Forth that it's based on works well only on Macs and Amigas. That ruled it right out, as I don't have either. There are other compositional languages, but they're either based on languages I wouldn't want anyone to have to compose in (e.g., C++), or they focus on acoustics and digital signal generation (e.g., Common Lisp Music) more than I or MIDI care to.

But if I implemented my

music language as an

Extensional Language based on Perl, composer-users could write their programs and be able to run them on any Perl interpreter, on all the platforms Perl runs on. People new to programming would have at their disposal all the documentation and support that all Perl novices have. Algorithmic composers would have at their disposal all the data structures and mathematical functions Perl provides. And—no small consideration—it'd save me a ton of work that would otherwise go into creating the Novel Language's rudimentary data structures and functions.

So the Extensional Language approach won. I scrapped the Novel Language and reimplemented its semantics (and a lot more) as the

MIDI::Simple module that comes in the MIDI-Perl suite, Version 0.7 or later. The remainder of this article gives a guided tour of its features, with insights into my thinking as I designed the module.

Behold MIDI::Simple!

For all the tortured thought I put into its design,

MIDI::Simple ended up easy to implement. It was almost anticlimactic. (Maybe I should have done it in C++ so that it would given me a heroic sense of accomplishment!) MIDI::Simple is only 600 lines of relatively unsurprising code—much of it highly redundant. Its only quirk is that it provides *both* a procedural and an object-oriented interface—I wanted beginners to be able to use a purely procedural interface, and advanced users to benefit from the power of object-oriented design if they wished.

MIDI::Simple operates on a data structure I call a *score*, which is basically a list of events and times, and some state variables for storing defaults. Here's a sample of the state variables:

\$Time

The time (measured in *ticks*, each 1/96th of a quarter note) at which the next event will take place

\$Duration

The number of ticks in the next note or rest

\$Channel

The channel number of the next note

\$Volume

The volume of the next note, from 0 to 127

@Notes

The pitches added to the score by the next call to the n routine, discussed below

Now, I expect that only the most advanced users will have to deal with the contents of a score directly, because everyone else can use this interface:

- new_score, which initializes a score object—a score plus the state variables
- n and r, to add notes and rests to the score
- functions to add arbitrary MIDI events to the score, such as patch_change to set the patch (a simulated instrument, like a piano or banjo)
- write_score, to write the newly composed score as a MIDI file
- read_score, to read a single-track MIDI file as a score
- synch, to take the output of several user-provided functions and place them in the score at the same time—useful for blending different instruments

Using the language specified as above, the Westminster Chimes could be notated like this:

```
use
MIDI::Simple;
new_score;
patch change 1, 8;
```

Set

```
Channel 1 to Patch 8 = Celesta
n c1, f, qn, Cs2; n F2; n Ds2; n hn, Gs1;
n qn, Cs2; n Ds2; n F2; n hn, Cs2;
n qn, F2; n Cs2; n Ds2; n hn, Gs1;
n qn, Gs1; n Ds2; n F2; n hn, Cs2;
write score 'chimes.mid';
```

Much more concise than the forty-odd lines of code you'd need to express low-level MIDI events and calls!

"Relative" Notes

At this point I noticed that in most music, notes aren't thought of as belonging to a particular numbered octave so much as a *current octave*, with notes being in it, below it, or above it. So I added another way to specify pitches: instead of number (n25) or note-and-octave (Cs2), one can specify them in terms of just note-letter: Cs, meaning "C sharp in the current octave."

I call this a *relative* note specification, in distinction to Cs2 and n25, which I call *absolute*.

The current octave is a number stored in a state variable called SOctave, which can either be set directly or by calling n or r with a parameter in the form o6 (where the 6 can be replaced by any number from 0 to 10) or an absolute note specification. All of these set SOctave to 2:

```
$Octave = 2;
$Octave = 3; --$Octave;
n o2 Cs;
n Cs2;
r n25;
```

I also added a way to denote "... an octave above the current one" or "... an octave below the current one":

```
Cs_ul ("u" for "up")
Cs dl ("d" for "down")
```

where 1 can be replaced by any positive integer, just so long as the resulting note is within the note range of MIDI devices: C0 to G10.

The result is that these four notes:

n c1, f, qn, Cs2; n F2; n Ds2; n hn, Gs1; can be expressed as:

\$Octave = 2; n c1, f, qn, Cs; n F; n Ds; n hn, Gs_d1;

or as:

n c1, f, qn, Cs2; n F; n Ds; n hn, Gs_d1;

In the last line, only the first note is specified absolutely, setting the current octave to 2. This means that all you have to do to move all these notes up two octaves is change Cs2 to Cs4.
Percussion, Uniformity, and noop

MIDI has a special reserved channel,

channel 9, where numbers for pitches are interpreted as a special percussive instrument. For example, n35 (that is, B2) on channel 9 doesn't mean a B2 on the current patch for channel 9, but instead a (largely untuneable) note on an acoustic bass drum. Lines of code to generate bunches of

percussion notes often look like this: n c9, ff, n41, qn; r; n; r;. This specifies a quarter note, a quarter rest, a quarter note, and a quarter rest, all played on the acoustic bass drum.

However, this seemed like a violation of *uniformity*, a principle I learned about from Weinberg's

The Psychology of Computer Programming. Weinberg holds that users expect things that look similar to do similar things, and things that look different to do different things. But the first call to n above looks very different from the second and third calls to n even though all they do the same thing. What I wanted was a way to set up all the state variables, and then be able to just say n; r; n; r;, perhaps like this:

```
$Channel = 9; $Volume = 112; $Duration =
96; @Notes = (41);
n; r; n; r;
```

But that seemed inelegant. What I ended up doing was adding another function, called noop (for "no operation"). noop parses options just like n and r, and has all the same side effects, but doesn't actually affect the score. For example, consider these three lines of code: n qn, C3; # C3 = n36, by the way r qn, C3; noop qn, C3;

The first line adds a note to the score, and increments \$Time by the duration of a quarter note. The second line just increments \$Time because it's a rest. The third line alters neither the score nor \$Time, but has all the same side effects as the first two: it sets \$Duration to the duration of a quarter note, and it sets @Notes to (36). With noop, you can write code like this:

This not to say that you *have* to do it this way, but allowing the organization of code to reflect different ways of organizing thought is the Perl way.

The Object-Oriented Interface

So far I've described functions (or procedures, really, seeing as how they don't have useful return values, and have more side effects than radical chemotherapy) for manipulating a score and for setting the state variables that ride along with the score.

This is great if you just want to manipulate one score at a time. For

manipulating several scores, I've provided the OOP interface shown in Table 16-1.

Procedural	ООР
new_score	<pre>\$score = MIDI::Simple->new_score</pre>
\$Channel = 3	\$score->Channel(3)
\$Octave = 4	\$score->Octave(4)
@Notes = (30,34)	\$score->Notes(30,34)
push @Notes, 36	<pre>push @{ \$score->Notes_r }, 36 or \$score->Notes(\$score->Notes, 36)</pre>

Table 16-1. Procedural interface and OOP equivalents

Procedural	OOP
n qn, Cs3	\$score->n(qn, Cs3)
noop o7, ff	\$score->noop(o7, ff)
write_score 'X.mid'	<pre>\$score->write_score(`X.mid')</pre>

Using synch, and Some Actual Music

synch takes a list of code references (generally of the form \&foo, where foo is a subroutine the user has defined for adding notes to the score). For each coderef, synch calls the user's routine, and then notes the value of \$Time. After calling all the routines, synch advances \$Time as necessary. In other words, it makes multiple subroutines *synchronous*—occurring (or at least starting) at the same time.

Each user routine, incidentally, should expect its first parameter to be the score object, and should add to that score object via the object-oriented interface to

MIDI::Simple instead of the procedural interface. A simple use for synch might look like this:

```
use MIDI::Simple 0.7;
new_score;
@subs = (\&tom_two, \&double_clap);
foreach (1 .. 10) { synch(@subs) }
write_score("rhythm1.
midi");
exit;
sub tom_two {
    my $it = shift;
    # n41 on c9 = low floor tom
    $it->n(c9, ff, n41, qn); $it->r;
    # qn = 1/4 note, ff = very loud
```

```
$it->n(f); $it->r;
# f = loud
}
sub double_clap {
    my $it = shift;
    # n39 on c9 = hand-clap
    $it->n(c9, ff, n39, sn); # sn = a 16th
note
    $it->n;
    # This only takes up 2 16th-notes of
time, but that's fine.
}
```

Now, this generates twenty monotonously identical measures. My instrument subroutines vary their effect from measure to measure because of a trick: The first subroutine is a call to a measure counter, and the other subroutines pay attention to it. Here's the measure counter:

```
sub measure_counter {
    my $it = shift; $it->r(wn); # a whole
rest
    ++$measure;
}
```

Using the measure counter, tom_two can now do two different things, depending whether \$measure is greater than 4:

```
$it->n(f); $it->r;
}
```

Then we just change this line in the program:

```
@subs = (\&measure_counter, \&tom_two,
\&double clap);
```

And voilà, simple percussion. From there it's not hard to get more ornate:

```
use
MIDI::Simple 0.7;
new score;
@subs = ( \&measure counter, \&boom,
\&tboom, \&clap );
foreach (1 .. 24) { synch(@subs) }
write score("rhythm2.midi");
exit;
sub measure counter {
    my $it = shift;
    $it->r(wn); # a whole rest
    ++$measure;
}
sub boom {
    my $it = shift;
    return if smeasure  4 < 2;
    $it->n(c9, ff, n41, qn); $it->r;
    $it->n(f); r;
}
sub tboom {
    my $it = shift;
    return if measure  4 < 2;
```

```
# 42 = 'Closed Hi-Hat' ; 43 = 'High
Floor Tom!
    # In guick succession...
    $it->n( c9, ff, n43, sn); $it->n( n42
); $it->r(dqn);
    # dqn = dotted quarter note/rest
     $it->r( c9, ff, n43, sn); $it->n( n42
); $it->r(dqn);
}
sub clap {
    my $it = shift;
    return if $measure < 4;</pre>
    $it->n(c9, ff, n39, sn); $it->n;
    $it->r(dgn);
    $it->r(hn);
}
```

Now, I promised that I'd show you a little Novel

Language

based on an

Extensional language. While I was tossing together the above code, originally just to test synch's functionality, I decided I wanted a more complex instrument. For some reason, I had the rhythm of the Talking Heads' "Psycho Killer"

stuck in my mind, and decided to code it up. I tried it with combinations of eighth notes and rests, but I couldn't quite do it. So I made a very simple Novel Language where whitespace is ignored, ! means to hit the "side stick" (note 37 on that magic channel 9) for a sixteenth note, and anything else makes a sixteenth rest:

```
sub psycho {
    my $it = shift;
    my $pattern = " !.!. !.!. !.!. ";
# just a start
    $pattern =~ tr<\cm\cj\t ><>d; # kill
```

```
whitespace
    warn "<$pattern> doesn't add up to a
whole measure\n"
    unless length($pattern) == 16;
    $it->noop(c9, mf, n37, sn);
    # setup: n37 on c9 = side stick
    foreach (split('', $pattern)) {
        if ($_ eq '!') { $it->n } else {
    $it->r }
    }
}
```

From here I just monkeyed around with the quoted string on the line after my \$pattern.I eventually arrived at:

" !.!.!. !!!!!! !.!. " ;

This was exactly the rhythm I was thinking of! To hear it, paste the psycho subroutine into the program above, and add it to the @subs line, like so:

```
@subs = ( \&
measure_counter, \&psycho, \&boom,
\&tboom, \&clap );
```

Mod, Canons, and Rounds

Having your subroutines use *\$measure* to decide what notes to generate is the most straightforward way to produce higher-level structures in

music. Earlier, you saw a subroutine use \$measure \$ 4 < 2 to control what notes it adds to the score. The % operator ("modulus,"

also known as "remainder"—X % Y means "the remainder of dividing X by Y") is one that a surprising number of people are unfamiliar with, but it's absolutely necessary for algorithmic composition. Consider the

Westminster Chimes' four measures:

```
my @phrases = (
    [ Cs, F, Ds, Gs_d1 ], [Cs, Ds, F, Cs],
    [ F, Cs, Ds, Gs_d1 ], [Gs_d1, Ds, F, Cs]
);
```

These four measures can be repeated ad infinitum (starting, appropriately, with the first measure), with this code:

If you change (\$measure + -1) % 4 to (\$measure + 0) % 4, everything happens one measure later than before. Change -1 to 1 and it's two measures later. That way, when the piece starts, with a \$measure of 1, you get the third element of <code>@phrases</code>. Since you probably want it to keep quiet until measure 3, just add return if \$measure < 3; to the start of the subroutine.

Now you can have several subroutines like &first that play all the same notes, but in different measures. This structure, called a *canon*, may seem very abstract, but it's common in songs, where it's called a *round*. If you remember singing "Row, Row, Row Your Boat,"

you were singing exactly the same kind of musical structure you get out of &first, copied and adjusted for different Nsin (\$measure + N) % 4. Compare &first, &second, &third, and &fourth in the program shown on the previous page, which plays a round based on the Westminster Chimes.

```
@bass line = ( F, Cs, Ds, Gs d1, Gs d1,
Ds, F, Cs );
new score;
# Some
MIDI meta-information:
copyright text event "1998 Sean M. Burke";
text event "Title:
Westminster Round":
# Patch inits:
# Patch 16 = Drawbar Organ. 8 = Celesta.
patch change 0, 16;
patch change 1, 8; patch change 2, 8;
patch change 3, 8; patch change 4, 8;
for (1..8) { synch(\&count, \&bass,
\&first, \&second, \&third, \&fourth); }
                                          #
r hn;
Pause, take a bow!
write score("round2c.mid");
dump score;
exit:
sub count {
    my $it = shift;
    ++$measure;
    $it->r(wn); # whole rest
}
sub first {
    my $it = shift;
    $it->noop(c1,mf,o3,qn);
    my phrase number = (smeasure + -1) %
```

```
4;
                       @phrase
                   mγ
                                   =
@{$phrases[$phrase number]};
        foreach my $note (@phrase) {
$it->n($note) }
}
sub second {
   my $it = shift;
   return if smeasure < 2 or smeasure > 5;
   $it->noop(c2,mf,o4,qn);
   my phrase number = (smeasure + 0) \% 4;
                   my @phrase
                                    =
@{$phrases[$phrase number]};
        foreach my $note (@phrase) {
$it->n($note) }
}
sub third {
   my $it = shift;
   return if $measure < 3 or $measure > 6;
   $it->noop(c3,mf,o5,qn);
   my $phrase number = ($measure + 1) % 4;
                   my
                      0phrase
@{$phrases[$phrase number]};
        foreach my $note (@phrase) {
$it->n($note) }
}
sub fourth {
   my $it = shift;
   return if smeasure < 4 or smeasure > 7;
   $it->noop(c4,mf,o6,qn);
   my $phrase number = ($measure + 2) % 4;
                   my
                      @phrase
                                    =
@{$phrases[$phrase number]};
        foreach my $note (@phrase) {
$it->n($note) }
```

```
}
sub bass {
    my $it = shift;
    my $basis_note = $bass_line[($measure
- 1) % 4];
    $it->noop(c0,fff,o3, wn); # fff = REAL
LOUD.
    $it->n($basis_note);
}
```

Future Features

MIDI::Simple is by no means finished. There are three areas where I hope to improve it:

- The language has plenty of functions for dealing with notes, but larger structures are either oddly implemented (as with set_tempo and time_signature) or missing (there are no functions for *crescendo* or *sostenuto*, for example).
- 2. n, r, and noop accept abbreviations like qn, hn, and den, for quarter note, half note, and dotted eighth note. However, these are counterintuitive if you call a quarter note a *crotchet* (as much of the English-speaking world does), or, for that matter, a *Viertelnote, semiminima, negyed hangjegy*, or a *neljännesnuotti*. Future versions of MIDI::Simple will provide an interface for adding or

changing abbreviations.

3. At present, MIDI-Perl generates files only, and can't send to other

MIDI devices, since interfacing with devices attached to a MIDI port requires OS- and hardware-specific programming well beyond what I can produce or support. However, Alex

McLean has done some fine work getting this working under Linux; see his MIDI::Realtime module in CPAN. As examples of applications based on real-time MIDI access, programs using MIDI::Simple can generate real-time streams that portray any incoming numeric data stream, such as stock market data or the status of a computer network.

References

Abelson, Harold, and Gerald Jay

Sussman. *Structure and Interpretation of Computer Programs*, 2nd ed. MIT Press, 1996. The full text is now available online at http://mitpress.mit.edu/sicp/

Burk, Phil. *HMSL, The Hierarchical Music Specification Language*. http://www.softsynth.com/hmsl/.

Langston, Peter S. "Little Music Languages." In *Handbook of Programming Languages*, vol. 3, Peter H. Salus, editor. Macmillan Technical Press, pp. 587-656, 1998.

Sturdy, John C. G. *The Cambridge Chimes*, 1998. http://www.ely.anglican.org/parishes/camgsm/bells/ chimes.html.

Walshaw, Chris. *The abc Music Notation Language Home Page*. http://www.gre.ac.uk/~c.walshaw/abc/.

Weinberg, Gerald M. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.

Xenakis, Iannis. Formalized Music: Thought and Mathematics in Composition. Stuyvesant, NY: Pendragon Press, 1992.

Chapter 17. Braille Contractions and Regular Expressions

Sean M. Burke

Sheri Wells-Jensen

This article is about how I used regular expressions to turn common English text into Braille, correctly using all the possible shorthand-like contractions and abbreviations that Braille provides. This is basically a case study in how a messy problem got solved straightaway with Perl—but I'll also mention some of the new features to be found in Perl 5.005's regular expressions, which I used along the way; and in the end, I find a surprising commonality between regexes and natural language writing systems.

Braille and Contractions

When I was a little kid, I had a children's book about Helen Keller. I don't remember reading it; I just remember the back cover, which had the Braille alphabet printed on it—well, embossed, actually. They had the Roman letter "a" in ink, and then below that the embossed dot pattern for Braille "a", and so on up to "z". So I got the idea that Braille printing is just like a letter-for-letter substitution cipher for the Roman alphabet.

Then I started noticing on men's room doors that below "MEN" in big Roman letters, there'd be the same word in Braille—but sometimes the word would have three Braille characters, and sometimes just two. And that I found perplexing. I couldn't imagine how the word "men" could end up only two characters long.

So I asked my friend

Sheri, who's been reading and writing Braille since she was a kid, how "men" could be just two characters long. She explained that Braille has contractions: the first character in "men" is "m", and the second character is a contraction for "en". (For the rest of this article, I'll use wedges to denote contractions, so "<en>" will mean the single Braille character that is a contraction for the two Roman letters "en".) Braille readers typically use the term "sign" to mean a Braille character.

Moreover, some contractions are context-specific—there's a contraction for "com", but it can apply only at the start of words, as in "computer." These shorthand-like contractions make reading

Braille faster, and they make books in Braille less expensive and less bulky.

There are contractions for "en", "st", "ea", "er", "the" (the letter sequence, not just the word "the"), and lots more. It occurred to me that there were cases where I could imagine more than one way to apply contractions. Consider the two ways to encode the word "leather":

```
l e a t h e r
. \ / / / / /  (contract "ther" as
"><er>";)
```

leather . _/ ___/ . "<the>r") (contract "ther" as

In each case, you end up with a word four letters long, so brevity can't be a tie-breaker. I asked

Sheri whether both were acceptable spellings and she said no; the second spelling, with "<the>r", is the only correct way to encode "leather". This reminded me of English

hyphenation. For example, you could hyphenate "boggling" as "bogg-ling" (sort of like "gos-ling", maybe), but you could also hyphenate it as "bog-gling", and that seems the best way to do it, so that gets to be the correct way.

or.

Sample Text in Braille

For a more in-depth Braille sample, let's look at the first paragraph of Chapter 3 of Sincair Lewis' *Babbitt*. The English text is below, and the Braille version is shown in Figure 17-1.

It was the best of nationally advertised and quantitatively produced alarm-clocks, with all modern attachments, including cathedral chime, intermittent alarm, and a phosphorescent dial. Babbitt was proud of being awakened by such a rich device. Socially it was almost as creditable as buying expensive cord tires.

Under each Braille line in Figure 17-1 is the "Braille ASCII" version—simply the Braille characters mapped onto 7-bit ASCII codes. You'll notice that the character mapped to "5" is the contraction "<en>"; the character mapped to "/" is the contraction "<st0>"; and the character mapped to "," means "the next word is capitalized" when it precedes a word, but on the inside of words, it's a prefix for particular abbreviations (e.g., it plus "y" means "-ally").

The e-text of all of *Babbitt* is 717K unencoded (ASCII), and 581K encoded (grade-2 Braille, in Braille ASCII)—a size savings of 19%.

There are even

alphabet blocks for Braille, shown in Figure 17-2. (Thanks to Tack-Tiles (http://tack-tiles.com) for donating the set for Sheri's continued research on Braille, and thanks to David Ondrik for taking the picture of them.)

Figure 17-1. The first paragraph of chapter 3 of Sinclair Lewis' Babbitt, in Braille



Figure 17-2. Alphabet blocks for braille, spelling out "The Perl 5 compiler reached line number 850 and died."

Linguistic Rule Systems

I was curious how the rules for Braille could be implemented as an encoding algorithm.

Sheri explained that programs do exist for taking conventional English

text, and applying all the contractions possible, in the right places—what I'll just call "Braille encoding."

Braille

encoding programs can be embedded in tactile displays for computers, so that when I send

Sheri email (unencoded), it shows on her display in encoded Braille. Or a Braille encoder can be a program that encodes a whole file at a time, typically for preparing an unencoded text file for sending to a Braille embosser, a kind of impact printer.

One encoder program is from the National Federation of the Blind, and is called

NFBTRANS. NFBTRANS is by no means the only such program around. Another widely used program is DBT,

Duxbury Braille Translator.

DBT, however, is a commercial product, with no source available. NFBTRANS is free, and comes with source code.

I hunted down a copy of NFBTRANS, and looked at the source code that came with it, and discovered that it was not just a Braille encoder, but more a typesetting system like troff. It was about seven thousand lines of C, and in the guts of it, I did manage to find the encoding routines—and I couldn't make heads or tails of them. NFBTRANS reads the list of contractions from an external data file that come with the program, but I couldn't figure out exactly how the rules were applied—namely, in what order or priority.

In other words, the file had rules like:

```
"the" contracts in any context
"er" contracts in any context
"th" contracts in any context
```

but how or why would NFBTRANS manage to always contract the end of the word "leather" to "<the>r" and not "<er>"? Were there other "-ther-" words that could get ">er>" instead of "<the>r"?

To fill in the blanks in my understanding of Braille, and to try my hand at modeling a linguistic system in Perl, I decided to try to develop an algorithm in Perl to correctly apply these rules, and to check them against NFBTRANS's output.

I'd had some experience with formal linguistic models of the subsystems that make up natural languages. Every time I'd seen a linguist create a formal model of a subsystem of a language (whether English phonology, or Hindi syntax, or something less abstract, like hyphenation), the rules that model the system have been in one of the two kinds of frameworks I'll explain below—either the kind I call a "generativity system," or the kind I call an "optimality system." So I expected Braille encoding to be one of these two sorts of rule

systems, and the first task was to figure out which it was.

Generativity Systems

In a generativity system, there's a list

of rules, to be applied in a particular order. Linguists often use this kind of rule system to model phonology, the study of the way the sounds of language are represented in the brain. Phonological models assume that words are stored in a somewhat simplified, abstracted form, and that to work out the details of how these are to be pronounced, we apply a series

of rules.

For example, a phonological model of my dialect of American English would assume that the "long i" sound is always stored as simply that. But I, like most Americans, pronounce it differently in "kite" than in "fly." To account for this difference, phonological models say that this sound starts out as just itself (here written "/aj/"), but that rules may apply to change it in some contexts. This isn't just about vowels—the two different-sounding t's in "write" and in "writing"

supposedly both start out as just /t/, but a rule happens to change the latter one to sound like a /d/.

The rules describing these two phonological changes could look like this:

- The diphthong /aj/ (as in "fly") changes to /Uj/ (as in "kite") when it's before /p/, /t/, or /k/.
- The consonant /t/ changes to /d/ when it's between two vowels.

These rules, when applied in that order, are supposed to take words in a representation that doesn't have these finer distinctions (like the difference between /aj/ and /Uj/), and change those representations to flesh them out with those distinctions.

Now, translating rules like these into regular

expressions used to be hard, but now, with Perl 5.005 regexes, it's a snap, with "lookbehind" and "lookahead":

```
s/ai
                       # Target: the vowel
/ai/
  (?=[ptk])
                      # Following context:
/[ptk]/
         && print "R1 applies. Output:
 /Ui/ax
$ \n";
s/(?<=[aeiouIUj])
                    # Preceding context:
any vowel
                    # Target: /t/
 t.
                      # Following context:
  (?=[aeiouIUj])
any vowel
 /d/qx && print "Output of system: $ \n";
```

What this gets you over /aj[ptk]/ or /[aeiouIUj]t[aeiouIUJ]/ is that with lookbehind (as in (?<=[aeiouIUj])) or lookahead (as in (?=[aeiouIUj])), the

text matching the lookahead and lookbehind parts of the regex aren't part of what gets matched by the regex as a whole.

That is, text that matches the lookbehind expression is free to precede the "cursor" (the part of the regex mechanism that pos reports the position of), and text matching the lookahead expression doesn't move the cursor forward, as normal matching does. (Also, the lookbehind text doesn't end up in

\$ a, which you probably know as the "what matched?" regex variable, but which you can better think of as, "Now that we've matched something, what was between where the point started and where it ended up?")

Now, the above

phonetic rules are greatly simplified for the sake of discussion, but you can see their application to "write + ing", starting from its simplified abstract form /rajt + IN/:

```
$_ = 'rajtIN';
print "Input to system: $_\n";
s/aj
 (?=[ptk])
 /Uj/gx && print "R1 applies. Output:
$_\n";
s/(?<=[aeiouIUj])
 t
 (?=[aeiouIUj])
 /d/gx && print "R2 applies. Output:
$_\n";
```

print "Output of system: $\n";$

This prints:

Input to system: rajtIN R1 applies. Output: rUjtIN R2 applies. Output: rUjdIN Output of system: rUjdIN

And this gives the correct pronunciation for "writing" in my dialect, /rUjdIN/. Change the first line to "rajdIN" ("riding"), and neither rule applies, and you get "rajdIN" out. (So, yes, when I speak, "writing" and "riding" sound different.) And, importantly, if you swap the rules so that R2 applies before R1, you get:

```
Input to system: rajtIN
R2 applies. Output: rajdIN
Output of system: rajdIN
```

(So, if *your* dialect has "writing" and "riding" sounding the same, it might be that rule-ordering is the only difference between your dialect and mine.)

The

ordering of the rules in

generativity systems is crucial; if you have the right rules in the wrong order, you get the wrong answer. If

Braille encoding were a generativity system, I'd need to figure out how to order the rules from the NFBTRANS data table.

Optimality Systems

If a generativity system is one that gives rules that get you from input to correct output, then an optimality system is one that takes all kinds of even remotely conceivable possible output, and ranks them in order of desirability ("optimality"). The highest ranked one is the "correct form" and becomes the output of the system. (And if it seems to you that generativity is like imperative programming, and optimality is like logical programming—à la Prolog—then you're basically right.)

The algorithms I've seen that implement English hyphenation are basically optimality systems. I encourage interested readers to look at the hyphenation algorithm in TeX (which you can see reiterated in Perl in CPAN's TeX::Hyphen), but for the sake of discussion, suppose you can model English hyphenation with these

rules for ranking candidate forms:

- Hyphenating between consonant letters is good (as in "gos-ling").
- Hyphenating between a double consonant is good (as in "bit-ter").
- Hyphenating between a consonant and a vowel is bad (as in "gosl-ing").
- If hyphenating leaves a word fragment of just consonants, that's *really* bad (as in "g-osling" or "gosli-ng").

In Perl, you could implement this as:

```
use strict;
my $in = 'boggling';
my $best = $in;
my $best_score = 0;
my $Cons = 'bcdfghjklmnpqrstvwxz';
my $Vowel = 'aeiouy';
foreach my $i (1 .. (length($in) - 1)) {
    $_ = $in;
    my $score = 0;
    substr($_, $i, 0) = '-';
    ++$score if /[$Cons]-[$Cons]/oi;
    ++$score if /[$Cons]-[$Cons]/oi;
    ++$score if /[$Cons]-[$Vowel]/oi;
        $score -= 10 if /^[$Cons]+-/oi ||
/-[$Cons]+$/oi;
```

```
print " \"$_\" : score: $score\n";
if($score > $best_score) {
        $best_score = $score;
        $best = $_;
    }
}
print "Best: \"$best\" with score of
$best score\n";
```

The output of this is:

```
"b-oggling" : score: -11
"bo-ggling" : score: 0
"bog-gling" : score: 2
"bogg-ling" : score: 1
"boggl-ing" : score: -1
"bogglin-g" : score: -10
"bogglin-g" : score: -9
Best: "bog-gling" with score of 2
```

These

rules seem to work right—the highest-ranked form ("bog-gling") is the best; and incidentally, the second best ("bogg-ling") is not too bad, and from there on out it's all quite bad ("bo-ggling", "boggli-ng", etc.).

Note that it doesn't matter here what order you apply the rules in; it just matters what weight gets attached to each rule. In the above example, I've kept it simple, but suppose we now add a rule that means "the closer to the middle of the word you hyphenate, the better," such as:

```
$score += .5 * (length($in) - abs($i -
length($in) / 2));
```

If we leave that weighting constant at 0.5, you still get "bog-gling" coming out on top. Change it to a 1, and we have a tie between "bog-gling" and "bogg-ling", since the point

that "bog-gling" gets for hyphenating between a double consonant is offset by the point that it loses out on for the hyphen not being at the exact middle of the word. And if we change the constant to 1.5, we get "bogg-ling" coming out on top.

If NFBTRANS's

rules somehow interacted with an optimality system with ranking-rules like "give K points for every letter this contraction saves," or possibly "give (or subtract?) L points if this word ends up with contractions next to each other" or "subtract a half-point if the word ends in a contraction," then I'd need to first puzzle out what these ranking-rules were, and then figure out what their values for K and L were.

Regex Replacement as a First Hack

I intuitively felt that

Braille

encoding somehow had a lot in common with hyphenation (except that it was about contracting letters instead of sticking hyphens in between them), suggesting that optimality was at least part of the system. On the other hand,

optimality systems often have some generative component to them (since you need to generate some candidates to apply the ranking rules), so I figured that whether the real answer would eventually be in a generativity system or an optimality system, I'd probably have to work up something generativity-based.

So I decided to take the rules from NFBTRANS's rules file, cook them up into a

regular expression, and use that to perform all the substitutions, in a one-pass regex like s/(\$re)/ &lookup(\$1)/eg, an

approach I borrowed from Chapter 14. I figured that this approach would by no means behave correctly, but that the cases where it didn't behave correctly would give me some kind of hint as to whether a generativity system or an optimality system was called for, and from there I could start worrying about the

ordering or weighting of rules.

Granted, next to either generativity

systems or

optimality systems, a big one-pass regular expression replacement seems pretty strange, but Perl makes regexes so easy that it seemed the path of least resistance for a first hack.

Contexts in Regular Expressions

Suppose the rule file consists of:

```
"the" contracts to "<the>" in any context
"er" contracts to "<er>" in any context
"th" contracts to "" in any context
"ea" contracts to "<ea>" only in the
middle of words
"ar" contracts to "<ar>" in any context
"ear" contracts to "e<ar>" in any context
```

Now, a regex for "the' in any context" would be a simple <the>. But how should you express "<ea> only in the middle of words"? You could do it with something like / (?<=\w) ea(?=\w) / to assert that there be a word character before and after the "ea", but Perl already provides a

metacharacter for matching a word boundary (\b), or the absence of one (\B). In other words, what's needed here is the absence of

word boundaries before and after the "ea", and you can do that with simply /\Bea\B/.

Translating the above mini-list of Braille contraction rules into a regex gives:

```
%contraction_for = (
  'the' => '<the>',
    # Remember: I'm using "<the>" here
    # in place of the Braille ASCII, or
Unicode,
    # code for the single Braille character
    # for that contracts "the"
    'er' => '<er>',
    'th' => '',
    'ea' => '<ea>',
    'ar' => '<ea>',
    'ar' => '<ea>',
    'ar' => 'e<ar>',
    ';
    s/(the|ear|ar|\Bea|er|th)/
$contraction for($1)/eg;
```

Now, notice that I moved the longer strings to the start of the regex. This is a crucial point in understanding Perl's implementation of

regular expressions: given alternates, Perl will match the first one it can match, regardless of whether there may be later alternates that are longer. So if th came before the in that regex, the th would always match, never letting "the" match. Moreover, if there were a Bea B and an ea in the regex, if the ea came first, it would block the Bea B from ever matching. So, in producing this regular expression, I had to make sure that the alternates started out with the longest strings, in the most specific contexts (like $\Bfoo\B$), and worked their way down to the shortest match strings, in the most general contexts (like just plain foo). This was a simple matter of a

Schwartzian transform, where the sorter function looked like:

By this time, I had 180 lines of code for reading the contraction rules file and transforming it into a regex. Only about 20 lines were necessary to perform the contractions, and they were basically a wrapper around this:

```
$word =~ s/($my_re)/$contraction_for{$1}/
oeg;
```

Embedding Code in Regular Expressions

I'd forgotten about one thing—there could be rules that say something like:

```
"foo" contracts to "<X>" at word-start
"foo" contracts to "<Y>" elsewhere
```

Now, you could model this pair of (fictitious!) rules with a regex that contains (... | b foo|foo|...), but then whether bfoo or foo matches, you end with simply \$1 holding foo. In other words, while you can use b or real

lookbehind/lookahead to select for certain contexts, you don't end up knowing which alternative actually matched.

However, a new feature in Perl 5.005 is exactly what I needed here: (? { CODE }) in a

regular expression will cause the Perl snippet *CODE* to be evaluated when the regular expression engine reaches that point. So, to differentiate between bfoo and foo matching, I add to the regex a bit of code to be executed at the end of each:

\$word =~ s/(| \bfoo (?{ \$x=1 }) | foo (?{ \$x=2 })) /\$contract[\$x]{\$1}/eg;

Here, if \bfoo is the bit of the regex that matches, the last thing that the match does is to execute x=1, so then the string that got matched (say, the "foo" in "foolish") gets replaced with the value of \$contract[1] { 'foo' }. If it's the second bit of the regex ("foo") that matches (as in the "foo" in "buffoon"), x=2 will get executed, and so "foo" will get replaced with the value of \$contract[2]{`foo'}. This array-of-hashes accessed like (which is Qcontract \$contract[context flag]{string to be replaced}) will have been filled by the same part of the program that read the rules in and created the regular expression.

Rules as Exceptions

Most linguistic models deal with exceptional cases (like, say, irregular verbs) by exempting certain forms from the rules. However, NFBTRANS's rules table knows only rules and

more rules, where more specific rules stop more general rules from applying. For example, given:

```
"the" contracts to "<the>" in any context
"er" contracts to "<er>" in any context
"th" contracts to "" in any context
"ea" contracts to "<ea>" in the middle of
words
"ar" contracts to "<ar>" in any context
"ear" contracts to "e<ar>" in any context
```

the rule that replaces "the" with the single character "<the>" blocks the more general rule of replacing "th" with "" from applying. Likewise, the rule that targets "ear" blocks the rule that targets "ea" from applying—which is why "heart" contracts to "he<ar>t", not "h<ea>rt".

Now, for some reason, it's incorrect to contract the "ear" in "tearoom" at all. This could be modelled by saying that "tearoom" is on a list of exception words that are exempt from contractions, but then you have two kinds of data in the system—the

rules table, and the list of exceptions. The way NFBTRANS models this is to simply add another rule:

```
"tearoom" contracts to "tearoom" in any context
```

A rule that replaces a word with itself seems entirely pointless, but the point of it is to block the more general rules.^[3]

NFBTRANS consists of four types of rules:

• A few dozen general rules that implement the regular contractions, like "ea"
- A few dozen more rules that implement more specific (yet still pretty general) contractions like "ear"
- About a hundred rules for whole-word abbreviations like "qk" for "quick"
- About a thousand rules that are there simply to block the more general rules

The above "tearoom" rule is an example that happens to block all contractions for that word, but the average rule is meant to stop only some of the contractions from applying. For example, applying the normal contraction rules to "pineapple" would contract it as "p<in><ea>pple". However, this spelling is considered incorrect, so there's a rule specifically to fix this with the correct spelling:

```
"pineapple" contracts to "p<in>eapple" in any context
```

And so on for "phoenix" (to block the "en" from contracting), "coworker" (to block the "ow" from contracting, lest they become cow orkers), and all the way up to the longest rule, which replaces "psychedelic" with "psy<ch>edelic" (blocking the "<ed>").

Incidentally, as with exceptions in

hyphenation, most of these exceptions to the normal

Braille contraction rules are motivated by wanting to not contract (or wanting to hyphenate between) letters that belong to different syllables or to different morphemes (roots, prefixes, or suffixes, like "tea-room," "co-worker," etc.). But the "ea" in "create" *does* contract to the single "<ea>" character, even though it's clearly across two syllables. Since the conventions

for Braille contractions evolved naturally—just as general English spelling did—it's full of patterns, and messy exceptions, and messy exceptions to the messy exceptions.

Testing It

When I wrote the code that generated the big

regular expression to match everything that could be contracted, I initially fed it just a few rules, then a few dozen, and when I did feed it the whole rule file, it ended up making a single regular expression 14KB long. The longest regex I'd ever heard of was the 6.5KB regex for matching RFC822-valid email addresses (in Appendix B of the first edition of Jeffrey

Friedl's

Mastering Regular Expressions), and this was more than twice as long—worrisomely long. I anticipated a screaming "OUT OF MEMORY" error, or that the regex would take two minutes of swapping to compile. But it compiled imperceptibly fast, and without errors.

I took the "Unix wordlist" (a motley wordlist of 25,000 English words, one per line), and encoded it with a copy of NFBTRANS, so that I could compare it to my algorithm's output. I wrote a small program to use my algorithm to encode the wordlist, print all cases where my algorithm's output disagreed with the output of the inscrutable NFBTRANS algorithm, and then report my algorithm's accuracy.

Guessing wildly, I expected no better than 50% similarity between the output of my algorithm, and the output of NFBTRANS.

It's Alive!

My first-hack one-pass regex algorithm agreed with NFBTRANS 99.59% of the time. I was boggled that it was over 50%, much less over 99%.

Most of the cases of disagreement were words like "R&D" or "O'Leary"—since this was a first hack, I had not bothered to properly deal with words with internal punctuation or complex capitalization, which require somewhat special treatment in

Braille

encoding. The remaining dozen words where my algorithm encoded differently than NFBTRANS were because I'd misimplemented the regex context for one of the more obscure rules. (And incidentally, I've only implemented my encoder for words in isolation—in

encoding of actual sentences, there's punctuation to deal with, and a few Braille contractions in words are sensitive to the context of the word, like whether there's punctuation after the word, which makes things a bit messier.)

This one-pass regex

approach was just a stab in the dark, and I expected to use the list of its failures to refine the algorithm—say, developing a generativity system that would make one pass to handle word-initial contractions, then another to handle word-final contractions, then a final pass for everything leftover in between. Or maybe the answer would lie in an optimality system to consider all possible ways to contract a word, and select the one that had the fewest characters, or maybe the fewest dots total, or maybe the most contractions closest to the beginning or end of the word. But it was nothing so complicated—this big regex, in one pass, did the whole job. Not only did this seem much too easy (although this is a not unfamiliar sensation in programming Perl), but it was quite unprecedented—I'd never heard of a phenomenon in natural language that could be treated with a regular expression.

I'd always considered regular expressions to be essentially abstract and mathematical. After all, the formalism of regular expressions was devised just this century, coming out of the work of mathematician Stephen

Kleene; and the workings of the regex engine (as in Mark Jason Dominus's article in *Computer Science and Perl Programming: Best of the Perl Journal* are explained in terms of automata. And whatever an automaton is (a big scary robot?), it sure doesn't sound like anything to do with something that people do unconsciously when they write Braille. (Incidentally, people have been writing English in Braille this way for decades longer than regexes have been around.)

But in spite of the formal origins of regular expressions, I mechanism think underlying that the basic regex replacement—scanning left to right, matching as much as you can, context permitting—is not so unnatural and abstract. That's what we do when we use systems of contraction or abbreviation, whether in Braille, Gregg shorthand, or our own idiosyncratic abbreviations: we start at the beginning of the word (as opposed to inking in the end, and working backwards from there), and we abbreviate as much, and as early, as possible.

Granted, Perl regexes have plenty of features that have no parallel in this process of Braille contractions (like backtracking, notably). But there's the commonality in that what's called

"greedy matching" for regexes is effectively the same as what we do when we abbreviate/contract, because of a mix of economy (wanting to save space) and laziness (not having to operate on the whole word at a time, or to consider lots of alternatives, but just contracting in one pass, as you write, starting at the beginning of the word). Perl programmers know from experience that regexes are immensely useful, and my experience with Braille encoding is just another case of that. But the fact that regexes have a basic commonality with a linguistic phenomenon like Braille encoding suggests that not only are regexes a useful tool, but that they're also linguistically natural and, hopefully, more learnable and valuable because of it.

The final Braille-encoding code is available, along with some other Braille links, at http://www.speech.cs.cmu.edu/~sburke/braille/.

^[3] Actually, the rule says "replace 'tearo' with 'tearo' in any context," not specifying the whole word—but I think the people who thought up that rule thought of "tearoom" as the only word it could apply to, and were not aware of the rarer "-tearo-" words: "Aotearoa" or "stearolactone."

Chapter 18. Hypernyms, Hyponyms, Pertainyms, and Other Word Relationships

Dan Brian

Author's Note: Since this article was published, development on the Lingua::Wordnet modules has been superseded by the Linguana project, an effort to create a dynamic, open source lexicon based on Wordnet. The author won the Damian Conway Award for Technical Excellence in 2001 for papers concerned with the project. More information is available from the Linguana web site at http://linguana.net.

My two-year-old son is just starting to grasp concepts like "A 'car' on the road is different from a 'car' on a railroad track." Computer programs can begin to exhibit this understanding as well; this article discusses the use of the Perl module Lingua::Wordnet to answer questions like, "What are some different kinds of cars?" and "What articles of clothing are made from leather?"

Wordnet is a lexical database of the English language organized according to current psycholinguistic theories of human lexical memory. Developed at Princeton's Cognitive Science department in the early 1990s, Wordnet was possibly the first undertaking to produce a machine-interpretable collection of English on a large scale. And like all really helpful and important software projects, it's open source and just waiting to be used and extended.^[4]

Concepts in Wordnet

The Wordnet package consists of several text database files, text indexes for those files, binaries for searching the files, and the source code for those binaries. A brief example can illustrate the functionality of the system:

```
% wn canary -n1 -hypen
Synonyms/
Hypernyms (Ordered by Frequency) of noun
canary
Sense 1
  fink, snitch, stoolpigeon, stoolie,
sneak, canary
  => informer, betrayer, rat, squealer
   => informant, source
    => communicator
        => person, individual, someone,
somebody, mortal, human, soul
          => life form, organism, being,
living thing
         => entity, something
         => causal agent, cause, causal
agency
         => entity, something
```

This example of the wn program searches for "sense #1" (-n1) of the noun "canary," and displays its hypernyms (which I'll talk about shortly). Entries in the Wordnet databases are called *synsets* (sets of synonyms); in the case of this entry, "fink," "snitch," "stoolpigeon," "stoolie," "sneak," and "canary" are all considered synonyms for this particular word sense, and thus are displayed as members of the same synset. A *synset* can then be understood to be all words sharing the same essential meaning. Consequently, the second sense (-n2) of "canary" in Wordnet is that of a "singer," the third refers to the color "canary" or "canary yellow," and the fourth is the bird. The collection of synsets for a given word is equivalent to the entries you might typically find in a dictionary entry, but includes a plethora of other data. This isn't your grandpa's dictionary; it's your grandson's.

A hypernym is the type to which something belongs: a boat is a type of transport, and a fish is a kind of animal. The hierarchy of hypernyms shown above proceeds from the most specific synset to the most general synset: entity, something. You can read this hierarchy as "a fink/canary is a kind of informer, which is a kind of informant, which is a communicator, which is a person, which is a life form; a person is also a causal agent, and both life form and causal agent are kinds of entities."

This hierarchy is generated using word relation pointers between synsets. First, let's look at the index entry for "canary":

```
canary n 4 3 @ ~ #m 4 0 07263970 07137082
03881697 01055943
```

I won't describe all the elements of this entry; the only things to note are that it indicates that there are four senses of the word, and the four big numbers are the four offsets of those senses (or synsets) in the data file. Here is the first entry for "canary" in the data file (offset 07263970):

```
07263970 18 n 06 fink 0 snitch 0
stoolpigeon 0 stoolie 0 sneak 0 canary 1
```

001 @ 07338772 n 0000 | someone acting as an informer or decoy for the police

Look familiar? Besides conjuring memories of the kid who told on me for urinating on the Holiday Inn sauna rocks when I was twelve, it lists the words that belong to the synset, has a pointer to a hypernym at synset 07338772 (preceded with the @), and contains a short *gloss* (definition) for the word. Consequently, the synset at 07338772 is the "informer" entry we saw listed above directly below "fink…canary". That entry will have a hypernym pointer to the synset below it, and so on. This basic pointer system forms a large network of word relations in Wordnet. The database contains pointer types for many linguistic relationships other than

hypernyms; they're depicted in Figure 18-1. A complete list of pointer and data types is included in the

Lingua::Wordnet distribution in *docs/terms.txt*, and is fully documented in the Wordnet documentation.



Figure 18-1. Wordnet relationships

In addition to pointers, the database contains glosses for each synset. The glosses consist of brief dictionary-like definitions, "sentence frames" for verbs to illustrate their basic usage, and other less significant information.

Lingua::Wordnet alters these databases to permit additional data types, including "attribute" and "function" pointers for nouns. We will discuss these later. First, let's discuss installation.

^[4] I have to apologize in advance for any heartache that "dynamicizing" the databases may have caused the Princeton researchers. While they informed me that such a change had been considered and deemed a possibility (given the funding), I can't help but feel like a hack invading their esteemed lexicographer territory. The "5 Papers" documentation reads, "WordNet's source files are written by lexicographers. They are the product of a detailed relational analysis of lexical semantics." That means no adding specialized definitions for "mongers," people.

Converting the Data

The Wordnet distribution stores all of the lexicon information in static text data files, with accompanying index text files to increase lookup speed. The Lingua::Wordnet installation rewrites these files in the

Berkeley DB database format, allowing these files to be dynamically edited.^[5]

With this change, users can edit existing

synsets, add new synsets, and even create new pointer types. The new files also increase the speed for data retrieval for the indexes, since words are mapped to synsets with a hash, eliminating the need for a manual binary search.

There are many programs on the Wordnet web site, but you'll need only the

Unix Database Package (wn16.unix.tar.gz).

Lingua::Wordnet does not use the Wordnet programs, so the Unix package should work for any operating system supporting Berkeley DB 1.x. If you install the Wordnet distribution, the Wordnet database files will typically be in */usr/local/wordnet-1.6/dict/*. If you unpack (but do not install) the distribution, these files will be in *wordnet-1.6/dict*. (You can install Lingua::Wordnet without formally

installing the Wordnet distribution.)

^[5] I get some complaints in this regard, because some people object to Berkeley DB. The fact is that it's the easiest to implement, it scales remarkably well in this and other, larger projects, and most of us already have it. Also, if you don't

like the Lingua::Wordnet data delimiters (| and ||), change them with the globals *\$DELIM* and *\$SUBDELIM* in *convertdb.pl* and *Wordnet.pm*.

Installing Lingua::Wordnet

To install Lingua::Wordnet, grab the distribution from CPAN, and unpack the files. Prior to installation, the script scripts/convertdb must be run to convert the databases to the new file format. This program will prompt for the location of the dictionary files, as well as the location for the new DB files. It then sorts through the Wordnet index and data files and writes out the new files, requiring approximately forty megabytes. Once the files are converted, you can delete the old files or the entire Wordnet distribution. The standard module installation can then be completed Makefile.PL, (perl make, make test, install). The module also contains a framework make for an HTML web interface for browsing the lexicon under mod perl, shown in Figure 18-2. If you manage to extend this to edit Wordnet information, please send me your changes!

Basic Usage

Throughout the

Lingua::Wordnet module, synsets are represented and manipulated as objects of type

Lingua::Wordnet::Synset, with methods that typically return arrays. The simplest use of Lingua::Wordnet is to look up English definitions:

```
use Lingua::Wordnet;
my $synset;
my $wn = new Lingua::Wordnet;
my @synsets = $wn->
lookup_synset("travel", "v");
for $synset(@synsets) {
    for ($synset->words) {
       print "$_ ";
    }
    print " - ", $synset->gloss, "\n";
}
```

Note that the lookup_synset method actually returns an array of Lingua::Wordnet::Synset objects, which can then be used to call other functions for that individual synset. The second parameter passed to lookup_synset is the part-of-speech (POS), which in this case is v for "verb." This code will look

up all senses of the verb "travel," and print the words of the synset, along with their gloss.^[6]

ile Edit View Go Comm	unicator					Hel
Back Forward Reload	Home	2 Search	Netscape	😅 Print	Security	N
🐠 Bookmarks 🌛 Location:	http://127.	0.0.1:80	80/wn#hypony	7n.s	V 🗗 Wha	at's Relate
🗶 WebMail 🦧 Contact 🦧 Pe	eople 🦧 Yello	w Pages	& Download	& Find S	ites 🖆 Chann	nels
	<u>.</u>					
opographic point, place, spot gloss	. (06381267%n)	refr rdille	resh are	lace for a p	ienie*	
opographic point, place, spot goss a point located with respect to surfi	(06381267%n) ace features of s	refr stitlb	resh ere ; "this is a nice p [(cilior=]=d1]440	place for a p	icnic*	
pographic point, place, spot gloss a point located with respect to surf antonyms: 0 hypernyms: 1	(06381267%n) ace features of s	refr refit b come region, correct correct	resh in ; "this is a nice p inclusing half and inclusing half and	place for a p	icnic*	
popgraphic point, place, spot goss a point located with respect to surf antonyms: 0 hypernyms: 1 point	(06381267%n) ace features of s <u>sem</u> l «	este la come region, estrate estrates riete neces	resh de ; "this is a nice p (callent [141] http (callent [141] http	alace for a p	icnic*	
popgraphic point, place, spot gloss a point located with respect to surfa antonyms: 0 hypernyms: 1 point hyponyms: 22	(06381267%n) ace features of s see 4	estille come region, estrats corras tete move extrats	esh dz ; "this is a nice p (causer (201 jang (causer (201 jang (causer (201 jang	vlace for a p	icaic*	
opographic point, place, spot gloss a point located with respect to surfa antonyms: 0 hypernyms: 1 point hyponyms: 22 grave, tomb	(06381267%n) acc features of s (2010)	refs zitt b come region erration erration erration erration erration erration erration erration	resh dia ; "this is a nice p [collect_left]ent [collect_left]ent [collect_left]ent	place for a p	icaic*	
opographic point, place, spot gloss a point located with respect to surfu- antonyms: 0 hypernyms: 1 point hyponyms: 22 grave, tomb unction	(06381267%n) acc features of s same features of s same features same features same features	etit ji etit ji ome region essass essass iste incos espass iste incos iste incos	esh dz ; "this is a nice p leditor: bill beig leditor: bill beig leditor: bill beig	ilace for a p	icnić*	
opographic point, place, spot gloss a point located with respect to surfa- sutonyms: 0 hypernyms: 1 point hyponyms: 22 grave, tomb unction birthplace, place of birth	(06381267%n) ace features of s gam) == gam) == gam) == gam) ==	etti ja etti ja etti ja erret erret erret itete jasse tete jasse tete jasse	resh dz ; "this is a nice p [colore_lest]en [colore_lest]en [colore_lest]en	ilace for a p	ionie*	

Figure 18-2. The HTML interface to Lingua::Wordnet

To format this information as a dictionary entry, only the glosses would be used:

```
my $i = 0;
my $synset;
my @
synsets = $wn->lookup_synset("travel",
"v");
print "travel, v.: ";
for $synset (@synsets) {
    $i++;
    print "$i. ", $synset->gloss;
}
print "\n";
```

This yields:

```
travel, v: 1. change location; move,
travel, or proceed; "How
fast does your new car go?" "We traveled
from Rome to Naples
by bus"; "The policemen went from door to
door
looking for
the suspect"; "The soldiers moved towards
the city in an
attempt to take it before night fell" 2.
travel as for
pleasure and sightseeing; go on travels 3.
make a trip for
pleasure 4. travel upon or across; "travel
the oceans" 5.
undergo transportation, as in a vehicle
```

But if you only use Wordnet for dictionary lookups, you're missing the reason it was created. Besides, the glosses aren't

all that great anyway. Let's talk about some real world applications instead.

Suppose that you were

writing a password checker to ensure that user passwords are hard to guess. You could amaze your users by warning them if their new password resembles the old:

```
old password: sneaker
new password: loafer
Why not pick something other than a shoe
this time?
```

To do this, we first generate two arrays of synsets, one containing all the matching synsets for the first (old password) term, and one containing the matching synsets for the new password:

```
my @oldsynsets =
$wn->lookup_synset("$oldpass", "n");
my @newsynsets =
$wn->lookup synset("$newpass", "n");
```

To determine the "type" of a given thing, we must know the hypernyms of that term. Hypernym relationships in Lingua::Wordnet are represented as object methods, and are

called with their name. To construct

a list of the hypernyms for the two passwords entered, we loop through all of the hypernyms for all the synsets, pushing their index keys (offsets) onto the @offsets array:

```
my ($hyper, @offsets);
foreach (@oldsynsets) {
    foreach $hyper ($_->hypernyms) {
        push (@offsets, $hyper->offset);
    }
}
```

Lastly, we can do the same for the array of new password synsets, and look for a match in their

hypernyms. The offset method returns the key for that synset:

```
my $hyper;
foreach (@newsynsets) {
    foreach $hyper ($_->
hypernyms) {
        my $offset = $hyper->offset;
        if (grep /$offset/, @offsets) {
            my $word = ($hyper->words)[0];
            $word = ($hyper->words)[0];
            $word =~ s/\%/w//;
                print "Why not choose
something other than a $word this time?\n";
               exit;
        }
    }
}
```

Of course, to use this with words other than nouns, it would need to search all four Wordnet parts of speech: noun (n), verb (v), adjective (a), and adverb (r). Also, while you might expect the entries "robin" and "canary" to come back with a clever objection, they won't, because of the detail Wordnet maintains for tiers of hypernyms. For example, a "robin" in Wordnet is a kind of "thrush," which is a kind of "oscine," a type of "passerine," which is a type of "bird." To generalize the hypernyms for our purpose, we could collect an array of all these hypernyms, stopping at a preset level. But There's More Than One Way To Do It.

^[6] It is important to note that while

sense numbers are printed along with the
Lingua::Wordnet::Synset::words function,
these numbers are not consistent with the ordering of
synsets in the Wordnet indexes. If anyone knows why, please
tell me.

Movin' It Up a Level

The

Lingua::Wordnet::Analysis

module allows

synset relationships to be analyzed from a slightly higher level, with methods to traverse trees, compute intersections of arrays of synsets, and scan arrays of synsets for matching entries. We'll see how these methods can enhance our already-clever, highly-useful password checker. But first, a tangent.

The concept of polysemy (or

familiarity) in Wordnet is determined by the number of times a word occurs in the Wordnet synsets. The lexicographers at Princeton determined that these occurrence frequencies corresponded closely enough to the familiarity of the human lexicon to equate them for their purposes. Retrieval of this information (the "polysemy count") in

Lingua::Wordnet is straightforward:

```
my $bird = $wn->familiarity("bird", "n");
my $passerine =
$wn->familiarity("passerine", "n");
print "The familiarity of 'bird' is
$bird.\n";
print "The familiarity of 'passerine' is
$passerine.\n";
```

The output is:

```
The familiarity of 'bird' is 5.
The familiarity of 'passerine' is 1.
```

The higher the count, the more likely that the word is the most familiar. The next hypernym above "birds" happens to be "vertebrate" with a polysemy count of 1. By traversing hypernyms and stopping at the hypernym with the highest familiarity, we can make a reliable guess that the hypernym represents the most appropriate category for the clever password checker. Given two

synsets and an array of their hypernyms up to the top level, we could compute the

intersection of the arrays and choose the resulting synsets with the highest familiarity. Then we could sleep soundly knowing that our system, while completely insecure, is clever. But There's A Better Way To Do It.

To perform this traversal, rather than use a foreach loop or recursive function to collect and compute the hypernym lists, we can use the traverse, union, and intersection methods of

Lingua::Wordnet::Analysis. One other method, distance, will help in determining the best response to the question.

```
use
Lingua::Wordnet;
use
Lingua::Wordnet::Analysis;
$wn = new Lingua::Wordnet;
$analysis = new Lingua::Wordnet::Analysis;
print "old password: ";
chomp($oldpass = <STDIN>);
print "new password: ";
chomp($newpass = <STDIN>);
```

```
@oldsynsets = $wn->lookup_
synset("$oldpass", "n");
@newsynsets =
$wn->lookup synset("$newpass", "n");
```

As before, we have an array of synset objects for both the old password and the new password. Next, we traverse the hypernyms of the @oldsynsets array, compute their union with those synsets already traversed, and store the results in a new array @oldhypersets. The same is done for the array @newsynsets:

```
foreach (@oldsynsets) {
          @oldhypersets =
$analysis->union(@oldhypersets,
$analysis->traverse("hypernyms", $_));
}
foreach (@newsynsets) {
@newhypersets =
$analysis->union(@newhypersets,
$analysis->traverse("hypernyms", $_));
}
```

By placing the traverse method within the arguments for union, the need to duplicate arrays is eliminated. Now we have two arrays (@oldhypersets and @newhypersets) that contain the union (that is, it contains no duplicate entries) of all possible hypernyms for any sense of the nouns typed in as the old and new passwords. Now to determine whether these two words' hypernym trees intercept (to see if they share any category in common), we can compute the intersection of the two arrays using references:

```
@intersection =
$analysis->intersection(\@oldhypersets,
\@newhypersets);
```

If we printed out the contents of @intersection now, we would see that it contains

synset objects common to both <code>@oldhypersets</code> and <code>@newhypersets</code>. However, since all Wordnet hypernym trees have a top level, it is likely that the two entries intersect no matter how different they seem. For example, "chicago" and "book" will have an intersection of

synsets "object" and "entity." To determine how applicable the results are, we can use the familiarity method discussed earlier, as well as the Lingua::Wordnet::Analysis::distance method. This takes an origin synset, a destination synset, and the pointer type to use as a "path" for the search. For now, we can use the first synset of @oldsynsets for the search, and compute the distance to each synset in our intersected array:

```
foreach $intersect (@intersection) {
    print $intersect->words;
        print " - distance: ",
$analysis->distance($oldsynsets[0],
        $intersect,"
hypernyms"), "\n";
}
```

For old password "sandal" and new password "loafer," this yields:

```
artifact%0artefact%0 - distance: 4
covering%0 - distance: 3
object%0physical_object%0 - distance: 5
shoe%0 - distance: 1
entity%0something%0 - distance: 6
footwear%0footgear%0 - distance: 2
```

(Spaces between words in Wordnet are replaced with underscores.) The synset with the shortest distance is "shoe," as we had hoped. We can finish the program by finding the shortest distance, checking it for a maximum value, and then outputting the message:

```
\$ shortest = 10;
for (@intersection) {
    foreach $oldsynset (@oldsynsets) {
                         $thisdistance =
$analysis->distance($oldsynset,
                                     $,
"hypernyms");
        next if ($thisdistance < 1);
        if ($thisdistance < $shortest) {
            $shortest = $thisdistance;
            \$word = (\$ ->words)[0];
            \$word = < s/\%/w//;
        }
    }
}
if (\$shortest < 5) {
    \$word = < s / / / g;
    print "Why not choose something other
than a $word this time?\n";
}
```

The clever password checker will now prompt correctly for the following examples:

old password: girl
new password: mother

```
Why not choose something other than a
woman this time?
old password: robin
new password: canary
Why not choose something other than an
oscine bird this time?
old password: germany
new password: france
Why not choose something other than a
European country this time?
```

```
If you're wondering why we have a foreach loop to go
through all the @oldsynset entries, it's because
$oldsynset[0] might not uncover any matches at all.
Recall that "canary the bird" was the fourth sense of "canary"
in the database, not the first.
```

Other Word Relationships

So far we have talked about only hypernyms, which are just one of the many linguistic relationships in Wordnet. Here's a list of what Lingua::Wordnet works with:

Meronyms

"An apple is a meronym of apple tree." A constituent part, substance, or a member of something. X is a meronym of Y if X is a part of Y. The subtypes for meronyms in Lingua::Wordnet are:

```
component-object: X is part of Y
member-collection: X is a member of
group Y
stuff-object: Y is made of X
portion-mass: X is a segment or portion
of Y
feature-activity: X is part of the
activity Y
place-area: X is a place in Y
phase-process: X is part of the process Y
```

(The last four are provided by Lingua::Wordnet, but not by Wordnet itself.)

Holonyms

"An apple tree is a holonym of apple." The opposite of meronym; the name of the whole of which the meronym names a part. Y is a holonym of X if X is a part of Y. The subdivision of types is the same as meronyms above.

Hypernyms

"Fruit is a hypernym of apple." The generic term used to designate a whole class of specific instances. Y is a hypernym of X if X is a (kind of) Y.

Hyponyms

"Apple is a hyponym of fruit." The opposite of hypernym; the specific term used to designate a member of a class. X is a hyponym of Y if X is a (kind of) Y.

Attributes

"Hardness is an attribute of the adjective soft." A noun for which adjectives express values. The noun "weight" is an attribute, for which the adjectives "light" and "heavy" express values.

Antonyms (direct)

"Heavy is an antonym of light." A pair of words with opposite meanings. In adjective clusters, direct antonyms appear only in head synsets.

Antonyms (indirect)

"Fatty is an antonym of light (via nonfat)." An antonym not of the word, but of the head synset.

Pertainyms

"Slow is a pertainym of slowly." A relational adjective. Adjectives that are pertainyms are usually defined by such phrases as "of or pertaining to" and do not have antonyms. A pertainym can point to a noun or another pertainym.

Noun attributes

"Yellow is a color attribute of canary." An adjective that defines a quality of a noun. (This is provided by Lingua::Wordnet, but not by Wordnet itself.)

Functions

"Sing is a function of canary." A verb that defines an action performed by a noun. (This is provided by Lingua::Wordnet, but not by Wordnet itself.)

In addition to analyzing hyponymy relationships (hypernymy/ hyponymy, which engineers might prefer to call "subsets" and "supersets," or IS-A relationships), *meronymy* (the HAS-A relationship) can be examined using the same methods. In many ways, meronymy is significantly more complicated than hyponymy. The Wordnet databases specify three

types of meronym relationships:

- Part meronym: a "tire" is a part of a "car"
- Member meronym: a "car" is a member of a "traffic jam"
- Substance (stuff) meronym: a "wheel" is made from "rubber"

Lingua::Wordnet creates the databases with the following four additional types, defined in the Wordnet documentation (see *5papers.pdf* at the Wordnet web site):

Activity meronym: "paying" is part of the activity "shopping"

- Phase meronym: "solstice" is a phase of "summer"
- Portion meronym: "slice" is a portion of a "pie"
- Area meronym: "Bronx" is a place in the area "New York"

The same objects are used to access these pointer types also. Suppose that you needed a different kind of password utility, but one no less clever. This utility would challenge potential password hackers by offering a kind of twenty questions game. Again, it's not good security, but it is fun.

Now, if we used an advanced parser (like

Lingua::LinkParser) we could allow a wide variety of questions to be asked. In this case, the game lends itself well to simply pattern-matching the questions, which we will limit to:

```
"Are you a(n) NOUN?"
"Are you ADJECTIVE?"
"Do you have NOUN?" or "Are you made from/
of NOUN?"
"Do/can you VERB?"
```

We again load

Lingua::Wordnet::Analysis and create the \$wn and \$analysis objects. First, we initialize and build arrays containing all the

hypernyms, attributes, meronyms, and functions that we'll need to answer these questions. Note that we need to add the direct attributes, meronyms, and functions of *\$password* to the arrays, since traverse will start at the level after \$password:

```
my $password = $wn->lookup
synset("oven", "n", 1);
mv (
Chypernyms, Cattributes, Cmeronyms,
@functions, $i);
ß
hypernyms
                          _
$analysis->traverse("hypernyms",
$password);
for (@hypernyms) {
       push (@attributes, $ ->attributes,
$password->attributes);
     push (@meronyms, $ ->all meronyms,
$password->all meronyms);
       push (@functions, $ ->functions,
$password->functions);
 }
```

Next, we will loop ten times, input the questions, and match a pattern against them. First, the "Are you a/an ..." question:

```
for $i (1 .. 10) {
    print "$i>";
    chomp ($_ = <STDIN>);
    if (/are\syou\san*\s(.+)\s*\?/i) {
        my $noun = $1;
            $noun =~ $/\s+$//; $noun =~ $/\s/
    \_/g;
        my @
synsets = $wn->lookup_synset($noun, "n");
        my $found = 0; my $synset;
        foreach $synset (@synsets) {
```

```
if
                        ($synset->offset
                                           eα
$password->offset) {
                 print "You guessed it!\n";
                 exit;
             }
                                          if
($analysis->match($synset,@hypernyms)) {
                 print "Yes.\n";
                 \$found = 1;
                 last;
             }
        }
        print "No.\n" unless ($found);
    }
}
```

In this block of code, the noun provided by the user is looked up and matched against the @hypernyms array. The \$analysis->match does this, telling us whether the word is a hypernym of "oven." The code for the other four questions is identical, with the exception of the actual pattern to match the question, the part-of-speech used to look up the synset, and the array that is used for the match. See Example 18-1 for the complete program. Here is how a sample run looks:

```
1> Are you an animal?
No.
2> Are you a plant?
No.
3> Are you an object?
Yes.
4> Are you food?
No.
5> Are you electrical?
No.
```

```
...
10> Are you made from wood?
No.
Sorry, you blew it.
```

As we can see, the program misled the user at question #5, since an oven is electrical; not all the information that a user will ask is in the Wordnet databases. For this reason,

Lingua::Wordnet allows the database to be extended very easily. Since we already will know what the password will be, we can add that information ourselves. The information we add will address questions that a user is likely to ask.

Before adding information to the databases, it is important to determine the

proper location for that information. To define an "oven" as "electric," you might be tempted to create a pointer directly from "oven" to the adjective "electric." However, inheritance of properties is one of the keys to Wordnet's usability. An attribute pointer should be created as high in the lexicon as possible; if we trace the

hypernyms upwards from "oven" we get "kitchen appliance," then "home/household appliance," then "appliance," then "consumer durables," and so on.

Looking at the glosses for these

synsets as well as the hyponyms under each shows that the "electric" pointer should be placed at the "appliances" synset, since all synsets under it are electrical devices. By placing the pointer at this synset, we eliminate the need to duplicate it in the synsets below.

To write to the database, we must first unlock it and have write permissions to it. The following code will add pointers to their appropriate locations. (If you're wondering which is the easiest way to look up the correct

sense numbers for synsets, I recommend writing a text-based browser and sending it to me. Or you can just use the LWBrowser.pm mod_perl Apache module.)

```
# The '2' and '1' in the lookup synset()
parameters specify the
# sense number of the word.
     $appliance
mv
                                         =
$wn->lookup synset("appliance", "n", 2);
mv $oven
                                          _
$wn->lookup synset("oven", "n", 1);
    $electrical
mv
                                         =
$wn->lookup synset("electrical", "a", 2);
my
   $prepare
                                          _
$wn->lookup synset("prepare", "v", 2);
     $kitchen ap
mv
                                         _
$wn->lookup synset("kitchen appliance",
"n", 1);
mv $bake
                                          =
$wn->lookup synset("bake", "v", 1);
# Allow writing to the databases
$wn->unlock;
# Create the pointers
$appliance->add attributes($electrical);
$kitchen ap->add functions($prepare);
$oven->add functions($bake);
# Write the entries
$appliance->write;
$oven->write;
$kitchen ap->write;
$wn->lock;
```

It is important to note that while a verb function like "prepare" does not immediately seem to connote "food preparation," this synset in fact refers to exactly that; the complete synset contains the words "cook, fix, ready, make, prepare." Remember that Wordnet already includes many concepts that you might be inclined to add yourself.

We could continue

adding pointers like part meronyms to "oven" for "door," "window," "dial," and "rack," since the "oven" synset doesn't already have them, and they are typical of ovens. For now, let's see how the game would turn out with the new lexicon entries:

```
1> Are you an animal?
 No.
2> Are you a plant?
 No.
3> Are you an object?
 Yes.
4> Are you food?
 No.
5> Are you electrical?
 Yes.
6> Are you an appliance?
 Yes.
7> Can you cook?
 Yes.
8> Do you bake?
 Yes.
9> Are you an oven?
 You guessed it!
```

Granted, we supplied all of that information to Wordnet beforehand. But as this simple application shows, the flexibility of Lingua::Wordnet allows it to grow in usefulness and power the more it is expanded. One more example of this program will illustrate the point. Let's finish our lexical feast by adding synset pointers for "oreo." This time we will add meronyms to describe the ingredients of my favorite cookie, and add a hypernym for "dessert." (A synset can have multiple

hypernyms. In fact, adding proper hypernyms to a synset can dramatically increase the inherited properties of that synset.)

```
$oreo
mν
$wn->lookup synset("oreo",
                              "n", 1);
mv
      $cream
                                    =
$wn->lookup synset("cream",
                               "n", 2);
mv $sugar
$wn->lookup synset("sugar",
                               "n", 1);
my $cake
                                     =
$wn->lookup synset("cake",
                              "n", 2);
my $chocolate
$wn->lookup synset("chocolate",
                               "a",
                                   1);
   $cocoa
my
                                    =
                              "n", 2);
$wn->lookup synset("cocoa",
my $dessert = $wn->
lookup synset("dessert", "n", 1);
mv
        $filling
                                   =
$wn->lookup synset("filling", "n", 3);
mv
      $vummv
$wn->lookup synset("yummy", "a", 1);
       $nourish
my
$wn->lookup synset("nourish", "v", 2);
$oreo->add attributes($yummy);
$oreo->add part meronyms($cream);
$oreo->add part meronyms($filling);
$oreo->add stuff meronyms($cocoa);
$orem->add hypernym($dessert);
```

```
$oreo->add_attributes($chocolate);
$orem->add_attributes($yummy);
# Not quite concise
$oreo->add_functions($nourish);
# A big stretch
```

Since the 10questions.pl program performs meronym lookups with the all meronyms

method, both part and stuff meronyms will be included, and generalize well enough to answer the "Do you have …" and "Are you made from …" questions.
Other Useful Functions

In addition to the hierarchical organization of synsets,

Lingua::Wordnet has other useful methods, including basic morphology, antonym lookups, and verb entailments:

```
# Prints "heavy"
$word = $wn->morph("heaviest", "a");
print "$word\n";
# Prints "light%1"
$heavy = $wn->lookup_synset($word, "a", 1);
print (($heavy->antonyms)[0]->words);
# Prints "buy%0purchase%0"
$shop = $wn->lookup_synset("shop", "v", 1);
print (($shop->entailment)[0]->words);
# A makeshift thesaurus ... er,
"similar-thingies-aurus"
@coordinates =
$analysis->coordinates($thesaurus);
```

What's Next?

The examples here demonstrate functionality, not philosophy. If lexical processing interests you, I highly recommend reading all the materials on the Wordnet web site, and especially the "5 Papers" document. The researchers at Princeton have given the Open Source community a comprehensive tool that can greatly benefit AI, search technology, and linguistic applications. It was my motivation for

Lingua::Wordnet that any weaknesses resulting from static data could be remedied to the user's specification. Lastly, if you significantly expand the Wordnet databases, please let me know. I plan on setting up a repository of expanded files in the near future.

Wordnet can be obtained from http://www.cogsci.princeton.edu/~wn/, and the Lingua::Wordnet module is on CPAN.

Example 18-1. 10_questions.pl

```
#!/usr/bin/perl -w
use
Lingua::Wordnet;
use Lingua::Wordnet::Analysis;
use strict;
my $wn = new Lingua::Wordnet;
my $analysis = new
Lingua::Wordnet::Analysis;
```

```
my
             $password
$wn->lookup synset("oven", "n", 1);
my (@
hypernyms, @attributes, @meronyms, @functions, $i);
@hypernyms
$analysis->traverse("hypernyms", $password);
for (@hypernyms) {
                                  push
(@attributes, $ ->attributes, $password->attributes);
                                  push
(@meronyms,$ ->all meronyms,$password->all meronyms);
                                  push
(@functions, $ ->functions, $password->functions);
}
for my $i (1 .. 10) {
    print "$i> ";
    chomp ($ = <STDIN>);
    # Are you a(n) NOUN?
    if (/are\syou\san*\s(.+)\s*\?/i) {
         my snoun = s1; snoun = s/s+s//;
snoun = < s/(s)/(q;
        my @
synsets = $wn->lookup synset($noun,"n");
        my $found = 0; my $synset;
        foreach $synset (@synsets) {
                    if ($synset->offset eq
$password->offset) {
                print " You guessed it!\n";
                exit;
            }
                                        if
($analysis->match($synset,@hypernyms)) {
                print " Yes.\n";
                 found = 1;
```

```
last;
            }
        }
        print " No.\n" unless ($found);
    # Are you ADJECTIVE?
    } elsif (/are\syou\s(.+)\s*\?/i) {
          my $adj = $1; $adj =~ s/\s+$//;
adj = < s/\s/\/g;
                        my @synsets
$wn->lookup synset($adj,"a");
        my $found = 0; my $synset;
        foreach $synset (@synsets) {
                                       if
($analysis->match($synset,@attributes)) {
                print " Yes.\n";
                found = 1;
                last:
            }
         print " No.\n" unless ($found);
    # Do you have (a(n)) NOUN? Are you
made of/from NOUN?
                              elsif
                                          ((/
do\syou\shave\s(?:an*\s)(.+)\s*\?/i) ||
(/are\syou\smade\s(?:of|from)\s*(.+)\s*\?/
i)) {
         my snoun = $1; snoun = < s/s+$//;
snoun = < s/\s/\/g;
                        my @synsets
$wn->lookup synset($noun,"n");
        my $found = 0; my $synset;
        foreach $synset (@synsets) {
                                       if
($analysis->match($synset,@meronyms)) {
                print " Yes.\n";
```

```
found = 1;
                 last;
             }
        }
        print " No.\n" unless ($found);
      } elsif (/(?:do|can)\syou\s(.+)\s*\?/
i) {
         my \$verb = \$1; \$verb = < s/\s+\$//;
verb = \langle s/\langle s/\rangle /q;
                          my
                               @synsets
                                            =
$wn->lookup synset($verb,"v");
        my $found = 0; my $synset;
        foreach $synset (@synsets) {
                                           if
($analysis->match($synset,@functions)) {
                 print " Yes.\n";
                 found = 1;
                 last:
             }
         }
        print " No.\n" unless ($found);
    }
}
print " Sorry, you blew it.\n";
```

Chapter 19. Parsing Natural Language

Dan Brian

I See a Pattern Developing

Regular expressions are one of the triumphs of computer science. While often intimidating to beginning programmers, the ability to capture complex patterns of text in succinct representations gives developers one of the most powerful tools at their disposal. Perl's

pattern matching abilities are among the most advanced of any language, and certainly rank among those features that have served to make it one of the most popular languages ever created.

However, regexes can't do everything. When the patterns in your data are complex, even Perl's regular expressions fall short.

Natural languages, like English, aren't amenable to easy pattern matching: if you want to find sentences that express a particular sentiment, you need to first understand the grammar of the sentence, and regular expressions aren't sufficient unless you throw a little intelligence into the mix. In this article, I'll show how to do that.

We'll make it possible to write code like this:

```
# create an array of everything cool
while ($sentence =~
```

```
/\G($something_that_rocks)/g) {
    push (@stuff_that_rocks, $1);
}
```

Our notion of "what's cool" can depend not just on simple character patterns, but upon the words in a sentence, and in particular their role in the sentence and relationships to one another. In brief, this article explores the application of regular expressions to grammar. Note that I am *not* suggesting another syntax for regular expressions. From a Perl hacker's perspective, what I demonstrate here is an interesting application of Perl's overloading ability, and its usefulness when applied to a domain that's tough to parse: natural language.

Before explaining the program, we'll explore the intelligence that we'll use to parse natural language: the Link Grammar.

Link Grammar

Many approaches to NLP (

Natural Language

Processing) have been pursued in the past few decades, but few are as popular as the

Link Grammar parser, by Drs. Daniel Sleator, Davy Temperley, and John Lafferty. Rather than examine the basic context of a word within a sentence, the Link Parser is based on a model that words within a text form "links" with one another.

These links are used not only to identify parts of speech (nouns, verbs, and so on), but also to describe in detail the *function* of that word within the sentence. It's one thing to know that a phrase consists of two adjectives and two nouns—but what you really want to know is which adjective modifies which noun. The Link Grammar does that for you.

The Link Grammar is based on a characteristic that its creators call *planarity*. Planarity describes a phenomenon present in most natural languages, which is that if you draw arcs between related words in a sentence (for instance, between an adjective and the noun it modifies), your sentence is ungrammatical if arcs cross one another, and grammatical if they don't. This is an oversimplification, but it'll serve for our purposes.

In Link Grammar vernacular, a *linkage* is a single successful parse of a sentence: a set of links in which none of the connecting arcs cross. A sample parse of the sentence A

camel is a horse designed by a committee is depicted in Figure 19-1.



Figure 19-1. A sample parse, with links

In Link Parser output, the primary parts of speech are labeled with .n and .v to indicate that these words are nouns and verbs, respectively. The labels of the links between words indicate the type of link. For example, the J connector in Figure 19-1 indicates a connection between a preposition and its object; in this case, the verb designed is connected to by a committee, identifying a prepositional phrase. The following list summarizes the links used above and elsewhere in this article:

A

Connects pre-noun ("attributive") adjectives to subsequent nouns: "The BIG BLACK UGLY DOG chased me."

AN

Connects noun-modifiers to subsequent nouns: "The TAX PROPOSAL was rejected."

С

Connects conjunctions and some verbs to subjects of clauses: "He SAID HE was sorry."

D

Connects determiners to nouns: "THE DOG chased A CAT and SOME BIRDS."

G

Connects proper noun words together in series: "GEORGE HERBERT WALKER BUSH is here."

\boldsymbol{J}

Connects prepositions with their objects: "The man WITH the HAT is here."

М

Connects nouns to various kinds of post-noun modifiers (prepositional phrases, participle modifiers, and so on): "The MAN WITH the hat."

0

Connects transitive verbs to their direct or indirect objects: "I GAVE HIM the BOOK."

S

Connects subject nouns to finite verbs: "The DOG CHASED the cat."

W

Connects subjects of main clauses to the beginning of the sentence, or the "wall."

X

Connects punctuation symbols to words or each other.

The Link

Parser 4.0 provides 107 primary types of links (indicated by the uppercase letters), with many additional subtypes further detailing the relationship of words (indicated by the lowercase characters). While the accuracy of the parser is remarkable, it is tailored to newspaper-style grammar, and will fail with more conversational statements.

The inner workings of the parser are fairly complex, but they use principles that might be familiar. A

link grammar considers a sentence to be proper if it satisfies three conditions:

- 1. Planarity: The link arcs above words do not cross.
- 2. Connectivity: The links connect all of the words together.
- 3. Satisfaction: Individual linking requirements of each word are satisfied.

The parser uses a dictionary that contains the linking requirements of each word. For example, the words the, chased, dog, and cat are shown in Figure 19-2 with their linking requirements. The D within the box above the indicates that another word must connect with a D to its right in order for the link requirements to be satisfied for that word.



Figure 19-2. Some linking requirements

For these words to form a sentence, the

parser must find them in an order that satisfies the above three requirements. When a word has more than one row of connectors, only one side (left or right) of each row may be connected (e.g., cat has a row D and a row O/S, so D must be connected along with either O or S). When only one row exists on a single level (e.g., cat has D), one connector must be linked. The meaning of each

link used here is indicated above. Thus, the arrangement in Figure 19-3 is correct: The cat chased the dog.



Figure 19-3. Linking requirements and inferred links

The unused connectors are grayed out in Figure 19-3. Since our second the connects to dog as a determiner, chased actually spans the length, connecting to dog. You can mentally shuffle these words to see that cat and dog could be swapped, and likely would be if our program had any semantic knowledge. Moving other words around, however, will break the

link criterion and deem the parse ungrammatical.

All of these requirements are stored in the Link Parser's dictionary files. The files use a "link dictionary language" to list the requirements for each word, and are themselves an interesting study in

pattern representation. A highly optimized custom algorithm processes the data in these files, analyzing the possible links. This algorithm is yet another fascinating study in and of itself. (For those interested, the algorithm is similar to the dynamic algorithm for triangulation of a convex polygon, and has a running time of $O(N^3)$. The general theory, optimizations, and formalisms are all detailed in the researchers' papers. Frankly, it's remarkable that it runs as fast as it does given the computation required.)

Because the researchers at CMU had the generosity and intelligence to make their project research open to developers like us, we can examine the ingenuity of their methods. We can use and modify their well-conceived API. We can extend and combine the functionality of their system with that of other language

processing technologies. And, of course, Perl makes it all possible, practical, and inevitable.

Lingua::LinkParser

The

Link Grammar parser itself is a complex piece of software implementing a complex theory of language. The Perl module Lingua::LinkParser (available on CPAN) directly embeds the parser API, providing an object-oriented interface that you can use from your Perl programs. Objects may be created to represent sentences, linkages, links, individual words, and the parser itself. As an example, consider the following code:

```
use Lingua::LinkParser;
use strict;
         $parser =
my
                              new
Lingua::LinkParser;
                                # Create
the parser
my $text = "Moses supposes his toses are
roses.";
            $sentence
my
$parser->create sentence($text); # Parse
the sentence
         $linkage
my
                               # Use the
$sentence->linkage(1);
first linkage
print
$parser->get diagram($linkage);
# Print it out
```

This code outputs:

+----Xp-----+

```
+----Ce----+ |
+---Wd--+--Ss---+
+--Dmc--+--Spx--+--Opt-+ |
| | | | |
LEFT-WALL Moses supposes.v his toses[!].n
are.v roses.n .
```

Without delving into all the details, this diagram reveals some interesting things about the parser. First, supposes and are have v labels, indicating that they're verbs. The word roses is labeled n for noun, as is toses. The [!] tag next to toses indicates that the parser isn't familiar with this word, which usually means that it isn't a word at all. So even with a word it's never seen before, the

Link Grammar can identify the part of speech.

The diagrams help us understand the link grammar, but to use the information within a program requires access to the links themselves. Continuing with the program above, we will extract from the *\$linkage* object an array of *\$word* objects. These will spring into existence, along with a *links* method to return an array of *\$link* objects. Well, just watch:

```
my @words = $linkage->words;
foreach my $word (@words) {
    print "\"", $word->text, "\"\n";
    foreach my $link ($word->links) {
        print " link type '",
    $link->linklabel, "' to word '",
    $link->linkword, "'\n";
    }
}
```

An excerpt from the output:

```
"Moses"
link type 'Wd' to word '0:LEFT-WALL'
link type 'Ss' to word '2:supposes.v'
"supposes.v"
link type 'Ss' to word '1:Moses'
link type 'Ce' to word '4:toses[!]'
"his"
link type 'Dmc' to word '4:toses[!]'
"toses[!]"
link type 'Ce' to word '2:supposes.v'
link type 'Dmc' to word '3:his'
link type 'Spx' to word '5:are.v'
```

Knowing the part of speech and linkages of each word allows us to use grammatical constructs in a program. Let's write one.

Irregular Regular Expressions, Overloaded

Returning to our original problem, how can we expand our pattern matches to handle grammatical constructs rather than simple combinations of metacharacters? We have two tools: the Link

Parser just described, and Perl's *overloading*, which allows us to redefine how Perl's operators operate. In particular, we're going to redefine how Perl processes strings.

Normally, operator

overloading is used to extend the definition of common operators, like +, so that you can say things like \$madness = \$vanhalen_mp3 + \$vivaldi_mp3 and overlay two MP3s as a result.

For our purposes, we overload

double-quote processing in the

Lingua::LinkParser::Linkage package so that when you print a linkage object, it displays a linkage diagram. Furthermore, to pattern match the data, we need a format that is more easily parsed, but still just a single string. Something like the following would be nice, listing the words with their links on each side in an ordered, linear format:

```
(Wd:0:LEFT-WALL "Moses"
Ss:2:supposes.v) \
(Ss:1:Moses "supposes.v"
Ce:4:toses[!]) \
( "his"
Dmc:4:toses[!]) \
```

```
(Ce:2:supposes Dmc:3:his "toses[!]"
Spx:5:are.v) \
```

We can get this type of output with print \$linkage by modifying the file *Linkage/Linkage.pm* and changing the overload behavior. Now, printing the object \$linkage from the previous examples will output the text shown above, in one long string.

Finally, we can pattern match that text to find what we're looking for. In this case, we're going to look for the Ss link from Moses, indicating a connector to the verb for our subject:

```
$Moses = '"Moses" ';
$does_something = 'Ss:\d+:(\w+)\.v';
$action_by_moses = "$Moses$does_something";
if ($linkage =~ /$action_by_moses/o) {
    print "What does Moses do? He $1.\n";
}
```

This prints What does Moses do? He supposes. We could take the idea further by overloading the right side of our regular expressions and getting them to return word objects, but we won't.

Peeking under the hood, here's how the overloading is implemented.

```
use overload q("") => "new_as_string";
sub new_as_string {
    my $linkage = shift;
    my $return = '';
    my $i = 0;
    foreach my $word ($linkage->words) {
```

```
my ($before,$after) = '';
        foreach my $
link ($word->links) {
                         my $position =
$link->linkposition;
            my $text
                      = $link->linkword;
                      my $type
                                         =
$link->linklabel;
            if ($position < $i) {
                               $before .=
"$type:$position:$text ";
            } elsif ($position > $i) {
                                $after .=
"$type:$position:$text ";
            ļ
        }
        $return .= "(" . $before . " \.
$word->text . "\" "
                  $after . ")" ;
        $i++;
    }
   "(" . $return .")";
}
```

What Sucks? What Rocks?

The "Operating System Sucks-Rules-O-Meter" by Don Marti (http://srom.zgp.org/) inspired Jon Orwant's "What Languages Suck" program, later adopted by Steve Lidie (http://www.lehigh.edu/~sol0/rules.html). It blesses all of our lives by counting the web sites that state visual basic sucks, perl rules, and so on. The numbers are then plotted on a graph, giving us a crude and comical sampling of the Net's public opinions about languages.

What if someone wanted to perform a search that would produce lists of anything that people think sucks or stinks, and rules or rocks? A quick web search for rocks reveals plenty of geology links, and news headlines like Senate confrontation rocks Capitol Hill. We just want those phrases that state that something rocks, so we need to analyze the grammar of the search results.

First, we need to determine the syntax for the data we want to collect. We use the first script listed in this article to experiment, or we could think through the grammar a bit: the rock we are looking for is not only a verb, but a verb without an object. This would serve to differentiate our meaning from the two others mentioned above.

```
+----Xp-----+ |
+--Wd--+--Ss-+---Op---+ |
| | | | |
LEFT-WALL he studies.v rocks.n .
```

Note that this diagram displays only one linkage, but there might be many. In the above output from our script, He

studies rocks has been parsed and labeled. The subject of the sentence (he) is shown with an Ss label, connecting a singular noun to the singular verb form (studies). This will be the connector we are looking for in our rocks phrase, but here it occurs with the wrong verb. It has identified rocks as a noun here, and linked rocks to studies with an Op connector. The Link

Parser documentation tells us that O connects transitive verbs to direct or indirect objects, and so the p subscript reflects the plurality of the object, rocks.

```
+-----Wd-----+
+---Os----+ |
| +---Ds--+---Ss---+
+--Ds-+ |
| | | | |
LEFT-WALL the earthquake.n rocks.v the
city.n .
```

This example recognizes the verb usage of rocks as an action being performed by earthquake. Do earthquakes rock? Perhaps, but not in the sense we are looking for, since rocks has a connector to the city as a singular object (indicated by Os). Objects suck, at least for our purposes. Let's try another.

+-----Xp-----+ +-----Wd-----+ | | +-----D*u----+

Again, rocks here is correctly recognized as a verb, and again, it is connected via Ss to a subject. But this time rocks is not a transitive verb, since it has no objects. The grammar of this sentence would satisfy our requirements for rocks. So now that we have the correct usage, how do we extract the subject? We don't want to use just language to compile our statistics—we want Perl programming language. To find a solution, take note of the AN connectors that span these words. The

Link Grammar reference identifies this link type as connecting modifiers to nouns. In the case above, both Perl and programming are modifiers for language. We can plan at the outset to always look for modifier links to our subject, and include them in the data we extract from the sentence. And there's more that we'll need, as you'll see.

What Sucks, Regex Style

Once we have determined the general grammatical elements for which to search, we can write a program that finds those elements in a given text. Since we overloaded this object's string handling to return the linkage information in a linear format, we can now formulate a

regular expression that represents the grammar we want to match.

```
\$what rocks = 'S[s|p]'.
                                          #
Singular and plural subject
                '(?:[\w*]{1,3})*' . # Any
optional subscripts
                  ':(\d+):'.
                                          #
Number of the word
                ( w+(?: . w) *)'; # And
save the word itself
other stuff = '[^\)]+';
                                          #
Junk, within the parentheses
                = '"(rocks*)\.v"';
                                          #
Śrocks
Singular and plural verbs
$
              = "$what rocks $other stuff
pattern
$rocks";
if ( $linkage =~ /$pattern/mx ) {
  print "$2 rocks.\n";
}
```

Our \$what_rocks portion of this pattern looks for an S link from rocks, and stores the word itself in \$2, with optional tags like .n after the word. (We will use the stored word number in a moment.) This regular expression works, but it works for *every* verb sense of rocks that has a subject, including the earthquake rocks the city. We need to limit our pattern to match only those usages of rocks that have no objects at all. Here, we add a pattern after \$rocks to be sure that no 0 connectors exist for rocks:

```
# match anything BUT an 'O' link, to the
end parenthesis
$no_objects =
'[^(?:O.{1,3}:\d+:\w+(?:\.\w)*)]*\)';
$pattern = "$what_rocks $other_stuff
$rocks $no objects";
```

With these changes, the pattern only matches the verb rocks when it has no objects. But one problem remains: when we use our regex with proper nouns like Pat Metheny rocks or noun modifiers like the Perl programming language rocks, we get only one word as the thing that rocks. Our pattern is getting a bit messy, so rather than add to it, we'll add a statement within the if block to scoop up the rest of the names. Proper nouns are strung together with G connectors, and noun modifiers with AN.

```
if ( $linkage =~ /$pattern/mx ) {
    $wordobj = $linkage->word($1); # the
stored word number
    $wordtxt = $2;
    $verb = $3;
    @wordlist = ();
    foreach $link ($wordobj->links) { #
process array of links
         if ($link->linklabel =~ /^G|AN/) {
$wordlist[$link->linkposition] =
```

```
$link->linkword;
        }
        print join (" ", @wordlist,
$wordtxt), " $verb\n";
   }
```

Note how although we are looking for matches in \$linkage, we are

using a method, \$linkage->word, in the next line. (Seeing objects used in both scalar and dereferenced context may look confusing at first.) Also, we store the words in the @wordlist array to maintain the order of these words. When run with sentences provided by the user, this block of code prints the following:

```
Enter a sentence> He studies rocks.
Enter a sentence> The earthquake rocks the
city.
Enter a sentence> The Perl programming
language rocks.
   -> Perl programming.n language.n rocks
Enter a sentence> Linux rocks!
   -> Linux rocks
Enter a sentence> He was telling me why he
thinks that San Francisco rocks.
   -> San Francisco rocks
```

The final listing for this program is at the end of this article, and includes additional modifications to permit possessive pronoun inclusions, grouping of possessive proper nouns, conjunctions, past tense, and attributive adjectives. A demonstration is shown below.

```
Enter a sentence> Roland Orzabal's music
rocks.
   -> Roland Orzabal 's.p music.n rocks
```

(Thanks to my wife for pointing out all of the grammar that would not work with my first attempts.)

Although the parser has no clue what the shogimogu and pyonuki mean (nothing at all, as it happens), it is still able to identify these as attributive adjectives. Anyone who has ever used another grammar-based parser will appreciate this feat.

We could compile literally thousands of patterns to match various grammatical phenomena, store them in constants within a module, and end up with a direct regex interface for analyzing grammar.

The What-Sucks-Ometer

Using this framework, the "What languages suck?" application could be extended to retrieve web links to the pages resulting from a search engine query for rocks, rules, sucks, and so on. The text of each page could then be split into sentences, parsed with the code shown here to find specific word usages, and graphing the results. I won't outline how such a utility would be developed; the documentation for the LWP and GD modules tells you all you need to know. If anybody does write it, I suggest having multiple exclamation points count progressively against the rocks rating of the subject, rather than for it. We need to discourage that. Thank you.

Certainly the possible applications of this type of "regex grammar" extend far beyond the toy application I've shown here. Smart search engines, document categorizers, and automated response systems all can make use of similar frontends to natural language.

There's Lots More Here

The

Link Grammar can hardly be fully described in this article, and I encourage anyone interested to delve further into the research in this field. There is lots of room here for continued innovation, and the parser itself has much more to offer than what's been described here.

References

Link Grammar web site: http://link.cs.cmu.edu/link

Daniel Sleator and Davy Temperley, *Parsing English with a Link Grammar*, Third International Workshop on Parsing Technologies, August 1993.

Daniel Sleator and Davy Temperley, *Parsing English with a Link Grammar*, Carnegie Mellon University Computer Science technical report CMU-CS-91-196, October 1991.

Dennis Grinberg, John

Lafferty, and Daniel Sleator, *A robust parsing algorithm for link grammars*, Carnegie Mellon University Computer Science technical report CMU-CS-95-125.

A Full Lingua::LinkParser Example

The following code uses Lingua::LinkParser to determine what really sucks:

```
#!/usr/bin/perl -w
# For this to work, the overload parameter
in ::Linkage and
    ::Sublinkage must point
                                      to
"new as string".
use Lingua::LinkParser;
use strict;
my $parser = new Lingua::LinkParser;
$parser->opts('disjunct cost' => 2);
$parser->opts('linkage limit' => 101);
while (1) {
  print "Enter a sentence> ";
  my $input = <STDIN>;
             mv
                     $sentence
                                     =
$parser->create sentence($input);
  my $linkage = $sentence->linkage(1);
  # Computing the union and then using the
last sublinkage permits conjunctions.
  $linkage->compute union;
            mv
                   $sublinkage
$linkage->sublinkage($linkage->num sublinkages);
  my \$ what rocks = 'S[s|p]'.
# Match the link label
                     '(?:[\w^*]{1,2})*'.
# Match any optional subscripts
                     ':(\d+):'.
```

```
# Match number of the word
                    '(\w+(?:\\w)*)';
# Match and save the word itself
 my $other stuff = '[^{)}]+';
# Match other stuff within parentheses
 my $rocks
              = '\"(rock[s|ed]*).v\"';
# match and store verb
 my $no objects = '[^{(2:0.{1,2}:')}]
# don't match objects
                   '\d+:\w+(?:\.\w)*)]*\)';
    my $pattern
                          = "$what rocks
$other stuff $rocks $no objects";
  if ( $sublinkage =~ /$pattern/mx ) {
   my $wordobj = $sublinkage->word($1);
   mv \$wordtxt = \$2;
   my $verb = $3;
   my @wordlist = ();
      # We could put all of the below
functionality in the regex above.
    foreach my $link ($wordobj->links) {
         # Proper nouns, noun modifiers,
pre-noun adjectives
      if ($link->linklabel =~ /^G|AN|A/) {
          $wordlist[$link->linkposition] =
$link->linkword;
        # Possessive pronouns, via a noun
determiner
      if ($link->linklabel =~ /^D[s|m]/) {
                         my $wword
$sublinkage->word($link->linkposition);
        foreach my $llink ($wword->links) {
          if ($llink->linklabel =~ /^YS/) {
$wordlist[$llink->linkposition]
```

```
$llink->linkword;
```

```
$wordlist[$link->linkposition]
                                        =
$link->linkword;
                            my
                                 $wwword =
$sublinkage->word($llink->linkposition);
                       foreach my $111ink
($wwword->links)
                  {
                  if ($lllink->linklabel =~
/^G|AN/)  {
$wordlist[$lllink->linkposition]
                                       =
$lllink->linkword;
              }
            }
          }
        }
      }
    }
               -> ", join (" ", @wordlist,
    print "
$wordtxt);
  }
}
```

Chapter 20. Word Morphology

Khurshid Ahmad

Duncan C. White

Natural Language Processing (NLP) is a branch of Artificial Intelligence involving computer programs that understand or generate documents written in "natural language"—that is, any human language, like English, Hebrew, Swahili, or Xhosa. Creating programs that exhibit full understanding of natural language has long been a goal of AI. Some typical NLP applications might be:

- Word assistance to users. For example, a human might ask: "What is the adverb form of 'accident'?," and the computer might reply: "'accidentally' is probably the word you want, although 3% of users spell it 'accidently'."
- A smarter web search engine that lets you search for a keyword such as compute to retrieve all documents that contain that keyword, or conceptually related keywords like computationally.
- A smart document categorizer that reads a series of documents and sorts them into different categories based upon disproportionate use of particular keywords ("This document appears to be about nuclear physics, because it mentions keywords like atom and nucleus far more than an average document would").

• A smart document summarizer that summarizes one or more documents into a more compact form and presents a digest (see "Summarizing Web Pages with HTML::Summary" in *Web, Graphics, and Perl/Tk: Best* of the Perl Journal).

All of these programs require some understanding of natural language. For perfect summarization or perfect translation, we'd need *total* understanding, and that's far too difficult because of the intricacies of human communication.

Fortunately, we don't need complete language understanding to obtain some utility. Even a little knowledge of language is useful for some applications, and in this spirit, let's start at the lower levels and pick one small part of natural language that we can process relatively easily: the structure of single words.

Morphology: Word Form and Structure

Natural languages like English often express shades of meaning by changing the form of a single word, such as by adding a prefix or a suffix to an existing word. Adding s to computer changes the meaning from "a single computer" to "multiple computers": a noun-to-noun change. We can also move between

nouns, verbs, and adjectives with similarly simple changes (move becomes movement). The study of the form of words is called *morphology*.

This article shows how to implement a program that uses morphological rules to generate many related words from one given word. This will be useful in search tools, because when the user searches for computer, the search engine can also find documents with related words like compute, computing, and computationally.

So let's look at four common morphological processes and how we might model them in Perl. The first is called *inflection*, in which a word is transformed to indicate the tense (for example, look becomes looks, looking, and looked) or the number (for example, husky becomes huskies).

As soon as we start thinking about such transformations from a Perl perspective, we can't help but think of regular expressions. If you have the verb look and you want to generate the present participle (looking), this regular expression springs to mind:

 $s/\$/ing/ # add "ing" to \$_$

In this spirit, Table 20-1 shows some typical inflections that pluralize nouns, and Table 20-2 shows some tense inflections on verbs.

Rule	Singular	Plural	Perl implementation
Regular	dog, cat, computer	dogs, cats, computers	s/\$/s/
Ending in s, x, z, sh, ch	gas, tax, wish	gases, taxes, wishes	s/\$/es/

Table 20-1. Some rules for pluralizing English nouns

Rule	Singular	Plural	Perl implementation
Ending in f	calf, elf, dwarf	calves, elves, dwarves	s/f\$/ves/
Ending in o	bamboo, folio domino, torpedo	bamboos, folios dominoes, torpedos	s/o\$/os/ s/o\$/oes/
Ending in y	sky, husky decoy, donkey	skies, huskies decoys, donkeys	s/y\$/ies/ s/y\$/ys/
Ending in sis	crisis, analysis	crises, analyses	s/sis\$/ses/
Ending in us	radius, genius omnibus, genius	radii, genii omnibuses, geniuses	s/us\$/i/ s/us\$/uses
Irregular	man	men	s/an\$/en/

Table 20-2. Some rules for inflecting English verbs
	Verb	Present tense	Present participle	Past tense	Perl implem	entations	
Regular verbs	look	looks	looking	looked	s/\$/ s/	s/\$/ ing/	s/\$/ed/
	walk	walks	walking	walked	s/\$/ s/	s/\$/ ing/	s/\$/ed/
	decide	decides	deciding	decided	s/\$/ s/	s/e\$/ ing/	s/e\$/ed/
	watch	watches	watching	walked	s/\$/ es/	s/\$/ ing/	s/\$/ed/
Irregular verbs	see	sees	seeing	saw	s/\$/ s/	s/\$/ ing/	s/ee\$/ aw/
	flee	flees	fleeing	fled	s/\$/ s/	s/\$/ ing/	s/ee\$/ ed/
	come	comes	coming	came	s/\$/ s/	s/\$/ ing/	s/ome\$/ ame/
	go	goes	going	went	s/\$/ es/	s/\$/ ing/	s/go\$/ went/
	take	takes	taking	took	s/\$/ s/	s/e\$/ ing/	s/take\$/ took/

Unfortunately, English has many irregularities, ranging from whether or not to suppress a trailing e, up to the impressive past tense of go: went, with no letters in common!

Sometimes two different rules can apply to a

word, but only one of the inflections is actually correct. Is the plural form of omnibus really omnibi according to the s/us\$/i/ rule? Or is it actually omnibuses because of s/us\$/uses/? In some cases (geniuses and genii, syllabuses and syllabi) both inflections are permitted. How can a program know what to do?

We'll downgrade our rules to "rules of thumb"—or, as AI researchers call them, *heuristics*. We'll accept that they will sometimes be wrong, and later we'll develop a way to filter out incorrect applications.

The second morphological process is called *derivation*, and involves taking a base

word (the *stem*) and transforming it from one part of speech into another (for instance, from the verb ignite to the noun ignition). Table 20-3 shows some typical derivations.

Table 20-3	Examples	of derivational	mornhology
1 auto 20-3.	L'Amples	of derivational	morphology

Noun	Derivation	Class
nation	national	adjective
nation	nationalize	verb
Verb	Derivation	Class

Noun	Derivation	Class
slow	slowly	adverb
observe	observation	noun
observe	observable	adjective
nationalize	nationalization	noun
develop	development	noun
Adjective	Derivation	Class
national	nationally	adverb
thick	thickness	noun

Nominalization is an especially important type of derivation—deriving a noun from a verb or adjective. Let's look at one example of derivation in detail—the verb "observe." As well as the usual inflectional tenses ("observing," etc.), we derive the adjective "observable" and, by nominalization, the noun "observation," the plural inflection "observations," another adjective, "observational," and an adverb, "observationally."

There are two other major morphological processes in which words are formed in English: *compounding* (ear + ache = earache, work + load = workload) and affixation (dis + arrange = disarrange). Compounding combines two words; affixation combines a word with a prefix or suffix. These actions can also be expressed in regular expressions ($s/^/ear/$), but for the purposes of this article we'll ignore compounding and affixation.

To summarize, we can transform words by adding certain prefixes or suffixes to a stem, or by joining two stems together. Salton (1989) estimated that there are about 75 prefixes and around 250 suffixes in English. Most importantly for us, Perl regular expressions are perfectly suited to performing all these transformations.

Now let's build a set of derivation and inflection rules and then use these rules to analyze a *corpus*, a collection of carefully selected documents. Several large

corpuses, representative of different types of documents, have been built. A typical representative corpus of everyday written language is the

British National Corpus. Details about the

BNC research project can be found on the Web at http://info.ox.ac.uk/bnc/. It is a collection of over 4,000 different documents with around 100 million words in over 6 million sentences. The number of distinct words—the *vocabulary*—is a little under 400,000. We'll use it for the examples below.

Morphological Analysis and Perl

We now present a series of Perl programs performing simple morphological inflections and derivations on single words. Our starting point is a series of documents, forming our corpus. Since we're working only with single words, we don't need to retain the original

word order in the documents. All we need is a list of the words that appear in the documents, and how many times each word appears.

Our first step, therefore, is to transform the documents into a word frequency list that tells us how many distinct word forms there are, and how many times each form occurs. The rest of the programs can then operate on the word frequency lists, which are much smaller and more easily managed than the documents themselves.

Constructing a Word Frequency List

Our first program, mkfreq, creates a word frequency list from a set of documents. The only hard part is defining what constitutes a word. One definition might be a series of alphanumeric characters; that is, anything not matching $/[\backslash W_] + /$.

This regular expression isn't perfect: it splits abbreviations like don't into two words. Let's live with that, and instead of calling the things identified by our regular expression words, we'll call them *tokens*. We'll routinely lowercase all tokens, since "work" and "Work" shouldn't be treated as different words. Here's mkfreq:

```
#!/usr/bin/perl
#
# mkfreq: take a series of documents and
produce a token
frequency file.
%freq = ();
%tokens = $maxw = 0;
while (<>) {
    chomp;
    foreach $token (split( /[\W_]+/)) {
        next unless $token;
        $freq{lc($token)}++;
        $tokens++;
        $width = length($token);
        $maxw = $width if $width > $maxw;
    }
```

```
}
$
vocab = keys(%freq);
$format = "%-${maxw}s %s\n";
print "====== Vocabulary: $vocab\n======
Total tokens: $tokens\n";
printf $format, "Token", "Freq";
foreach $token (sort {$freq{$b} <=>
$freq{$a}} keys(%freq)) {
    printf $format, $token, $freq{$token};
}
```

mkfreq uses a hash to store the

word frequencies, and displays two simple but useful measures of the linguistic diversity in the documents—the total number of tokens and the vocabulary. The

frequency list is displayed in descending order of frequency using a custom sort order. For cryptographic applications, a knowledge of the most frequent words in the English language is useful. A tabulation of the most frequent ten words in the

British National Corpus is shown in Table 20-4.

Table 20-4. The ten most frequent words in the British National Corpus

Word	Frequency
the	6,198,660
of	3,112,676
and	2,695,784

Word	Frequency
to	2,672,790
a	2,241,128
in	1,996,243
that	1,150,942
it	1,091,584
is	1,004,574
was	924,132
Total	23,088,513

Some observations: Repetitions of these ten words are responsible for 22.5% of the words in the entire corpus. They're all common determiners, conjunctions, pronouns, and prepositions, carrying no information about the subject of a document. Most search engines exclude these uninformative words from their indexes. Excluded words are called *stopwords*.

The

frequency list can also help us out by identifying misspelled words; we'd expect to find uncommon spelling mistakes at the bottom of our frequency list. Pragmatically, we might decide to exclude all tokens with a frequency below some threshold (say, one or two occurrences) in order to exclude many spelling mistakes (and, inevitably, some legitimate words) from our processing. In the

British National Corpus discussed above, 154,883 tokens occur exactly once and 49,465 tokens occur exactly twice. Together, these comprise 204,348 tokens out of the total vocabulary of 381,807 tokens—53% percent of the vocabulary occurs less than three times!

Excluding

low-frequency tokens is a simple addition to mkfreq. We omit that here, but it's in the version of mkfreq on the web page for this book (http://www.oreilly.com/catalog/tpj3). We also omit tokens containing digits.

Morphological Inflections and Derivations

Some

derivation rules apply only to particular

word categories—you wouldn't try to find the plural of a verb. However, there's no algorithmic way of identifying whether an English

word is a verb or noun. No rules or even heuristics are possible.

Ideally, we'd have some way to take a

word and find out which category it belonged to. However, this would require a knowledge base containing every base word from the corpus. Constructing and maintaining such a knowledge base would be a major undertaking.

So, let's turn the problem on its head: how far can we get with *no knowledge base whatever*? Could we, in fact, make reasonably accurate derivations based only upon our corpus? If we have no knowledge of

word categories, then we must start by

applying all derivation rules to all tokens in the corpus, and then see if there's any way of eliminating the duds.

Representing a Single Rule

A rule is basically an s/// regular expression. However, we need to store a collection of rules in an array and apply them all to a token. Having decided to ignore affixation and compounding, if we look back to Tables Table 20-1 and Table 20-2, we notice that all our

rules for inflection and

```
derivation are of the form s/ending$/suffix/.
```

This suggests that we can represent each rule as an ending-suffix pair with the meaning:

If token ends with ending, replace the ending part of token with suffix.

Given this, we could write a function that attempts to apply a rule pair (*ending*, *suffix*) to a given *token*, returning either the derived token or the empty string (if the token does not end in *ending*). This subroutine, derive, is shown below.

We embedded this into a test harness that we named applyruletotoken, which, at its core, is simply this:

```
my $d = derive( @ARGV );
print "Derivation is '$d'\n" if $d;
print "No derivation\n" unless $d;
```

We can now perform a few simple tests, shown in Table 20-5.

Table 20-5. Commands and associated responses

Command				Response
applyruletotoken	look	, ,	ing	Derivation is looking
applyruletotoken	look	's'	sing	No derivation
applyruletotoken	take	'e'	ing	Derivation is taking

Representing Many Rules

The next step is to figure out how to represent all the derivation rules so that we can apply all of them to each token and determine which succeed. For convenience we'll collapse our rule pair (*ending*, *suffix*) into a single string: *-ending+suffix*, or just *+suffix* if *ending* was empty. Below, we see the function setuprules, which creates an array of rules @rule and then processes it into an array of removal-strings @remove and an array of addition-strings @add. Only a few rules are shown, although we routinely use many more.

```
Set up the derivation rules and
#
processed forms.
sub setupRules {
    my ( $r, $a );
    @rule = (  # A rule is a disguised
pair of the form -R+A or +A
                "+s", "+es", "-us+i",
"-is+es",
                     # singular -> plural
noun
             "+ion", "+tion", "+sion",
"-te+tion",
             # verb -> noun
      "-ify+ification", "+ment",
       "+ness", "+fulness", "+ful", "+ic",
"+al", # noun -> adjective
           "-e+ing", "-s+ssing", "+ing",
"+ed",
              # verb -> tense forms
                                 "+ly"
);
                                         #
adjective -> adverb
    (add = (remove = ());
    foreach (@rule) {
```

We had to make a few small changes to derive, and our new version is shown below. Whenever it finds a valid derivation, it calls handlederivation, providing the token, the rule, and the derived token.

```
# Our updated derive(), which now calls
handlederivation()
sub derive {
    my ($t) = @ ;
    my ($i, $a, $r, $d);
    for ($i = 0; $i < @add; $i++) {</pre>
        a = add[si];
        $r = $remove[$i];
        d = t;
        if ($r) {
            next unless $t =~ /.$r$/;
            $d =~ s/$r$//;
        }
        $d = "$d$a";
          handlederivation( $t, $rule[$i],
$d );
    }
}
sub handlederivation {
    my ($token, $rule, $derivation) = @ ;
    print "$token\t$rule\t$derivation\n";
}
```

We embedded these subroutines into a test harness called applyallrules, whose heart is simply this:

```
setuprules();
derive(@ARGV);
```

A test such as applyallrules look generates the following output:

look	+s	looks
look	+es	lookes
look	+ion	lookion
look	+tion	looktion
look	+sion	looksion
look	+ment	lookment
look	+ness	lookness
look	+fulness	lookfulness
look	+ful	lookful
look	+ic	lookic
look	+al	lookal
look	+ing	looking
look	+ed	looked
look	+ly	lookly

As we would expect, this generates some good derivations (*looks, looking, looked*) but the majority of the derivations are not real English words.

Telling Good from Bad

We need some way to distinguish the real English words from the bogus, and this is the point at which an English

word knowledge base could be used. We could use an external wordlist such as the Unix dictionary */usr/dict/words*, rejecting any

derivation that doesn't appear, but

many uncommon words (and even some common ones) aren't included. Augmenting a wordlist with a large collection of specialized words is another major endeavor.

The Key Insight!

Fortunately, we have another wordlist available that is a source of exactly the specialized words that we need—the frequency list derived from the corpus! Using this insight, we propose the following method:

- Take an existing token in the frequency list.
- Apply all derivation rules to that token as above.
- Reject any derivation that is not itself present in the frequency list.

Implementing It

Our new system, filterallrules, is just applyallrules with two changes: the readfreqfile subroutine shown below reads a frequency list into %freq, and derive calls handlederivation only if the

word is in the frequency list: handlederivation(\$t, \$rule[\$i], \$d) if \$freq{\$d}.

```
sub readfreqfile {
                               # Read the
frequency list
    my ($filename) = @ ;
    my ($line, $
word, $count);
      open (IN, $filename) || die "can't
open $filename\n";
    %freq = ();
    while ($line = <IN>) {
        last unless $line =~ m/^======/;
        chomp $line;
         totaltokens = $1 if $line =~ m/=
Total tokens: s+(d+)/;
            vocab = 1 if sline = m/=
Vocabulary:\s+(\d+)/;
    while ($line = <IN>) {
        chomp $line;
        next if sline = ~ /^ d/;
        ($
word, $count) = split(/\s+/, $line);
        $freq{$
```

```
word} = $count;
        }
        close IN;
}
```

Applying this to look with a

frequency file derived from the

British National Corpus now correctly sifts out the true English words from the duds:

look	+s	looks
look	+ing	looking
look	+ed	looked

These inflections are rather straightforward and obvious. Consider a more complex example—the derivations of approximate:

```
approximate +s
approximates
approximate -te+tion
approximation
approximate -e+ing
approximating
approximate +ly
approximately
```

This technique is not perfect. There are two main types of false derivations produced: first, when two completely unrelated English words happen to match a derivation rule (for example, if the +s rule is applied to asses, we get assess). There are surprisingly few cases of this type, however, because English has strong morphological constraints.

The second and more common type of false derivation arises from specialized terms, proper names, foreign words, and spelling mistakes in our corpus. For example, the electrical term AC is lowercased to ac and then is used to generate seemingly related words such as aced and action. This is inevitable without deeper knowledge about the words we're processing.

Applying the Derivation Process to All Tokens

The final step is to apply this derivation process not just to one token, but to all tokens within the corpus, and to display the results in a pleasing format. To apply the derivation rules to all tokens, we define a new subroutine called deriveall that iterates over all tokens:

```
# Call derive() for every token in the
frequency table
sub deriveAll {
    my ( $t );
    %b2d = ();
    foreach $t (sort(keys(%freq))) {
        derive( $t );
    }
}
```

To tabulate the results, we redefine handlederivation as shown below. The successive calls to handlederivation now build a hash called %b2d (base-to-derivations) that maps each base token to an array of all of its derivations. We use references so that our hash can have many values for each key:

```
sub handlederivation {
   my ( $token, $rule, $
   derivation ) = @_;
   my ( $ref );
    $ref = $b2d{$token};
        $ref = $b2d{$token} = [ ] unless
defined $ref;
```

```
push( @$ref, $derivation );
}
```

Once deriveall completes, the entire %b2d hash has been constructed. To display all the derivations, we define a subroutine named tabulate, which displays each word and its frequency:

```
# Tabulate the results in the %b2d array.
sub tabulate {
    my (@b, $base, $list, $t);
    # @b contains all the base tokens (the
kevs of %b2d)
    @b = sort(keys(%b2d));
    foreach $base (@b) {
        # $list is a reference to an array
of derivations
        $list = $b2d{$base};
        f = freq{base};
        print "$base/$f\t";
        foreach $t (@$list ) {
            f = freq{t};
            print "$t/$f\t";
        }
        print "\n";
    }
}
```

We assembled a final test harness called tabulate which did little more than this:

```
readfreqfile($ARGV[0]);
setuprules();
```

```
deriveall();
tabulate();
```

Now the user can receive a summary of the derivations obtained, where each token is displayed followed by its frequency. Here is an excerpt of the output:

```
approximation/258
approximate/39
approximating/2
                    approximately/35
                  arguments/3
argument/3
arise/34
                     arises/19
arising/41
arrange/4
                      arrangement/17
arranging/1
artificial/10
                  artificially/1
aspect/2
                  aspects/4
assess/23
                      assessment/10
assessing/10
                    assessed/7
assistant/3
                  assistants/1
assume/68
                     assumes/10
assuming/69
assumption/49
                  assumptions/19
asymptotic/17
                  asymptotics/8
```

Summary

We have managed to obtain remarkably accurate heuristic derivations by taking a set of real texts, turning them into a frequency distribution, applying a series of very simple addition and substitution rules to every token, and rejecting derivations that do not appear in the frequency distribution. Some derivations are wrong, but a high percentage are accurate. The results are certainly good enough to enable Web searches for morphologically related words—a significant improvement over a dumb keyword search.

Future Work

To enhance our system, more rules could be added (we routinely use a much bigger set). More categories of replacement rules (such as affixation and compounding) could be added relatively easily.

We could make mkfreq retrieve an arbitrary web page, discard its HTML tags, and generate its frequency list using the LWP::Simple module, exactly as shown by Jon Orwant and Dan

Gruhl in *Web*, *Graphics*, *and Perl/Tk: Best of the Perl Journal*. A version that does this (webmkfreq) is available from the web page for this book.

Reading in the entire token frequency file cost a lot of time and memory, so we implemented it as a DBM file instead. This significantly speeds up programs that access only a few frequency elements (such as plural and adverb), but slows down programs like tabulate that access a very large number of frequency elements many times. We concluded that in our era of large high-speed computers and bountiful memory, reading a token frequency file into memory is realistic.

We could easily add the ability to include known word lists such as Unix's */usr/dict/words*, so that we could include entries for derivations of a base token that happens not to be present in the corpus.

We have also implemented two small applications called plural and adverb, which use expanded single-purpose

sets of rules to report on the probable plural and adverb forms (if any) of a given word. It answers the very first question we posed in this paper:

```
Adverb form of accident: accidently (3.5%) accidentally (96.5%)
```

Chapter 21. Smart Matching for Human Names

Brian Lalonde

A few months ago, I needed to synchronize our groupware's address book with our employee database. Since the address book provided only minimal data about employees, and the employee database didn't contain any of the exact fields in the address book, I had to synchronize them using nothing more than first and last names

This was not as easy as it sounds. It quickly became apparent that only a small portion of the >3500 records would match directly. A simple SQL join between imported tables would not work; the data was too inconsistent. After matching a few names manually, I became increasingly obsessed with the problem of matching names, e.g., identifying that "Bill Gates" was the same person as "William Gates III."

The problem was tenacious; I would add

processing to catch misspellings or hyphenations, and new issues would come up. The script quickly grew into a module, Lingua::EN::MatchNames, and then a second,

Lingua::EN::Nickname, in response to the bizarre and arbitrary conventions for shortening first names. Many first name forms have little or no similarity whatsoever: Peggy = Margaret = Midge, and several can follow an almost endless mutation path: Peggy > Margaret > Martha > Mary > Maryanne > Anna > Roseanne > Rosalyn > Linda > Melinda > ...).

When the initial versions were complete, my script was able to match the vast majority of the records on its own (with greater than 85% certainty per match), and most of the rest either had no real-world match, or were suggested by the highest confidence ranking of the script. While this is by no means a statistically significant sampling, it is extremely encouraging.

Once the

databases had been matched, I had convenient test data for the modules, since they could now check their work. This allowed me to emphasize successful test methods, and de-emphasize ones that provided too many false positives.

This article will show you how to use these modules, and explains a little about how they work.

Installing the Modules

Lingua::EN::MatchNames has several dependencies: Lingua::EN::NameParse, Lingua::EN::Nickname, String::Approx, Text::Metaphone, and Text::Soundex. Kim Ryan's Lingua::EN::NameParse, in turn, requires Parse::RecDescent. Lingua::EN::Nickname has no dependencies. All modules mentioned in this article are on CPAN. Although they aren't bundled, your CPAN module should be able to follow the dependencies:

% perl -MCPAN -e
"install('Lingua::EN::MatchNames')"

Windows users without a compiler will likely need to first use PPM to retrieve binaries of two of the requisites:

```
% ppm install String-Approx Text-Metaphone
```

Module Contents

The main module, Lingua::EN::MatchNames, exports one function by default: name_eq. You can either feed it four parameters:

```
name_eq( $firstname0, $lastname0,
$firstname1, $lastname1 )
```

or two (thanks to Lingua::EN::NameParse, which breaks full names into their constituent components):

```
name eq( $name0, $name1 )
```

and it will return a certainty score between 0 and 100, or undef if the names cannot be matched via any method known to the module.

If you ask for them, Lingua::EN::MatchNames will also export fname_eq or lname_eq, for matching first and last names. Both take two parameters, and each returns a certainty score between 0 and 100.

Lingua::EN::Nickname exports nickname_eq for matching first names solely on the basis of

nicknames, and nickroot, which attempts to look up the full (formal) first name(s), given a nickname. If asked, it will also export nickmatch, which returns a regex for matching all known full names given a nickname; or nickfollow, which recursively searches for a path of related names joining the two names passed to it, and returns the number of "hops" between the two. In practice, unless you have a specific need, you will probably use Lingua::EN::Nickname only indirectly, through Lingua::EN::MatchNames.

Using the Modules

These modules were designed for completeness over speed or size. You can expect that matching thousands of records will take hours when testing every possible match.

Typically, your script will build an array of [uniqueid, firstname, lastname] elements from the first database, then iterate over the records in the second database, collecting matches as shown below:

use Lingua::EN::MatchNames; my \$certainty threshold = 85; # 85% certain is reasonable # Set up the pool of potential matches open NAMES, '<listA.tsv' or die \$!; # Assume a tab-delimited file of idnum, firstname, lastname $\text{@pool} = \text{map} \{ \text{chomp}; [\text{split} / t /] \} <$ NAMES>; # Again, assume a tab-delimited file of idnum, firstname, lastname open NAMES, '<listB.tsv' or die \$!; # Iterate over list B, finding matches while(<NAMES>) {

```
chomp;
     my ($idnum, $fname, $lname) = split
/\t/;
      my Qmatches = sort { $b \rightarrow [0] <=>
$a->[0] } # most certain first
      map {
       my $score=
name eq( \$fname, \$lname, \$ ->[1], \$ ->[2]
);
        $score ? [ $score, @$ ] : ()
      } @pool;
    unless (@matches) {
        # no matches found
         print "$idnum ($fname $lname): no
matches\n";
   }
         elsif ( @matches == 1 and
$matches[0]->[0] > $certainty threshold ) {
        # one solid match
         my ($score, $m idnum, $m fname,
m lname) = Q{matches[0]};
         print "$idnum ($fname $lname) =
$m idnum ($m fname $m lname) $score%\n";
    } else {
        # several potential matches, or an
uncertain one
            print "$idnum ($fname $lname)
matches:\n";
        foreach (@matches) {
                    my ($score, $m idnum,
$m fname, $m lname) = @$ ;
              print " $m idnum ($m fname
$m lname) $score\n";
        }
    }
}
```

The above code assumes that your names are in a file with some ID number, a first name, and then a last name, all separated with tabs. Obviously, since the functions exported by a

module accept simple strings, you can extract your names from anything you like, or even the command line.

What They Do

name_eq simply requires that some certainty exist for both
first and last

names, and returns a score that combines the two, weighting the last name more heavily (70%).

```
fname_eq matches first names like so:
```

1. Simple equality is tested.

Trivial matches return a certainty of 100.

2. Informal names, delimited by parentheses or quotes, are recursed.

Portions of the name wrapped in parentheses or quotes are recursively checked against the potentially matching name with fname_eq. "William (Chip)" would therefore match "Chip," with full certainty of the best submatch.

3. Extraneous initials are removed.

"H. Ross" matches "Ross" with a high certainty level; "Ross H." would also match at this stage. This step must be performed before the next step to avoid trivially poor chunk matches.

4. Name chunks, separated by symbols or mixed case, are recursed.

Names are broken into pieces at nonword characters, or capitalization changes. These parts are each recursively

checked against the potentially matching name with fname_eq. "Mary" would therefore match "MaryAnn," "Mary-Ann," or "Mary Ann," again with full certainty of the best submatch.

5. Inconsistent case is flattened to all uppercase, and spaces and symbols are removed.

Legacy databases, particularly mainframes, tend to favor ALL UPPERCASE DATA. At this stage, "Arthur" matches "ARTHUR."

 Nicknames are followed, matched, and ranked based on proximity, using Lingua::EN::Nickname.

This is an alarmingly difficult problem that eventually grew into its own module.
Matching Nicknames

Lingua::EN::Nickname is a much more terrifying animal than Lingua::EN::Match-Names, due to the capricious nature of nicknames. Basically, it builds four giant hashes: one for looking up nicknames that have a single known root name (that is, full/formal form), one for looking up ambiguous nicknames, one for finding a regular expression matching all known forms of a root name, and one for mapping related root names. The functions in this module are fairly straightforward users of these hashes.

This vast hash data should not be edited by hand. Although I would prefer to receive mail regarding omissions, a utility for generating the Perl code for these hashes (nickhash.pl) is provided, and is a good starting point for other languages (Lingua::XX::Nickname). The original datafile used by my utility, *nicknames.txt*, is also included; each line consists of three tab-separated fields: root name, nicknames (a space/ nonword-separated list), and related root names.

Any certainty returned by nickname_eq is scaled back a little, and returned.

Continuing where we left off, the remaining four steps for matching first names are:

1. Misspellings are detected, within a threshold, using String::Approx.

Data-entry errors are tolerated by determining how many character insertions, transpositions, and deletions would be required to change one of the names into the other, and ensuring that this number is below the default threshold for String::Approx (10% of the characters, rounded up) for matches. "Bart" and "Bort" would now match one another, and "Brian" and "Brain" would now match

2. Phonetic matching catches similar pronunciation, using Text::Metaphone.

Homonymous names are now matched using the surprisingly accurate Text::Metaphone module, which allows us to find "Sandy" = "SanDee," and "Cindy" = "Sindy," but with relatively low certainty.

3. Somewhat similar last names are caught with Text::Soundex.

Out of completeness, I've included soundex matching, which is very tenuous (especially for first names). Soundex matches return a very low certainty.

4. Regular expressions are checked for simple truncation and relevant initials.

Initials are re-checked to see if they may not have been pure noise ("H. Ross" = "Herman"), and simple truncation is checked ("Bri" = "Brian"). Because of the tiny amount of matching data, this is the least certain match.

Matching last names with lname_eq is a slightly different process:

- 1. Simple equality is tested.
- 2. Extraneous suffixes are removed.

Persons usually include unused names as initials formally; frequently, this means initials are simply noise. "Smith, Jr." matches "Smith" with a high certainty level; "Smith II" would also match at this stage.

3. Hyphenated surnames are recursed.

Names are broken into pieces at hyphens; "Bouvier-Simpson" would be recursively checked to match the other name against "Bouvier" and "Simpson," and if any matches are found, the score of the best submatch is returned.

4. Inconsistent case is flattened to all uppercase, and spaces and symbols are removed.

Legacy databases, particularly mainframes, tend to favor ALL UPPERCASE DATA. Also, handling of names that contain apostrophes or spaces is terribly inconsistent. At this stage, "O'Neil" matches "O NEIL" and "ONEIL."

5. Misspellings are detected, within a threshold, using String::Approx.

"Hanson" and "Hansen" would now match, as well as "Simpson" and "Smipson."

- 6. Phonetic matching catches similarly pronounced last names, using Text::Metaphone.
- 7. Somewhat similar last names are matched with Text::Soundex.

8. Regular expressions are checked for nonstandard hyphenation or simple truncation.

Nonstandard hyphenation (like lower-to-upper case changes) are caught and recursed.

The certainty scores returned by fname_eq and lname_eq are generated from weights I assigned after ranking each of these steps, based on frequency of use, reliability, some final tweaking after checking against my test data, and on my fondness for round numbers. These scores seem to be pretty reliable relative to each other, but ultimately have an inadequate scientific basis. Anyone willing to do research that would result in more accurate scores for these weights is encouraged to do so, and will receive credit in future versions of this module, plus my thanks.

Conclusion

So there you have it: how to match names with Lingua::EN::MatchNames and Lingua::EN::Nickname, and a peek under the hood to see how they work. If you are looking for a project, and have access to a vast database of names, the most critical work is precisely refining the weight of each step, and filling in any gaps in the list of nicknames.

Chapter 22. Localizing Your Perl Programs

Sean M. Burke

Jordan Lachler

Once upon a time, when the Internet was merely an experiment in getting a few dozen machines to talk to each other without actually melting, and when computer science was about getting your accounting program to run in 5K of core, it didn't matter that program output was in English, only in English, and MAYBE EVEN ALWAYS IN CAPITALS. After all, computers were basically designed by and for a few American engineers, and as long as packets were swapped and numbers were crunched, everyone was happy.

But nowadays, computers are becoming part of daily life for much of the planet, and that means that the average user is less and less likely to be a native speaker of English. And software that doesn't work in your native language is very annoying, even if it does work in some other language you understand fluently.

The first step to making software "work," in your language of choice, is called *internationalization* (often abbreviated "118N"

). Internationalizing a piece of software, or a file format, or a protocol, basically means making sure it can convey text in any language. Mercifully, this has been mostly taken care of;

modern protocols and data formats, like MIME-encoded email, HTTP, HTML, and XML, do a fine job of identifying the character set of your text, so that whatever program receives your text will know how to display it. And unlike in the old days, we now have standard character sets capable of representing text in almost any language: notably, there's

Latin-1, which does fine for English and most other Western European languages, there's

Unicode, which works for all languages, and there's also a slew of other language-specific character sets like

KOI8 (among others) for Russian,

JIS (among others) for Japanese,

VISCII for Vietnamese, and so on. (That's the great thing about standards—there are so many to choose from.)

You can use an email program to write email in whatever language you want, but chances are the interface is still only in English. That software doesn't *really* "work" in your language of choice.

Making the *interface* to a program work in the user's language of choice is called *localization* (often abbreviated "L10N"

), and that's what this article is about. For the programmer, localization means an extra bit of bookkeeping; instead of having bits of text hard-coded in your program's interface, they get looked up in a little lexicon module—so that if the user is using the program's French interface (assuming one has been provided), your program won't say "File not found," but instead will look up the French phrase for that and say "Fichier non trouvé." And where a GUI button used to say "Search," it now says "Cherchez."

The most widely used localization system is GNU *gettext*, and while it's a definite advance over previous systems, it and similar systems suffer from some basic deficiencies. Simply put, they don't deal well with the different ways that different languages phrase things. Before I propose solutions to these problems, I have devised a tale of woe to illustrate how frustrating these problems can be.

A Localization Horror Story: It Could Happen to You

Imagine that your task for the day is to localize a piece of software someone else in your company wrote. Suppose it's a simple search tool of some sort, the exact details of which aren't important. Luckily for you, the only output the program emits is two messages, like this:

```
I scanned 12 directories.
Your query matched 10 files in 4 directories.
```

How hard could that be? You look at the code that produces the first item, and it reads:

```
printf("I scanned %g directories.",
$directory_count);
```

First you have to look up what %g does—it performs number interpolation with nice formatting. But then you think about the above code, and you realize that it doesn't even work right for English, as it can produce this output:

I scanned 1 directories.

So you rewrite it to read:

```
printf("I scanned %g %s.",
$directory_count,
        $directory_count == 1 ? "directory"
: "directories");
```

which does the Right Thing. (While looking up %g in the Perl docs for sprintf, you learned that %s is for interpolating strings.)

But you still have to localize it for all the languages spoken by your users, and after a little poking around in CPAN, you find the Locale::gettext module, which is an interface to *gettext*, a set of C routines that seem well suited to this task. After some Googling, you find the *gettext* manual. You browse through the tutorial, and, following its examples, you start to write:

But you see later in the *gettext* manual that this is not a good idea, since how a single word like "directories" is translated depends on context. In languages like German or Russian, the "directories" of "I scanned 12 directories" demands a different case than the "directories" of "Your query matched 10 files in 4 directories." The first is the object of a verb, and the second is the object of a preposition.

So, on the advice of the gettext manual, you rewrite:

```
printf($dir_scan_count == 1 ?
    gettext("I scanned %g directory.") :
        gettext("I scanned %g
directories."), $dir_scan_count);
```

The boss decides that the languages du jour are

Chinese, Arabic, Russian, and Italian, so you hire one translator for each and ask for translations of "I scanned g directory" and "I scanned g directories." When they reply, you'll put that in the lexicons for *gettext* to use when it localizes your software, so that when the user is running under the zh (

Chinese) locale, gettext("I scanned βα directory.") returns the appropriate Chinese text, with a %g in there where printf can then interpolate the number of directories scanned. (Locale primarily means a choice of language. accompaniments like and character sets. preferences for expressing numbers-for example, whether one and a half is 1.5 or 1,5—and preferences for sort order, since not all languages have the same alphabetical order. Since we don't talk about those other preferences in this article, just think "language" whenever you see "locale".)

Your Chinese translator mails right back—he says both of these phrases translate to the same thing in Chinese, because, to use linguistic terminology, Chinese "doesn't have number as a grammatical category" like

English does. That is, English has grammatical rules that depend on whether something is singular or plural; one of these rules is the one that forces nouns to take a suffix (usually "s") when there's more than one ("one dog, two dogs"). Chinese has no such rules, and so has just one phrase where English needs two. No problem; you can have this one Chinese phrase appear as the translation for the two English phrases in the zh gettext lexicon for your program.

Emboldened by this, you dive into the second phrase that your software needs to output: "Your query matched 10 files in 4 directories." You notice that if you want to treat phrases as indivisible, as the *gettext* manual wisely advises, you need four cases to cover the permutations of singular and plural on each of \$dir_count and \$file_count. So you try this:

(The case of "1 file in 2 [or more] directories" could, I suppose, occur with symbolic links in the filesystem.)

This isn't the prettiest code you've ever written, but this seems the way to go. You mail the translators asking for translations for these four cases. The

Chinese guy replies with the one phrase that these all translate to in Chinese, and that phrase has two %gs in it, as it should—but there's a problem. He translates it word-for-word: "To your question, in %g directories you would find %g answers." The %g slots are reversed. You wonder how you'll get *gettext* to handle that.

But you put it aside for the moment, and optimistically hope that the other translators won't have this problem, and that their languages will be better behaved—that they'll be just like English. The

Arabic translator is the next to write back. First, your code for "I scanned %g directory" or "I scanned %g directories" assumes there's only singular or plural. But, to use linguistic jargon again,

Arabic has grammatical number, like English and unlike Chinese. However, it's a *three-term* category: singular, dual, and plural. In other words, the way you say "directory" depends on whether there's one directory, two of them, or more than two of them. Your test of (\$directory == 1) no longer does the job. And it means that where English's grammatical category of number necessitates only two permutations of the first sentence, Arabic has three—and, worse, in the second sentence ("Your query matched %g file in %g directory"), Arabic has nine possibilities where English had only four. You sense an unwelcome, exponential trend taking shape.

Your Italian translator emails you back and says that "I searched 0 directories" (a possible output of your program) is stilted, and if you think that's fine English, that's your problem, but that just will not do in the language of Dante. He insists that where \$directory_count is 0, your program should produce the Italian equivalent of "I *didn't* scan *any* directories." And ditto for "I didn't match any files in any directories," although he adds that the last part about "in any directories" should probably be omitted altogether.

You wonder how you'll get *gettext* to handle this; to accommodate the ways Arabic, Chinese, and Italian deal with numbers in just these few very simple phrases, you need to write code that asks *gettext* for different queries depending on whether the numerical values in question are 1, 2, more than

2, or in some cases 0, and you still haven't figured out the problem with the different word order in Chinese.

Then your

Russian translator calls, to personally deliver the bad news.

Russian, like German or Latin, is an inflectional language; that is, nouns and adjectives take endings that depend on their case (nominative, accusative, genitive, and so on; what role they play in the syntax of the sentence)—as well as on the gender (masculine, feminine, neuter) and number (singular or plural), as well as on the declension class of the noun. But unlike other inflected languages, putting a number-phrase (like "ten" or "forty-three") in front of a Russian noun can change the case and number of the noun, and therefore its ending as well.

He elaborates: In "I scanned %g directories," you'd expect "directories" to be in the accusative case (since it is the direct object) and a plural, except where \$directory_count is 1—then you'd expect the singular, of course. Just like Latin or German. But! Where \$directory_count % 10 is 1 (assuming \$directory_count % 10 is 1) (assuming \$directory_count % 100 is 11), "directories" is forced to become grammatically singular, which means it gets the ending for the accusative singular.

You begin to visualize the code it'd take to test for the problem so far, and still work for

Chinese and Arabic and Italian, and how many *gettext* items that'd take. But he keeps going. Where \$directory_count % 10 is 2, 3, or 4 (except where \$directory_count % 100 is 12, 13, or 14), the word for "directories" is forced to be genitive singular—which

means another ending. The room begins to spin around you, slowly at first... And with all other integer values, since "directory" is an inanimate noun, when preceded by a number and in the nominative or accusative cases (as it is here, just your luck!), it does stay plural, but it is forced into the genitive case—yet another ending. And because the floor comes up to meet you as you fade into unconsciousness, you never get to hear him talk about the similar but subtly different problems with other Slavic languages like Polish.

The above cautionary tale relates how an attempt at

localization can lead from programmer consternation to program obfuscation to a need for sedation. But careful evaluation shows that your choice of tools merely needed further consideration.

The Linguistic View

The field of

linguistics has expended a great deal of effort over the past century trying to find grammatical patterns that hold across languages; it's been a constant process of people making generalizations that should apply to all languages, only to find out that, all too often, these generalizations fail-sometimes failing for just a few languages, sometimes whole classes of languages, and sometimes nearly every language in the world except English. Linguists can make broad statements about the "average language," but the "average language" is as unreal a concept as the "average person"-no language (or person) is entirely average. The wisdom of past experience suggests that any given language can do just about whatever it wants, in any order, with any kind of grammatical categories—case, number, tense, real metaphoric or characteristics of the concepts that the words refer to, arbitrary classifications of words based on what endings or prefixes they accept, degree of certainty about the truth of statements expressed, and so on.

Mercifully, most

localization tasks are a matter of finding ways to translate fixed phrases in their entirety, and where the only variation in content is in a number being expressed, as in the example sentences above. Translating specific, fully-formed sentences is, in practice, fairly foolproof—which is good, because that's what's in the phrasebooks that so many tourists rely on.

Breaking gettext

Most sentences in a tourist phrasebook are of two types: ones like "How much do these ______ cost?" where there's a blank to fill in, and "How do I get to the marketplace?" where there isn't. The ones with no blanks are no problem, but the fill-in-the-blank phrases may not be straightforward. If it's a Swahili phrasebook, for example, the authors probably didn't bother to tell you the complicated ways that the verb "cost" changes its inflectional prefix depending on the noun. The trader in the marketplace will still understand what you're saying if you say "How much do these potatoes cost?" with the wrong inflectional prefix. After all, you can't speak proper Swahili, you're just a tourist. Tourists are supposed to be stupid. Computers are supposed to be smart. The computer should be able to fill in the blank, and have the result be grammatical.

In other words, a phrasebook entry accepts a parameter (the word that goes in the blank), and returns a value based on that parameter. In the case of Chinese, this operation is simple; in the case of Russian, it's quite complex.

This talk of parameters and complexity is just another way to say that an entry in a phrasebook is what we programmers call a "function." Just so you don't miss it, this is the crux of the article: *A phrase is a function; a phrasebook is a bunch of functions*.

The reason that *gettext* runs into walls is that it tries to use strings to do something that requires a function, which is futile. Performing printf interpolation on the strings you get back from *gettext* allows you to do *some* common things passably well, sometimes, sort of. But to paraphrase what some people say about csh script programming, "It fools you into thinking you can use it for real things, but you can't, and you don't discover this until you've already spent too much time trying, and by then it's too late."

Replacing gettext

So, what we need to replace *gettext* is a system that supports lexicons of functions instead of lexicons of strings. An entry in a lexicon from such a system should *not* look like this:

J'ai trouvé %g fichiers dans %g répertoires

but instead like this, although this is just a first stab:

```
sub I_found_X1_files_in_X2_directories {
  my ( $files, $dirs ) = @_[0,1];
  $files = sprintf("%g %s", $files, $files
== 1 ? 'fichier' : 'fichiers');
  $dirs = sprintf("%g %s", $dirs, $dirs ==
1 ? "répertoire" : "répertoires");
  return "J'ai trouvé $files dans $dirs.";
}
```

Now, there's no particularly obvious way to store anything but strings in a *gettext* lexicon, so it looks like we just have to start over and make something better, from scratch. I call my shot at a *gettext*-replacement system "Maketext," or, in CPAN terms,

Locale::Maketext.

When designing Maketext, I planned its main features in terms of "buzzword compliance."

Buzzwords: Abstraction and Encapsulation

The complexity of a language is abstracted inside (and encapsulated within) the

Maketext module for that interface. When you call:

```
print $lang->maketext("You have
[quant,_1,piece] of new mail.",
scalar(@messages));
```

you don't know (and in fact can't easily find out) whether this will involve lots of figuring, as in Russian, or relatively little, as in Chinese. That kind of abstraction and encapsulation may encourage other pleasant buzzwords like modularization and stratification, depending on what design decisions you make.

Buzzword: Isomorphism

"Isomorphism" means "having the same structure or form"; in discussions of program design, the word takes on the special, specific meaning that your implementation of a solution to a problem *has the same structure* as, say, an informal verbal description of the solution, or maybe of the problem itself.

What's wrong with gettext code like this?

First, it's not well abstracted. These ways of

testing for grammatical number should be abstracted to each language module, since how you get grammatical number is language-specific.

Second, it's not isomorphic. The verbal "solution" to our problem is "The way to say what you want in Chinese is with the one phrase 'For your question, in y directories you would find x files"—and so the implementation should be a straightforward way to spit out that one phrase with the

numerals properly interpolated. The complexity of one language shouldn't impede the simplicity of others.

Buzzword: Inheritance

There's a great deal of reuse possible for sharing phrases between modules for related dialects, or for sharing

auxiliary functions between related languages. (By *auxiliary functions*, I mean functions that don't produce phrase-text, but answer questions like "Does this number require a plural noun after it?" Such auxiliary functions would be used internally by functions that actually do produce phrase-text.)

Let's assume that you have an interface already localized for American English.

Localizing it for UK English should be just a matter of running it past a British person with the instructions to indicate which phrases need rewordings or minor spelling tweaks. The UK English

localization

module should have only those phrases that are UK-specific; all the rest should inherit from the American English

module. The same situation should apply with Brazilian and Continental Portuguese, possibly with some closely related languages like Czech and Slovak, and possibly with the slightly different versions of written Mandarin Chinese, as I hear exist in Taiwan and mainland China.

For auxiliary functions, consider the problems with

Russian numbers. Obviously, you'd want to write only once the hairy code that, given a numeric value, returns which case and number a noun should use. But suppose you discover, while

localizing an interface for, say, Ukrainian (a Slavic language related to

Russian, spoken by several million people), that the rules are the same as in Russian for quantification, and many other grammatical functions. While there may well be no phrases in common between Russian and Ukrainian, you could still choose to have the Ukrainian module inherit from the Russian module, just for the sake of inheriting all the various grammatical methods. Or, better, you could move those functions to a module called East_Slavic, from which Russian and Ukrainian could inherit, but which itself has no lexicon.

Buzzword: Concision

Okay,

"concision" isn't a real buzzword. But it should be, so I decree that as a new buzzword, *concision* means that simple common things should be expressible in very few lines (or maybe even just a few characters) of code—call it a special case of "making simple things easy and hard things possible." It played a role in the MIDI::Simple language, discussed earlier in Chapter 16. Or just think of it this way: usefulness plus brevity equals concision.

Consider our first stab at an entry in our phrasebook of functions:

```
sub I_found_X1_files_in_X2_directories {
    my ( $files, $dirs ) = @_[0,1];
        $files = sprintf("%g %s", $files,
$files == 1 ? 'fichier' : 'fichiers');
        $dirs = sprintf("%g %s", $dirs, $dirs
== 1 ? "répertoire" : "répertoires");
        return "J'ai trouvé $files dans
$dirs.";
}
```

You may sense that a lexicon consisting of functions like these would quickly get repetitive. And you may also sense that you don't want to bother your translators with having to write

Perl code—you'd much rather that they spend their very costly time on actual translation.

In a first-hack implementation of Maketext, each language-module's lexicon looked like this:

```
%Lexicon = ( "I found %g files in
                                        Яd
directories"
             => sub {
                       my( $files, $dirs )
= @ [0,1];
                                  $files =
guant($files, "fichier");
                       $dirs = guant($dirs,
"répertoire");
                        return "J'ai trouvé
$files dans $dirs.";
                }.
                  ... and so on with other
phrase = sub mappings ...>
           );
```

but I immediately went looking for a more concise way to denote the same phrase-function—a way that would also serve to denote *most* phrase-functions in the lexicon for *most* languages. After much time and thought, I decided on this system:

- Where a value in %Lexicon is a string instead of a subroutine, it is interpreted as a sort of shorthand expression of what the subroutine does. When accessed for the first time in a session, it is parsed, turned into Perl code, and then eval 'd into an anonymous subroutine; then that subroutine replaces the original string in that lexicon. (That way, the work of parsing and evaluating the shorthand form for a given phrase is performed no more than once per session.)
- Calls to maketext happen through a "language session handle" very much like an IO:: handle, in that you open one at the start of the session, and use it for sending signals to an object to have it return the text you

want. So this code means: look in the lexicon for \$lang (which may inherit from any number of other lexicons), and find the function that we happen to associate with the string "You have [quant,_1,piece] of new mail."

```
$lang->maketext("You have
[quant,_1,piece] of new mail.",
scalar(@messages));
```

If we find such a function, we call it with \$lang as its first parameter, and a copy of scalar (@messages) as its second. If that function was found in string shorthand instead of as a real subroutine, parse it and make it into a function before calling it.

• The shorthand uses brackets to indicate method calls that should be performed. For instance, this string will trigger the quant method:

```
"You have [quant, _1, piece] of new mail."
```

That string is shorthand for this anonymous subroutine:

```
sub {
  my ($handle, @params) = @_;
    return join '', "You have
",$handle->quant($params[0],'piece'),"of
  new mail.";
}
```

where quant is a method you've written to quantify the noun (piece) given a number (\$params[0]).

A string with no brackety calls, like this:

```
"Your search expression was malformed."
```

is a degenerate case, and just gets turned into:

```
sub { return "Your search expression
was malformed." }
```

- Starting with perl-5.7.3, Locale::Maketext is in the Perl core distribution, along with I18N::LangTags (a module for dealing with RFC 3066 language tags) and I18N::LangTags::List (a module that maps language tags to their English names, like "uk" to "Ukrainian").
- As this book goes to press, Lincoln D. Stein's popular Apache::MP3 module-distribution is being modified to use Locale::

Maketext to make its interface available in dozens of different languages. It too is available in CPAN.

However, not everything you can write in Perl can be expressed in this shorthand—not by a long shot. For example, consider our Italian translator, who wanted the Italian for "I didn't find any files" as a special case, instead of "I found 0 files." That couldn't be specified (at least not easily or simply) in our shorthand system, and it would have to be written out in full, like this:

Next to a lexicon full of shorthand code, this sticks out like a sore thumb—but it *is* a special case, after all, and at least it's possible, if not concise.

As to how you'd implement the

Russian example from the beginning of the article, well, There's More Than One Way To Do It. It could be something like this (using English words for

Russian, just so you know what's going on):

```
"I [quant,_1,directory,accusative] scanned."
```

This shifts the burden of complexity to the quant method. That method's parameters are: the number, the Russian word it's going to quantify, and the parameter accusative, which means that this sentence's syntax wants a noun in the accusative case.

Now, the Russian quant method here is responsible not only for implementing the strange logic necessary for figuring out Russian number-phrases, but also for inflecting the Russian word for "directory." How that inflection is to be carried out is no small issue, and among the solutions I've seen, some are straightforward but not very scalable, and others involve more complexity than is justifiable for all but the largest lexicons.

Mercifully, this design decision becomes crucial only in the hairiest of inflected languages, of which Russian is by no means the worst. Most languages have simpler inflection systems; for example, in English or Swahili, there are generally no more than two possible inflected forms for a given noun ("error/errors"; "kosa/makosa"), and the rules for producing these forms are fairly simple. A simpler inflection system means that design decisions are less crucial to maintaining sanity, whereas the same decisions might incur overhead-versus-scalability problems in languages like Russian. It may *also* be likely that code has already been written for the language in question, as with Lingua::EN::Inflect for English nouns.

Moreover, there is a third possibility simpler than anything discussed above: Just require that all possible forms be provided in the call to the given language's quant method, as in "I found [quant,_1,file,files]." That way, quant just has to chose which form it needs, without having to look up or generate anything. While possibly suboptimal for Russian, this should work well for most other languages, where quantification is not as complicated.

The Devil in the Details

There's plenty more to Maketext than described above—for example, the details of how language tags interact with module naming. Module tags are the things you see in an HTTP Accept-Language header (en-US, x-cree, fi, and so on) or locale IDs like you'd see in \$ENV{ `LANG' } (they have underscores instead of hyphens: en_US for US English, po_BR for Brazilian Portuguese). There are the details of how to stipulate what character encodings Maketext will return text in (UTF8? Latin-1? KOI8?). There's the interesting fact that Maketext is for

localization, but nowhere actually has a use locale in it. For the curious, there are the somewhat frightening details of how I implement something like data inheritance so that searches across %Lexicon hashes of modules parallel how Perl implements method inheritance.

And, most importantly, there are all the practical details of how to go about using Maketext for your interfaces, and the various tools and conventions for starting out and maintaining individual language modules.

That is all covered in the documentation for

Locale::Maketext and the modules that come with it, available in CPAN. After having read this article, which covers the "why" of Maketext, the documentation, which covers the "how" of it, should be quite straightforward.

But to give just a taste of it, here is the outline of code for English and French in a mythical application called BogoQuery. Here's the BogoQuery/ L10N.pm file:

```
use Locale::Maketext;
package BogoQuery::L10N;
@ISA = qw(Locale::Maketext);
If you wanted any new methods accessible
to all
of your lexicons, they'd go here.
Otherwise, just inherit from
Locale::Maketext, which provides some sane
defaults.
```

1;

The file BogoQuery/ L10N/en.pm:

```
"Your query matched
[quant,_1,file,files]
in
[quant,_1,directory,directories]."
=> "Your query matched
[quant,_1,file,files]
in
[quant,_1,directory,directories].");
1;
```

The file BogoQuery/L10N/fr.pm:

```
package BogoQuery::L10N::fr; # French
@ISA = qw(BogoOuery::L10N);
...methods specific to French go here...
%Lexicon
         =
                          "Ι
                                 scanned
                    (
[quant, 1, directory, directories]."
                   => "[quant, 1, répertoire
lu, répertoires lus].",
                       "Your query matched
[quant, 1, file, files]
                                           in
[quant, 1, directory, directories]."
                          => "J'ai trouvé
[quant, 1, fichier, fichiers]
                                       dans
[quant, 2, répertoire, répertoires]." );
1;
```

And finally, the main module, named BogoQuery.pm:

```
package BogoQuery;
use BogoQuery::L10N;
my $language =
BogoQuery::L10N->get_handle()
        || die "Can't find an acceptable
language module!?!";
```

```
When called with no parameters like
this, get handle()
 returns a handle to an appropriate
language lexicon,
# based on things like $ENV{'LANG'}, or if
running as a
# CGI program, $ENV{'HTTP ACCEPT LANGUAGE'}
... code that runs a BogoQuery...
print $language->maketext("I scanned
[quant, 1, directory, directories].",
$directory count);
... then code that counts up the matches...
print $language->
maketext("Your guery matched
[quant, 1, file, files]
                                        in
[quant, 1, directory, directories].",
$matched file count,
$matched directory count);
```

Adding support for new languages is now just a matter of having a translator provide the text for a new BogoQuery/L10N/zh.pm (zh for

Chinese), it.pm (it for Italian), and so on.

Because of Russian's complicated handling of numbers, BogoQuery/L10N/ru.pm would have to provide a quant method of its own, but that wouldn't require any change to the other modules. The same is true for

Arabic, since its quant method would deal with the singular/dual/plural distinction in the language.

Chinese, which was so problematic for *gettext*, is easy with Maketext, with a %Lexicon entry like this:

```
"Your query matched [quant,_1,file,files]
in [quant,_1,directory,directories]."
=> "To your question, in
[quant,_2,directory] you would find
[quant, 1,answer]."
```

(I'm using English words in place of the actual Chinese text, just for the sake of this article.) Incidentally, the quant method in Chinese wouldn't need to do anything more than put a number in front of the noun, since there's no grammatical pluralization in Chinese.

The case of Italian requiring "I didn't scan *any* directories" instead of "I scanned 0 directories"—well, that's the one case so far that can't be treated via our shorthand notation. It requires actual Perl code:

```
"I scanned
[quant,_1,directory,directories]."
=> sub {
    my ($lg, $dir_count) = @_[0,1];
        return "I didn't scan any
directories." if $dir_count == 0;
    return "I scanned 1 directory." if
$dir_count == 1;
    return "I scanned $dir_count
directories.";
}
```

However, such cases are relatively rare. Most phrases can be translated either as fixed strings, or fixed strings with a few bracket shorthand bits, meaning that the translators can focus on the translating.

Proof in the Pudding: Localizing Web Sites

Maketext and *gettext* have a notable difference aside from their approaches to languages: *gettext* is in C, accessible through C library calls, whereas Maketext is in Perl, and can't work without a Perl interpreter. Unlucky accidents of history have made C++ the most common language for the implementation of applications like word processors, web browsers, and even many in-house applications like custom query systems. Current conditions make it somewhat unlikely that the next one of any of these kinds of applications will be written in Perl, albeit more for reasons of inertia than what tool is best for the job.

However, other accidents of history have made Perl a well-accepted language for design of server-side programs (often CGI programs) for web site interfaces. Localization of static pages in web sites is trivial, either with simple language-negotiation features in servers like Apache, or with some kind of server-side inclusions of language-appropriate text into layout templates. However, the localization of Perl-based search systems (or other kinds of dynamic content) in web sites, be they public or access-restricted, is where Maketext will see the greatest use.

The ever-increasing internationalization of the Web makes it increasingly likely that the interface to the average dynamic content service will be localized for two or maybe three languages. It is my hope that Maketext will make that task as simple as possible, and will remove previous barriers to localization for languages dissimilar to English.
References

Alvestrand, Harald Tveit. *RFC 3066: Tags for the Identification of Languages.* ftp://ftp.isi.edu/in-notes/rfc3066.txt.

Callon, Ross, editor. *RFC 1925: The Twelve Networking Truths*. ftp://ftp.isi.edu/in-notes/rfc1925.txt.

Drepper, Ulrich and Peter

Miller. GNU *gettext*. Available at ftp://prep.ai.mit.edu/pub/ gnu/. The manual is accessible at http://www.gnu.org/ software/gettext/gettext.html. There's a handy Perl module for interfacing with *gettext* called Locale::gettext, available at http://www.cpan.org/modules/by-module/Locale.

Forbes, Nevill. *Russian Grammar*. Third Edition, revised by J. C. Dumbreck. Oxford University Press, 1964.

Locale::Maketext is on CPAN.

Chapter 23. Internationalized Sorting

Sean M. Burke

In my first semester of Spanish class in high school, I went to look up an unfamiliar word, "chaleco," in a Spanish-English dictionary. I looked under "C", and found that the dictionary went right from "cetro" to "cía". Someone had expurgated all the "ch" words! I had a brief nightmarish vision of a world without chorizo, chimichangas, chicharrones, or churros. After some frantic page-turning, I discovered that the "ch" words were in a separate section, "Ch", between "C" and "D". I asked the teacher about this, and he explained that it was normal practice for Spanish

alphabetical order to consider "Ch" a letter after "C". But it seemed ludicrous to me—two letters that counted as one. "*How pointlessly complicated*!" I thought. "*Why not just keep it simple, A to Z, like normal? Like English.*"

I later learned that *every* language has its own particular idea of what "alphabetical order" means; the fact that English's conception of it seems so "normal" is partly because English doesn't use any accents and partly because of accidents of history.

But many other languages use

accented characters that have to be sorted with the 26 letters of the "normal" A-through-Z alphabet. And with other languages, some combinations of characters, like the "ch" in Spanish, count as letters on their own. But in almost every case, if you want to sort according to the conventions of a particular language, the default behavior of Perl's sort won't sort that way. This article is about how to get Perl to sort according to the conventions of whatever language you have in mind—even if it's English!

Default sort Versus "Normal" English Sorting

Let's say you want to sort a list of words (or phrases) in what you think of as normal English alphabetical order. So you try using normal sort:

```
@stuff = ("canine", "cantaloupe",
             "cant",
                                            # As
  in an underworld jargon
             "Canberra", "can't",
             "Cantonese", "cannery",
             "Cannery Row", "canonicity",
              "Cañon de Chellv"
                                             Τn
  north-eastern Arizona, also
                                               #
  spelled "Canyon de Chelly" and
                                               #
  "Cañon de Chelle"
             );
  @sorted = sort @stuff;
                                          # The
  sorting happens here
  print map "[$ ] ", @sorted;
That prints:
```

```
[Canberra] [Cannery Row] [Cantonese]
[Cañon de Chelly]
[can't] [canine] [cannery] [canonicity]
[cant] [cantaloupe]
```

Whoa. All the capitals are

sorting first. That's because sort's default behavior (what you get without a "sort criterion"

or without use locale, both of which we'll discuss later) is

ASCIIbetical sorting—where the

sorting is based on ASCII order. Since "C" comes before "c" in ASCII, all the "C" items in @stuff (like "Cantonese") get sorted before all the "c" items (like "cantaloupe").

So you happen to remember an idiom for

case-insensitive sorting, and you change the line that sets @sorted to:

```
@sorted = sort { lc($a) cmp lc($b) }
@stuff;
```

and then you rerun the code. It prints:

```
[can't] [Canberra] [canine] [cannery]
[Cannery Row]
[canonicity] [cant] [cantaloupe]
[Cantonese] [Cañon de Chelly]
```

Closer. Here's what we want:

```
[Canberra] [canine] [cannery] [Cannery
Row] [Cañon de Chelly]
[canonicity] [cant] [can't] [cantaloupe]
[Cantonese]
```

The phrases "can't" and "Cañon de Chelly" are out of place. "can't" is out of place because { $lc(\$a) \ cmp \ lc(\$b)$ } treats "can't" as a five character string that sorts before anything else in @stuff. Consider this code:

```
print ( "can't" cmp "canal" );
```

That prints -1, meaning that "can't" comes before "canal". This is because cmp is doing simple

ASCIIbetical comparison, and when it compares "can't" to "canal" it gets as far as comparing "can" to "cana". At that point it sees that the apostrophe character comes before a, because the apostrophe is ASCII 39, and "a" is ASCII 97.

Now, this is also why "Cañon de Chelly" is coming last: because "ñ" is a character after "n". For sake of argument, I'll assume that you are, like me, using Latin-1 as opposed to UTF8, so I can say that "ñ" is a single byte: byte 241, in particular. (If you're using MacPerl and therefore probably using MacASCII, it's a different code, but it's still one byte with a value over 127, so my point stands. If you don't know what encoding you're using, you're probably using Latin-1.)

So what you want is to sort this list according to your idea of English

alphabetical order—ignoring apostrophes, treating "ñ" and "n" as the same letter, and of course ignoring case.

What you need is a subroutine we can call like this:

```
@sorted = sort {
normalize($a) cmp normalize($b) } @stuff;
```

where your normalize subroutine lc's things, turns "ñ"s to "n"s, and removes apostrophes (in no particular order). That function could consist of:

```
sub normalize {
    my $in = $ [0];
    $in =~ tr/Ñn/Nn/;
    $in =~ tr/'/d; # d for delete
    return lc($in);
}
```

Paste that into our original code, run

it, and it'll display this (lined up vertically just for better perusal):

```
[Canberra]
[canine]
[cannery]
[Cannery Row]
[Cañon de Chelly]
[canonicity]
[can't]
[cant]
[cantaloupe]
[Cantonese]
```

And that's basically right. Now, the only peculiarity there is "cant" versus "can't". It so happens that when you feed both into your normalize subroutine, you get "cant". So when vour sort criterion compares them using { normalize(\$a) cmp normalize(\$b) $\left.\right\}$. it's performing "cant" cmp "ant", which returns 0, meaning that these two sort identically. But since your use of sort produces a list where "can't" either comes before or after "cant", having your sort criterion return a 0 means that you don't care which of the two items comes first in the output, which effectively means that you can't predict which will end up first. Personally, I don't want my sort criterion to ever be unpredictable, so I add something that kicks in to avoid returning 0 when comparing different strings:

In other words, when normalize(\$a) cmp normalize(\$b) evaluates to 0, the routine falls through to returning the value of \$a cmp \$b. That makes this a completely predictable sort criterion, since \$a cmp \$b never returns 0 for different strings.

However, if you wanted something smarter than just \$a cmp \$b, you could use some second subroutine, normalize2, that could be a bit more fine-grained than normalize. Maybe it would implement the idea that, in case of a tie in normalize, words with apostrophes (like "can't") should always come after words without them (like "cant"), or that "Chile" (the country) should always be after "chile" (the hot sauce), and so on. You'd call that second part of your

sort criterion as:

To be really thorough, you could add in a cmp at the end:

Incidentally, falling back on a second comparison as a sort of "tie-breaker" in a sort criterion is basically what people mean when they refer to "bi-level sorting."

We'll return to this idea later.

Locale-Based Sorting

The idea of being able to sort things according to the conventions of other languages is not a new one. The *perllocale* documentation bundled with Perl describes how to take advantage of locales built into many OSes. Ideally, you'd set the locale to the language that sorts the way you want to sort, and then your calls to sort or cmp do the right thing. So if I set my locale to fr_CA.ISO8859-1 (meaning "French Canadian, using Latin-1"), "étude" will sort (correctly) with the "e"'s, instead of after the "z"'s, which is how it'd be sorted ASCIIbetically.

But locales might not be available on all computers. As *perllocale* points out: "The available locales, the location in which they are kept, and the manner in which they are installed, vary from system to system. Some systems provide only a few, hard-wired locales, and do not allow more to be added; others allow you to add 'canned' locales provided by the system supplier; still others allow you or the system administrator to define and add arbitrary locales."

In other words, if you want to sort a list of French words according to French

sorting conventions, even if you can get a French locale to work on one system, and even if that locale's idea of French sort order is the same as *your* idea of French sort order, there's still no guarantee that your

locale-based sorting will be able to use the same locale on someone else's system.

Because of these basic problems with locales, I consider locale-based

sorting (even where available) to be fine for one-shot programs, but these portability problems make it unacceptable for use in code that I'd actually want to distribute.

So, in short, much of the code in this article basically duplicates the functionality of some of the

sorting you *might* be able to get from locales, but in a more portable and flexible way.

Spanish: Cana y Caña

Earlier, we saw how to treat "ñ" as just an alternate form for "n", which is appropriate for English. But suppose you actually wanted to sort a list of Spanish words according to Spanish sorting conventions. In that case, you want to treat "ñ" not as an alternate form for "n", but instead as a letter between "n" and "o". In that case, you'd develop a sort criterion, as above, based on a normalize subroutine, wherein you'd have to move the letters of the alphabet around like so:

In other words, you want to keep "a" thru "n" as they are, and then have "ñ"—and this means bumping "o" thru "z" down by one character code to make way for the "ñ". That "[" is there just because it's the character after "z" in ASCII.

That's one way to do it. However, I find it a bit confusing, since that way makes the Spanish alphabet look like a strange decoder-ring substitution cypher. What I prefer is this:

If I add or remove characters from the alphabet on that first line, all I have to remember to do is change the 1B there to reflect however many characters are in the alphabet of characters that I'm starting with. (Since I'm just going to end up feeding the output of this normalize subroutine to cmp, it doesn't matter whether I'm mapping the alphabet to the range a-[or to the range x01-x1B.)

Here's how you'd work this into your normalize subroutine:

Then you can test it with this:

```
@stuff = ("cana", "Cantata", "caña",
"cantina",
            "canoso", "cañonero", "capa");
@sorted = sort { normalize($a) cmp
normalize($b) or $a cmp $b } @stuff;
print map "[$ ] ", @sorted;
```

When run, this program returns:

```
[cana] [canoso] [Cantata] [cantina] [caña]
[cañonero] [capa]
```

which is right! But change @stuff to these:

```
@stuff = ("cana", "caña", "cánula",
"cantina",
"cantó", "canto", "cantor");
```

and you (re)discover a problem:

```
[cana] [cantina] [canto] [cantor] [cantó]
[caña] [cánula]
```

And that's quite wrong. Spanish, you see, uses

acute accents (like over the "o" in "cantó")—but "ó" isn't considered a separate letter from "o". This is the same problem you faced in the English data set from the start of the article, except that here it's not "n" and "ñ"

we want to treat as alternates, but "o" and "ó"—and, while we're at it, "á/a", "é/e", "í/i", "ú/u", and the somewhat rare "ü". So you change the normalize subroutine:

Run this code, and you get:

```
[cana] [cantina] [canto] [cantó] [cantor]
[cánula] [caña]
```

Which (ta-daa!) is The Right Thing.

Spanish: Chorizo, Chimichangas, Chicharrones, y Churros

We now have a sort criterion and an associated subroutine (normalize) that together implement Spanish sorting conventions as far as treatment of \tilde{n} , \tilde{N} , and the accented vowels. But recall from the start of this article that Spanish has a letter "ch" between "c" and "d".

So far we've been massaging all the data using character-to-character substitution (using the tr operator), so that cmp's

ASCIIbetical character-by-character comparison would do what we want. However, that all assumes that

sorting is about *single* characters. But since "ch" consists of two ASCII characters, it won't fit well into our plan of using normal cmp. And "ch" is not alone: Spanish has one other two-character letter, the double-ell "ll", as in "llama", "quesadilla", and so on. Now, you could break down and write a subroutine that basically does the same work as Perl's built-in cmp but considers character-clusters like "ch" that you want to treat as single elements. However, that would be *very* inefficient compared to the speed of Perl's builtin cmp. A more efficient way of doing it consists of simply turning the clusters into single characters, so that cmp can be made to work right on them. So if you simply turn all occurrences of "ch" to, say, "¢" (which is presumably not to be found in any of the items we're sorting), you can pretend that "chimichanga" is really "¢imi¢anga" and then you can treat "¢" as just another strange letter, like "ñ"

is. Similarly, you could turn "ll" to "£", say. This would look like:

And then you can test it with:

The output, which is correct is:

```
[encanto] [Enciclopedia de México]
[encogido] [enchilada] [enchufe] [endibia]
```

Your normalize subroutine now correctly implements Spanish-style

sorting.

Bi-Level Sorting to the Rescue

There is a problem with our approach so far—and it might not even be a real problem for you, depending on why you're sorting your data.

Earlier, I talked about what happens when a sorting subroutine returns the same value for a pair of items, like "cant" and "can't" for an English normalize, or "canto" and "cantó" for a Spanish normalize, or "Chile" and "chile" with either. So far we've been sort of cheating, with criteria like these:

```
@sorted = sort { normalize($a) cmp
normalize($b) or $a cmp $b } @stuff;
```

This worked because the last expression, \$a cmp \$b, just happens to have correctly resolved ties that arise with normalize(\$a) cmp normalize(\$b). That was just dumb luck. And if, like many dictionaries, you want "Chile" to come after "chile," then plain old cmp as a tiebreaker does the wrong thing. So we need bi-level sorting with a normalize2 function as a better tiebreaker:

Let's implement a normalize2 subroutine that correctly breaks normalize ties. Let's continue with Spanish, and let's suppose that given a tie between variants of the letter "e", the order they should come out in is:

еЕ́е́É

Now, you could use the same sort of code as in normalize, this time implementing an alphabet consisting of:

a A á Á b B c C ch Ch CH d D e E é É ...

However, consider that normalize2 is just a tie-breaker—it doesn't need to distinguish "a" from "b". It would never be called in a case where an "a" in one position would need to be compared to a "b" in another, since that would *not* have resulted in a tie between normalized strings. In other words, all normalize2 needs to do is distinguish letters that normalize obliterated the difference between—letters in the same "family". In other words (grouping these letters into families), you need only map from these:

a A á Á b B c C ch Ch CH d D e E é É ...

onto these:

1 2 3 4 1 2 1 2 1 2 3 1 2 1 2 3 4 ...

And you can implement that this way:

```
sub normalize2 {
  my $in = $_[0];
  # Digraph things...
```

```
\sin = \sqrt{s/ch/c/q};
                            # chimichanga
=> ¢imi¢anga
  \sin = - s/Ch/*/q;
                            # Chimichanga
=> *imi¢anga
  \sin = \frac{s}{CH} / \frac{s}{q};
                            # CHIMICHANGA
=> *IMI*ANGA
  \sin = \frac{s}{ll/f/q};
                         # llama => £ama
  \sin = \frac{s/Ll}{s/q};
                         # Llama => §ama
  \sin = \frac{s}{LL}/\frac{q};
                         # LLAMA => ¶AMA
  # Now the big whammy...
                           $in
=~tr<aAbBcC¢**dDeEéÉfFqGhHiIíÍjJkKlL£§¶mMnNoOóÓpPqQrR
return $in;
}
```

To get a better feeling for the output of this function, consider:

normalize2("chile") is "1111"
normalize2("Chile") is "2111"
normalize2("CHILES RELLENOS!") is "32222
2232222!"
normalize2("cantó") is "11113"
normalize2("Canto") is "21111"
normalize2("CANTÓ") is "22224"

Consider what happens when

sorting "chile" and "Chile"; the sort criterion considers the expression:

This simplifies to:

```
"chile" cmp "chile"  # First
subexpression
  or "1111" cmp "2111"  # Second
subexpression
  or "chile" cmp "Chile"  # Last
subexpression
```

The first cmp subexpression evaluates to 0, falling through to the expression consisting of the two values from normalize2. Between them, "1111" (from "chile") comes first ASCIIbetically, so "1111" cmp "2111" returns -1, to signal that "chile" should come before "Chile". (Perl never gets as far as evaluating the last subexpression, "chile" cmp "Chile".)

English: Résumé and Resume

Now, this whole business of

bi-level sorting may all seem very abstract and, well, foreign, if the only thing you've ever sorted is English. But consider if you're

sorting this list of English words:

rot résumé resume rabble return

and you want it to sort correctly:

rabble resume résumé return rot

In other words, you want "resume" to always sort before "résumé". If you use a one-level sort like this:

```
@sorted = sort { normalize($a) cmp
normalize($b) } @stuff;
```

then you have a choice. Either treat "e" and "é" as the same letter (as with "ñ" and "n" in our Canberra/canine/cannery example), or treat "é" as a letter after "e". If you treat "e" and "é" as the same letter, then the ordering of "resume" and "résumé" would be unpredictable, since your normalize will return the same value for both.

But if you treat "é" as a letter after "e" (and that seems to be many people's first guess at a solution, I've found), that means that "é" will be a letter between "e" and "f", and all the "ré-" words will come after all the "re-" words—so that your list will sort as:

rabble resume return résumé rot

That's wrong. So if you want this to sort right, you need at least two levels in your

sorting. And since I've yet to see a case where *more* than two levels of

sorting were needed, that pretty much leaves you with bi-level sorting.

Like it or not, the only way to get really correct sorting in English is to use bi-level sorting. And this is not just a problem with English having

foreign words like "résumé"—the same problem arises with wanting to sort "Bath" and "bath", say.

Optimizing with Memoization

As Perl evaluates a sort criterion while sorting a list, it will ask that criterion to compare several of the items against each other. To see it at work, you can run:

```
@stuff = sort { print "$a & $b ; "; $a cmp
$b } qw(A B C D E F);
```

and you'll see something along the lines of:

```
A & D ; B & D ; C & D ; D & F ; D & E ; E
& F ; A & B ; B & C ;
```

If your criterion, like most of the ones in this article, will have to call normalize (and maybe normalize2) for whatever items they're asked to compare, then you can see that you're going to be calling normalize ("D") several times. There's no point in *re*-computing it, since normalize ("D") always gives the same answer, so all the calls after the first are just wasted effort. To make your sort criterion more efficient, you can cache the results of the function calls. Caching the results of a function like this is commonly called *memoization*.

In other words, instead of evaluating the expression normalize(\$a), you look to see if you computed it earlier and saved the result. otherwise, you compute the value and stow it in the cache for next time. So wherever you have:

```
function($INPUT)
```

you would use:

```
exists($cache{$INPUT}) ? $cache{$INPUT} :
($cache{$INPUT} = function($INPUT))
```

Worked into our bi-level sort criterion, this would look like:

```
{
 my(%cache, %cache2);
  @sorted = sort {
     ( exists($cache{$a}) ? $cache{$a}
          : (\cache{\sabel{a}} = normalize(\sabel{a}))
  cmp
     ( exists($cache{$b}) ? $cache{$b}
          : ($cache{$b} = normalize($b)) )
  or
     ( exists($cache2{$a}) ? $cache2{$a}
          : (\cache2{\alpha} = normalize2(\alpha))
  cmp
     ( exists($cache2{$b}) ? $cache2{$b}
          : ($cache2{$b} = normalize2($b)) )
  or
  $a cmp $b
     } @stuff;
}
```

It's not pretty, but it does avoid having to needlessly recompute normalize(ITEM) several times for each item being sorted. And the only thing better than correct sorting is faster correct sorting.

(Note: I've presented memoization only in the context of sorting. For more general applications, see Mark Jason Dominus's article on the topic in *Computer Science and Perl Programming: Best of the Perl Journal*, or the Memoize module in CPAN.)

Sorting it All Out

Now, I've heard that in the years since I took my last Spanish class, the Spanish Academy has decided to stop giving "ch" special treatment, so that "churro" will be, they decree, under "C", somewhere between "cesto" and "cicatriz". However, I don't know to what degree this has been accepted by the average Spanish speaker, much less the people who make the phone books and dictionaries in all the Spanish-speaking countries.

But even if everyone's idea of Spanish sorting conventions suddenly gets simpler (by doing away with those "ch" and "ll" digraphs), it'll still need bi-level sorting, just like English.

In fact, because implementing the bi-level sorting presented in this article is so common, I've written a module called

Sort::ArbBiLex that does it for you. The module allows you to specify a

sort order (possibly including multi-character letters like Spanish "ch") for which it builds a subroutine that sorts that way. The internals of Sort::ArbBiLex are frightening, but they're just an elaborate version of the techniques discussed in this article, adapted to the kinds of sorting found in most languages.

Here's some example code that defines then uses a sort order for Spanish:

```
use strict;
use Sort::ArbBiLex;
*sort_es = Sort::ArbBiLex::maker( #
defines &sort_es
```

```
"a A á Á
           \n b B
                         \n c C
\n ch Ch CH
               еЕ́е́Е́
                          \n
                              f F
 d D
           ∖n
\n q G
                ίΙίΙ
                              jJ
 h H
           \n
                          \n
\n
   kК
  1
                ll Ll LL
   T.
            ∖n
                          \n
                              тM
    n N
\n
           ∖n o O ó Ó
 ñÑ
                          \n
                              рP
\n qQ
                s S
                              t T
 r R
           \n
                          \n
\n uUúÚüÜ
v V
           \n
               wW
                          \n
                              хΧ
∖n yY
zΖ
...
);
my @stuff = ("cana", "caña", "cánula",
"cantina",
  "Canal", "cantó", "canto", "cantor");
print map "[$ ] ", sort es(@stuff);
```

This code prints:

```
[cana] [Canal] [cantina] [canto] [cantó]
[cantor] [cánula] [caña]
```

And there we have it: a simple sort that sorts Spanish text correctly.

Chapter 24. Simulating Typos with Perl

Sean M. Burke

Quoth the raven, "Nwvermpre!"

About two years ago, I switched to typing on the Dvorak keymap. That meant going from the Sholes "QWERTY" keymap:

1 2 3 4 5 6	7 8 9 0 - = \
qwert	yuiop[]
asdfg	hjkl; '
zxcvb	nm,./

to August

Dvorak's more efficiency-minded keymap:

` 1 2 3 4 5 6	7890[]\
', • ру	f g c r l / =
aoeui	dhtns-
; q j k x	b m w v z

It was just a matter of switching the keymap preferences on whatever computers I had to type on, and then a few days of acclimating to all the keys having moved. This had the two desired effects: my hands would no longer ache after marathon coding sessions, and no one ever touched my computer again. But there was one side effect I hadn't anticipated: a different keymap means different kinds of

typos. This became evident to me first on IRC. Since IRC is a medium characterized by people typing faster than they can think,

typos abound:

```
<Wuglife> I hear it's out on video now
me> I know, I sow it a wook age.
<Wuglife> sow?
<Koolmodey> wook age?
<Mugsv> GWAWRR! BEWARE THE AGE OF THE WOOK!
me> I mean I sAw it a wEEk agO.
<Koolmodey> guh, how do you manage to aim for
            'e' and hit 'o' instead? they're
on
            different sides of the keyboard
me> They're right next to eachother on mine.
    I use a Dvorak keyboard. The middle row
    goes: "aoeuidhtns".
<Koolmodev> that's because you're a communist
me> columnist
<Koolmodey> yea like dvorak
me> different Dvorak. August, not John.
<Mugsy> whatEVERRRR
<Wuglife> i like pie
```

Over time I did get the feeling that

typos on a Dvorak keyboard were really consistently different. At least for me, the typos I'd made on QWERTY keyboards were either transposition ("hte" for "the") or hitting a key adjacent to the intended one. On a Dvorak, transposition errors are more or less the same, but adjacent-key errors are, naturally, rather different—if you miss to the left or right of a QWERTY "e", you hit "w" or "r", but to the left or right of a Dvorak "e" is "o" and "u". So the equivalents of "fwlt" or "frlt" on a QWERTY become "folt" or "fult" on a Dvorak.

I had the feeling that Dvorak typos were, on the whole, much less likely to look like typos, compared to QWERTY typos. Whereas "fwlt" and "frlt" couldn't possibly be words, "folt" and "fult" look like plausible words that happen not to exist. And sometimes the typo does make for an existing word—one off from "seen" is "soon", one off from "be" is "me", and so on. This isn't something completely exclusive to a Dvorak—on a QWERTY, "fear" and "dear" are just one key off—but I had a feeling it was happening much more frequently with the Dvorak.

Now, looking at the keymap, it stands to reason—but then, lots of things stand to reason that don't actually happen (like, say, everyone abandoning the

QWERTY keymap, or having done so decades ago). So I decided that the best way to test this would be to write some sort of program to simulate typos on a Dvorak and on a QWERTY, have it generate lots and lots of typos, and see what the results would be.

First off, this might tell me whether I was just imagining things, or whether this was a measurable (and simulatable) property of typing on a Dvorak versus typing on a QWERTY. Moreover, the code developed could be of use in catching common typos—a capability important in spelling-correction algorithms, whether in actual spellcheckers or programs that, given a failed URL or email address, can suggest to the user perversely, alternative. More an one could use typo-simulating code to lend a hint of authenticity to a chatbot (see Chapters Chapter 12 and Chapter 13).

Simulating the Typos

For sake of simplicity, I figure I'd model the kind of typo I make most: trying to hit one key, but hitting a key either to the left or to the right. And since most of the keys I hit are letters, I decided to ignore typos on other keys, like hitting "%" instead of "\$", or even shift typos (for example, "THe" for "The").

The first thing any typo-simulating program needs to know is what keys are next to what. So I the first thing I wrote was a data table for the keys, @rows, and then a bit of code to expand that into two hashes, %Left and %Right:

```
use strict;
my @rows;
if (1) { # Change to 0 to get
OWERTY.
      @rows = ( # Yes, I use a split
keyboard...
              ...
                  py fgcrl ",
             "
               aoeui dhtns ",
             " gjkx bmwvz " );
} else {
    @rows = ( " qwert yuiop ",
             " asdfg hjkl
             " zxcvb nm " );
}
# To simulate an un-split keyboard:
# for (@rows) { substr($ , 6, 2) = '' }
my (%Left, %Right);
 # So $Left{$x} is what letter, if any,
 # to the left of the letter x.
```

```
foreach my $r (@rows) {
    for (my $i = 1; $i < length $r; ++$i) {
        my $x = substr($r,$i,1);
        next unless $x =~ m/[a-z]/;
        $Left{$x} = substr($r,$i - 1,1)
unless substr($r,$i - 1,1) eq ' ';
        $Right{$x} = substr($r,$i + 1,1)
unless substr($r,$i + 1,1) eq ' ';
      }
}
# And add the uppercase letters:
%Left = (%Left, map uc($_), %Left);
%Right = (%Right, map uc($_), %Right);</pre>
```

Then, after some tinkering, I came up with a function that, given a word, would try to think of some way to make a typo in it:

```
sub
typo on word {
    my \ $word = $ [0];
    my $typo word;
    my $tries = 0;
Make typo:
 {
   if (++$tries > 4) {
       # after too many do-overs, give up
       $typo word = $word;
       last Make typo;
   }
   my @strokes = stroke groups($word);
   my $where = int rand @strokes;
   my $char = substr($strokes[$where],0,1);
   my \$instead = (rand(1) < .5)
      ? ($Left{$char} || $Right{$char} ||
redo)
```

```
: ($Right{$char} || $Left{$char} ||
redo);
   $strokes[$where] = $instead
                                  x length
$strokes[$where];
     # So 'e' => 'r' or 'w', 'ee' => 'rr'
or 'ww'
    $typo word = join '', @strokes;
    redo Make typo unless
rep pattern($word)
     eq rep pattern($typo word);
     # That's so that we don't create any
stroke
    # groups that weren't there before, as
in
    # turning "soar" into "soor", which is
а
     # kind of mistake that I rarely if
ever make.
  }
 return $typo word;
}
sub
stroke groups {
      'eat' => qw(e a t)
    #
    # 'eel' => qw(ee l)
    # 'fool' => qw(f oo l)
   my @out;
   while (\$ [0] = \ m < (.) (\1^*) > g) {
        push @out, $1 . $2;
    }
    return @out;
```

Now, there's a lot going on here, so I'll break it down: every word is seen as an array of stroke groups—where each stroke group is a character plus any immediately following repetitions of itself. So "cat" is three stroke groups, "c", "a", and "t"; and "food" is three: "f", "oo", and "d".

Modeling things based on stroke groups captures the fact that if I miss the first "o" in "food", I'm also going to miss the following "o" the same way. And it also captures the fact that I wouldn't make a typo that would create a new stroke group—while I could mistype "pen" as "pes", I would *not* mistype "pens" as "pess" or "penn". So if the typo-generating code tried doing exactly that, turning "pens" into "pess", then rep_pattern (\$word) eq rep_pattern (\$typo_word) would be false (rep_pattern of "pens" is "1.1.1.1" but rep_pattern of "pess" is "1.1.2"), and the redo would start the block over. (Yes, you can have redo s and last s in non-loop blocks!)

So if we use the above subroutines and then try:

```
for (1..15) {
    print
typo_on_word("nevermore"), " ";
}
```

you'll get output like this with the Dvorak keymap:

nevecmore	nevelmore	nuvermore	nevermoro
severmore			
neverbore	nevermole	nevurmore	nevurmore
novermore			
nevecmore	nevermare	nevermoru	nevormore
nevermare			

With a

QWERTY keymap you'll get this:

mevermore	nevermote	nwvermore	nevermorw
nevernore			
nevermpre	nevermorw	nebermore	nevwrmore
nevwrmore			
nrvermore	nwvermore	mevermore	nevermorw
nevermire			

Now, these look to me like plausible

typos of the sort I've made on Dvoraks and QWERTYs. This is not to say that every possible typo I'd make would be generated by the above typo_on_word function. For example, typo_on_word doesn't attempt to simulate transposition, as in "hten" for "then". Moreover, it fails to account for the fact that I now and then make

typos like "moro" for "mere"—where, in effect, "e-e" functions as a sort of stroke group, because the left hand never leaves its key, regardless of the fact that the right hand is meanwhile off hitting the "r".

But there are diminishing returns to this. I think that if I wrote a function that modeled *every* kind of typo I make, with the appropriate frequency, it alone would be longer than this article, but wouldn't be vastly more realistic than what I hacked together. The exhaustive and exhausting detail of Dvorak's book

Typewriting Behavior certainly convinced me of the fact that errors are not simple things. However, typo_on_word does simulate most of the sorts of typos I do make, on each kind of keyboard.

And notice that most of the simulated Dvorak typos for "nevermore" look more or less like plausible (if not actually existing) English words to me, whereas most of the QWERTY typos contain character sequences that no English word could contain, like "nwv", "vwrm", and so on.
How to Identify Words

Being able to say that the string "tevermore" *could* be an existing word but "nevwrmore" couldn't be (and maybe that "nevecmore" and "nevermoru" *might* be) is something we can do intuitively based on some pretty complex implicit knowledge about how letters (and, at another level, sounds) can co-occur in English. Expressing that knowledge and then teaching it to a computer would be pretty difficult.

However, it's possible to teach the computer to acquire, on its own, a simple model of letter co-occurrence.

Consider the word "nevermore" as a sequence of overlapping three-character sequences, including, for good measure, enclosing brackets, to stand for the word boundaries:

```
[nevermore]
[ne
nev
eve
ver
erm
rmo
mor
ore
re]
```

If we scan a large amount of existing and presumably typo-free text (a corpus), and look at all such three-character clusters (

trigraphs), then we'll be able to scrutinize the simulated typo "nevwrmore", and we'll see that it consists of never-before-seen clusters like "evw", "vwr", and "wrm". Then we can note that it's got three things wrong with it, which makes it rather implausible as a word.

First, to build the frequency table:

```
my $text = '
babbitt.txt';
open(TEXT, "<$text") or die "Can't
read-open $text: $!";
my
%Known clusters;
while (<TEXT>) {
   my @words = words in($);
    foreach my $w (@words) {
        w = 1c "[w]";
        for (my \$i = 0; \$i < length(\$w) -
2; ++$i) {
              ++$Known clusters{substr $w,
$i, 3};
        }
    }
}
close(TEXT);
sub words in {
              return " $ [0]" =~
m/\s([a-zA-Z]+[a-zA-Z']*)(?=[\s,.`?!;])/q;
               #
                   return
                             $ [0]
                                    =~
m/(s([a-zA-Z]+[a-zA-Z']*)(?=[(s,.])/g;
     # See perlfaq6 for more on matching
words
}
```

This builds a hash, %Known_clusters, where the keys are all the three-letter clusters in all the words in a file. The file I happen to be using is a 700K text file comprising Sinclair

Lewis's novel *Babbitt*, available from Project Gutenberg (http://gutenberg.net).

We can test whether a cluster occurred in the text by just testing exists

```
$Known_clusters{$cluster}—and that's the basis
of this routine that gives a measure of the "plausibility"
```

of a word, by simply figuring what proportion of the word's clusters occur in %Known_clusters:

```
my $Debug = 1; # set to 0 to make plaus
silent
sub plaus {
    die "don't feed plaus a null string!"
unless length $ [0]; # sanity checking
    my $w = 1c "[$ [0]]";
    my $plaus count = 0;
    my $cluster count = 0;
    print "$w: " if $Debug;
     for (my \$i = 0; \$i < length(\$w) - 2;
++$i) {
        # Loop over three-character
clusters
        ++$cluster count;
         if (exists $Known clusters{substr
$w, $i, 3}) {
            ++$plaus count;
        } else {
                 print ' <',substr($w, $i,</pre>
3), '>?' if $Debug;
        }
    }
```

```
my $p = $plaus_count / $cluster_count;
printf " = %0.2f\n", $p if $Debug;
return $p;
}
```

We can test this by giving it two variations on "nevermore", and a few (typo-free) phrases chosen at random from my mail file, and then some random odd-looking words and names from a dictionary:

```
foreach my $w (qw(
  nevermore neverbore nwvwrmore
 potatoes cheese power and solidarity
   as metrics in language survey data
analysis
   assessing ethnolinguistic vitality it
seems to
  me that this homogenization of language
parallels
  what took place a couple hundred years
ago and
  is still going on
 Tokyo Xhosa Zanzibar yoghurt amphioxis
 Kleenex Yaqui quetzal
)) {
   plaus($w);
       Since we're in debug mode, just
      #
figuring
    # out plaus will print things.
}
exit;
```

This processes all the above words, noting three-letter clusters not found in the most frequent half of the clusters in Babbitt, and figuring the score (which is just the proportion of clusters that were known). All of the words get straight 1.0's (i.e., all clusters known), except for these:

```
[nwvwrmore]: <[nw>? <nwv>? <wvw>? <vwr>?
<wrm>? = 0.44
[tokyo]: <yo]>? = 0.80
[xhosa]: <[xh>? <xho>? = 0.60
[zanzibar]: <[za>? <anz>? <nzi>? <zib>? =
0.50
[yoghurt]: <ghu>? = 0.86
[amphioxis]: <iox>? = 0.89
[kleenex]: <[kl>? = 0.86
[yaqui]: <yaq>? <ui]>? = 0.60
[quetzal]: <etz>? <tza>? <zal>? = 0.57
```

So, for example, "neverbore" consists entirely of clusters seen in *Babbitt*. (The near-rarest cluster, incidentally, is "rbo", but that appears in "carbon", "Arbor", "Bourbon", and a few other words in the

Babbitt corpus.) But "nwvwrmore" gets a very low rating from plaus because it contains all sorts of

clusters that don't appear anywhere in Babbitt: "*word-start* n w", "n w v", and so on.

The words from "Tokyo" on are all marked as somewhat implausible; while they *are* all either English words or existing names usable in English sentences, plaus has no way to know that. But note that "nwvwrmore", with a

plausibility of 0.44, scores much lower than any of these. So plaus does a pretty good job of being able to tell gibberish from the "background radiation" of merely odd words and names.

Now, to test it on the "nevermore" typos we simulated in the previous section:

```
sub avg plaus {
    mv @words = @;
    return undef unless @words;
    my plaus sum = 0;
    foreach my $w (@words) {
        $plaus sum += plaus($w);
    return($plaus sum / @words);
}
print "
Dvorak 'nevermore' typo plaus: ",
   avg plaus(qw{ nevecmore nevelmore
nuvermore nevermoro severmore
                       neverbore nevermole
nevurmore nevurmore novermore
                       nevecmore nevermare
nevermoru nevormore nevermare }), "\n";
print "
QWERTY 'nevermore' typo plaus: ",
   avg plaus(qw{ mevermore nevermote
nwvermore nevermorw nevernore
                       nevermpre nevermorw
nebermore nevwrmore nevwrmore
                       nrvermore nwvermore
mevermore nevermorw nevermire }), "\n";
```

This returns:

Dvorak 'nevermore' typo plaus: 0.9555555555556 QWERTY 'nevermore' typo plaus: 0.851851851852

So plaus 's simple algorithm captures our observation that the simulated QWERTY typos on "nevermore" are more gibberish-like than the simulated Dvorak typos.^[7] But that's just one word—a real test of this would be to simulate

typos in a real text. We can deal with any amount of text (either in files named on the command line, or piped via STDIN), try to make a typo in every word, and then report the average plausibility (via plaus) of the typo-ridden words in the text:

```
my (@typo words);
while (<>) {
   foreach my $w (words in($ )) {
       push @typo words, typo on word($w);
    }
}
print "Typo
                plaus:
avg plaus(@typo words), "\n";
print
          "Input words:
                                  ۳,
scalar(@typo words), "\n";
         "Typo plaus:
                              ۳,
print
avg plaus(@typo words), "\n";
                                  ۳,
print
          "Input
                     words:
scalar(@typo words), "\n";
print "Start of typo text: ",
    join(' ', (@typo words > 100) ?
@typo words[0 .. 100] : @typo words), "\n";
```

When we feed text through this program, we get (after some minutes of frenzied calculation) a report of the average plaus rating for the simulated

typos in the text. We also get to see the beginning of the typo-filled text.

Typo-free Babbitt starts out: The towers of Zenith aspired above the morning mist; austere towers of steel and cement and limestone, sturdy as cliffs and delicate as silver rods.

But the above program, simulating typos on a split Dvorak keymap, gives us:

> Thu nowers af Venith aspured abowe tho mornisg bist; austece tomers og sheel anh cument ond liwestone, sturhy an criffs anh dericate an nilver rodn.

And for a split QWERTY, we get:

Rhe rowers pf Zenirh asoired sbove tje mirning nist; ausrere rowers pf steek amd cemenr amd limestonw, srurdy ad clidds anf delicare as dilver rids.

The average plaus of the whole of *Babbitt*, all 115,826 words of it, is about 0.87 for simulated Dvorak typos, but only 0.75 for simulated QWERTY typos.

There may be something a bit odd about using the same text to simulate typos as the %Known_clusters was built from, but it turns out that if we use the %Known_clusters from *Babbitt* but simulate typos on other texts (here, a 48,000-word

Project Gutenberg e-text of Charles Babbage's *Reflections on the Decline of Science in England, and on Some of its Causes*; and the first few paragraphs of William Gibson's *Neuromancer*), we find that the average plaus ratings are basically the same as for *Babbitt*! Table 24-1 at the end of this article shows the results.

Errors typed on a Dvorak, at least as modeled by my simulator, seem to be consistently more plausible (looking less like errors and more like real words) than errors on a QWERTY—at least for English text.

^[7] The only unknown clusters in the Dvorak nevermores were: nevecmore, *nuv*ermore, and nevermoru. However, in the QWERTY nevermores, there were *mev*ermore, *nwve*rmore, nevermore, nevermore, nevermore, nevermore, nevermore.

Typos in Other Languages

I was wondering, however, to what degree this might be specific to typing just in English. After all, both the Dvorak and QWERTY keymaps were designed with only English in mind, although both (with some degree of modification) are used for typing in any language that uses the Roman alphabet.

Now,

simulating typos in typing another language begs the question of exactly what keymap is used—languages with lots of accents have to add or alter the Dvorak or QWERTY keymaps to accommodate typing those accents. To keep things simple, I decided to try text in

Dutch, a language with few accents. (I do wonder how Polish typos would come out on a

QWERTY and a

Dvorak, but I know of no Dvorak keymaps that support Polish accents.)

A quick trip over to the

European Parliament's web site (www.europarl.eu.int) got me about 22,000 Dutch words: the text of four days' worth of the *Dagelijks Presbericht*, the EP Daily Notebook. An example phrase, with Dvorak and QWERTY typos:

Dutch:	Maar	met	twee	amendementen	wordt
er bij	de Raad	nog	maals	op	
Dvorak:	Moor	mot	hwee	amendomenten	mordt
el mij	du raah	sog	maals	ap	
QWERTY:	Naar	net	rwee	amensementen	wirdt
wr bih	dr rssd	nog	naals	ip	

The results over the mini-corpus of Dutch were comparable to the English results: the average plaus on Dvorak was about 0.82, and on QWERTY it was about 0.72. So the average typo on each for Dutch was a bit less plausible than for English, although interestingly enough, the difference (about 0.10) remains the same.

But then, Dutch is a Germanic language like English, with similar restrictions on how many consonants you can pack into each syllable (relatively many when compared to most other languages). A typical Italian syllable, however, is just a consonant and a vowel, and possibly a consonant at the end. So, to see how Italian would work with Dvorak and QWERTY typos, I rebuilt %Known_clusters from the clusters in Dante's *Inferno*, and then simulated typos on the text. The text, with typos, starts out:

```
Italian:
         Nel mezzo del cammin di nostra
vita
Dvorak: Ner mevvo dol commin hi
                                    sostra
zita
         Nwl nezzo dek cammim si nostrs
OWERTY:
bita
Italian: mi ritrovai per una selva oscura
Dvorak: wi ritrozai pel uno sulva oscira
        ni rotrovai oer yna sekva oscurs
QWERTY:
Italian: che' la diritta via era smarrita.
Dvorak:
         cho' lo duritta vio ora nmarrita.
OWERTY:
         cje' ka dititta vua eta amarrita.
Italian: Ahi quanto a dir qual era e` cosa
dura
         Ahu quanta o hir jual eca o` casa
Dvorak:
hura
```

```
Shi quamto s fir wual wra w` cisa
OWERTY:
dira
Italian: esta selva selvaggia e aspra
                                       e
forte
Dvorak:
         esto selvo selvoggia u asyra
                                        0
ferte
         eata swlva sekvaggia w asprs w
OWERTY:
fprte
Italian: che nel pensier rinova la paura!
Dvorak:
        ghe ner pensuer rinovo ra paira!
OWERTY:
        xhe nek prnsier riniva ls psura!
```

Midway upon the road of our life I found myself within a dark wood, for the right way had been missed. Ah! how hard a thing it is to tell what this wild and rough and dense wood was, which in thought renews the fear!

—from the Norton translation, also available from Project Gutenberg.

Simulating Dvorak typos on *Inferno* (about 30,000 words) gives an average plaus of about 0.81, like

Dutch, and not far off from English's 0.88. But

QWERTY typos have a much lower plaus: 0.61. The plaus figures are the same with *Paradiso* (also about 30,000 words).

Just to see if I could throw a wrench into the works, I decided to try feeding through some texts in written

Tibetan (in Romanization). While spoken Tibetan is pretty normal as languages go, written Tibetan has (silent) consonants in patterns and quantities I'd never have thought possible. (See Beyer 1992 for a fascinating discussion of how the writing system got to be that way.) Luckily for my purposes, the

Asian Text Input Project (http://asianclassics.org) has megs and megs of ASCII text in Tibetan. I decided at random on an 833KB file called 'Phags Pa Rgya Cher Rol Pa Zhes Bya Ba Theg Pa Chen Po'i Mdox (The Exalted Sutra of the Greater Way entitled The Sutra of Cosmic Play, or Arya Lalitavistara Nama Mahayanasutra).



Figure 24-1. A line of Tibetan text

Here is a sample line from the Tibetan text (with the actual phrase shown in Figure 24-1), with simulated Dvorak-typo and QWERTY-typo versions:

Tibetan: gcig na, bcom ldan 'das mnyan yod na rgyal bu rgyal Dvorak: gcug no, bcow ldon 'dan bnyan yad no rgyar bi cgyal QWERTY: fcig ns, bcon lsan 'fas mnyam uod ns rfyal bi rfyal

You'd think that a language that admits "rgyal" as a syllable isn't too terribly choosy about syllable structure—since "gcig" is a word, you'd bet "gcug" and "fcig" are just as plausible as words.

But you'd be wrong.

Simulating typos on Tibetan text gives results not far from typos on the other languages' texts: the Tibetan text's average plaus for a split

Dvorak keymap is 0.80, a few points below the 0.82 for Italian, but well above the average plaus score of just 0.59 for QWERTY-typo'd Tibetan.

The principle at work seems to be that on a Dvorak, if you miss a vowel, you'll probably get another vowel, and similarly for consonants. Moreover, there's a decent likelihood you'll get a consonant of the same articulatory class: most of bottom-right on a Dvorak is letters whose typical values are sounds articulated with the lip ("bmwvz"—"z" being the odd man out), and most of the middle-right row are sounds articulated with the tongue-tip right behind the top front teeth ("dhtns"—"h" being the exception this time). Substituting one of these for another of the same class typically will give you a plausible word.

On a QWERTY keyboard, however, there is relatively little such phonetic patterning of the keys, and so being one key off will get you a letter with basically no relationship to the letter you were aiming for.

While I find typing on a Dvorak to make for less work (muscularly) than typing on a QWERTY, the typos will stick out less, apparently regardless of language. So using a Dvorak means that careful proofreading has to be even more careful—at least until someone writes a use strict pragma for Tibetan, Italian, Dutch, and maybe even English.

Results

The average plausibility of simulated typos, on different keymaps, for texts in various languages is shown in Table 24-1.

	Dvorak		QWERTY	
	Split	Unsplit	Split	Unsplit
Sinclair Lewis's Babbitt	.874	.864	.756	.749
Charles Babbage's <i>Reflections on the Decline of Science in England, and on Some of its Causes</i> (plaus based on Babbitt)		.865	.773	.757
First few paragraphs of William Gibson's <i>Neuromancer</i> (plaus based on Babbitt)	.885	.863	.770	.766
Dutch: <i>Dagelijks Presbericht</i> 2000-10-24	.836	.828	.724	.692
<i>Dagelijks Presbericht</i> 2000-10-23, 2000-10-25, and 2000-10-26 (plaus based on 2000-10-24)	.831	.821	.715	.686

Table 24-1. Plausibility of simulated typos

	Dvorak		QWERTY	
	Split	Unsplit	Split	Unsplit
Italian: Dante's Inferno	.821	.806	.616	.600
Dante's <i>Paradiso</i> (plaus based on <i>Inferno</i>)	.821	.804	.612	.604
Tibetan: 'Phags Pa Rgya Cher Rol Pa Zhes Bya Ba Theg Pa Chen Po'i Mdo [Sutra of Cosmic Play]	.804	.754	.585	.607

References

Beyer, Stephan V. *The Classical Tibetan Language*. State University of New York Press, Albany, 1992.

Dvorak, August, Nellie L. Merrick, William L. Dealey, and Gertrude Catherine Ford. 1936. *Typewriting Behavior*. American Book Company, New York City. [Out of print and rather hard to find.—SB]

Chapter 25. Correcting Typos with Perl

Dave Cross

Symbol::Approx::Sub is a

Perl module that allows us to call

subroutines even if we spell their names wrong. Using it can be as simple as adding this to your programs:

use Symbol::Approx::Sub;

Once we've done this, we never have to worry about spelling our

subroutine names correctly again. For example, this program prints This is the foo subroutine!, even though & foo was misspelled as & few.

```
use Symbol::Approx::Sub;
sub foo {
    print "This is the foo subroutine!\n";
}
&few;
```

Why Was It Written?

This is obviously a very dangerous thing to want, so what made me decide to write Symbol::Approx::Sub?

In July 2000 I attended the O'Reilly Perl Conference and took Mark Jason Dominus's "Tricks of the Wizards" tutorial. He explained a number of concepts that can take your Perl programs to a new level of complexity and elegance. The most important of these concepts are

typeglobs and the AUTOLOAD function. It was the first time that I'd really tried to understand either of these concepts and, thanks to Dominus's clear explanations, I began to understand their power.

One example that Dominus used in this class was a demonstration of how we can use AUTOLOAD to catch misspelled subroutine names and perhaps do something about it. He showed a slide containing code like this:

```
sub AUTOLOAD {
    my ($sub) = s/.*::(.*)/;
        # Work out what sub the user really
meant
    $sub = get_real_name_of_sub($sub);
    goto &$sub;
}
```

On the following slide, he went into some detail about what a really bad idea this would be and how it would make your code completely unmaintainable. But it was too late; I was already thinking about how I could write a "get the real name of the subroutine" function and put it into a module that could be used in any Perl program.

How Does It Work?

During the twelve-hour flight home from California to England I thrashed out the implementation details. Here are the four required stages:

- When the module is loaded, it needs to install an AUTOLOAD function in the package that called it.
- When AUTOLOAD is called (as the result of invoking a non-existent subroutine) it needs to get a list of all the subroutines in our calling package.
- The AUTOLOAD function needs to compare each of those subroutine names with what the user actually called, and choose the most likely candidate.
- It then invokes the chosen subroutine.

The key to the first two stages was the other main topic of Dominus's talk—*typeglobs*.

In every Perl package, there is something called a *stash* ("symbol table hash"

) that contains the package's variables and subroutines. The stash is like a normal hash, with the keys being the names of the variables, and the values being references to the typeglobs. A *typeglob* is a data structure containing references to all of the objects with the same name. We know that in a Perl program you can have a, a, a, a, and a, and they are all completely separate—but they all live in the same typeglob.

The first stage is achieved with a useful typeglob trick. We can assign values (which should be references) to the various slots of a typeglob. This has the effect of *aliasing* the typeglob's name to the referenced value. For example, if we execute the following line of code:

```
*a = \@array_with_a_really_long_name;
```

@a will become an alias to @array_with_a_really_long_name and any changes we make to to @a will actually happen to the other array.

Furthermore, we can do this with any typeglob object, not just arrays. In particular, we can do it with subroutines, which is what I needed for

Symbol::Approx::Sub. The two objects don't even have to be in the same package, as we can see from the following code:

```
package other;
sub foo { print "This is &other::foo\n" }
*main::bar = \&foo;
package main;
&bar;
```

In this example we create a

subroutine called foo in the package other. We then alias that subroutine to &main::bar. This means that within the main package, if we call bar we actually call &other::foo. (This is how the Exporter module works.)

When

Symbol::Approx::Sub is loaded, we alias our caller's AUTOLOAD function to the one in our module. We know what our AUTOLOAD needs to do, but how do we get a list of subroutines in the calling package?

Let's look at a simple typeglob example. The next piece of code declares three package variables and a subroutine. We then write a simple foreach loop to print out the contents of %main::stash. If we run this program we'll see the names of our package objects a, b, c, and d. (We'll also see the standard filehandles STDIN and STDOUT and other built-in

Perl variables like @INC and %ENV.)

```
use vars qw($a @b %c);
sub d { print "Hello, world!\n" };
foreach (keys %main::) {
    print "$_\n";
}
```

Having listed the

typeglobs, our next task is to work out which of them contain subroutines. For this, we can use the $*FOO\{$ *THING* $\}$ syntax. In the same way that scalar names always start with a \$ and array names always start with a @, typeglob names always start with a *. *FOO therefore refers to the typeglob called FOO (which will contain \$FOO, @FOO, &FOO, and &FOO). With the $*FOO\{$ *THING* $\}$ syntax, we can find out whether the typeglob FOO contains an object of type *THING*, where *THING* can be SCALAR, ARRAY, HASH, IO, FORMAT, CODE, or GLOB. The next piece of code uses this syntax to show which of the typeglobs in our current package contain a subroutine:

```
#!/usr/bin/perl -w
use strict;
use vars qw($a @b %c);
sub d { print "sub d" };
while (my ($name, $glob) = each %main::) {
    print "$name contains a sub\n" if
defined *$glob{CODE};
}
```

We now know enough to create an AUTOLOAD function that generates a list of the subroutines that exist in the package.

Inside the AUTOLOAD function, the name of the subroutine that the program attempted to invoke will be available in the \$AUTOLOAD variable. All we need to do is carry out some sort of fuzzy matching on the set of subroutine names and the

misspelled subroutine name to find the best match.

Unfortunately, this isn't as simple as it sounds. I didn't want to write my own fuzzy matching algorithm, so I decided to borrow someone else's.

Perl comes with a

Text::Soundex

module that converts any word to a single letter and three digits that collectively correspond to the pronunciation of the string. This is what I initially used for my fuzzy matching.

The module computes the Soundex value for the

misspelled subroutine, and then computes the Soundex values for each of the subroutines in the caller's package. If none match, it mimics Perl's standard "undefined subroutine called" error message. If one matches, it's assumed to be the right subroutine. But what if there are multiple matches? This can happen, since Soundex compression can map two similar-sounding subroutine names to the same Soundex value. I thought about this for a while before deciding that the only option would be to pick one at random. I really couldn't see any other reasonable approach.

The Sub::Approx Module

That's pretty much how the *original* version of the module worked. I called it Sub::Approx and released it to CPAN. People started to talk to me about the module, and one of the most common things they said was, "Really interesting idea, but you should do the fuzzy matching using Some::Other::Module."

So Version 0.05 of Sub::Approx included what I called "fuzzy configurability" (or "configurable fuzziness") and with the help of Leon

Brocard, we made the process of matching a subroutine more modular. We introduced the concept of a *matcher*, which is a subroutine called with two things: the name of a subroutine that we're trying to match, and the list of subroutines in the package. The matcher returns an array of the subroutine names that match the required name. We supplied a matcher for each of Text::Soundex,

Text::Metaphone, and

String::Approx. You can therefore now use Sub::Approx like this:

use Sub::Approx (matcher =>
'text_metaphone');

This makes matching be carried out with Text::Metaphone instead of Text::Soundex.

To make it even more flexible, we allowed programmers to define their own matching subroutines; the subroutines are passed by reference into Sub::Approx. Here, we provide our own subroutine, named reverse:

```
use Sub::Approx (matcher => \&reverse);
sub reverse {
    my $sub = reverse shift;
    return grep { $_ eq $sub } @_;
}
sub abc {
    print "In sub abc!\n";
}
&cba;
```

If our subroutine doesn't exist, this matcher searches for a subroutine whose name is the reverse of the subroutine we tried to call.

One last feature was the ability to define a chooser function, which decides what to do if more than one subroutine matches. This function, when passed a list of matching

subroutine names, should return the name of the one it chooses. The default chooser still picks one at random, but you can define your own like this:

```
use Sub::Approx (chooser => \&first);
sub first {
    return shift;
}
```

This example isn't very bright—it'll always choose the first item in the list of matching subroutines.

The Symbol::Approx::Sub Module

This was how things remained until I gave a lightning talk on Sub::Approx at YAPC::Europe 2000. Afterward, a number of discussions took place that changed the shape of Sub::Approx, resulting in four changes:

- Perl RFC 324 was drafted, which suggested that in Perl 6, the AUTOLOAD function should be renamed to AUTOGLOB and invoked when *any* typeglob object that doesn't exist is called. This would allow us to create Scalar::Approx, Array::Approx, and so on.
- A mailing list was set up to discuss Sub::Approx and related matters. You can subscribe to the list at http://www.astray.com/mailman/listinfo/subapprox/.
- The typeglob walking code from Sub::Approx was abstracted out into a new module called GlobWalker so that it could be reused in Scalar::Approx and friends. (Later, I discovered that the Devel::Symdump module on CPAN did much the same thing and switched to that.)
- We realized that to produce Scalar::Approx and friends, we would be polluting a number of module namespaces. After some discussion on the modules and subapprox mailing lists, we decided on the name Symbol::Approx::Sub.

Symbol::Approx::Sub Version 1.60 is currently on CPAN.

Robin Houston has started work on a Symbol::Approx::Scalar module. Variables are trickier than subroutines for two reasons. First, there is currently no AUTOLOAD facility for variables the way there is for subroutines; Robin gets around this by tying the scalar variables. Second, most variables (at least in good programs) are *lexical variables*, rather than package variables, and therefore don't live in typeglobs. Robin (who knows more about Perl internals that I do) has written a

PadWalker module that does the same for lexical variables as GlobWalker (or Devel::Symdump) does for typeglobs.

Future Plans

On the mailing list, we are already planning Symbol::Approx::Sub Version 2.0. Planned features include:

• Separating the matcher component out into two separate stages:

canonization and matching. *Canonization* takes a subroutine name and returns some kind of canonical version, which might include removing underscores or converting all characters to lower case. This suggests having chained canonizers, each of which carries out one transformation in sequence.

- Developing a plugin architecture for canonizers, matchers, and choosers. This would make it easy for other people to produce their own modules that work with Symbol::Approx::Sub.
- Trying to accommodate calling packages that already define an AUTOLOAD function.

Even with all of this development, I have yet to find a real use for the module. As far as I can see, it's simply a very good demonstration of just how easy it is to do things in Perl that would be impossible in other languages. If you think you have an interesting use for Symbol::Approx::Sub, please let the mailing list know.

Afterword

Development of Symbol::Approx::Sub continues. Version 2.00 of the module was released during the Open Source Convention in July 2001. This version implements the plug-in architecture discussed in the article. When

Google released the API to their search engine in April 2002, Tatsuhiko

Miyagawa combined it with the Symbol::Approx::Sub plug-in architecture to create

Symbol::Approx::Sub::Google, which uses Google's spellcheck

feature to do the fuzzy matching.

In the summer of 2001 I gave a talk called "Perl for the People" at both the Open Source Convention and YAPC::Europe. In it I looked at some of the more extreme things that will be possible with Symbol::Approx::Sub. The slides for this talk are online at http://www.mag-sol.com/talks/ppl/.

And we're eagerly awaiting Larry Wall's Apocalypse 10, which will tell us whether or not RFD 324 has been accepted for implementation in Perl 6.

Chapter 26. Learning Japanese with an HTML Filter

Tuomas J. Lukka

I like to learn new languages by plunging into a good book. For Italian it was Pinocchio, for English *The Moon is a Harsh Mistress*. I keep a dictionary handy, and I spend many hours rechecking words until I remember them. It's tedious, but in the end it's still a more interesting way to learn than the usual route of beginning with very simple phrases and vocabulary and building up slowly, reading childish stories about uninteresting subjects.

I tried this with a book on Go strategy written in Japanese, and quickly hit a wall. With many languages you can infer the meaning of words by what they look like and remember them by how they sound. But in everyday Japanese text, there is no way for a beginner to know how a given phrase is pronounced until he learns the two thousand characters in common use. Furthermore, each character can usually be pronounced in two or more ways depending on the context (see the sidebar on Japanese Characters).

It might still be possible to learn Japanese with this method, but the task is complicated further still by the fact that the character dictionaries are tough to use—Japanese has 2,000 characters, so you have to find the words graphically, which is much more time-consuming. You can't leaf through the dictionary as you can with Western writing systems. So I ended up auditing the Japanese course at the university where I work. Even though the teacher made the course as much fun as a language course can be,

learning

kanji was difficult because of the feeling of not seeing them in real, interesting contexts.

The Web

Eventually I found an invaluable resource

for learning and using Japanese on the Web: ftp://ftp.monash.edu.au/pub/nihongo. This site has two freely available dictionaries that convert Japanese to English: *edict* and *kanjidic*. There are also instructions on how to view and edit Japanese on various operating systems.

Japanese Characters

There are four different character sets used for

Japanese:

hiragana,

katakana,

romaji, and

kanji. Hiragana and katakana both contain less than fifty characters and are purely phonetic writing systems. They can be used interchangeably, but usually hiragana is used for text and katakana is used for loanwords or special emphasis, like italics in English text. Romaji are simply the familiar letters you're reading right now. It is the last character set, kanji, that motivated this article. These characters, mostly borrowed from Chinese, relate to meanings, not sounds. There are over 6,000 kanji in all, but in 1946 the Japanese Ministry of Education settled on a list of 1945 characters for common use and 166 for names. Most kanji have at least two readings: *on* and *kun*. Which reading is used depends on the context, but usually the Japanese (kun) reading is used for single kanji and the Chinese (on) reading is used for compounds.

Japanese verbs and adjectives are usually written with kanji for the stem and hiragana for the ending. The format of kanji dictionary entries usually includes the readings in hiragana or katakana.

There were a few Japanese web pages about Go, and I'd visited them several times, each time hoping that my proficiency had improved enough to let me read them. Each time I found that I didn't know enough, and so I came up with an idea: Why not simply look up the characters automatically?

The simplest design I could think of was a CGI script that fetches the page and inserts the definitions of the kanji. Now I can browse any web page I like, and the kanji are automatically translated to English. Perl and CPAN made this nearly as simple as it sounds. I called the result wwwkan.pl, and it's shown at the end of this article. It automatically translates web pages in Japanese like Figure 26-1 into web pages like Figure 26-2.

Dictionary Database

The

dictionaries are fairly large and it would take too long to load and parse them again whenever the script is called. There are several solutions. You could have a dictionary server that sits in memory all the time, responding to queries as they arrive, or you could store the dictionary in a

hashed database. For simplicity I chose the latter. The script that converts the dictionary files into hash entries is shown in gendb.pl:

```
# gendb.pl - generate a database file from
the
# kanji dictionaries.
# Copyright (C) 1997,1998 Tuomas J.
Lukka.
# All rights reserved.
#
# Get the files "
kanjidic" and "
edict" from
# ftp://ftp.monash.edu.au/pub/nihongo
```



Figure 26-1. The subject listing of the Japanese Yahoo! web site

```
use AnyDBM_File;
use Fcntl;
$dir = ".";
$dir = $ARGV[0] if defined $ARGV[0];
# Interval to show that we're alive
$report = 4000;
tie %
kanji, AnyDBM_File, 'kanji.dbmx', O_CREAT
| O_RDWR | O_TRUNC, 0755;
open DIC, "$dir/
```

```
edict" or die "Can't open $dir/edict";
while (<DIC>) {
    next if /^{\#}/;
    /^(S+) \ or \ die("Invalid line '$ '");
    $kanji{$1} .= $ ;
    print("E: $nent '$1'\n") if ++$nent %
sreport == 0;
}
close DIC;
open DIC, "$dir/
kanjidic" or die "Can't open $dir/
kanjidic";
while (<DIC>) {
    next if /^#/;
    s/\s[UNBSMHQLKIOWYXEPCZ][\w-.]*//q; #
Leave G and F
   /^(S+) or die("Invalid line '$ '");
    $kanji{$1} .= $ ;
    print("K: $nent '$1'\n") if ++$nent %
report == 0;
}
close DIC;
untie %
kanji;
```
Help ile Edit View Go Window Back Forward Reload Home Search Guide Print Security 💕 Bookmarks & Netsile http://komodo.media.mit.edu/~tjl/ogi-bin/weekanl.ogi 芸術 <[[[芸術 [げいじゅつ]/art/the arts/]]]> と <[[[と /if (conjunction)/]]]> 人文 <[[[人文[じんぶん]/humanity/civilization/culture/人文[じんもん] /humanity/civilization/culture/]]]> - - 写真 </// 写真(しゃしん)/photograph/]]]>, 建築 <// [建築[けんちく]/construction (vs)/architecture/]]]>,美術館 <[[[美術館]びじゅつかん]/art galleny/art museum/]]]>と<[[[と/if (conjunction)/]]]>画廊<[[[画廊[がろう]/(picture) gallery/]]]>, 歴史<[[/歴 史[れきし]/history/]]]>,文字 <[[[文学[ぶんがく]/literature/]]]>,... ビジネス
 ビジネス /business/))> と
 ど (ff (conjunction)/))> 経済 済(けいざい)/economics/business/finance/economy/]])> [ニュース ([[ニュース/news/)]]>]--企業<///企業/きぎょう]/enterprise/undertaking/]]]>,雇用<///雇用/こよう]/employment (long term y]]]>, マーケット <[[[マーケット/market/]]]>と <[[[と/if(conjunctiony]]]>投資 <[[投資[とうし]/investment/]]]>, 求む <[[[求む[もとむ]/ito want/to seek/to request/]]]>, ... • コンピュータ <//> インターネット <[[[インターネット/the Internet/]]]> [ニュース <[[ニュース http://komodo.media.mit.edu/~tjl/cgi-bin/www.kan1.cgi?url=http%3A%2F%2Fwww.yahoo.co.jp%2FBusiness_and_Eco 100

Figure 26-2. Figure 26-1 viewed through www.kan.pl

The format of the *edict* dictionary is straightforward: first the kanji, then a space, and finally the definition. The loop to parse the file is:

```
open DIC, "$dir/edict" or die "Can't open
$dir/edict";
while (<DIC>) {
    next if /^#/;
    /^(\S+)\s/ or die("Invalid line '$_'");
    $kanji{$1} .= $_;
}
close DIC;
```

The second dictionary file, *kanjidic*, is slightly more complicated, as there are several fields on each line explaining different aspects of the kanji in question:

3027 U6328 N1910 B64 S10 I3c7.14 L2248 P1-3-7 Wae Yai1 Yai2 Q5303.4 MN12082 MP5.0229

The various numbers represent different ways of indexing the kanji, e.g., N1910 means that this kanji is number 1910 in Nelson's

Modern Reader's Japanese-English Character Dictionary and Wae means that the romanized Korean reading of this kanji is "ae". However interesting this information might be, it clutters up our web page, so let's remove most of it:

```
s/\s[UNBSMHQLKIOWYXEPCZ][\w-.]*//g;
```

In the parsing loop, %kanji isn't just any hash. It's a *tied* hash:

```
tie
%kanji, AnyDBM_File, 'kanji.dbmx', O_CREAT
| O_RDWR | O_TRUNC, 0755;
```

This ties %kanji to the file *kanji.dbmx* using AnyDBM_File, a handy module bundled with Perl that lets hashes stored on disk appear to be in memory. (*Editor's note:* Chapter 13 uses the same technique.)

Adding entries to the database is then as simple as saying:

\$kanji{\$1} .= \$_;

This stores the entry in the file. I use the .= operator instead of = because there can be multiple entries for different meanings of characters or character sequences. After we are done with it, we untie %kanji to break the connection between the hash and the disk file.

The CGI Script

wwwkan.pl uses two different libraries as its front and back ends:

libwww-perl (LWP, available on CPAN) is used to fetch the HTML document from the server and

CGI.pm (provided with the latest Perl distribution) to parse the request from the HTTP daemon and create the HTML to be returned.

The script begins with:

```
tie %kanji, AnyDBM_File, "$libdir/
kanji.dbmx", O RDONLY, 0;
```

which opens the kanji database created by the other script—the contents of %kanji are read back from the file when requested. Next we print the CGI header and a form for the new URL:

For more explanation of what is happening, see Lincoln Stein's documentation in *CGI.pm* or any of his columns in *Web, Graphics, and Perl/Tk: Best of the Perl Journal.*

After printing the form, the script retrieves the web page:

```
$url = $query->param('url');
$doc = get $url;
```

Now we have the

HTML code of the page that was specified in the url field of the form in \$doc.

The next task is to replace all the links to other HTML files with links through our engine:

See the

HTML::Parse documentation for further details. The anonymous subroutine ($sub \{ ... \}$) merely checks whether this tag has a URL field, using the hash that we initialized at the beginning of the program:

The anonymous subroutine in the call to \$h->traverse rewrites any URLs that appear on the page. URLs that are believed to contain text are rewritten with getlink so that my translation engine filters them. URLs that represent images are replaced with absolute links (i.e., prefaced with http://) by the abslink subroutine:

After modifying the tags in the parsed form, this line retrieves the modified HTML:

\$doc = \$h->as HTML;

Next, the climactic ending of the script:

```
print;
}
```

This converts the text into explained

kanji one line at a time. The regular expression matches one or more

Japanese characters: each is stored in two bytes with the highest bit in the first byte set. The /e modifier is used to replace them with the output of the explainstr subroutine, which converts a string of kanji into an English explanation:

```
sub explainstr {
    my $str = 0 ;
    my $res = "";
    my ($pos, $mlen, $s);
     for ( \$pos = 0; \$pos < length(\$str);
$pos += $mlen ) {
        my $expl;
        smlen = 20;
                 while (!defined($expl =
$kanji{$s=(substr(($str),$pos,$mlen))})
               and \$mlen > 2) {
            smlen -= 2;
        }
        $res .= $s;
        if (defined $expl) {
                                          ...
                               $res .=
<small><[[[".($expl)."]]]></small> ";
        }
    }
    return $res;
}
```

The inner loop is necessary because we wish to find the longest match available in the dictionary. (We want to

translate "word processor," not the shorter matches "word" and "processor.")

Taking It a Step Further

This design is good if you don't know any

Japanese, but once you've learned the basic characters (e.g., "one," "large," and so on), it's tedious to see their definitions over and over again. We need a way to categorize the difficulty of characters, and luckily, the Ministry of Education has done most of our work for us. They have divided kanji into grades for school. The kanjidic file contains the grade number of each kanji, so we can include an option that disables translation below a particular grade. This can be /G([0-9])/ in achieved with the regex the explainstr loop and checking \$1 to see whether we should explain this character.

Of course, different people have different interests. For example, I have learned several terms relating to Go but far fewer that relate to, say, art history. It would be nice to be able to provide a detailed list of what kanji I know. It is easy to envision CGI interfaces to a personalized database containing the kanji that you know, but let's KISS (Keep It Simple, Stupid) for now. The easiest solution is to write the numbers of the kanji I know into a file. As a bonus, I can use the same file to generate a selection of words from *kanjidic* and *edict* to use with the *kdrill* program to test myself on the kanji I should know.

Also, some

Japanese pages use an alternate encoding called

Shift-JIS. To handle both encodings without degrading performance, I adapted the code used by the *xjdic* program

(from the Monash archive) and made it into an XS module, available from my author directory in the CPAN.

Even though all these changes would be useful, they are fairly trivial so I won't show the code here. It's all on CPAN.

Conclusion

This tool has proven itself quite useful—I am able to keep up my study of

Japanese by reading interesting material. The effort that went into making these scripts was not large; only about 5 hours to get the most complicated (messy) version, and a few more to clean them up for TPJ.

There are several problems with this approach. The most serious is that images of characters cannot be translated—you have to resort to a traditional dictionary (I recommend *xjdic* from the Monash archive). Another problem is the fact that Japanese inflects verbs and has short particles all over the sentence, which is why the displayed text wwwkan.pl is sometimes odd-looking. A good rule of thumb is that all entries with one or two hiragana characters should be viewed with suspicion.

As a teaser, I might mention that my study of Japanese is related to my work on a Go-playing program, which I'm writing mostly in Perl (PDL for the speed-critical parts—see Chapter 11) but that's a story for another time.

wwwkan.pl

Here's wwwkan.pl, a program to translate kanji in Japanese HTML:

```
#!/usr/bin/perl
#
# www.kan1.pl - translate kanji or
compounds in Japanese HTML.
# Copyright (C) 1997, 1998 Tuomas J.
Lukka. All rights reserved.
# Directory to the kanji dictionary
database
$libdir = "/my/home/dir/japanese files/";
# The url of this CGI-script, for mangling
the links on the page
$my url = "http://komodo.media.mit.edu/
~tjl/cgi-bin/wwwkan1.cgi";
# Link types to substitute. 0 = absolute,
1 = relative.
%links = (a => ['href', 1], img => ['src',
01,
           form => ['action', 1], link =>
['href', 1],
          frame => ['src', 1]);
use CGI;
use LWP::Simple;
use HTML::Parse;
use URI::URL;
use Fcntl;
```

```
use AnyDBM File;
tie %
kanji, AnyDBM File, "$libdir/kanji.dbmx",
O RDONLY, 0;
$query = new CGI;
print $query->header, "CONVERTED By TJL's
kanji explainer on ",
      'date', '. Mail comments to
lukka@fas.harvard.edu.<P>',
       $query->startform(), "<b>Go To:</b>
۳,
      $guery->textfield(-name => 'url',
                             -default =>
'http://www.yahoo.co.jp/', -size => 50),
      $query->submit('Action','Doit'),
      $query->endform, "<HR>\n";
# Get the original document from the net.
$url = $query->param('url');
$doc = get $url;
# Substitute web addresses so that text
documents are fetched with
# this
script and pictures are fetched directly.
$h = parse
html($doc);
$h->traverse(
    sub {
        my($e, $start) = @ ;
        return 1 unless $start;
```

```
my $attr = $links{lc $e->tag} or
return 1;
         my \$url = \$e->attr(\$attr->[0]) or
return 1;
        $e->attr($attr->[0], ($attr->[1] ?
                           getlink($url) :
abslink($url)));
},
1);
doc = h->as HTML;
# Substitute kanji for English
for (split "\n", doc ) {
            s/((?:[x80-xFF][x40-xFF])+)/
explainstr($1)/ge;
   print;
}
exit;
# SUBROUTINES
# Make an absolute URL from a relative URL
in the original document
sub abslink {
                       return
                                   (new
URI::URL($ [0]))->abs($url)->as string;
}
# Make a new URL which gets a document
through our translation service.
sub getlink {
               my $url to = (new
URI::URL($ [0]))->abs($url);
    my $proxy url = new URI::URL($my url);
          $proxy url->query form(url =>
$url_to->as string);
    return $proxy url->as string;
}
```

```
# Insert explanations into a string of
kanji
sub explainstr {
    my $str = @ ;
    my $res = "";
    my ($pos, $mlen, $s);
    for ( $pos = 0; $pos < length($str);</pre>
$pos += $mlen ) {
       my $expl;
        smlen = 20;
                while (!defined($expl =
$kanji{$s=(substr(($str),$pos,$mlen))})
                 and \mbox{mlen} > 2) {
            smlen -= 2;
        }
        $res .= $s;
        if (defined $expl) {
                                         ...
                               $res .=
<small><[[[".($expl)."]]]></small> ";
       }
    }
   return $res;
}
```

Part IV. Games and Quizzes

In this part:
Chapter 27
Chapter 28
Chapter 29
Chapter 30
Chapter 31
Chapter 32
Chapter 34
Chapter 35
Chapter 36

In this section, ten articles explore games that you can play and manipulate from Perl, and quizzes to test (and increase) your Perl knowledge. The first article is my survey of the world of Perl games, from ready-to-run programs to utilities that help you build your own games. I follow up with an article on the Prisoner's Dilemma, one of the foundational puzzles of game theory.

Next, Michael Edmonson introduces his Rezrov interpreter for Infocom games, which allows you to control Infocom's text-based adventures from Perl.Greg Bacon then describes a graphical solitaire game implemented in Perl/Tk. The remainder of the section consists of six Perl quiz shows. First, the four quiz shows I emceed at the O'Reilly Perl and Open Source conferences, followed by one that I wrote for the Perl Whirl conference/cruise. And in case those are too easy for you, the section concludes with Tom Christiansen's Perl Wizard's Quiz.

Chapter 27. Games in Perl

Jon Orwant

This is a brief survey of the free Perl games and game utilities available on CPAN and elsewhere on the net. It's hard to find games in Perl, in part because the module-oriented philosophy of Perl makes people more likely to distribute utilities than the standalone applications that players want. And people distributing standalone apps don't typically advertise the language they wrote their program in, nor do they provide source code.

Most Perl games are simple affairs, either ASCII games or graphical games developed with Perl/Tk. (http://ptktools.sourceforge.net/ has a collection of Perl/Tk tools, including some card games and Tetris variants not listed here.)

If you're a budding game designer and would like to distribute your game as a standalone application on Windows so that players don't need to install Perl or Perl/Tk, I recommend the *Perl2Exe* utility at http://www.indigostar.com/perl2exe.htm.

Strategy Games

A list of Perl utilities for strategy games follows:

Chess

There are several chess resources available for Perl programmers. On CPAN, there's the

Games::Chess module for representing chess positions and games, and Games::Chess::Referee, which uses Games::Chess to track piece movements—the start of a system that could one day be used to create chess bots.

http://cgi.resourceindex.com/Programs_and_Scripts/Perl/ Games/Chess/ lists three

chess programs: *BeholderBoard*, which lets others play chess on your server, *KewlChess*, a two-player real-time chess game, and *CGI Chess*, a full-featured chess application for non-real-time games.

There's a mailing list for Perl developers to discuss chess programs: perl-chess-subscribe@yahoogroups.com.

Go

Now that computers can play chess better than any human, it's time for them to tackle Go. Two unpolished utilities on the CPAN might help:

Games::Go::GMP, which speaks the Go modem protocol, and Games::Go::SGF, which manipulates Smart Go Format files.

Concentration

A Perl CGI application for Concentration-style memory games: http://www.tiger-marmalade.com/~james/concentration/.

A.I. Wars

Games::AIBots is a Perl clone of the commercial game A.I. Wars, in which players build AI insects that compete with one another.

Kugel

A Perl/Tk game called *Kugel*, a strategy game in which you arrange objects into lines, can be found at http://www.vlptel.com/~vlatko/kugel/kugel.html.

Mathematical Smurfs Expedition

An educational game for ages 4–6: http://www.perltk.org/ ex/.

Alak

In *Computer Science and Perl Programming: Best of the Perl Journal*, Sean Burke discussed tree data structures, and showed how they were used to create computer players for certain types of games. He demonstrated using trees for a simple strategy game called Alak, and you can see the result of his efforts in the Games::Alak CPAN module.

NIM

Games::NIM is a basic CPAN module for playing NIM (in which players alternate removing pegs, trying to be the last to do so).

Life

You can find a Perl implementation of

Conway's game of Life at http://archive.develooper.com/ perl-ai@perl.org/msg00646.html.

You can find a four-line PDL version in Chapter 48.

Cards

Greg

Bacon describes his Perl/Tk solitaire game in Chapter 30, but if you're interested in utilities to create your own card games, I recommend the Games-Cards distribution on CPAN. The sample code from the documentation shows what the package lets you do:

```
use Games::Cards;
my $Rummy = new Games::Cards::Game;
# Create the correct deck for a game of
Rummy.
my $Deck = new Games::Cards::Deck ($Rummy,
"Deck");
# Shuffle the deck and create the discard
pile
$Deck->shuffle;
my $Discard = new
Games::Cards::Queue "Discard Pile";
# Deal out the hands
foreach my $i (1 .. 3) {
      my $hand = new Games::Cards::Hand
"Player $i" ;
    $Deck->give cards($hand, 7);
    $hand->sort by value;
    push @Hands, $hand;
}
# Print hands (e.g. "Player 1: AS 2C 3C
3H 10D OS KH")
```

```
foreach (@Hands) { print
($_->print("short"), "\n") }
$Hands[1]->give_a_card ($Discard, "8D"); #
Discard 8 of diamonds
```

The distribution also includes the

Games::Cards::Tk module, which allows you to create Perl/ Tk versions of card games.

Dice

There are two

dice packages on CPAN: Games::Dice and Games::Die. To be honest, I wouldn't recommend either—not because I have any opinions about the quality of the code, but because dice rolling is simple enough that the overhead of subroutine calls makes it preferable to just inline the code using rand. Just remember to seed the random number generator using srand, and remember that rand(6) returns a random floating-point number in the range from 0 to 6. To simulate a conventional die, you'd use this:

 $die_roll = int(rand(6)) + 1;$

In general, to generate a random integer between \times and y, use this:

\$random = int(rand(y - x + 1)) + x;

Word Games

A list of Perl utilities for word games follows:

Cryptograms

There are two Perl-based cryptogram solvers: the Games::Cryptoquote module on CPAN, and *cryptosolve.pl*, available at http://thayer.dartmouth.edu/ ~rjk/perlwords/.

MUDs

Various utilities for creating or accessing MUDs (Multi-User Dungeons) are at http://www.linux.org/apps/all/Entertainment/MUD.html.

Crossword puzzles

http://cgi.resourceindex.com/Programs_and_Scripts/Perl/ Games/Crossword_Puzzles/ contains two utilities for building your own web-based crossword puzzles.

Dissociated text

The Games::Dissociate module implements a Dissociated Press algorithm, which garbles text in amusing ways.

Word finds

Games::WordFind generates

Word finds from a list of words. A sample from the module's documentation:

		Ŵ¢	ord	ds	to	o I	Fir	nd	:		
									-		
CAMEL				GI	REA	ΑT					LINUX
LLAMA				Pł	ERI	-					
	W	Z	J	I	Q	Х	D	L	R	М	
	0	K	Ζ	С	S	Т	Ε	A	0	I	
	С	Ζ	V	Ζ	А	М	Ρ	Y	I	М	
	D	Y	Ρ	Ε	А	М	D	Ν	Ι	С	
	W	R	R	С	Т	Т	V	L	F	J	
	Α	G	W	Е	L	L	R	Е	Ρ	0	
	F	М	А	W	L	Ι	Ν	U	Х	Ι	
	А	S	А	Η	Q	R	Q	D	U	0	
	S	G	R	L	F	0	U	С	G	Ν	
	V	В	Y	В	L	Κ	U	F	W	Ι	

Googlewhacking

To *googlewhack* is to find a pair of words that has exactly one hit when searched for on Google. Some googlewhacks from http://www.googlewhack.com:

```
ambidextrous scallywags
illuminatus ombudsman
squirreling dervishes
assonant octosyllable
fetishized armadillo
panfish interrogation
```

Whether these also constitute good names for rock bands is a matter of interpretation.

Mad Libs

WebLibs is a web-based version of the "Mad Libs" party game, in which people provide adjectives, nouns, and verbs for insertion into a canned story, with humorous results (for some definition of "humor"). It's available at http://awsd.com/scripts/weblibs/index.shtml.

Hangman

http://cgi.resourceindex.com/Programs_and_Scripts/Perl/ Games/Hangman/ has two web-based

Hangman games, in which you try to guess the identity of a word with as few letters revealed as possible. The Games::GuessWord module, available on CPAN, also provides a simple text-based Hangman game.

Jumble

The

Games::Jumble module creates and solves Jumble word puzzles.

Quizzes

Games::QuizTaker is a CPAN module for creating and taking online quizzes.

Twitch Games

http://www.frozen-bubble.org/ is a Perl-based version of the arcade game sometimes named "Puzzle Bobble" and sometimes named "Bust-a-Move." It's pictured in Figure 27-1.



Figure 27-1. Frozen Bubble, a Perl/Tk game

http://sourceforge.net/projects/davidsperlgames/ has a few ASCII-style

arcade games: PerlBlaster, PerlRacer, and PerlArena.

Games::Worms is a Perl/Tk implementation of the classic Worms game, in which you're a worm that races around trying to eat food and avoid running into yourself. Not zoologically accurate, but fun anyway.

Also on CPAN is the Games::Quake::Log module, which allows you to access the information held in a Quake logfile.

One of the original twitch computer games was

SpaceWar, which ran on a venerable computer called the PDP-1. You can play it on a PDP1 assembler implemented in Perl and running under a Java emulator at http://lcs.www.media.mit.edu/groups/el/projects/spacewar/.

Chapter 28. The Prisoner's Dilemma

Jon Orwant

The police break down your door and hustle you downtown Seems you've for interrogation. been using illicit cryptography to exchange information about-well, they're not exactly sure, because you used cryptography, but they know it must be sinister because you used cryptography. And they know who you were talking to; their packet sniffers (and subpoenaed logs) revealed router that vou were communicating with your friend a few miles away. They've arrested him too.

You're going to jail. The question is, for how long?

The police don't have enough evidence to convict you of the new crime, Encoding In The First Degree, carrying a penalty of five years in jail. But they can convict you of Second Degree Encoding, and that's two long years in the overcrowded Minimum Security Federal Penitentiary for Computer Abusers.

They offer you a deal: if you agree to confess, and testify against your friend, they'll let you go free. They offer your friend the same deal. But if you both decide to testify against one another, you each get four years in jail. You must both decide what to do in solitude, without communicating. (That's why they split you up.) You can either Testify (T) or Hold Out (H), and your friend can do the same. The outcomes are shown in the payoff matrix depicted in Figure 28-1.



Figure 28-1. Payoff matrix for a Prisoner's Dilemma

What should you do? You might think to yourself:

If I testify, I'll get either four years or zero years. And if I hold out, I'll get either five years or two years. I have no idea what my friend will do, and I can't talk to him. Maybe I should assume that he'll choose at random, in which case I'm better off testifying.

If your friend thinks the same way, you'll both testify and get four years each. That's unfortunate, since the outcome with the fewest number of man-years in jail occurs when you both hold out.

This problem is called the

Prisoner's Dilemma, and it's the foundation for the mathematical discipline of *game theory*. It's been used to represent scenarios in gambling, genetics, business, and thermonuclear war, by using different payoffs: money, offspring, more money, and death.

The Iterated Prisoner's Dilemma

There's not a whole lot to say about the one-shot

Prisoner's Dilemma: either you should testify or you shouldn't, and you can construct compelling arguments for both decisions.

Here's when things get interesting: forget about the prison terms and think of the payoff matrix as abstract "points." Now pit you and your friend against one another in a series of matches, say 100. Your goal is to minimize the number of points you accrue during the 100 matches. This is the Iterated Prisoner's Dilemma.

Now there's a little bit of communication between you and your friend: for any given match, you can consider all of the previous matches before making your decision. If your friend held out on all the previous matches, you might think that he'll remain consistent, and hold out again. But maybe that's just what he wants you to think, so that he can testify, adding five points to your score and none to his. (Remember, he's a criminal too!)

Here's a simple always-hold-out strategy:

```
sub nice_guy {
   return "H";
}
```

Here's a strategy that chooses at random:

```
sub random {
  return "H" if rand() < 0.5;
  return "T";
}</pre>
```

Here's parting_shot, which holds out on the first 99 matches, and testifies on the last (100th) match. The history of your choices is stored in the array reference <code>\$my_choices_ref</code> (which becomes the array

<code>@my_choices</code>). parting_shot uses that array only to determine when the 100th match has been reached.

```
sub parting_shot {
  my ($my_ref, $friend_ref) = @_;
  my (@my_choices) = @$my_ref;
  if (@my_choices == 99) {
      return "T"
  } else { return "H" }
}
```

Here's a strategy called tit-for-tat, which holds out on the first match, and then chooses whatever the friend chose on the previous match:

Tit-for-tat variants usually perform well in

Prisoner's Dilemma contests. Random strategies aren't so bad either. Of course, that all depends on which other strategies participate. There's no single best strategy—just the best strategy for a given match.

The Three-Way Prisoner's Dilemma

Let's add another level of complexity: a three-person Prisoner's Dilemma, in which three strategies compete simultaneously. The payoff matrix (actually a payoff cube) is shown in Figure 28-2.

		Friend1, Friend2									
		T, T	T, H	H, T	H, T						
lou	т	You: 4 Friend1: 4 Friend2: 4	You: 1 Friend1: 1 Friend2: 7	You: 1 Friend1: 7 Friend2: 1	You: 0 Friend1: 5 Friend2: 5						
	H	You: 7 Friend1: 1 Friend2: 1	You: 5 Friend1: 0 Friend2: 5	You: 5 Friend1: 5 Friend2: 0	You: 2 Friend1: 2 Friend2: 2						

Figure 28-2. Payoff cube for a three-person Prisoner's Dilemma

When one person testifies and the other two hold out, the fink gets off scot-free, and the holdouts get five years each. When two people testify, they get one year each, and the holdout gets seven.

As before, the only communication between the prisoners is through their actions on previous matches. Here's a sample strategy for a three-way contest:

The Prisoner's Dilemma Programming Contest

We invite you to design your own three-way strategy, to be pitted against the rest of the Perl community.

Every strategy should be a single subroutine. During a match, the subroutine will be passed three array references (as with fool_me_twice above): the first will contain an array of your past decisions, and the second and third will contain the decision arrays for friend1 and friend2 respectively. Here are the rules:

- Your subroutine must always return either H or T.
- Your subroutine will play every other subroutine at least once, in a series of exactly 100 matches.
- The winning strategy will be the one with the lowest cumulative score over all matches.
- Your entry must have comments before the subroutine with your name, email address, and a truthful explanation of the strategy.
- The random number generator will be initialized before every series of 100 matches, should you care to use it.
- One entry per person.

Good luck!

Results of the Contest

Each strategy, encoded as a single Perl subroutine, was pitted against every other pair of strategies in a "duel" of exactly 100 matches. The distinction between a duel and a match is important, since in a series of matches each strategy can observe the others and decide how to act based on their past behavior.

A total of 31 strategies were submitted, yielding:

$$\binom{31}{2} = \frac{31!}{3!28!} = 4495$$

duels, or about 450,000 matches, in which each strategy returned a single letter: either an $\rm H$ to Hold out (or cooperate, in

game theory parlance) or a T to Testify (or defect).

This particular

contest was inspired by Profs. Mitch

Resnick and Mike

Eisenberg, who in 1987 conducted a three-way

Prisoner's Dilemma contest at MIT. I entered that contest (in which the strategies were Scheme functions instead of Perl subroutines), noticed that the four contest rules didn't prohibit functions that modified their inputs, and wrote a revisionist function that changed its opponents' histories, yielding the highest possible score by tampering with the past. I was disqualified, and a fifth rule excluding such strategies was added the next year.
I've been seething over this for a decade, and wanted some moral validation for what I thought was a clever insight. That's why I left the door wide open for similar strategies in this contest: there was nothing prohibiting subroutines from poking around the filesystem or the symbol table to locate, dissect, or modify other strategies, or even the judging program itself.

A few hackers, notably Felix Gallo and Randal Schwartz, spotted these possibilities, but no one actually submitted such an entry. (I don't know whether this was because of a lack of time or a surplus of scruples.) Felix also pointed out another devious tactic: collude with other contestants so that their strategies could recognize one another by their behavior. Then a previously-elected "master" could defect while the other "slaves" all cooperated, sacrificing themselves for the good of the master.

That's exactly what Ken

Albanowski, Earle

Ake, and Dan

Wisehart did: their three gestalt entries identify whenever Ken's strategy is playing one of the other two. If so, the slave holds out while the master testifies, resulting in a bad score for Earle or Dan but a good score for Ken. It worked: Ken's strategy finished first, and martyrs Earle and Dan finished 27th and 28th.

The top ten, along with the number of years their strategy spent in jail, are shown in Table 28-1.

Table 28-1. Top ten finalists in the Prisoner's Dilemma

Contestant	Years in jail	spent
Kenneth Albanowski (with sacrifices by Earle Ake and Dan Wisehart)	135,164	
Peter Frodin	147,341	
Eric Le Saux	147,624	
Francisco Azevedo	147,678	
Dave Pavlick and Beirne Konarski	148,527	
Bill England	149,053	
Peter Seibel	149,317	
Steve Frost	149,328	
Ravi Kolagotla	149,396	
Giovanni Marzot	149,412	

Peter

Frodin's second-place strategy is a less tolerant version of the fool_me_twice strategy explained earlier in this chapter: his fool_me_once testifies if either of the opponents testified during the previous match. (An honorable mention goes to Brian

Gough, whose subroutine is identical to Peter's, but didn't score in the top ten because of other strategies that behaved nondeterministically.)

Eric Le Saux's elephant placed third by noticing whether opposing strategies retaliate against testifying. Francisco Azevedo's strategy was nice but unforgiving: it holds out until someone testifies, and then testifies forever after. Dave Pavlich and Beirne

Konarski collaborated on the most complex strategy submitted: their subroutine contains five additional subroutines, implementing a temporary "probation" state for opponents. It testifies if both opponents are on probation and either one violated his probation by testifying last round.

Then the backstabbing began: the bottom half of the top ten, and ten out of the 31 strategies overall, simply testified each and every time. (The differences in score were due to the random behavior exhibited by other strategies.) Peter

Seibel used genetic programming to breed winning strategies, and found nothing that performed better than a pure testifier. Georg

Rehfeld's be_a_good_boy_all_the_time was the exact opposite: it cooperated regardless of the other strategies' actions. In his description, Georg said:

I think this is the only strategy to save the world: don't do any harm to anybody, be helpful and cooperative all the time. This holds true in real life, I believe...

His strategy finished dead last.

Chapter 29. The Rezrov Infocom Game Interpreter

Michael Edmonson

Interactive fiction, or

IF to its devotees, is a genre of text-based computer games. IF games present a virtual environment to the player, and respond to commands typed by the player. For example:

```
West of House
You are standing in an open field west of
a white house,
with a boarded front door.
There is a small mailbox here.
>open mailbox
Opening the small mailbox reveals a
leaflet.
```

IF traces its heritage to the early days of computer gaming. The first game was known simply as

Adventure, or

Colossal Cave, in which the player explored a mammoth underground cavern, collecting treasures and solving puzzles. Adventure caused a sensation when it arrived on the scene in the 1970's, but the game had a very limited vocabulary and could only accept simple commands. Inspired, a group of friends at MIT set about developing a new game with a vastly expanded vocabulary and the ability to understand complete sentences. This game became known as Zork, and its developers went on to form a company called Infocom.

Zork was originally developed on a DEC PDP-10 mainframe, a very expensive computer few people had access to. To bring the game to a wider audience, Zork's programmers wanted to find a way to make it run on the fledgling home computers of the time. Further complicating matters was the fact that home computers were much rarer than they are today, and there was a much wider variety of brands. To maximize the size of their market, Zork needed to run on as many of these computers as possible, and adapt to new machines as they emerged.

Infocom's solution was ingenious: they invented a design for something called the *Z*-machine, a

virtual computer. The idea was that the functions of this machine could be emulated on different computers with a program called a

ZIP, or Z-machine

Interpreter Program. The Z-machine's programs—that is, the games themselves—were written in a new, compact language called the Zork Implementation Language (ZIL or

Z-code for short). Because the emulated Z-machine behaved the same way on every platform, the games were 100% portable from computer to computer. The design was pared down to the point where it became possible to write ZIPs that ran on TRS-80s and Apple IIs, quite an accomplishment when you consider that the original version of Zork strained even the august PDP-10.

When a new computer was released, the programmers didn't need to touch any of the game programs; all they had to do was create a new ZIP for that platform, and it would be able to run all the games. Likewise, new games could be developed and released simultaneously for all platforms. These efficiencies allowed Infocom to nimbly adapt to the gyrations of the personal computer market in the 1980's; they released Z-machine interpreters for the TRS-80 series, Apple II and Macintosh, Atari 800/XL/XE and ST, Commodore 64 and Amiga, and the IBM PC, to name a few. This strategy also allowed all players to experience Infocom's games in exactly the same way, regardless of their computer. The Z-machine concept was astonishingly ahead of its time—consider that Java, whose "write once, run anywhere" virtual machine has received so much attention, wasn't developed until more than a decade later.

Technical merits aside, what ultimately determined Infocom's success was the quality of their games, whose immersive, sharply-written style and dry wit quickly earned them a loyal following. Zork I described itself as "a game of adventure, danger, and low cunning." Infocom's heyday was in the early to mid-1980's, when it released a string of solid titles, but the company stumbled with a failed venture into business software and was eventually subsumed by Activision.

Resurrecting the Z-Machine

The original Infocom is long gone, but the Z-machine is alive and well. A team of dedicated hackers known as the InfoTaskForce was largely responsible for

reverse-engineering the Z-machine's inner workings. Various open source Z-machine interpreters were developed and circulated. Eventually the Z-machine was understood completely enough that a formal specification for it was drawn up by Graham Nelson. Graham also developed

Inform, an

interactive fiction authoring system and compiler that allows new games to be written for the Z-machine architecture. Inform has a wide following, and the annual

IF contest brings in many new quality Z-code games every year. Perhaps the ultimate affirmation of Inform's success came with Activision's 1997 promotional release of

Zork: The Undiscovered Underground, a game co-authored by two of Infocom's original authors (Marc

Blank and

Michael

Berlyn), and compiled with Inform.

Enter the Camel

Z-code

interpreters exist for almost every platform imaginable. Two of the most widely-used interpreters, *zip* and *frotz*, are written in C and have been ported to everything from mainframes to Palm Pilots. One thing I've always thought was rather a shame about the available crop of interpreters is the way they've tended to fragment around different platforms' user-interface models. For example, *xzip* is a derivative of the character-based *zip*

interpreter, adapted to the graphical X Window System. Likewise, the character-based *frotz* interpreter begat a port to Windows called *winfrotz*. Z-code interpreters have also been written in Java; unfortunately, when it comes to cross-platform compatibility, Java's user-interface classes are finicky and unstable—just ask anyone who's tried to reason with the AWT on different JVMs.

I decided to write a Z-code interpreter in pure Perl, so the games could be played anywhere Perl could be used. Moreover, I wanted the program to be able to adapt itself to the variety of I/O models available on different systems, so you could play the games on anything from a basic character display to a fancy graphical window. I decided to name it rezrov after a magic spell from Infocom's classic game Enchanter. In the game, the spell means "open even locked or enchanted objects," and I figured Z-code program files qualified as both. I remember playing Enchanter and thinking I was really clever for using the spell to circumvent a puzzle involving a jeweled egg:

>take egg then examine it Taken

This ornamented egg is both beautiful and complex. The itself is mother-of-pearl, but eaa decorated with delicate gold traceries inlaid with jewels and other precious metals. On the surface are a lapis handle, an emerald knob, a silver slide. qolden crank, а and a diamond-studded button carefully and unobtrusively imbedded in the decorations. These various protuberances are likely to be connected with some machinerv inside.

The beautiful, ornamented egg is closed.

>learn rezrov then rezrov egg

Using your best study habits, you learn the rezrov spell.

The egg seems to come to life and each piece slides effortlessly in the correct pattern. The egg opens, revealing a shredded scroll inside, nestled among a profusion of shredders, knives, and other sharp instruments, cunningly connected to the knobs, buttons, etc. on the outside. Oops.

What's It Do?

rezrov is a Z-code

interpreter. An interpreter's job is to fetch the instructions, or *opcodes*, that form a program and execute them; that's exactly what Perl itself does under the hood. Interpreters spend most of their time in a loop:

- Retrieve the next opcode
- Perform the task specified by the opcode
- Repeat

Each opcode performs a single, basic logical operation. The Z-machine has

opcodes for reading from and writing to memory, manipulating variables, performing mathematical operations, reading input from the keyboard, displaying output on the screen, and changing the control flow of the program. Z-machine opcodes can be thought of as small subroutines or functions, which may accept arguments ("operands") and return results. It is the execution of many combinations of opcodes in series which form the programs, and thus the games. This basic process is echoed at many levels inside your computer, both in hardware and software, from your CPU on up through your operating system and applications.

So, rezrov emulates the workings of a virtual computer, and the Z-code games are programs that run in this computer. But where and what are the programs? For every computer, Infocom supplied an interpreter program and a data file: on MS-DOS systems, you would usually find an executable file for the interpreter (for example, *zork1.com*) and a large data file (*zork1.dat*). This data file, called the *story file*, is the actual Z-code program. It is completely portable between systems—you can extract the story file from an Apple II diskette and play it on a PC with a PC interpreter. Neat, huh?

On the home computers of yore, the games were usually too large to fit into memory all at once; for example, the Zork I story file is about 94K, while a typical PC had only 64K of RAM. To cope, Z-code interpreters swapped small chunks or "pages" of data from the diskette in and out of memory as they were needed by the game. This may have been the first use of virtual memory on a microcomputer—yet another revolutionary feature of the Z-machine design.

The Joy of vec

Most Z-code interpreters still use these paging schemes to minimize memory consumption. My feeling is that life is too short, especially since loading the entire game image into memory is such a breeze with Perl:

```
open(GAME, $filename) || die "can't open
$filename: $!\n";
binmode GAME;
my $size = -s $filename;
read(GAME, $story_bytes, $size);
```

This loads the entire story file

into a variable called *\$story_bytes*. The Z-machine's memory is basically a fixed-length array of 8-bit bytes; while 16-bit words are used for certain operations, virtually everything in memory is indexed by a byte offset. So, my first instinct was to simply convert the data into an array of bytes:

```
@story bytes = unpack("C*", $story bytes);
```

This was easy to do, but it came at a significant cost. Because each byte is converted to an individual scalar variable, much more memory is consumed. Consider the following program:

```
#!/usr/bin/perl -w
use strict;
use Benchmark;
my $filename = shift || die "specify
filename\n";
my $b1 = new Benchmark();
```

```
my $story bytes;
open(GAME, $filename) || die "can't open
$filename: $!\n";
binmode GAME;
my $size = -s $filename || die "no file";
read(GAME, $story bytes, $size);
if (@ARGV) {
    print "Creating array with unpack\n";
        my @story bytes = unpack "C*",
$story bytes;
} else {
  print "Not converting data to array\n";
}
my $td = timediff(new Benchmark(), $b1);
printf "%s\n", timestr($td);
print "Sleeping...\n";
sleep 10000;
```

This program reads a file

into memory, and if a second argument is provided, it converts that file into an array of bytes. Finally, it prints the time elapsed as recorded by the Benchmark module, and sleeps (to allow the user to check the size of the process in memory).

I ran this program on the 94 kilobyte *zork1.dat* on my PC running Linux. It completed almost instantaneously, and the Perl process consumed 1.9 megabytes of memory. When it converted the data into an array of bytes, the process consumed 6.5 megs of memory, and took almost a half-second to finish. This discrepancy became even more pronounced with the 256K *trinity.dat*: the first version consumed 2 megabytes of memory and still finished virtually

instantaneously, while the array of bytes consumed almost 14 megabytes of memory and took more than 1.3 seconds to complete. While Perl's typeless scalar variables are very convenient to use, the overhead they impose becomes apparent in large arrays like this.

Happily, there's a way out: the undersung vec function. vec lets you treat a variable as an array of unsigned integers where you get to specify the number of bits of storage to allocate for each entry. This is great for arrays of fixed-size data: 8-bit bytes, 16-bit words, etc. There are some restrictions: the elements of a vec array can only be unsigned integers, and the number of bits of storage must be a power of two from 1 to 32. But for my purposes, vec was perfect: I could grab all the data with a single read command, my program would start up quickly because I could avoid converting the data into a huge array, and I would save a lot of memory as well.

The vec function takes three arguments: the name of the variable to hold the data, the index in the array you want to reference, and the number of bits used for each entry in the array. Here are two object methods from rezrov that use vec to access single bytes in \$story bytes:

```
sub
get_byte_at {
    # get the 8-bit byte at the specified
storyfile offset.
    return vec($story_bytes, $_[1], 8);
}
sub
```

```
set_byte_at {
    # set the 8-bit byte at the specified
storyfile
    # offset to the given value.
    vec($story_bytes, $_[1], 8) = $_[2];
}
```

This initial excitement aside, much of the guts of rezrov's interpreter aren't particularly interesting. Most of the opcodes are fairly straightforward subroutines; the trickiest part was just making sure the bits came from and wound up in the right place. The basic nuts and bolts of

Z-code interpreters are considered a solved problem these days; Graham

Nelson wrote, "If your system isn't supported, adapting the C source code for one of the main interpreters is only a trial of patience, not of strength."

Tinkering with the Z-Machine

>read dusty book

The first page of the book was the table of contents. Only two chapter names can be read: The Legend of the Unseen Terror and The Legend of the Great Implementers.

>read legend of the implementers

This legend, written in an ancient tongue, speaks of the creation of the world. A more absurd account can hardly be imagined. The universe, it seems, was created by "Implementers" who directed the running of great engines. These engines produced this world and others, strange and wondrous, as a test or puzzle for others of their kind. It goes on to state that these beings stand ready to aid those entrapped within their creation. The great magicianphilosopher Helfax notes that a creation of this kind is morally and logically indefensible and discards the theory as "colossal claptrap and kludgery."

—--Enchanter, 1983

One of the most entertaining things about writing rezrov was adding features that putter around under the hood of the Z-machine while it's running. I found fertile ground for these experiments in the

Z-machine's *object table*. The object table is a block of memory containing information about every significant object in the game—every room, every item that can be picked up or otherwise interacted with, even the player: each is represented by an entry in the object table. Entries contain, among other things, a short description of the object, and pointers to a parent, a sibling, and a child object. Thus the object table describes a tree whose branches connect the items in the game. For example, take Zork I's

kitchen, which the game describes like so:

```
>look
```

```
Kitchen
You are in the kitchen of the white house.
A table seems
to have been used
                      recently for
                                      the
preparation of food. A
passage leads to the west and a dark
staircase can be
seen leading upward. A dark chimney leads
down and to the
east is a small window which is open.
A bottle is sitting on the table.
The glass bottle contains:
          A quantity of water There is a
brown sack here.
The brown sack contains:
       A lunch
        A clove of garlic
```

This scene is represented in the object table as shown in Figure 29-1.



Figure 29-1. A tree of Zork objects

Here the Kitchen room object (object #203), has a child, the kitchen table (#204). The table has a child, a brown sack (#224). The sack has a child, the lunch (#225), which itself has a sibling, the clove of garlic (#189). Likewise the sack has a sibling, the glass bottle (#236), which has a child, the quantity of water (#237).

The Z-machine has a set of opcodes that manipulate the entries in this table; the games execute these opcodes to make changes to their environment. I added a feature to rezrov, activated by the -snoop-obj command-line switch, which prints a message whenever an object is moved from one location to another. It prints out the name of the object being moved and the name of its new parent so you can watch what's happening. Using this feature you can see the name Infocom assigned to the "player" object in a number of their early games:

```
West of House
You are standing in an open field west of
a white house,
with a boarded front door.
There is a small mailbox here.
```

>north [Move "cretin" to "North of House"]
North of House
You are facing the north side of a white
house. There is
no door here, and all the windows are
boarded up. To the
north a narrow path winds through the
trees.

Later games changed this to the more diplomatic "yourself."

Once I had implemented all the opcodes necessary to manipulate the

object table, the temptation to make changes to it in mid-game proved irresistible. This led to the creation of a few new verbs, which rezrov intercepts without the game's knowledge, enabled by the -cheat command-line option: teleport, bamf, and pilfer, among others. I'll discuss each in turn.

Teleport

This command moves the player to any room in the game. Since the player and the locations in the game are all simply entries in the object table, teleporting the player is simply a matter of modifying the player's object so that it becomes the child of the room to be moved to. The code for the teleport subroutine is shown below:

```
sub teleport {  # Cheat command: move the
player to a new location
    my (\$self, \$where) = 0;
    mv $story = $self->story();
    unless ($where) {
        $story->write text("Please tell me
where you want to go.");
    } else {
                   my $object cache
                                        =
$self->get object cache();
                        my
                              @hits
                                       =
$object cache->find($where, "-room" => 1);
                     mv
                          @item hits
                                        =
$object cache->find($where);
if (@hits > 1) {
                                           #
Ambiguous destination
          $story->write text(sprintf 'Hmm,
where you mean: %s?',
                             nice list(sort
map {$ ->[1] } @hits));
        } elsif (@hits == 1) { # Only one
possible destination: proceed
        my $room id = $hits[0]->[0];
                            my $zo
                                       =
```

```
$object cache->get($room id);
if ($zo->is current room()) {
              # Destination object is the
current room: be rude
$story->write text($self->random message(
TELEPORT HERE MESSAGES));
        } else {
          # Teleport to the new room
$story->insert obj($story->player object(),
$room id);
          # Make the player object a child
of the new room object
$story->write text($self->random message(TELEPORT MES;
          # Print an appropriate message
          $story->push command("look");
            # Steal player's next turn to
describe new location
        }
      } elsif (@item hits == 1) {
            # User has specified an item
instead of a room; try to teleport
        # to the room the item is in
        my \sin a room = 0;
        my $item id = $item hits[0]->[0];
                              $zo =
                            mγ
$object cache->get($item id);
        my \ = 0;
        my $last;
        while (1) {
          \$last = \$zo;
          $zo = $zo->get parent();
          $levels++;
```

```
last unless defined $zo;
                                       if
($object cache->is room($zo->object id()))
{
             # Aha: this parent looks like
a room; go there.
            if (\$ levels == 1) {
                # Item is a top-level child
of the room
$story->write text($self->random message(TELEPORT TO )
            } else {
                 # Item is probably inside
something else visible in the room
              my $desc = $last->print();
              $story->write text(sprintf
                  "I think it's around here
somewhere; try the %s.", $$desc);
                    # Print description of
item's toplevel container
            }
$story->insert obj($story->player object(),
$zo->object id);
            $story->push command("look");
             # Move the player to the room
and steal turn to look around
            \sin a room = 1;
            last;
          }
        }
        unless ($in a room) {
            # Can't determine parent (many
objects are in limbo until
          # something happens)
                               $random
                           mv
                                        =
$object cache->get random("-room" => 1);
```

```
$story->write text(sprintf
               "I don't where that is; how
about the %s?", $$random);
      }
     } elsif (@item hits > 1) {
                                           #
Ambiguous item
          $story->write text(sprintf 'Hmm,
which do you mean: %s?',
                             nice list(sort
map {$ ->[1] } @item hits));
     } else {
                                           #
No clue at all
                        my $random =
$object cache->get random("-room" => 1);
       $story->write text(sprintf
         "I don't where that is; how about
the %s?", $$random);
     }
   }
}
```

The teleport subroutine uses several objects to do its dirty work:

- \$story is a reference to StoryFile.pm, a package that contains most of the Z-machine's data and opcodes, and manages communication with the user interface.
- \$zo is a reference to ZObject.pm; ZObject instances represent individual objects in the object table. ZObject's methods provide access to the object's data and allow the caller to navigate its parent/child relationships.
- \$object_cache is a reference to ZObjectCache,
 which manages a pool of ZObject references. It also tries

to guess which objects represent rooms and which represent items, providing a find method to search this information.

The subroutine takes as a parameter the name of the location the player wants to move to. Using ZObjectCache, it then looks to see

if the player has specified a unique location name. If this is so, the insert_obj method of StoryFile is called. insert_obj implements the opcode responsible for making one object the child of another object. It's called in the ordinary course of games to move the player from room to room: with each move, the player's object is unlinked from its old "parent" room object and made a "child" of the new room object. By doing this ourselves, we can teleport from one place to another.

But most of the code in this subroutine is for handling special cases:

- If you specify the name of an item in the game rather than a room, it tries to determine what room the object is in by walking up the object's parent/child hierarchy so it can take you there.
- If you specify an ambiguous location or item name, it will print a message listing the alternatives and ask which you mean.
- If it can't figure out where you mean to go, it will assume you are a tourist and suggest visiting a randomly chosen location in the game instead.

After teleport is called, rezrov temporarily disables the output of the game until the next prompt. This prevents the player from seeing the game's confused complaint that it didn't understand the teleport command: remember, even though rezrov understands what to do, the underlying game program has no idea what that means. rezrov also steals the player's next turn to submit a look command on the player's behalf. The player sees only the final result, which looks something like this:

>teleport kitchen

You are momentarily dizzy, and then... Kitchen You are in the kitchen of the white house. A table seems to have been used recently for the preparation of food. A passage leads to the west and a dark staircase can be seen leading upward. A dark chimney leads down and to the east is a small window which is slightly ajar. On the table is an elongated brown sack, smelling of hot peppers. A bottle is sitting on the table. The glass bottle contains: A quantity of water

This effectively modifies the behavior of the game without the player noticing that anything unusual has happened.

This technique is also used at the very beginning of the game to try to guess its title. rezrov steals the user's first turn to submit a version command to the game. version is a traditional command that prints out the name of the game and its revision information. rezrov extracts the title from the game's redirected output and uses it to set the title of the window. Usually version doesn't even cost the player an official turn, so no one's the wiser.

If the user interface can't support setting the window's title or the word version isn't in the game's dictionary, rezrov skips the attempt entirely.

Bamf

bamf is sort of the inverse of teleport: it unlinks an object from the object tree, effectively making it disappear from the game. This is convenient for moving monsters or other troublesome objects out of your way:

```
The Troll Room
This is a small room with passages to the
east and south
     а
       forbidding hole leading west.
and
Bloodstains and deep
scratches (perhaps made by an axe) mar the
walls.
A nasty-looking troll, brandishing
                                        а
bloody axe, blocks
all passages out of the room.
Your sword has begun to glow very brightly.
>wait
Time passes...
The axe sweeps past as you jump aside.
>bamf troll
The troll disappears with a pop.
>look
The Troll Room
This is a small room with passages to the
east and south
and a forbidding hole leading west.
Bloodstains and deep
scratches (perhaps made by an axe) mar the
walls.
Your sword is no longer glowing.
```

Pilfer

pilfer is a practical implementation of two of the cardinal rules of adventure gaming:

- Anything that is not nailed down is mine.
- Anything that I can pry loose is not nailed down.

pilfer moves the object you specify to your current location, by making the object a child of your current room's object (rezrov knows which object represents the current room by tracking the player's movements). It then steals a turn from the player to submit a take command to move the object into the player's inventory. In this fashion any "takeable" item can be moved from anywhere in the game into your hot little hands; any object that can't actually be picked up will simply remain in the player's current location.

Like teleport, pilfer uses the context information of the object table to respond appropriately.

If you pilfer something from another location, you may hear a distant rumble of thunder. Pilfering an item that's already in your inventory results in the sensation of invisible hands rummaging through your possessions. Pilfering things that are contained inside of other things produces special effects, and attempting to pilfer yourself results in ridicule.

Besides its obvious nefarious uses, the pilfer command raises the possibility of revealing

Easter eggs in old Infocom games. I remember a maddening puzzle from the game

Planetfall, involving a room that you could enter, but not see anything in. There was a lantern in the game, but it was located in a lab full of deadly radiation. You could enter the room and take the lamp, but would die of radiation poisoning in a few moves, just out of reach of where you needed it. In this way the player's natural curiosity was denied even if they sacrificed their life to get a peek. And as it turned out, you didn't need to see inside that room to finish the game. In fact, as pilfering the lamp and entering the room reveals, you were never meant to:

```
Transportation Supply
You have just located a serious bug.
```

Planetfall contains a number of these red herrings, and closes with a truly sadistic flourish: Your robotic companion Floyd rushes up, hands you several seemingly-critical items missing from the game, and tells you "maybe we can use these in the sequel."

Universal Command Set

Infocom made a number of revisions to the Z-machine and its interpreters, gradually adding new features that made the games more enjoyable to play. A number of these were quality-of-life concessions for weary typists, such as short aliases for certain frequently-used words; for example, x could be entered in place of examine. There was also the oops command, which allowed you to correct the spelling of a word you had mistyped on the previous line:

```
>give lmap to troll
I don't know the word "lmap".
>oops lamp
The troll, who is not overly proud,
graciously accepts the
gift and not having the most
discriminating tastes,
gleefully eats it.
You are left in the dark...
```

One of the most useful commands was undo, which rolled back the effects of your previous turn. This came in extremely handy, as it allowed you to recover from mistakes or foolish experiments even

if you hadn't saved your

game. Unfortunately, undo wasn't available in early Infocom games, so rezrov emulates the undo command by saving the game after every turn, and, in the manner of the cheating verbs described previously, intercepts the undo command and rolls back the game state. It also saves the data in an array of user-definable length, so you can undo multiple turns. By emulating undo, oops, and other convenient functions rezrov makes the Infocom command set even more portable than it was before.

Interface Abstraction

When it comes to writing

user

interfaces for their programs, Perl programmers have a wide range of options to choose from, from character-based toolkits such as the Curses module to full-blown graphical systems such as Perl/Tk. Unfortunately, most of these APIs are tied to the style and history of the operating systems they were developed for, and as a result tend to be not very portable. For example, the text-based Curses API, ubiquitous on Unix systems, is alien to most Windows machines.

Committing to one API meant handcuffing my program to whatever platforms supported it. And I didn't want to sacrifice exotic GUIs just to accommodate the most portable interface, which is ASCII text. My solution was to isolate the Z-machine's I/O operations from the main code. I created an object-oriented module, named

ZIO, that defined a set of methods allowing the Z-machine to function under multiple APIs. These methods define a set of basic tasks such as moving the cursor around the screen, drawing text, and reading

user input; many of them correspond directly to Z-machine opcodes. The goal was to have each separate ZIO implementation contain all of the user interface code, but none of the Z-machine's higher-level logic.

At this point I knew I had a flexible design that could work with many different user-interface APIs. But this still left the problem of figuring out which APIs were available on each Perl installation, and getting the main program off the ground. For example, if your code contains this statement, it won't compile unless your system has the Curses module installed:

```
use Curses;
```

Since I wanted to use modules that were sure to be missing on many platforms, I needed a way to detect which were available on the current system without stopping Perl in its tracks. My solution was to use eval, which interprets its argument as its own little program—but instead of quitting outright when a fatal error occurs, it traps the error message in the special variable \$@. You can take advantage of this feature to detect whether modules are installed. For example, the following code detects whether the local Perl installation has the Tk module available, and behaves differently depending on the result:

rezrov uses this technique to detect the best interface module available on the user's computer (the interface can also be specified manually when desired). After we know which API module we'll be using, it's safe to use require to dynamically load the ZIO package that depends on that module:

```
if
       ($zio type eq "tk")
{
                         # GUI interface
   require
Rezrov::ZIO Tk;
   $zio = new Rezrov::ZIO Tk(%FLAGS);
             ($zio type eq
                               "win32")
}
    elsif
                 # Windows console
{
   require Rezrov::ZIO Win32;
   $zio = new Rezrov::ZIO Win32(%FLAGS);
}
```

If none of the optional I/O modules are available, rezrov retreats to a minimalist interface that doesn't require any external code. In this fashion the program can optimally adapt itself to a variety of systems without needing to be configured by the user, while still retaining the ability to run on a bare-bones installation. The Tk version is shown in Figure 29-2.

Performance Considerations

Perl teaches the three programmer's virtues of Laziness, Impatience, and Hubris. Programmers are also occasionally motivated by Beauty, the desire to write programs that, in addition to merely doing their jobs, adhere to some higher aesthetic standard. In this case, I felt compelled for Beauty's sake to try to use objects wherever I could. This was motivated by the desire to keep the code as readable as possible; I figured that anybody who, like me, had tried to understand the source code of a C

interpreter would appreciate it. Using an object-oriented approach with the abstraction of I/O operations made perfect sense and turned out to be a big win. However, in the utilitarian core of the main interpreter, this proved more problematic than I had hoped. Ah, Hubris.


Figure 29-2. Tk one-upping Curses with a fixed-font-width popup window superimposed over a variable-font-width main window

Quantity Is Job One

As home computers grew more powerful, Infocom released new revisions of the Z-machine and more complex games designed to take advantage of this capacity. In a similar spirit, the post-Infocom Inform compiler and library provided an improved set of standard features for game authors to employ. One side-effect of these developments was a marked increase in the typical number of opcodes executed by games between the user's commands. For example, here are the number of opcodes required to process a single look command in a different number of games (you the can use -count-opcodes command-line switch to see this for yourself):

Zork I (1983, Z-machine revision 3)	652 opcodes
Trinity (1986, Z-machine revision 4)	1539 opcodes
Zork: The Undiscovered Underground (1997, Z-machine revision 5, Inform)	2186 opcodes

Though the amount of work done by any one opcode is typically small, the sheer volume of opcodes being executed places a lot of stress on the implementation. Small inefficiencies in frequently-performed operations can compound to the point where they exact a significant toll. Here are a few things I have learned in the course of writing lots of tiny, frequently-called object methods in Perl:

- Object methods are significantly slower than regular subroutine calls. When invoked as a method, each subroutine call requires an additional parameter, the instance variable. Inheritance can slow things down even more. Under the best of circumstances I've found method calls to be about 25% slower than ordinary subroutine calls.
- Instance data can be expensive, because it must be dereferenced before it can be used. Using blessed hashes, I've found accessing instance data to be about 25% slower than static data. This is a little faster with blessed arrays, but then the notion of "keys" becomes much more obtuse.
- Declaring lexically-scoped variables

 methods can cost you. Typical methods declare a
 variable for the object instance (often \$self) and
 others for any parameters that may be passed to the
 method. I've found that using the @_ array to directly
 access the subroutine's parameters is about 15–25%
 faster than declaring variables that copy @_. It would be
 nice

if there was a way to reference entries in @__ within the scope of a subroutine as if they were variables without the overhead of variable creation and destruction. Pseudovariables, anyone?

The program below demonstrates the costs of method calls and instance data:

```
#!/usr/bin/perl -w
#
#
Performance comparison for lots of calls
to a small subroutine:
#
      00 of varying degrees, static
functions, and inlined calls.
#
use strict;
use Benchmark:
$Bogus::DATA = "";
my $COUNT = shift || 100000;
printf
         "timing %d iterations;
initializing...\n", $COUNT;
for (my $i=0; $i < $COUNT; $i++) {</pre>
    vec(\$Bogus::DATA, \$i, 8) = 42;
}
package Bogus;
sub new {
    my $self = bless {}, shift;
    $self->{"data ref"} = \$Bogus::DATA;
    return $self;
}
sub data ref {
    return $ [0]->{"data ref"};
}
sub get byte superverbose instance data {
    my (\$self, \$where) = 0;
        return vec(${$self->data ref()},
$where, 8);
```

```
}
sub get byte verbose instance data {
   my (\$self, \$where) = 0;
       return vec(${$self->{"data ref"}},
$where, 8);
}
sub get byte terse instance data {
       return vec(${$ [0]->{"data ref"}},
$ [1], 8);
}
sub get byte terse static {
    return vec($Bogus::DATA, $ [1], 8);
}
sub get byte verbose static {
   my ($self, $where) = 0;
    return vec($Bogus::DATA, $where, 8);
}
package main;
sub get_byte_main_terse_static {
    return vec($Bogus::DATA, $ [0], 8);
}
sub get byte main verbose static {
   my (\$where) = @;
   return vec($Bogus::DATA, $where, 8);
}
$main::bogus = $main::bogus = new
Bogus(); # Shut up, -w
timethis($COUNT,
                       '$x
$main::bogus->get byte superverbose instance data(0)'
```

```
"superverbose instance 00");
                        '$x
timethis($COUNT,
$main::bogus->get byte verbose instance data(0)',
         "verbose instance 00");
timethis($COUNT, '$x = $main::bogus->
get byte terse instance data(0)',
         "terse instance 00");
timethis($COUNT,
                        'Śx
$main::bogus->get byte verbose static(0)',
         "verbose static OO");
                        '$x
timethis($COUNT,
$main::bogus->get byte terse static(0)',
         "terse static OO");
timethis($COUNT,
                        '$x
main::get byte main verbose static(0)',
         "verbose static call");
timethis ($COUNT,
                        'Śx
main::get byte main terse static(0)',
         "terse static call");
timethis($COUNT, '$x = vec($Bogus::DATA,
0, 8)', "fully inlined");
```

On my computer, it generated the results shown in Table 29-1.

Table 29-1. Performance comparison of function call techniques

Technique	Time (seconds)
Superverbose instance OO	9.47
Verbose instance OO	6.30

Technique	Time (seconds)
Terse instance OO	5.41
Verbose static OO	5.27
Terse static OO	3.81
Verbose static call	3.65
Terse static call	2.72
Fully inlined	0.72

A Plea for Inlining

The results highlight an issue I am still wrestling with, the notion of

inlining frequently-used code to maximize performance. Inlining is an

optimization technique whereby calls to a subroutine are replaced by the body of that subroutine. Since the Z-machine accesses its memory in nearly every operation, it's desirable to have the most efficient implementation possible. It's no surprise that the

ZIP

interpreter's C code used preprocessor macros to inline memory access:

```
#define get_byte(offset) ((zbyte_t)
datap[offset])
```

This declares the macro $\verb"get_byte"$, which can be used in the source code as

if it were a function call, but will actually be replaced before compilation with code that directly references the global array datap. ZIP uses datap to store the Z-machine's memory, much like \$story_bytes is used in rezrov. By inlining this array code, ZIP is able to access the Z-machine's memory without the overhead of calling a subroutine to do so. This provides a huge boost in performance. Even better, get_byte can be used in the source code just like a function, enhancing readability and maintainability, because the programmer doesn't have to manually duplicate the references to datap all over the program. Unfortunately, Perl has no built-in way to expand macros in source code, which seems odd considering its fantastic text-processing capabilities. The

Filter::cpp module (available from CPAN) does provide an interface to the standard C preprocessor, but I am reluctant to use it because it will limit the number of systems rezrov can run on. I have attempted various workarounds, from constant subroutines to manipulating source code in text variables with regexps and then eval 'ing it, but none of these approaches has been particularly satisfactory. I would really like to find a good solution for this problem; the desirability of inlining is obvious from the benchmark results above, where the fully inlined version is nearly four times faster than the fastest subroutine equivalent. Anyone?

Conclusion

The more I studied the Z-machine, the more I realized that many of the things that drew me to it were the same things that attracted me to Perl itself.

The story of the Z-machine is the story of an ongoing open source project that has been around nearly as long as Perl. But while the Z-machine architecture began to show cracks as it struggled to integrate graphics and sound in late versions, Perl's modular design has allowed these features to be introduced in the form of dynamically-loaded extension modules that do not weigh down the core. And while the code base of Z-code interpreters has fragmented somewhat between systems, Perl has been able to dodge this bullet, most recently with the reintegration of the Win32 port back into the standard distribution.

The Z-machine was designed to provide universal and consistent access to Z-code programs by making them run exactly the same way on as many different platforms as possible. One of Perl's greatest strengths is its success at doing exactly the same thing: Perl, like the Z-machine, transcends the foibles of individual systems with its vision of a common playground. And that's a beautiful thing.

References

Rezrov

http://www.edmonson.paunix.org/rezrov/

Graham Nelson's Z-machine specification

http://www.gnelson.demon.co.uk/zspec/

Mark Howell's ZIP source code

ftp://ftp.gmd.de/if-archive/infocom/interpreters/zip

Paul David Doherty's Infocom Fact Sheet

ftp://ftp.gmd.de/if-archive/infocom/info/fact-sheet.txt

Pete's Infocom page (various historical documents and articles.)

http://www.csd.uwo.ca/~pete/Infocom/

The rest of Graham Nelson's site

http://www.gnelson.demon.co.uk

The IF archive

ftp://ftp.gmd.de/if-archive/

The annual IF competition

http://www.ifcompetition.org

IF newsgroups

news://rec.games.interactive-fiction, news://rec.arts.interactive-fiction

Chapter 30. Tktk: A Perl/Tk Solitaire Game

Greg Bacon

One evening, I saw my wife playing a

solitaire game that I'd never seen before, and whose name I still don't know. I asked her to explain the rules. The game turns out to be a simpler variation of

Pyramid, which is described at http://www.semicolon.com/ solitaire/rules/pyramidrules.html.

The Rules

To play, you lay out the

cards in a seven row pyramid. The first row contains one card, the second two, and so on. When laying subsequent rows, you should cover each card in the previous row with two cards (picture the running bond pattern from brick masonry), as shown in Figure 30-1.

If you want to impress your solitaire aficionado buddies, call these 28 cards the *tableau*. The rest of the cards are the *stock*. To play, place the stock's bottom card face up on top of the stock. This top card is called the *base*. Your goal is to pick up all the cards in the tableau. You may pick up any uncovered card whose denomination or rank is adjacent to the denomination of the base card, regardless of suit. For instance, if your base card is a six, you can pick up a five or seven. Aces are considered adjacent to both deuces and kings.

When you pick up a card from the tableau, it becomes the new base card. When you decide that you want a new base card (usually when you can't pick up any more from the tableau), you flip from the bottom of the stock. The game is over when you empty the tableau (you win) or when you exhaust the stock (you lose).

It turned out to be a fun game. A

solitaire game has to be tantalizing to hold the player's interest, and this one definitely is. My wife would laugh cruelly when I bemoaned being so close to winning (she says she's only won about three times in all the years that she's played). I realized that she had infected me with an awful meme, so now I'll spread it to you.



Figure 30-1. The start of the game

A First Cut

The more I played, the more I became impatient with tedious tasks such as shuffling the deck and laying out the cards. I kept thinking, "I should implement this game in Perl," and eventually I did. I had a lot of fun in the process, and this article will describe the steps involved.

The first decision was whether the interface should be graphical or character-based. I chose graphical. (Sorry, I'm a wuss—I even play the tiled X11 version of *nethack*.) The next step was to find some card images (because I didn't want to wake up the Gimp). and а web search found http://www.waste.org/~oxymoron/cards/. The images have sensible names: *qs.gif* is the queen of spades, *th.gif* is the ten of hearts, and so on.

Shuffling

Before we can play, we have to shuffle the deck. The Fisher-Yates algorithm to the rescue:

```
# Fisher-Yates shuffle
sub shuffle {
    use integer;
    my $array = shift;
    my $i;
    for ($i = @$array; -$i; ) {
        my $j = int rand ($i+1);
        @$array[$i,$j] = @$array[$j,$i];
    }
}
```

Of course, we need a deck to shuffle. I decided to build a hash whose keys are the card denominations and suits (the basenames of the card images) and whose values are Tk::Photo objects created from the card images:

```
my %card;
sub init_
cards {
    my $c = shift;
    my @denom = ( 2 .. 9, qw/ t j q k a /
);
    my @suits = qw/ h d c s /;
    my @dim = ( -width => CARDWIDTH,
-height => CARDHEIGHT );
    foreach my $d (@denom) {
```

```
foreach my $s (@suits) {
    my $im = $c->Photo(-file =>
"cards/$d$s.gif", @dim);
    $card{"$d$s"} = $im;
    }
}
```

Layout

Now that we have a shuffled deck, we need to lay out the cards. I considered using Gtk, but I have more experience with Tk and wanted to play my game as soon as possible.

Using Tk, we create a

Tk::Canvas object and place Tk::Photo image items on it. The layout is pretty simple. The tableau's vertical axis of symmetry coincides with the vertical center line of the canvas. For each row, we compute half the width of the whole row, move left half that distance, and place cards from left to right. The code looks like this (note that I called the tableau the "field"):

```
# Place $
cards
        my @row = splice @$deck, 0, $;
        for (@row) {
            $c->create( 'image',
                        int $x, int $y,
                        -anchor => 'nw',
                                -image =>
$card{$ },
                              -tags => [
'field', "card=$ " ] );
            $x += CARDWIDTH + CARDSPACE;
        }
        $y += CARDHEIGHT/2;
    }
}
```

Notice that we didn't have to bother with *clipping*, the process of deciding which pixels should be displayed. We simply placed the rows from back to front (that is, from the top of the pyramid to the bottom), letting new items cover anything that might be under them. This is called the *painter's algorithm* and, conveniently, achieves the arrangement we want.

Show the Tableau!

The heart of the program ties it all together:

This program is enough to produce a layout similar to Figure 30-1.

Would You Like to Play a Game?

I once had a math professor who obsessively moved radicals out of the denominators of fractions because "our teachers like it better this way." You'd think he would have been able to cast off those chains when he took his Ph.D. Programmers (or at least pedants like me) feel the same sort of guilt about rampant use of globals. I decided to aggregate all the game state information into an anonymous hash and then pass the state to the different subroutines. There's a Rob

Pike quote that I like to repeat: "The O-O languages give you more of course—prettier syntax, derived types and so on—but conceptually they provide little extra." I bring it up here because too many programmers would have carried this aggregation a step further, to define a class implementing the entire game. That would be hunting rabbits with a tank, because it's not likely that we'll have more than one game running at once and because I don't plan on inheriting and overriding methods from the game.

Making the Moves

Our program is about to become at least a little interesting as we make it recognize and respond to

events. The events that we're interested in are when the user clicks on a card in the tableau and when the user clicks on the base card.

Tk operates under an *event-driven* model. This means that the programmer says to Tk, "When a certain type of event happens (like a mouse click or a key press), execute this code." The code that Tk executes in response to an event is known as a *callback*. The way to register

callbacks (that is, how you ask Tk to execute callbacks when certain events take place), is to use Tk's bind method. Here's a simple example:

```
$parent->bind(qw/field <1>/ => sub { print
"Saw a click!\n" });
```

This tells Tk to print a message when the user clicks Button-1 (usually the left mouse button) on an item or widget with the field tag. See the

Tk::bind documentation for details.

Tags are simple strings that we can associate with items and widgets. When we laid the

cards in the field, we gave them two tags: the field tag, and a tag indicating which card it is:

-tags => ['field', "card=\$_"]);

As you can see, tags allow the programmer to create a logical association among different widgets.

Laying the Base

Now that we know how to deal with events, the code to cycle through the stock is a SMOP (Simple Matter Of Programming). If there's a card on the bottom of the stock, we display and remember it. Otherwise, we display the back of a card and tell Tk to stop executing callbacks when the user clicks the base card:

```
sub next base {
   my $
game = shift;
   my $deck = $game->{DECK};
   mv $ht = $c->cget('height');
   my $wd = $c->cget('width');
   my \$x = 3 + \$wd/2 - CARDWIDTH/2;
   my \$y = \$ht - 3 - CARDHEIGHT;
     # Lose any base card that might be
there
   $c->delete('base');
   my $image;
   my $
up;
   if ($deck and @$deck) {
       $up = pop @$deck;
       $image = $card{$up};
    } else {
       $image = $card{b};
       $c->bind(qw/base <1>/ => '');
```

We register this callback with:

```
$c->bind(qw/base <1>/ => sub { next_base
$game });
```

Pickup Lines

To legally pick up a card, it must be completely uncovered. The Tk::Canvas class provides a method for asking which items overlap a certain rectangular area. Before we can do that, we have to know where (in terms of canvas coordinates) we laid each card. We add this line to init_field:

```
@{ $game->{POS}{$_} }{ qw/ X Y / } = ($x, $y);
```

Tk associates the current tag with the item or widget involved in triggering the current callback. We can ask Tk what other

tags the "current" item has, which helps us to figure out what card it is. The code looks like this:

```
my ($cur) = $c->find('withtag' =>
'current');
my ($card) = grep /^card=..$/,
$c->gettags($cur);
```

In the rules, I stated that it is legal to pick up a card if it is uncovered and its denomination is adjacent to the base card's denomination. We can ask Tk whether any cards are covering the selected card:

Armed with the knowledge of the denominations of the base card and the selected card, we need to be able to decide whether those denominations are adjacent. I originally used a big, overly verbose hash of hashes where the test for denomination adjacency looked like this:

Stephen Lidie suggested the following data structure:

The test for a legal move then becomes:

```
if ( index($rank{$base_denom},
$field_denom) >= 0 ) {
    $c->delete($cur); # Remove the card
from the tableau
    push @$deck, $card;
    next_base $
game; # ...and make it the new base
card
}
```

If we determine at some point that the insidious user is attempting to make an illegal move, we ring the bell with c->bell and return.

Oops!

As I was testing the game, I would pick up a card and then realize that I should have picked up another card. Because of the bookkeeping in init_field that kept track of where the cards were in the tableau, to undo we need only replace the card in the tableau and the last base card on the stock. The code is simple:

```
sub undo {
   my $c = shift;
   my $game = shift;
   my  $up = $game->{UPCARD};
    return unless $up;
         my(\$x, \$y) = map int, @{
$game->{POS}{$up} }{ qw/ X Y / };
    $c->create( 'image',
                $x, $y,
                -anchor => 'nw',
                -image => $card{$up},
                    -tags => [ 'field',
"card=$up" ] );
    game -> \{UPCARD\} = 0;
           push @{ $qame->{DECK} },
$game->{OLDBASE};
   next base $
game;
}
```

Notice the \$game->{UPCARD} check. This makes it so that players can undo only a pickup. (If we also allowed players to undo after flipping from the bottom of the stock,

they would enjoy the luxury of a one-card lookahead, and that would make the game less excruciating.)

Finishing Touches

At this point, I had a playable game, but I wanted to add a more polished feel. The first touch was to add a check to make sure that the directory of cards was where we expected:

And what

GUI would be complete without menus and menu accelerators?

```
my $restart = sub { $game = new_game $c };
my $undo = sub { undo $c, $game };
my $mod = 'Alt';
if ($^0 eq 'MSWin32') { $mod =
'Control' }
elsif ($^0 eq 'MacOS') { $mod =
'Command' }
my $menu = $mw->Menu;
$menu->cascade( -label => '~File',
```

```
-tearoff => 0,
                      -menuitems => [ [
command
            => '~New game',
-command => $restart,
-accelerator => "$mod+N" ],
                                        Γ
command
           => '~Undo pickup',
-command
           => $undo,
-accelerator => "$mod+U" ],
                               ۰۰,
                                        Γ
command => '~Quit',
-command => [ destroy => $mw ],
-accelerator => "$mod+Q" ] ] );
$mw->configure(-menu => $menu);
$mw->resizable(0, 0);
$mw->bind("<$mod-n>" => $restart);
mw->bind("<mod-q>" => [ destroy => mw
1);
$mw->bind("<$mod-u>" => $undo);
```

Any artist will tell you that the work isn't done until it's signed:

tktk (Tk timekiller)\n" .
"gbacon\@cs.uah.edu\n\n" .
"(updated \$UPDATED)\n\n" .
"Tk
version \$Tk::VERSION");

The result is shown in Figure 30-2.



Figure 30-2. A dialog box containing information about the game

I wish it would do something neat when the player empties the tableau, but I haven't come up with anything cool enough. If you have any ideas, please let me know.

As you've seen, Tk provides a flexible framework for developing graphical user interfaces. This program was easy: it only took a couple of days to develop. The result was a fun alternative to sol.exe for killing time. Grab the code from the web page for this book (http://www.oreilly.com/catalog/tpj3) and give it a try.

Chapter 31. The First Perl/Internet Quiz Show

Jon Orwant

At the Second Perl Conference in 1998, I wrote and moderated the first-ever Perl Quiz Show. Four three-person teams competed for a variety of delicious prizes: Perl Resource Kits, TPJ subscriptions, TPJ Magnetic Perl kits, Perl Mongers T-Shirts, and O'Reilly gift certificates.

There were two semifinal rounds and one final round; each round pitted two teams against one another with fifteen toss-up questions. Toss-ups are questions answered by individuals; each correctly-answered toss-up earns ten points and the right for the entire team to collaborate on a bonus question worth up to thirty points.

What follows are all of the toss-up and bonus questions I wrote, including several that weren't asked. If the toss-ups seem simple, bear in mind that merely knowing the answer isn't enough—you also have to answer before the other team buzzes in. If the bonuses seem hard, bear in mind that the entire team gets to confer on the answer.

Answers are at the end of the article. Since toss-up questions were tests of how *fast* you could answer correctly, and bonus questions were given to teams of people, neither translates very well to the printed page. But if you simply must quantify your score, count one point for each correct answer (rounding partial credit on bonuses) and sum.

Toss-up Questions

Toss-up 1: In a regular expression, you can use

curly braces to specify how many times you want to match something. For instance, $x \{5, 10\}$ matches at least five but no more than ten x's. What is the maximum number of occurrences you can match with this technique?

Toss-up 2: Guess that switch: Every once in a while, someone tells me they wish they could use #if and #ifdef statements in Perl. They can; all they need to do is use this flag.

Rate Your Knowledge

The five quiz shows each have around 90 questions, split into toss-ups and bonuses. Count one point for each correctly-answered toss-up, and one point for each bonus question where you get the answer at least half correct.

0–10:

Perl

Novice. Hit head with board, read FAQ, retake test.

11–25:

Perl

Adept. Keep telling yourself that you would have scored a lot higher if you'd used the online documentation, and really isn't that a better way to gauge programming competence.

26–50: Perl
Guru. Feel guilty for peeking at the answers, disagreeing with them, and deciding that you're really correct.

51–70: Perl Wizard. Your future holds satori and high consulting fees.

71+: Cheater. You already knew the answers because you attended the quiz show in person. Shame on you for pretending that you really did this well.

Toss-up 3: Guess that variable: What scalar contains the name of the operating system you're on?

Toss-up 4: A tangent function, a floor function, a function to generate temporary filenames. These are all part of what module bundled with

Perl, meant to provide a uniform way of calling functions independent of what operating system you're on?

Toss-up 5: Let's say you have a

digital clock that displays twelve-hour time (that is, it displays a.m. and p.m.). Ignoring the colon, what's the smallest power of two that will appear on the display?

Toss-up 6: Continuing with the previous question, what's the largest power of two?

Toss-up 7: Spin the Black Circle, Who You Are, Jeremy, Evenflow. These are songs by a Seattle grunge rock band with a two-word name of particular relevance to this conference.

Toss-up 8: It's the only Perl program I know of that has been

tattooed on a forearm. This cryptographic scheme was invented in 1977 by three MIT graduate students: Ron Rivest, Adi Shamir, and Len Adleman.

Toss-up 9: If you have multiple______in your Perl program, they'll be executed in reverse order: the last one will be executed

first, and the

first one last. This behavior is the opposite of the BEGIN block.

Toss-up 10: \$ has two characters, @ARGV has five. This symbol is arguably the only

single-character variable in Perl. It's a special filehandle used to speed up calls to stat, lstat, or any of the file tests.

Toss-up 11: The

first day of the year is always a Monday, somewhere between December 29th and January 4th. That's according to the standard year of this organization, which is responsible for POSIX.

Toss-up 12: The hex function converts a hexadecimal number to decimal. What function would you use to convert back to hex?

Toss-up 13:

Back Orifice '98 allows you to remotely control the Windows 95 and 98 desktops. Name the group responsible.

Toss-up 14: A 233-MHz 750 PowerPC processor, a 4GB hard disk drive, 32MB of memory, built-in networking, an

internal modem, a CD-ROM drive, and no floppy drive. What is this

computer, released by Apple in August 1998?

Toss-up 15: A

Frankenstein language created with pieces of JavaScript, cascading stylesheets, and HTML. That's how WIRED News describes this variant on HTML that lets you enhance HTML with animation.

Toss-up 16: "The Lurking Horror,"

"Planetfall," "Leather Goddesses of Phobos," and the classic "Zork" were all games produced by what company?

Toss-up 17: A pragma is like a module—you invoke it with use—but it affects how Perl interprets your program. Which pragma is most likely to speed up the typical Perl program?

Toss-up 18: Definitions of croak and confess are found in which module?

Toss-up 19: Sarah Ophelia

Cannon's hat resides at the National Museum of History. The hat, with the price tag still attached, symbolizes Cannon's character at the

Grand Ole Opry, a homophone with the little program that Perl uses to build itself.

Toss-up 20: If Perl 5 were designed from scratch, these two functions (which currently exist in both Perl 4 and Perl 5) probably wouldn't be there, because everything you can do with them you can now do with the tie function. They let you store hashes on disk.

Toss-up 21: Give me the two common ways to access (but not modify) the last element of the array @bar.

Toss-up 22: What does this display?

```
print print print print print
```

Toss-up 23: This selfish-sounding keyword didn't exist before Perl 5. It's a faster and safer way to declare variables than local.

Toss-up 24: This word is the name for what a pearl is before it's big enough to be called a pearl, but is better known in biology as the protoplasmic center of a cell in control of growth and metabolism.

Toss-up 25: Someone comes up to you and says that use strict is broken, because they say a = \$[; use strict; and no error occurs; they were expecting to see Global symbol 'a' requires explicit package name. What Perl function is responsible for this crime of omission?

Toss-up 26: There's no difference between the foreach and for operators, but those aren't the answers to this question. Two Perl operators that manipulate strings are 100% synonymous. What are they?

Toss-up 27: When I write a

smiley face, it's a colon followed by a hyphen followed by a right parenthesis. Consider the three scalar variables \$:, \$-, and \$). Which one will vary from user to user on the same system?

Toss-up 28: Gem, moo, six, miss. Which of these words couldn't you find immediately after the second slash of an m// pattern match?

Toss-up 29: \$x is set to fuzz. What is \$x++?

Toss-up 30: tr/a-z//. What does that do?

Toss-up 31: In a Perl regular expression, how do you create a grouping without creating a backreference?

Toss-up 32: What's the sum of the protocol numbers for SMTP, FTP, and telnet?

Toss-up 33: The 1997 Ig Nobel Prize for Communications went to this man, the president of Cyber Promotions and widely-acknowledged King of Spam.

Toss-up 34: "Join us now and share the software." This is the first line of a song by a MacArthur grant winner who hangs out at MIT, wrote Emacs, and founded the Free Software Foundation.

Toss-up 35: This built-in function constructs a linked list of every character in a string, and is sometimes invoked before expensive pattern matches. It's what good little boys and girls do after school.

Toss-up 36: What command-line switch checks the syntax of Perl programs?

Toss-up 37: "When you say it to your computer, the terminal may become slightly moist." This was said by Donald

Knuth, describing what text formatting package of his own design?

Toss-up 38: He'll be twenty years old next year and underwent a five-hour operation on a brain tumor in 1991. The operation was successful, allowing him to enjoy his title as recipient of the world's most postcards—33 million in all, thanks to a massive Internet campaign.

This line of Perl code comes from the file *hv.h* in the Perl distribution, and it's what Perl uses to convert the keys of your hash into its internal representation. The number that goes in the blank is a power of two plus one. It's two digits, and both digits are the same. What is it?

Toss-up 40: To help the Perl development team debug Perl, you decide you're going to write a program that generates and executes random Perl programs. You exclude functions like system and unlink and syscall and open and kill that directly affect your operating system or any other processes running on your computer. You leave your program on all night, and in the morning you discover that many of your programs crashed. What four-letter function is the culprit?

Toss-up 41: What's the longest function name in Perl?

Toss-up 42: Initial etic inquiry typically yields particles whose wave or field relationships (i.e., situatedness) to other particles are undetected, indistinct, or ambiguous and which must be identified before progress can be made toward emic understanding. This description of something called "eticity"

is a core definition of what field of computational linguistics that influenced Larry's design of Perl?

Toss-up 43: "If there is a Perl 6, Perl 7 will be built out of sticks and stones." I made that up, but it's an allusion to a statement about World Wars III and IV by what famous physicist known for his Theory of Relativity?

Toss-up 44: Perl is free software. If you redistribute it, you can choose between two completely separate licenses: the GNU General Public License, and what other license?

Toss-up 45: Assuming a default Perl configuration, which of these four code snippets will beep?

```
print "\a" if $[;
print "\007" if $0;
print "\x7" if $$$;
alarm;
```

Bonus Questions

For this book, each bonus question is worth a maximum of one point. As an example, if you get two or more correct in the

first question below, score one. Otherwise, score zero.

Bonus 1: I'm going to make four statements about values that a scalar can hold. After each, you tell me whether the statement is true or false.

- Every string is true except for the null string.
- Every number is true except for the quantity 0.
- Every reference is true.
- Every undefined value is false.

Bonus 2: I'll give you three lexically scoped variable declarations. Which are valid? Answer yes or no.

```
my $_;
my $::dog;
my $big{dog};
```

Bonus 3: Pencil and paper ready: give me a regular expression that matches these three misspellings of O'Reilly:

```
Oreilly O`Reilly O Riley
```

but does not match these two misspellings:

```
Or Reilly Oh really O'Rwant
```

Bonus 4: I'll give you four partial Larry sayings. You complete them.

"Down that path lies madness. On the other hand, the road to hell is paved with _____."

"The three principal virtues of a programmer are

Bonus 5: Write a Perl one-liner that can be executed from the command line. The one-liner should take a file and replace each tab with a space, but use no more than six letters to do so. You don't have to count Perl or the filename, but it does have to count command-line switches.

Bonus 6: If you say 3 + 4 * 5 without any parentheses, the answer is 23 and not 35 because of something called precedence. * has a higher precedence than +, so it steals the 4 away from the 3. Rank the precedence of these five Perl operators from lowest to

highest. ++ ** == .. ||

Bonus 7: I'll give you four comments. After each one, you tell me whether the comment appears in the Perl source code.

"And you'll never guess what the dog had in its mouth."

"Here there be dragons."

"I'm not insane, you're insane, and you're trying to steal my magic bag."

"It all comes from here, the stench and the peril."

Bonus 8: Collecting

demographic information about Perl is difficult because there's no central distribution point, and there's no registration process. So we'll treat the Perl Journal subscriber list as representative of the Perl community as a whole. After the United States and United Kingdom, which six countries have the most Perl programmers? You don't have to order them.

Bonus 9: Caffeine is the drug of choice for programmers. I'll give you four drinks; you rank them in order from least caffeinated to most caffeinated.

Mountain Dew,

Pepsi,

```
Jolt, and Coca Cola. Where does one tablet of
```

Excedrin fit into this list?

Bonus 10: People complain about Perl's unreadability, but there are many ways to write crystal-clear Perl code. One of those ways is the

English.pm module, which lets you use

verbose alternatives to the sometimes cryptic scalars that Perl predefines for you. For instance, instead of saying \$_, you can say \$ARG. I'll give you five English names; you give me the Perl scalar each represents. MATCH, INPUT_LINE_NUMBER, OUTPUT_AUTOFLUSH, FORMAT_LINES_PER_PAGE.

Bonus 11: Usenet started as a 300-baud connection between which two universities?

Bonus 12: Full credit if your answer is within a factor of 10: How many

black market kidneys could Microsoft buy? I'm assuming the August 14, 1998 market capitalization of Microsoft and the price of one kidney quoted in the

famous piece of Internet folklore. Assume that Microsoft wants to corner the kidney market and is willing to spend its entire net worth to do so.

Bonus 13: Rank the following in order of occurrence:

- IRC is launched.
- Perl is introduced to the public.
- The acronym "FAQ" first appears on Usenet.
- Intel introduces the 80386 CPU.

Bonus 14: Rank the following in order of occurrence:

- The Communications Decency Act is introduced.
- DOOM is released.
- Altavista is launched by Digital.
- 33.6 kilobit modems are introduced.

Bonus 15: You've heard of MP3s, the CD-quality audio format that's giving music producers the shivers about this Internet thing. MP3 stands for MPEG-3; MPEG is a digital video format. What do the four letters M, P, E, and G stand for?

Bonus 16: After I finish reading this question, I'm going to ask the crowd to choose whether they prefer vi to

Emacs. Will substantially more people choose vi, substantially more people choose Emacs, or will the two be roughly equal?

Bonus 17: Write a syntactically perfect program that prints "yes" if a number is evenly divisible by 10. The number will be provided as an argument to the program. I'll give you a little extra time.

Bonus 18: Rank these code snippets from slowest to fastest:

```
print "$c$d";
print $c, $d;
print $c . $d;
```

Bonus 19: Expand these four acronyms: CGI, FQDN, URL, ISDN.

Bonus 20: Expand these four acronyms: TCP, SMTP, MIME, XML.

Bonus 21: Expand these four acronyms: IANAL, CERT, API, FTP.

Bonus 22: I'm going to name six cities. After each, you tell me whether it had a

Perl Mongers chapter in 1998.

Toronto; London; Redmond, Washington; Paris; St. Louis; Helsinki.

Bonus 23: The Perl source code is peppered with Tolkein quotes. What are the three books in Tolkein's Lord of the Rings trilogy?

Bonus 24: Pencil and paper ready. Take the area code of this hotel (Editor's note: the

first Perl Conference was in San Jose) and subtract from it the number of bones in the adult human body. You'll end up with the area code for a U.S. city commonly known by a two-letter abbreviation, or four characters if you count punctuation. Give me a pattern that precisely matches that four-character abbreviation.

Bonus 25: I use the mSQL DBD to do CRUD. There are three database-oriented acronyms here: mSQL, DBD, and CRUD. What do they stand for?

Bonus 26: I'll name six modules. After each, you tell me whether it exists on the CPAN. Search::Patent, Text::GenderFromName, Mortgage::Calculate, D'oh, Lint, Business::UPS.

Bonus 27: I'll give you an operating system or computer, you tell me whether Perl runs on it, yes or no. BeOS, Cray, Amiga, PalmPilot.

Bonus 28: I'll give you three built-in Perl functions; you tell me what prototypes they would have if you were writing equivalent subroutines. join, keys, pipe.

Bonus 29: When you apply the built-in ref

function to an object, you get the name of the object. When you apply it to a non-object reference, you get one of six answers. What are they?

Bonus 30: Consider the scalar $\$X : : Y : : Z \cdot X$ and Y are both packages. What is the relationship between X and Y?

Bonus 31: I'm going to list five functions. For each, tell me whether it's built into Perl 4, Perl 5, both, or neither. reverse, bless,

eval, tag, abs.

Bonus 32: Many of Perl's operators can be made into assignment operators by adding an

equals sign. For instance, a + 4 adds a and 4, but a += 4 adds a and 4 and sticks the result in a. I'll give you four operators; you tell me which can be made assignable by adding an equals sign. Answer yes or no after each. x, \hat{a} , \hat{a} .

Bonus 33: Laptops in the movies! Name the movie in which alien ships are infested with a virus uploaded from a PowerBook. Next, name the Tom Cruise movie in which we briefly see lists of Usenet groups scrolling by on a laptop.

Bonus 34: I'll give you three code snippets. Tell me which module bundled with the Perl distribution contains the snippet.

```
print $query->textfield( -name =>
'field_name' ...
@type = qw/Author Bundle Distribution
Module/;
tr/AEHIOUWYBFPVCGJKQSXZDTLMNR/
000000011112222222334556/;
```

Bonus 35: Many people don't know that Perl comes with its own built-in

symbolic debugger, perhaps because Perl programs practically debug themselves. It's invoked with the -d switch. I'll give you a debugger command, and you tell me what it does. Lowercase b, lowercase x, lowercase h, lowercase s.

Bonus 36: What does this print?

perl -e '\$-=\$=; print \$--- --\$-'

Bonus 37: You have a hash, and you know it's been tied to some package. Which two functions should you use to find out what package that is?

Bonus 38:

Entropy coding (and, more to the point, common sense) dictates that you should use short terms to represent commonly-occurring concepts. That's why Morse code uses a dot for E and a dash for T. You see this all over Perl, too. Give me a three, four, five, six, and seven letter built-in Perl function.

Bonus 39: There are five tokens beginning and ending with double underscores that have special meaning to Perl. What are they?

Bonus 40: Name that

file test. I'll give you five operations. You give me the single-letter file test that performs that operation. Don't forget to specify whether the letter is lower- or uppercase. Is the file there? Is it a binary file? Is it empty? How long has it been since the last access? How big is the file?

Bonus 41: Which

modules are bundled for the

first time with Perl 5.005? I'll give you six modules that might or might not exist; answer yes or no after each. Test.pm, Bytecode.pm, Thread.pm, Java.pm, fields.pm, Compiler.pm.

Bonus 42: There's always more than one way to do it, but can you tell me the two most concise ways to make Perl add a newline after each print?

Bonus 43: The caller function returns information about the environment in which it was invoked. If it's invoked with no arguments inside a subroutine, it tells you three things. What are they?

Bonus 44: This little known technique lets you transform your Perl program as it's being interpreted. For instance, you can use this to encrypt your source code and have Perl decrypt it on the fly. What is this technique?

The Answers

I used an old College Bowl convention for the cards I used when emceeing the show: capitalizing the part of the answer that players needed to say. Even though I was both judge and jury, it occasionally helped to be able to show the card to players to avoid any perceptions of favoritism.

Toss-up Answers

T1: 65535 T2: -P (-p is incorrect) T3: \$^0 T4: POSIX T5: 128 T6: 1024 T7: PEARL JAM T8: RSA T9: END blocks (also accept END subroutines) T10: _ T11: ISO or

INTERNATIONAL STANDARDIZATION ORGANIZATION or INTERNATIONAL ORGANIZATION for STANDARDIZATION

T12:

SPRINTF

T13.

CULT OF THE DEAD COW (Sean Mahoney, age 16, got this one.)

T14: The IMAC

T15:

DHTML or DYNAMIC HTML

T16:

INFOCOM. No one got this. Depressing.

T17: INTEGER

T18:

CARP

T19:

MINNIE PEARL or MINIPERL

T20:

DBMOPEN and DBMCLOSE

T21: bar[-1] and bar[\$#bar]

T22: 1111 (print returns 1 on success.)

T23:

MY

T24: NUCLEUS

T25: SORT

T26: TR and Y

T27: \$) (which contains a list of group IDs.)

T28: GEM

T29: FVAA (You can increment letters as well as numbers. a plus one is b, y plus one is z, and z plus one is aa. So fuzz plus one is fvaa. You can't decrement them, though.)

T30: COUNTS the LOWERCASE LETTERS in \$. (The player buzzed in and said "It's a no-op." I declared that incorrect. Then some rube in the audience yelled out that if \$ were empty, the compiler would optimize away the statement, so that it actually would be a no-op. I don't think that's what he was thinking, but I awarded full points anyway.)

T31: (?:)

T32: 69 (FTP is 21, telnet is 23, and SMTP is 25. Someone pointed out that FTP could have another port number, and so the answer isn't unique. However, the team answered 69, getting full points, so it's moot.)

T33: Sanford WALLACE T34: Richard STALLMAN **Linked Bonus**: For all twenty points, your entire team has to sing the next line of the song in unison.

Answer: "You'll be free, hacker, you'll be free." No points awarded; the team didn't know the next line. After I read the answer, someone in the crowd yelled out for me to sing it. Security detained him after the show.

T35:

STUDY

T36: -c

T37:

TEX (According to Knuth, "Insiders pronounce the X of TeX as a Greek chi, not as an 'x', so that TeX rhymes with the word blecchhh.")

T38: Craig SHERGOLD

T39: 33

T40:

DUMP (Spoken from personal experience.)

T41:

GETPROTOBYNUMBER

T42:

TAGMEMICS

T43: Albert EINSTEIN

T44: ARTISTIC

Bonus Answers

B1: F (consider "0"), T, T, T.

B2: NO

B3: Answers can vary. /O.?Re?il/ (I shuffled this question to the end because it's too open-ended. It was never asked.)

B4:

LAZINESS, IMPATIENCE, and HUBRIS

B5:perl -pi -e 's/\t/ /g'

B6: . . | | == ** ++

B7: YES (mg.c), NO, NO, YES (Frodo said that in toke.c.)

B8: GERMANY, CANADA, AUSTRALIA, FRANCE, JAPAN, SWITZERLAND. (The next six are Sweden, Holland, Norway, Denmark, Finland, and Italy.)

B9: PEPSI (38 mg/12 ounces), COCA-COLA (45.6 mg), MOUNTAIN DEW (54 mg), JOLT (72 mg). Excedrin is between Mountain Dew and Jolt. (Tidbit: Canadian Mountain Dew has no caffeine.)

B10: \$& \$. \$| \$=

B11: UNIVERSITY of NORTH CAROLINA and DUKE

B12: 25,690,000 kidneys. (Accept between 2,569,000 and 256,900,000 kidneys). (When the quiz show began, the market cap of Microsoft was \$256.9 billion. Kidneys are a steal at 10 grand.)

B13: Intel introduces the 80386 CPU (June 86); Perl is introduced to the public (Jan 88); IRC is launched (May 89); The acronym "FAQ"

first appears on Usenet (Sept 89).

B14: DOOM is released (Dec 93); The Communications Decency Act is introduced (Feb 95); Altavista is launched by Digital (Dec 95); 33.6 kilobit modems are introduced (Jan 96)

B15:

MOTION PICTURE EXPERTS GROUP

B16: Depends on crowd response. (vi got more applause, much to my chagrin.)

B17: print "yes" unless shift() % 10 is one possible answer. (I didn't ask this because it was too open-ended.)

B18: A-C-B

B19:

COMMON GATEWAY INTERFACE; FULLY QUALIFIED DOMAIN NAME; UNIFORM RESOURCE LOCATOR; INTEGRATED SERVICES DIGITAL NETWORK (The team didn't fall into my trap. I'd hoped that they'd answer Universal Resource Locator.)

B20:

TRANSMISSION CONTROL PROTOCOL; SIMPLE MAIL TRANSFER PROTOCOL; MULTIPURPOSE INTERNET MAIL EXTENSIONS; EXTENSIBLE MARKUP LANGUAGE

B21:

I AM NOT A LAWYER;

COMPUTER EMERGENCY RESPONSE TEAM;

APPLICATION PROGRAM INTERFACE;

FILE TRANSFER PROTOCOL (The team got hung up on the

first one, trying desperately to complete "Internet Assigned Numbers Authority ______". Tom

Phoenix pointed out that

CERT had recently declared that their name was—poof!—no longer an acronym.)

B22: Toronto NO, London YES, Redmond NO, Paris NO, St. Louis YES, Helsinki NO (Mongers groups exist in some of these places now, but not in 1998.)

B23: The FELLOWSHIP OF THE RING, The TWO TOWERS, The RETURN OF THE KING

B24: $/D \setminus .C \setminus ./$ (The team said $\setminus w+$, but that won't handle the dots.)

B25:

MINI STRUCTURED QUERY LANGUAGE; DATABASE DRIVER; CREATE, READ, UPDATE, and DELETE.

B26: Search::Patent NO, Text::GenderFromName YES, Mortgage::Calculate NO, D'oh YES, Lint NO, Business::UPS YES

B27: BeOS YES, Cray YES, Amiga YES, PalmPilot NO (Ken Albanowski wrote a Perl API to the PilotLink library, which lets you develop Pilot applications in Perl.)

B28: join \$0; keys $\$; pipe **

B29: REF, SCALAR, ARRAY, HASH, CODE, GLOB (But there are more: LVALUE and IO.)

B30: There is no relationship.

B31: reverse BOTH, bless PERL 5, eval BOTH, tag NEITHER, abs PERL 5

B32: x YES, ^ YES, & YES, & & YES

B33: INDEPENDENCE DAY, MISSION: IMPOSSIBLE

B34:

CGI.pm, CPAN.pm, SOUNDEX.pm

B35: b sets a BREAKPOINT; x evaluates, and PRETTY-PRINTs (and prints nested data structures too); h displays HELP; s single-STEPS through the code.

B36: 2 (The print statement is equivalent to saying x - (x - 2), so it'll be 2 in almost every circumstance. Chip

Salzenberg came up to the stage afterward and pointed out that the order of evaluation of the arguments isn't guaranteed by Perl, so the answer wouldn't necessarily be two on all implementations. I stand by this answer, because I've yet to see a Perl implementation that evaluates the second argument of—before the first. Now Chip will patch Perl just to spite me.)

B37:

REF and TIED.

B38: One possible set of answers: pop, push, sleep, scalar, reverse.

B39: __FILE__, __LINE__, __PACKAGE_ _, __DATA__, __END__

B40: -e tests whether the file is there; -B tests whether the file is binary file; -z tests whether the file is empty; -A returns how long it's been since the last access; -s returns how big the file is.

B41: Test.pm YES, Bytecode.pm NO, Thread.pm YES, Java.pm NO, fields.pm YES, Compiler.pm NO

B42: -l and $\$ = `` \n''$

B43: The PACKAGE, the FILENAME, and the current LINE.

B44:

SOURCE FILTERing (*The answer given was "filter*." *I* declared it incorrect.)

Chapter 32. The Second Perl/ Internet Quiz Show

Jon Orwant

At O'Reilly's 1999 Open Source conference, I emceed and judged the

Second Perl/Internet Quiz show, which pitted teams of Perl hackers against one another to win enduring fame and a motley collection of prizes. Here are the questions, including a few I didn't ask.

Four teams played, with all participants winning one of the following:

- VA Linux: A cube fridge and \$40 to fill it with junk food
- perltoys.com: Magnetic Perl Poetry Kits
- Geek Cruises: 50% off a state room for the Perl Whirl Alaska Tour
- O'Reilly \$50 and \$25 gift certificates
- TPJ: Free two-year subscriptions

If you want to tally your score, you can use the ratings at the beginning of Chapter 31.

Sample Questions

The Perl Quiz Show isn't like Jeopardy or Win Ben Stein's Money. It's modeled after College Bowl, a family of collegiate tournaments that I participated in at MIT. Here's the sample toss-up I used to warm up the teams, with interspersed commentary.

Toss-up 0: This company started in a abarn in Newton, Massachusetts,

This question illustrates how the ideal toss-up question is written: with the most obscure information at the beginning.

and originally specialized in technical writing and consulting.

At this point, a few people in the crowd already knew the answer.

Their consulting business slowed down in 1985, so they tried publishing some of their material as books, and thought they might give them away to promote their consulting business.

Most people had a good guess after this question.

In 1988, they were mobbed by participants at the MIT X Conference for their Xlib manuals, and soon after they focused on publishing computer books.

Now just about everyone knew.

They now publish more than 120 books, many of which have pictures of animals on the cover.

If you haven't buzzed in by now, you shouldn't be playing.

They've recently branched out into the conference business.

And the kilowatt spotlight over the head:

For ten points, name this company that is hosting the conference YOU'RE AT RIGHT NOW.

Answer: O'REILLY & Associates

Because many toss-ups are designed with a giveaway at the end, the questions aren't as challenging in print as they are when spoken. Remember that when you read these questions: the skill is not in answering the question correctly so much as answering it before the other team.

The capitalization indicates the essential part of the answer that players had to utter. Here, anyone who buzzed in only had to say "O'Reilly"; they didn't have to give the full name of the company.

Here's a sample bonus question:

Bonus 0: I'll give you four book titles; you tell me whether O'Reilly published the book or not.

Occasionally, a bonus question is linked to the previous toss-up.

a. Using Samba

Answer: YES. That's an easy one.

b. The Adventure of Food

Answer: YES. Surprisingly, the team got this one right.

c. Danger!

Answer: YES. Sounds like a dime-store detective novel.

d. Curious George Learns Assembler

Answer: NO.

Toss-up Questions

Toss-up 1: Movies have their Oscars, music has Grammys, and TV has Emmys. This year, the Internet Movie Database (imdb.com) won this award, which is given to web sites.

Toss-up 2: This

hybrid data structure is available in Perl 5.005, allowing you to refer to array elements with names instead of numbers.

Toss-up 3: This newspaper has many more web visitors than actual paper subscribers, and has run stories entitled "Bill Gates Grants Self 18 Dexterity, 20 Charisma," "Chess Supercomputer Beaten Up By More Popular Computer," "Microsoft Patents Ones, Zeros," "Apple Employee Fired For Thinking Different," and "New Smokable Nicotine Sticks." What is this newspaper named after a spherical vegetable?

Toss-up 4: Guess that special variable. What magic scalar can you undefine to make Perl read an entire file directly into a single string?

Toss-up 5: The baud rate of regular fax machines is a common modem speed, although slow by today's standards. Name this speed.

Toss-up 6: There are many sorting algorithms; Perl uses this variety. What is it?

Toss-up 7: His relevance to Perl is somewhat obscure: the Perl source code is full of Tolkein quotes, and he is the director of the *Lord of the Rings* trilogy, shot on location in

New Zealand. Some of his other movies are *Bad Taste, Meet the Feebles*, and *Dead Alive*.

Toss-up 8: You can use it with Active Server Pages and the Windows Scripting Host. This package is an

ActiveX scripting engine that lets you incorporate Perl into any ActiveX scripting host. Name it.

Toss-up 9: It had both modules and classes back in 1977. Every official module had an identifier: one or two letters followed by a single digit, and a name, such as White Plume Mountain, the Village of Hommlet, and Glacial Rift of the Frost Giant Jarl. Some of the classes were fighter, thief, cleric, and magic-user. Name this game.

Linked Bonus. The game came with five types of dice: 4-sided, 6-sided, 8-sided, 12-sided, and 20-sided. I'll give you a situation, you tell me which die is used in it.

- 1. Determining your character's strength, intelligence, wisdom, dexterity, constitution, and charisma.
- 2. How many hit points a first-level magic user has.
- 3. You're hit by lightning and need to make a saving throw.
- 4. How many hit points a first-level cleric has.

Toss-up 10: What does this print?

```
#!/usr/bin/perl
use constant e => 2, pi => 3;
print e ** pi;
```

Toss-up 11: If Larry Wall is the

father of Perl, and Perl is the mother of the World Wide Web, and the Internet is the father of the World Wide Web, the Web is the sister of FTP, and Al Gore is the grandfather of the National Organ Transplant Act, what relation is FTP to the National Organ Transplant Act?

Toss-up 12: Perl doesn't have any Year 2000 bugs, but if you store the Unix time in a 32-bit integer, you'll wraparound to 0 in what year?

Toss-up 13: What will this print? You can provide your answer in exponential notation if you wish.

print (1 << (1 << 5))

Toss-up 14: A computer utilizing this medium was used to solve the

Traveling Salesman Problem in 1995 by executing 100 trillion operations per

second, 100 times faster than the fastest supercomputer of the day. This medium let the computer perform 20 quintillion operations for every joule of energy, and was able to store one bit per cubic nanometer, which is one trillion times the storage density of videotape. What is this medium, best known for its double helix?

Toss-up 15: Guess that scalar. Name the scalar that holds the time when your program began running.

Toss-up 16: (Tiebreaker; not asked.) Guess that operator: if you stick this operator in between \$x and \$y, it will set \$x to \$y unless \$x is already true.

Toss-up 17: This country has the highest number of Internet hosts per capita, and the CPAN master site is located there. On a clear day you can see Estonia from its tallest building, which is a mere twelve stories high. Linux started there.

Toss-up 18: Guess that module. It has Purity, Terse, and Deepcopy methods. It's bundled with Perl, and written by Gurusamy

Sarathy. It's most commonly used to pretty print complex data structures via its Dumper method.

Toss-up 19: The last meeting of this group was in Oslo, and at some of their workshops, they use humming to vote on proposals. Their motto is

"Rough consensus and running code," and they're charged with maintaining the

RFC Internet standards.

Toss-up 20: This character can be used as a shorter equivalent of the

double colon. What is this character, whose string quoting behavior is emulated by the q function?

Toss-up 21: In 1995, Perl won the Nobel prize in this discipline, in part for discovering the

tau lepton in the mid 1970's. Name this discipline.

Toss-up 22: What will this print?

```
my $i; if ($i = 6) { print "success" }
else { print "failure" }
```

Toss-up 23: You're Gordon

Freeman, a scientist in this popular computer game. After an explosion nearly destroys the underground plant you work in, you have to restart various machines and kill various aliens. Your fellow scientists are eager to be rescued by Marines until the Marines start massacring them in an attempt to cover up the aliens' existence. What is this game, which has the same name as a term having to do with radioactive decay?

Toss-up 24: You want Perl to make use of an already existing C library. What utility is commonly used to create a stub interface?

Toss-up 25: The

Netcraft web server survey has been compiled every August since 1995. In this year's survey, these two web servers place first and

second in terms of popularity percentages. What are they, in order?

Toss-up 26: In my opinion, the

scariest addition to

regular expressions in 5.005 is (?{ something }). Tell me what this does.

Toss-up 27: Hit your buzzer now.

Toss-up 28: New in 5.004, this pragma accounts for the fact that not every language uses a period as a decimal point, and that not every language has the same letters as English. Its name is related to

L10N, which stands for "localization."

Toss-up 29: "It is dark in here." "The door is closed." "The Diet Coke can is 41 degrees Fahrenheit." So reads the web page for this device hooked up to the Internet. The device itself is at 48 degrees, and its freezer compartment is 12 degrees.

Toss-up 30: Where is the Internet top level domain .to located?

Toss-up 31: travesty, server, client, who, findtar, rmfrom, wrapsuid, uudecode, rename. These are all programs in what directory of the

Perl distribution?

Toss-up 32: Two words, pronounced the same. One word is a U.S. army rank, the other is the name for the core of an operation system, and is also what you find plenty of on corn on the cob.

Linked Bonus:

Microkernels versus

monolithic kernels. (*Here, I spoke off the cuff about the difference between the two.*) I'll give you four operating systems, and you tell me whether it has a monolithic kernel or a microkernel.

- 1. MS-DOS
- 2. GNU Hurd
- 3. Linux
- 4. NT

Toss-up 33: Guess that scalar. By default, out-of-memory errors aren't trappable. But they can be if your Perl was compiled with PERL_EMERGENCY_SBRK and you create an

emergency memory pool with this scalar, which has a caret followed by the thirteenth letter of the alphabet.

Toss-up 34: According to searchwords.com, this is the most popular

search term. Surprisingly, it's not pornographic, but is instead a popular audio format. Name this format, whose name is derived from MPEG3.

Toss-up 35: If you turn on

warnings with -w, what error do you get if your Perl statement is nothing more than a plain string?

Toss-up 36: Pencil and paper ready: According to Win Treese's

Internet Index, there are 7.6 billion commercial email messages sent every day. If each message were a regular snail mail letter paying U.S. first class rates, tell me how much that would cost per year, within ten percent.

Toss-up 37: Guess that scalar. You could use it to cope with division by zero errors, because this variable contains the error message from the last eval command. Name this variable.

Toss-up 38: U.S. Senator Jon

Kyl introduced a bill known as the Internet Prohibition Act in 1997. The act makes it a crime to perform this activity online, even though you can do it on Indian reservations and in Nevada. What is this activity?

Toss-up 39: Perl almost always ignores your comments. Name something that you can put in your comments that Perl *will* pay attention to.

Toss-up 40: According to the 9/12/97 issue of the *Wall Street Journal*, this is the number of miles of undersea cable. It's a funny number, because light could travel it in almost exactly one

second.

Toss-up 41: In Unix, Ctrl-Z suspends a process; Ctrl-C interrupts a process. You can keep these from happening in your Perl program with
signal handlers. Give me the signal handler that lets you trap either Ctrl-Z or Ctrl-C.

Toss-up 42: Pick the most appropriate unit: days, weeks, months, or years. If you have a cable modem that gives you a constant 1.5 megabit downstream rate, how long would it take you to download the Library of Congress, compressed?

Toss-up 43: Guess that scalar. This scalar is true if you're inside an eval and false otherwise. Name this serpentine scalar.

Toss-up 44: Marvin

Minsky and Seymour

Papert wrote a disparaging book about these computational entities and single-handedly quashed neural net

research for almost two decades. What is the name of this single-layer neural network?

Toss-up 45: On August 16, 1999, they announced plans to provide Virtual Private Network (or VPN) support for their 594 series of network controllers. They specialize in IP connectivity with product lines that include remote access servers and switches for Ethernet and token ring LAN's, terminal servers and serial connectivity products for Unix and NT platforms, and network controllers for IBM AS/400s. They trade on the Toronto Stock Exchange under the symbol PL, and trade on the NASDAQ under the symbol PERL.

Toss-up 46: What does close do if you don't provide a filehandle to close?

Bonus Questions

Remember, bonus questions are typically tougher, because the entire team gets to confer before giving an answer.

Bonus 1: This is the first bonus of the game, so let's have a question about beginnings. What regular expression

metacharacter matches the actual beginning of the string? It's not the caret, since that matches the beginning of each line, and in a multiline string it'll match multiple times. The metacharacter I'm looking for matches only the very beginning.

Bonus 2: You can specify the

background color of a web page with the BGCOLOR attribute of the BODY tag. If you set BGCOLOR to a pound sign followed by six hex digits, the first two indicate the amount of red, the

second two indicate the amount of green, and the third two indicate the amount of blue. For instance, 00FF00 is a saturated green, and FFFFFF is white. Give me a good yellow.

Bonus 3: This

special variable holds the name of your Perl program. On some operating systems, you can set it, and make your program seem like it's named something else. Give me the short name of the variable, and the long name provided by the English module.

Bonus 4: A correctly functioning Perl program ends with the following two lines.

```
}BEGIN{
    @ARGV=<*M*>
```

These lines are not part of a string. "Here" documents don't come into play, and there are no sneaky eval tricks. There are no funky characters after these two lines, nor is there a _____DATA___ or ____END___ previously. The program has no signal handlers, it does not launch another process, and it doesn't make strange use of symbol tables or otherwise play with itself. The program does not use any module or pragma.

These two lines are executed just fine. If you put a print before or after the @ARGV line, it's executed too.

What's the simplest explanation?

Bonus 5: There's a

freeware porn detector that spiders through the web and flags an image as pornographic if the amount of continuous skin tone exceeds this threshold, expressed as a percent. As it turns out, this is the exact same threshold that Perl uses to determine whether a file is binary or text for the -B and -Tflags. If more than this percentage of the characters are "odd characters" (like control characters or characters with the eighth bit set), it's deemed a binary file. Name this amount, within ten percentage points.

Bonus 6: Give me the two reasons why laptop CPU speeds are typically slower than desktop CPU speeds.

Bonus 7: Describe what this displays:

print scalar(localtime);

Bonus 8: According to the *St. Petersburg Times*, in what year is the

U.S. expected to use the Internet to collect Census data?

Bonus 9: Name that command-line switch.

- 1. You'd use this switch to execute a Perl program in a larger message, because it tells the interpreter to skip down until it finds the first line beginning with #! and containing the word perl.
- 2. You'd use this to append a newline automatically to every print statement.
- 3. You'd use this switch to verify that your Perl program is syntactically correct.
- 4. You'd use this switch to specify a directory where modules are located.

Bonus 10:

Netcraft compiles an annual web server survey, and compiles its results by polling every server it can find, worldwide. Within a factor of two, how many sites was that, for the 1999 survey?

Bonus 11: This is a bonus in five parts; as soon as you get one wrong, the question is over. If the derived class Rabbit inherits from the base class Rodent and you invoke Rabbit::Forage, what subroutine does Perl look for next? (*This question doesn't work so well in print, since each subquestion gives away the answer to the previous subquestion.*)

- 1. After failing to find it in the Rabbit class?
- 2. After failing to find it in the Rodent class?

- 3. After failing to find it in UNIVERSAL?
- 4. After failing to find a Rabbit::AUTOLOAD?
- 5. After failing to find a Rodent::AUTOLOAD?

Bonus 12: I'll describe five pairs of

built-in Perl functions. The last letters of the first function are the first letters of the

second function: you mash them together and give me the combined name. For instance, if I say "the function that adds something to the end of an array" and "the function that adds something to the beginning of an array," you'd say "pushift." The "sh" is shared.

- 1. The function that removes something form the end of an array, and the function that readies a directory for reading.
- 2. The function that converts a binary structure into regular Perl variables, and the function that declares a global namespace.
- 3. The function that displays a formatted record, and the function that reports where a file pointer is.
- 4. The function that clears all variables beginning with a specified letter, and the function that you'd use to keep your server from hanging onto a port after it's no longer used.
- 5. Finally, what two single-letter built-in functions combine to make a third built-in function?

Bonus 13: Tell me whether these statements about DBI, the Database Interface, are true or false.

- 1. Both ODBC and DBI provide a generic interface to multiple database engines.
- 2. DBI supports multi-threading.
- 3. DBI runs on Windows 95.
- 4. You can manipulate Microsoft Access database from DBI.

Bonus 14: Guess that module. I'll name five modules that might exist on

CPAN; you tell me whether they do.

- 1. Net::Video
- 2. WWW::Robot
- 3. Games::WordFind
- 4. Modem::Dial
- 5. Tie::BikePower

Bonus 15: What does this do?

goto (qw(alpha beta gamma))[rand 3];

The rest of the question requires audience participation. I contend that goto is used far more often than people admit. I'll ask the crowd to applaud if they've used goto in the last month, and then if they've used redo in the last month. Tell me whether the amount of applause will be significantly higher for goto, significantly higher for redo, or about the same.

Bonus 16: What country has the largest number of top-level domains ending in its two-letter country code? Hint: it's not the U.S., since there aren't that many .us domains.

Bonus 17: There are four steps to building and installing a CPAN module on a Unix system once you've downloaded and unpacked it. What are the four steps? Each step is something you type at the command line.

Bonus 18: What CGI

environment variable can you use to determine which browser is visiting your pages?

Bonus 19: To the nearest ten percent, what proportion of Internet users use English as their primary language, according to a survey by Global Reach?

Bonus 20: I'll give you four Perl constructs; you tell me whether they're executed at

run-time or

compile-time when they appear by themselves in Perl programs.

- 1. require statements
- 2. BEGIN blocks
- 3. use statements
- 4. srand

Bonus 21: Here's a

palindromic program; it reads the same backwards as forwards. What will it display?

print \$;,(1 => "able was i ere i saw elba"
<= 1),;\$ tnirp</pre>

Bonus 22: Which of these

HTTP status constants exist? Tell me yes or no for each.

1. HTTP_NOT_FOUND

- 2. HTTP_GONE
- 3. HTTP_PAYMENT_REQUIRED
- 4. HTTP_TOO_MANY_USERS
- 5. HTTP_LANGUAGE

Finally, what is the numeric HTTP status code for HTTP_NOT_FOUND?

Bonus 23: Name two ways to embed comments in regular expressions.

Bonus 24: Tell me whether the following snippets are legal or illegal.

1.	sub _ { print 4 }			
2.	\$\$ = 17;			
3.	if (2 < 3 < 4) {	print		
4.	open (M,		"mail	
	<pre>president@whitehouse.gov");</pre>		print	Μ,
	"Release Mitnick die!";			

Bonus 25: You have a function that accepts either one or two arrays, and you want those arrays to be implicitly passed by reference. That is, if you're passing in <code>@red</code> and <code>@blue</code>, you want the advantages of

passing by reference but you don't want to have to include the backslashes. What prototype should you use?

Bonus 26: Four languages: ADA, Basic Plus, C, and Fortran. I'll give you four Perl features; you tell me which of the four languages it also appears in. 1. ++ and --2. ** 3. :: 4. ... if

Bonus 27: What are the three types of System V IPC structures?

Bonus 28: What meaningful difference, if any, is there between how these two function invocations pass arguments to the blork subroutine?

```
blork();
&blork;
```

Bonus 29: I'll give you five built-in Perl functions; you tell me the maximum number of arguments it accepts. I'm counting lists and expressions as single arguments, so for instance push takes two arguments: an array and a list.

- 1. open
- 2. tie
- 3. substr
- 4. splice
- 5. split

Bonus 30: Perl programmers love their weird symbols. I'll give you four esoteric symbol names; you tell me a more common name for that character. For instance, if I say "ampersand," you'd say "and."

2. Octothorpe

Solidus

- 3. Quadrathorpe
- 4. Pilcrow
- 5. Lemniscate

Bonus 31: Perl has a somewhat unearned reputation for being hard to read. We can all

show how clear it can be by writing lucid programs. But we could also show how murky it isn't by comparing it to something worse. There's a language called

Befunge that is two-dimensional, with a stack and an instruction pointer that moves character by character through the program. Here's a program that prints "Hello, world."

```
0".dlrow , olleH">v
,:
^ @
```

At this point, I explained how Befunge worked. Full details are at http://directory.google.com/Top/Computers/ Programming/Languages/Befunge/, but you can get the gist of it from the vcharacter. In the program above, the characters are processed from left to right (pushed down on an instruction stack) until that vis encountered. Then the characters are processed from top to bottom, until _is encountered. _is a "horizontal if," which sends control to the right or the left depending on the contents of the stack.

What will this print?

>v" ^,<u>@</u>

Bonus 32: Answer the question posed by this visual code snippet:

```
#!perl
# What does this print?
seek(DATA, 7, 0);
print scalar <DATA>;
___END_____This is
some test
data for
your program's
end.
```

Bonus 33: This

HTTP header field is used by a browser to avoid downloading a web page that is already in its cache. The web server compares the value of this field to the last time the web page changed, and returns it only if the web page is more recent. Name the field.

Bonus 34: What does this print? I'll give you some extra time.

perl -e "print substr(**.**, 7, 2)"

Bonus 35: XML documents often have associated DTDs, or

document type definitions. I'll give you four DTDs, you tell me if they exist in any of the big DTD repositories on the Net.

1. Music Markup Language

- 2. HTML
- 3. News stories
- 4. Real estate listings

Bonus 36: What does the following print?

```
perl -wle 'print +(8/2).".".0.0.0'
1. 4
2. 4.
3. 4.0
4. 4.00
5. 4.000
6. 4.0.0.0
```

Bonus 37: I'm going to give you some Perl features. Tell me which version of Perl added it: 1, 2, 3, 4, or 5.

- 1. The ability to handle binary data in strings
- 2. Henry Spencer's regular expression package
- 3. Support for object-oriented programming
- 4. The ampersand before function names became optional

The Answers

As with the previous quiz, the mandatory part of the answer is given in capital letters.

Toss-up Answers

T1.

WEBBY. Not one, but two recipients held their trophy aloft and declared, in an embarrassing imitation of *Titanic* director Jim

Cameron, "I'm the king of the World...Wide Web!"

T2. PSEUDOHASH

T3. The ONION

T4. \$/ or \$INPUT_RECORD_SEPARATOR. (It's a single newline by default. If you set it to the empty string, Perl will read your file in as paragraphs.)

T5. 9600 baud

T6. QUICKSORT

T7. Peter JACKSON

T8. PERLSCRIPT **T9**. DUNGEONS AND DRAGONS.

T10. 1. (pi is undefined because use constant only accepts a single assignment. For extra credit, try to prove which is greater, e to the pi'th power, or pi to the e'th power, without resorting to a computer or calculator.)

T11. FIRST COUSINS (This assumes that Gore is father of the Internet. In his defense, he never actually claimed to be.)

T12. 2038

T13. 0.1 << 5 is 32, so the whole expression is 2 to the 32. Even though you can print 2 * * 32 just fine,

Perl uses 32-bit integers when you're bit shifting, so the ones get shifted off the left end into oblivion.

T14.

DNA. (The work was done by Len Adleman, who's the A of RSA.)

T15. T or BASETIME

T16. | | =

T17.

FINLAND or SUOMI. (Source: Win Treese's Internet Index.)

T18.

DATA::DUMPER

T19.

IETF or

INTERNET ENGINEERING TASK FORCE

T20.

APOSTROPHE

('). (You can even say use LWP'Simple, but that looks screwy because when Perl 5 introduced use it also introduced the double colon syntax.)

T21.

PHYSICS, won by Martin Perl. (The

tau lepton is identical to the electron, but weighs 3,500 times as much and survives less than a trillionth of a

second. According to the Standard Model of particle physics, the elementary building blocks of matter appear in families, with two leptons and two quarks in each. The tau lepton is the first-known member of a third family. The other members are the bottom and top quarks.)

T22. SUCCESS. (That = should be ==.)

T23.

HALF-LIFE. (Someone about five or six rows back must have been a huge Half-Life fan, because he went absolutely spastic as I read the question.)

T24.

H2XS. (See the *perlxstut* documentation for more information.)

T25. APACHE and MICROSOFT. (Apache had 55% and Microsoft had 22%. Netscape was third with 7%.)

T26. It executes CODE or EVALuates code.

T27. (Who says Perl programmers don't care about speed?)

T28. The

LOCALE pragma. (See the *perllocale* documentation bundled with Perl for more information.)

T29.

REFRIGERATOR. (From the FAQ at http://www.hamjudo.com/cgi-bin/refrigerator: The Coke can is colder than the fridge itself because it's on the top shelf near the coils. People always assume that fridge temperatures are uniform, but that's never the case.)

T30.

TONGA

T31.

EG or EXEMPLI GRATIA. (This directory is no longer part of the Perl distribution.)

T32.

KERNEL

T32-Bonus. MONO, MICRO, MONO, MICRO

T33. \$^M

T34.

MP3. ("pokemon" was number 6.)

T35. USELESS

USE OF A CONSTANT in void context. (Mark Jason Dominus said: "The numbers 0 and 1 are exempt from this warning. I looked in the source code, and discovered three other exemptions: Strings beginning with 'di', 'ds', or 'ig' do not trigger this warning."

These exceptions were for the benefit of the *wrapman* program. They allowed you to include documentation in Perl programs—the pod format hadn't been invented yet—and

feed the programs to either Perl for execution or *nroff* for documentation.)

T36. \$915.42 BILLION. (Accept between \$823.878 BILLION and \$1.006962 TRILLION)

T37. \$@ or \$EVAL_ERROR

T38.

GAMBLING. (The Kyl bill also makes it a crime to provide information about how to use the

Internet for gambling. The Senate passed it 90–10, and it's now languishing in a House committee. By the way, several members of the U.S. Supreme Court have a regular poker game.)

T39. The LINE directive.

```
# line 300 "camel"
die;
Died at camel line 300.
```

T40. 186,000 (Only three significant digits required.)

T41. \$SIG{TSTP} or \$SIG{INT}.

T42. YEARS. (About four of them). The Library of Congress has about 20 terabytes (160 terabits) today.

```
T43. $^S
T44.
PERCEPTRON
```

T45.

PERLE systems, named after CEO Joseph Perle. (Correct spelling not necessary.)

T46. It closes the CURRENTly selected FILEHANDLE. (*That's usually STDOUT*.)

B1. ∖A

B2. FFFF00 (Or something close to it; FFFF66 looks pretty good too. The red and green values need to be close to each other and nearly maxed out; blue should be anywhere from 00 to 88. During the quiz

show, I showed pictures of the resulting colors.)

 $B3.\$ \$0 and \$program Name

B4. The Perl program used the -n or -p flags. When the documentation says that they wrap a loop around your code, it means that literally: Perl provides the opening and closing braces that make this program syntactically correct. (This trick is used in Chapter 46.)

B5. 30% (accept from 20% to 40%)

B6.

HEAT dissipation and

POWER consumption. (The Boston.pm team got both answers correct; impressive for such a poorly-worded question.)

B7. The current date and time in English.

B8. 2010. (Lisa

Nyman is a frequent Perl conference attendee who hacks Perl for the U.S.

Census. As I'd hoped, she was in the audience; I could tell, because she yelled out "WRONG!" as soon as I read the answer. Turns out that the Census will be accepting some online forms for the 2000 Census as a test. The team answered 2010, so I awarded full credit. If memory serves, the team was all Norwegian, so after posing the question I added that the

U.S. census happens every decade on the decade.)

B9.-x, -1, -c, -I

B10. 7,078,194. Accept between 3,539,097 and 14,156,388.

B11. Perl looks in: a) RODENT::Forage, b) UNIVERSAL::Forage, c) RABBIT::AUTOLOAD, d) RODENT::AUTOLOAD, e) UNIVERSAL::AUTOLOAD. After that, you could say it looks in \$SIG{__DIE_}}.

B12. POPENDIR, UNPACKAGE, WRITELL, RESETSOCKOPT, M, and Y

B13. TRUE, FALSE, TRUE (there's also a Win32::ODBC module), TRUE (you use the DBD::ODBC module)

B14. NO, YES, YES, NO—although there is a Win32::Serial module, YES (Tie::BikePower calculates power output and power consumption for bicycling. You give it things like riding speed, body weight, hill grade, and wind speed, and it shows you your power output and consumption.)

B15. Jumps to one of the three labels alpha, beta, and gamma at random. (FORTRAN also lets you compute gotos. For the audience participation question, the applause levels were about the same.)

B16. GERMANY. (The top four are shown below. There are 9.2 million registered domains, depending on who's paid their Internic fees this week. 5.5 million are .com sites.)

```
.de (Germany)
391,113
.uk (United Kingdom)
360,821
.au (Australia)
122,201
.dk (Denmark)
93,181
```

B17. perl Makefile.PL, make, make test, make install

```
B18. HTTP USER AGENT
```

B19. SIXTY percent. (Accept between FIFTY and SEVENTY.) European non-English, 25.5%; Asian languages, 15.5%.

B20. RUN, COMPILE, COMPILE, RUN.

B21. 11. (\$; is chr(28), => is a comma, a string evaluates to zero in a numeric context, and the next comma is followed by a null argument to print. \$ is a no-op, and tnirp is a bareword.)

B22. YES, YES (*it means that the document has been permanently removed*), YES, NO, NO, 404.

B23. The /x modifier and (?#comment)

B24. LEGAL, LEGAL but ineffective, ILLEGAL, ILLEGAL. (It's illegal to threaten the president, and there's a comma after the filehandle in the print statement. There's

also no pipe symbol before mail, and the @ isn't backslashed. Whoever wrote this should be in prison.)

B25. \@; \@ (Tom

Christiansen pointed out that prototypes should really be called "input context templates,"

because they're really not like prototypes in other languages.)

B26. C, FORTRAN, ADA, BASIC Plus

B27. SEMAPHORES, MESSAGE QUEUES, and SHARED MEMORY.

B28. blork() has no input arguments; &blork ends up with the input arguments from its caller.

B29. 2 (filehandle, expression. You can omit the expression!), 3 (variable, classname, list), 4 (expression, offset, length, replacement), 4 (array, offset, length, list), 3 (pattern, expression, limit).

B30. POUND or HASH or TICTACTOE, SLASH (it's also called a "virgule"), EQUAL sign (two quadrathorpes form an octothorpe), PARAGRAPH symbol, INFINITY symbol.

B31. P+e (This was inspired by Chris Howe's entry for the Obfuscated Perl Contest in Chapter 46, which printed out "The Perl Journal" by creating a Befunge interpreter and running a Befunge program that printed "The Perl Journal" through it. He did this in exactly 1,000 characters.)

B32. # WHAT DOES THIS PRINT? (Seeking the DATA filehandle can take you anywhere in the program. Nathan Torkington pointed out that you can even stat(DATA) to find out when the program was last modified.)

B33. IF-MODIFIED-SINCE

B34. **. (**.** looks like a shell glob, but it's actually a **

typeglob. ** is *main::*, so **.** is
*main::**main::*, and the eighth and ninth characters
of that are **.)

B35. YES, YES (HTML 4.0), YES, YES

B36. 4.00 (The subexpression 0.0.0 is actually the float 0.0 concatenated with 0. But 0.0 is the same as 0, hence we only get 2 0's.)

B37. 3, 2, 5, 5.

Thanks to Tom Christiansen, Chris Nandor, Abigail, Mark Jason Dominus, Nathan Torkington, and Jarkko Hietaniemi, all of whom provided a few question ideas (sometimes inadvertently).

Chapter 33. The Third Perl/Internet Quiz Show

Jon Orwant

Here are the toss-ups and bonus questions from the Third Internet Quiz Show, held in the summer of 2000 at the O'Reilly Open Source Convention. Answers are at the end of the article.

Three questions in the contest refer to "Kyle," the captain of my College Bowl team whom I described briefly when the quiz show began. I told a story about Kyle to make a point about why, during the 1999 quiz show, I discarded a question because I knew that one player had prior knowledge of the answer.

The abridged story was this: When Kyle walked into the classroom where our College Bowl match was being held, he noticed ".367" written on the chalkboard. He said, to no one in particular, "Hey, what's Ty Cobb's lifetime batting average doing on the board?" Later, in the middle of the round, the emcee discarded a question because it relied on that statistic, causing us to lose the round.

As with previous quizzes, you can keep score by giving yourself one point per question and using the ranks at the beginning of Chapter 31.

Toss-up Questions

Toss-up 1: This company hired David

Boies to defend it against a lawsuit by the RIAA. What is this company, which made Metallica even angrier than they normally are?

Toss-up 2: This computer pundit used the phrase "open sores" in his InfoWorld column. He was one of the architects of Ethernet and founded 3Com. Who is this bitter, bitter man?

Toss-up 3: It's the look or overall style of an

XUL file, and they're created with cascading stylesheets and images. What is this four-letter word used by the Mozilla project to separate appearance from content?

Toss-up 4: This well-known piece of software got its name from the combination of two English words that, when spoken aloud, sound the same as a non-English word. One of the words derives from the word that's used to apply a diff to source code. Name this web server.

Toss-up 5: VA Linux invested in this

free database system, which now supports transactions and is used by Slashdot. The system wasn't released recently, but it was recently changed to be under the

GNU General Public License. What is this database system, which has one letter more than mSQL?

Toss-up 6: It takes a lot of input: audio, S-video, cable, antenna, and a phone line. It uses embedded Linux and MPEG II to store video for up to 30 hours. What is this four-letter

consumer electronics device that lets you pause live TV?

Toss-up 7: It is "just too complex to ever be secure," wrote Bruce

Schneier, author of the

CRYPTO-GRAM newsletter. "What happens when somebody uses modifier characters in an unexpected way, or someone uses UTF-8 or UTF-16 to encode a conventional character in a novel way to bypass validation checks, or we start attaching semantics to characters like delimiters and whitespace?" Name the international character set he's describing.

Toss-up 8: In *The Hitchhiker's Guide to the Galaxy*, it was an animal you inserted into your ear to understand alien languages. It's also the name of an AltaVista service that translates between human languages, often with unintended humor. Name this fish.

Toss-up 9: This device is capable of transmitting speech over wires with low bandwidth, equivalent to about eight kilohertz, and with only one channel. Name this device, which in the U.S. used to be all black.

Toss-up 10: They own more trucks than any other entity in the United States, and they plan to provide an email address for every American, so we can have even more disgruntled postmasters. What is this quasi-private U.S. agency known for its eagle logo?

Toss-up 11: A procedural fraud, a technical failure, a basic misconception. Those are the subtitles of sections 2, 3, and 4 of a recently created document aimed at convincing people to reject a protocol. Name this protocol for the transmission of wireless data.

Toss-up 12:

ICMP is the protocol used by this command to find out if a networked computer is alive. What is this command, which derives its name from sonar?

Toss-up 13: On "The Simpsons," what recurring character has a PhD in computer science from MIT? Name this employee of the Kwik-E-Mart.

Toss-up 14: It piggybacks a

DOM, or Document Object Model, onto HTTP so that it can penetrate

firewalls. What is the name of this technology, which shares its name with something you can find in bars?

Toss-up 15: The

Apache/Java Integration Project is named after what capital of Java?

Toss-up 16: There are two

nybbles in each. What is this common computer word, most of which have eight bits?

Toss-up 17: This is a new programming language developed by Microsoft, and is included with Visual Studio.NET. What is this language, which shares its name with a musical note?

Toss-up 18: The name of his temple at Delphi is Pytho, which derives from the word Python. Who is this Greek sun god?

Toss-up 19: Give me the name of the potbellied penguin who is the mascot of Linux.

Toss-up 20: *Bloomberg News* said that they're getting close to an IPO, which should make Linus Torvalds happy, since he works for this Bay Area company that makes power-saving CPUs. Name that company.

Toss-up 21: You hear about a bauble called the Amulet of Yendor rumored to be in the Valley of Gehennom deep within the Mazes of Menace. What is this descendant of Rogue and Hack, which saw a Version 3.2 release in April of 1996?

Toss-up 22: It has over six thousand projects and forty thousand registered users. What is this VA Linux–sponsored central resource and repository for open source development?

Toss-up 23: This may be the most popular Tcl program. What is this utility that enables preprogrammed interchanges with a network service?

Toss-up 24: This system is best known for its ci and co commands, which check files in and check files out of a repository. Name this three letter utility for controlling revisions.

Toss-up 25: This protocol is ideal for storing information that is read frequently and updated less frequently, for instance, company directories. Your web browser might use this protocol to make use of a *de facto* company address book. Name this four-letter protocol.

Toss-up 26: One of the great things about editing the *Perl Conference Proceedings* is that I finally got to learn what absolutely everything on a copyright page means. O'Reilly now uses Cataloguing in Publication data for its books, so that copies are shelved in the U.S. Library of Congress with a Library of Congress

shelving number. What two letters will you find at the beginning of the Library of Congress numbers for nearly all computer books? The two letters also identify a group you'll find in software companies, which audits programs and ensures that they're bug free.

Toss-up 27: This field involves the application of computational techniques to biology. Name this discipline known primarily for its contributions to the Human Genome Project.

Toss-up 28: The half brother of Thor and scion of Odin, this god shares his name with a company that ported games to Linux, notably Railroad Tycoon and Civilization 3. Name this Norse god of mischief.

Toss-up 29: This

API for XML is simpler than DOM. What is this three-letter acronym that is also an abbreviation for a musical instrument?

Toss-up 30: This

network protocol is now specified in the IEEE standard 802.3, even though it's been around longer than some programmers have been alive. It was originally developed at Xerox, and a typical LAN using this protocol uses coaxial cable or twisted pair. What is this technology, which often goes hand in hand with 10BASE-T?

Toss-up 31: In 1980, he took home the Olympic silver medal in

archery. In a recent rant called "Systems Software Research is Irrelevant," this system architect and writer lamented the dearth of innovation in operating system design, taking even Linux to task as not being interesting enough. He co-authored the software engineering treatise "The Practice of Programming" with Brian

Kernighan, and is one of the creators of the Plan 9 operating system. Name this Bell Labs researcher.

Toss-up 32: This company hired private investigators to root through trash bins for dirt on

Microsoft. What is this company known for its 8*i* database?

Toss-up 33: It's the

web crawler that

AltaVista uses to collect the content of 3 million web pages per day, and shares its name with a vehicle that looks like a skateboard with a steering handle attached.

Toss-up 34:

XSL lets you choose the style of

numbering for things like ordered lists. For instance, you can use Roman numbering, Arabic numbering, Katakana numbering, and, if you use the attribute value

𐴐, what style, which is also a style of calendar and a style of chant?

Toss-up 35: These programs work by spoofing an IP address with packets that contain an

ICMP ping message addressed to an IP broadcast address. The echo responses to the ping message are sent back to the "victim" address. Enough pings and resultant echoes can flood the network, making it unusable for real traffic. What is the name of this denial of service attack, which shares its name with a village of stunted blue characters on a once-popular Saturday morning cartoon?

Toss-up 36: You can think of it as being HTML 5.0, although it's not officially called that. What is the name of the reformulation of HTML 4 in XML?

Toss-up 37: "Hello" is "danzhon," "duck" is "nachlele," and "bear" is "shash." I just gave you three words in this family of Indian languages. What is this language family, which shares its name with a web server we're all familiar with?

Toss-up 38: The basic routing philosophy on the Internet is "best-effort," which serves most users well enough but isn't adequate for the continuous stream transmission required for video and audio programs over the Internet. With the protocol, people who want to receive a stream of

_____ protocol, people who want to receive a stream of video or other high-bandwidth messages can reserve

bandwidth in advance of the program and be able to receive it at a higher data rate than they'd normally be able to. It's part of the

Internet Integrated Service (IIS) model, which ensures both best-effort service and

real-time service. Tell me the name of this protocol, which is four letters you might see on a party invitation.

Toss-up 39: Invented in 1960, the

RS232 serial port comes in two common flavors: the DB-9 and DB-25

type connectors. On both, pin 2 is send, or transmit. What's pin 3?

Toss-up 40: Andrew

Tridgell is known for work developing

Samba, the tool that allows Unix to mount and read various Microsoft filesystems and exports Unix filesystems to Windows clients. But before this bit of file transfer magic, he was better known for creating an open source utility that helped synchronize copies of files on various remote machines. What is the name of the utility?

Toss-up 41: It's a set of

programming interfaces you can use to build Mac OS X applications that also run on Mac OS 8 and 9. What is this technology, which shares its name with the sixth element of the periodic table?

Toss-up 42: This standard derives from

PICS, the Platform for Internet Content Selectivity, and the mysteriously named

Dublin Core. It uses XML to describe a resource in terms of properties. What is this three-letter abbreviation of the Resource Description Framework?

Toss-up 43: According to the *Los Angeles Times*, a dotcom called

Scour scans the Internet for unprotected _____

access ports, opens a file connection, and indexes your hard drive for public access. Fill in the blank. It's a three-letter word, and Samba got its name from it.

Toss-up 44: It's versatile; you can tell by its middle initial. This medium can hold up to 17 gigabytes of information, and uses

MPEG-2 to encode video. Consumers assume the V in its name stands for video, but it actually stands for versatile. What is the three-letter name of this popular movie medium?

Toss-up 45: XML, UML, XSL, DTD. Which of these has nothing to do with document markup?

Toss-up 46: I find Perl perfectly charming, but when you charm a python with music, it's important to wave your hands back and forth. Which of the six senses do pythons lack?

Toss-up 47: His web site bears the likenesses of such innovative geniuses as Thomas

Edison, Nikola

Tesla, and Albert

Einstein. He has posted a one million dollar reward for anyone who can "come up with a better health device" than his patented

Immortality Rings. Even under fear of an FDA raid, he claims that he is not "one of those stupid moron [sic] who don't know what I am doing." He also explains black holes in his penetrating article "Black hole is no magic." While explaining how to build a UFO, this inventor notes that a "UFO's saucer-shaped body is actually a big gyroscope which rotates endlessly to keep the UFO balanced in the air." Finally, his work became known to citizens of the U.S. in his seminal interview with Mo

Rocca for Comedy Central's "The Daily Show."

Name this Chinese inventor, whose last name is half of the sound a train makes.

Toss-up 48: This common Unix utility doesn't generate art, but it's an easy way to produce large signs composed out of ASCII letters. Name this utility.

Bonus Questions

Bonus 1: Rank the following protocols having to do with email, from oldest to newest: IMAP, SMTP, and POP.

Bonus 2: In his InfoWorld column, Bob Metcalfe referred to open source software as

"open sores" software, S-O-R-E-S. So now we'll test your knowledge about open sores. I'll give you

four definitions all at once, and you match them up with boil, lesion, cyst, and wart.

- 1. A horny projection on the skin, usually of the extremities, that is caused by a virus.
- 2. A closed sac having a distinct membrane and developing abnormally in a cavity or structure of the body.
- 3. A localized swelling and inflammation of the skin resulting from the infection in a skin gland, having a hard central core and forming pus.
- 4. An abnormal change in structure of an organ or part due to injury or disease, especially one that is circumscribed and well defined.

Now match them up: boil, lesion, cyst, and wart.

Bonus 3: I'll give you three slogans; you tell me the company it's associated with.

- 1. Where do you want to go today?
- 2. The Right Choice.
- 3. Supermarket to the world.

Bonus 4: Here it comes: the

obligatory product placement for our sponsor, ActiveState. Yes, ActiveState, friend to you and me. ActiveState, helping to make the world a better place to live in, and bringing you tomorrow's technology today. ActiveState, the dynamo powering the longest period of growth in the history of the U.S. economy. Even though they're Canadian. Because the ActiveState revolution *transcends national boundaries*, ladies and gentlemen. Oh, ActiveState, hallowed be thy name. We thank you for your clothing <<<<hod>we then how you are ternal fountain of knowledge <<<<hod>we then how you?we then how you are ternal fountain of knowledge <<<</td>we then how you?we how you?<

Anyway, ActiveState's motto is "Programming for the People." Answer the following questions about these related phrases.

- 1. Name the rock group that released the album "Automatic for the People" in 1992.
- 2. There was a 1965 TV series called "For the People" starring this man as an obsessive New York City assistant district attorney. He later went on to record a cover of "Lucy in the Sky with Diamonds" and authored a science fiction novel called *Tekwar*.
- 3. The Discovery biography of this man was also called "For The People," even though he had little in common with William Shatner, being the dignified and scrupulous sixteenth president of the United States.

Thanks to Dick Hardt and Lori Pike for having a sense of humor about this, as well as the other ActiveState question later.

Bonus 5: There's a new

game console coming, and it will compete with Sony, Nintendo, and Microsoft's X-Box. It uses embedded Linux and a network connection to allow anyone to be a game designer and release games for free. Name this consumer console. [*Editor's Note: the console was never manufactured.*]

Bonus 6: Kyle had to install a lot of memory in computers. I'll give you three memory chips, and you tell me the number of pins each has.

1. A

non-parity SIMM; the number of pins it has is a multiple of 24.

2. A

non-parity DIMM; the number of pins it has is a multiple of 24.

3. The Apple ASIC, commonly called an ASIMM, used in the Mac 2ci; Kyle pointed out that they worked in old PC XTs. The number of pins it has is a multiple of 30.

Bonus 7: Every Perl programmer worth his salt knows that the world's compendium of Perl programs is

CPAN, the Comprehensive Perl Archive Network. Obviously the Python folks couldn't call theirs the Comprehensive Python Archive Network. What *did* they call it?

Bonus 8: I'll give you four descriptions. You give me the

evil name of the thing I'm describing. For instance, if I said "A version of the Ultima game," you'd say UNDERWORLD, because UNDERWORLD has to do with evil.

- 1. A process that is invoked when some event occurs.
- 2. A sysadmin tool for finding security bugs in networks.
- 3. In Unix, an unwanted process left after a program terminates.
- 4. A security system that uses tickets and realms.

Bonus 9: In XML, you can define special marked sections of character data that XML processors won't attempt to interpret as markup. You do it with this five-letter tag. What is it?

Bonus 10: I'll give you four descriptions of types of DSL; you tell me their acronyms.

- 1. Home users tend to receive more data than they send, so this style of DSL has a slower upstream speed than downstream speed.
- 2. This style of DSL has a maximum of 144 kilobits per second.
- 3. This style of DSL is guaranteed to only need a single twisted pair line, and has equal upstream and downstream speeds.
- 4. This name for this style of DSL was officially adopted in June of 1995, and it offers a higher speed than traditional ADSL: up to 50 megabits per second.

Bonus 11: Red Hat, Slackware, or Caldera. Which Linux distribution appeared first?
Bonus 12: The common phone jack you find on most phones is called an RJ11 jack. There are other

RJ jacks. Which is the one commonly used for Ethernet connections?

Bonus 13: I'll give you two acronyms related to computer sales, and you expand it.

- 1. OEM
- 2. VAR

Bonus 14: Closed-caption information is sent during this period between frames of a TV broadcast. In Europe, there are some channels that carry IP packets and others that carry MP3s. What is this three-word name for this interval, which you can actually see when your TV loses its vertical hold?

Bonus 15: What does

BSD stand for? What was the first BSD project aimed at providing Unix on the Intel x86 architecture: FreeBSD, NetBSD, or 386BSD?

Bonus 16: Here's another open sores question. A 17th century theory of physiology held that the state of health and mind depended among a balance of four elemental fluids, called the

humors. I'll describe each of

the four humors, and you name it from the description.

1. This

fluid is associated with being cold and moist, and an excess makes you dull, pale, and cowardly, like Bill Gates. It's the phlegmatic fluid.

- This fluid is associated with being hot and moist, and an excess makes you happy and generous, like Larry Wall. It's the sanguine fluid.
- 3. This fluid is associated with being cold and dry, and an excess makes you gluttonous, lazy, and sentimental, like Marlon Brando. It's the melancholic fluid.
- 4. This fluid is associated with being hot and dry, and an excess makes you violent and vengeful, like Mike Tyson. It's the choleric fluid.

Bonus 17: I'll give you three catchphrases; you tell me the company it's associated with.

- 1. We're the dot in dot com.
- 2. Think different.
- 3. Inspiration becomes reality.

Bonus 18: Penguins are the most diverse group of flightless birds in the world. There are currently 17 species of penguin. Tell me how many of those species live above the Arctic circle.

Bonus 19: Pencil and paper ready for a product placement bonus. I'm going to name six projects. ActiveState is doing three of them. You tell me which three.

- 1. PerlMX, an add-on product for Sendmail that lets a site archive and filter mail, rewrite content, and control
- 2. spam.

Visual Python, a plug-in for Microsoft's next-generation development environment, Visual Studio 7.

- 3. VerticalBribe, a B2B web site where software agents broker deals between quiz show hosts and Vancouver-based software companies who think that just because they bought a platinum sponsorship of the quiz show, they can bribe the quiz show host with free T-shirts and beer in exchange for being able to submit quiz show questions, but instead get a quiz show host who uses his bully pulpit to draw attention to the bribe with oblique and self-referential questions.
- 4. Komodo, a cross-platform development environment for scripting languages based on the Mozilla framework.
- 5. Gecko, an open source eugenics project to breed Perl programmers for toil in Canadian coding mines.
- 6. RPCom, a peer-to-peer framework for sharing executable bits of Perl code across the network anonymously.

Bonus 20: I'll name three projects, you tell me the GUI toolkit most closely associated with it.

- 1. GNOME
- 2. KDE
- 3. Mozilla

Bonus 21: Kyle worked in computer repair, and had to lift a lot of printers. I'll give you three

HP printers, and you rank them from lightest to heaviest. The three printers are the HP Color Laserjet 2100, the HP 4050, and the HP Laserjet 5si.

Bonus 22: In the "WAP Trap," the author includes the following quote: "Sometimes when you fill a vacuum, it still sucks." This was first uttered by what founder of Sun Microsystems?

Bonus 23: In Mozilla, this word refers to the skin, content, and whatever localization and platform-specific files are necessary for a particular

part of the application or window. Name this five-letter metallic word.

Bonus 24: The O'Reilly *Smileys* book lists 650 smiley faces, ranging from barbershop quartet singer to Charlie Chaplin to Bugs Bunny with a carrot to a drunk, devilish chef with a toupee in an updraft, a mustache, and a double chin. The traditional ASCII smiley face has a colon, a hyphen, and a right parenthesis. I'll give you three descriptions of smileys, and you name another character it has. For instance, if I said 'A sad smiley,' you'd say 'left parenthesis.'

- 1. A winking smiley
- 2. Smiley with glasses
- 3. The apathetic smiley
- 4. Man wearing a dunce cap
- 5. Dolly Parton

Bonus 25: For the first time, the

U.S.

Census collected forms over the Internet. There are

about 270 million people in the U.S. To the nearest power of ten, how many forms were received online, excluding duplicate submissions and I'm looking at you, Chris Nandor.

Bonus 26: In olden days, there were seven liberal arts recognized as being part of a classical liberal education. They were divided into a group of three and a group of four. The group of three are called the

Trivium, and you can think of them as fields that prepare you for a job as a liberal arts professor. The group of four are called the

Quadrivium, and you can think of them as fields that prepare you for a job at Starbucks. I'll give you all but one of each grouping, and you give me the missing field.

- 1. The Trivium consists of Grammar, Rhetoric, and what third component?
- 2. The Quadrivium consists of Geometry, Arithmetic, Music, and what fourth component?

Bonus 27: Slashcode, the code behind Slashdot, is used to run this site, which has as its slogan "All the Perl that's Practical to Extract and Report."

Bonus 28: Which HTTP header contains a misspelling?

Bonus 29:

O'Reilly product placement time. I'll give you seven books, you tell me what animal is on the cover. *Programming Perl, Programming Python, Y2K in a Nutshell, ASP in a Nutshell, Sendmail, MP3: The Definitive Guide, Apache: The Definitive Guide.*

Bonus 30: Answer these question about Plan 9.

- The Plan 9 file server uses a WORM (write once read many) device for file storage. At a particular time every morning, Plan 9 backs up the filesystem to the WORM device. At what hour of the morning does this occur?
- 2. The Plan 9 windowing system creates each window in a separate namespace, so each window is similar to a shell environment. The windowing system has a number as a name, but the number isn't 9; it's a fraction that rounds up to 9. What's the number?
- 3. Plan 9 comes with embedded support for parallel programming, including a language for concurrent programming. The name of this language sounds the same as the first Hebrew letter.

Bonus 31: This was the year the computers came together, as we learned in the Terminator. It was also the year that HAL, from the movie *2001*, was created, on January 12. What was this recent year?

Bonus 32: Every

Apache web site includes three directories below the Apache root directory. Name them.

Bonus 33: JavaScript, Perl, PHP, and Python. Two of these languages support C and C++ style comments. Which two?

Bonus 34: Kyle had to study for the

MCSE Exam, or Microsoft Certified Systems Engineer. Poor Kyle. I looked at the O'Reilly MCSE books for the first time last week, and was actually surprised at how much good trivia it had. That makes sense, since standardized exams about

computers really have to be about trivia. Here's a question that you're told to memorize the answer to for the TCP/

IP test. You can tell the class of an IP address by looking at its first byte. Tell me what number between 0 and 255 separates addresses into class A and other addresses. You have to be within two.

Also, tell me what number between 127 and 255 separates class C addresses from Class A and B addresses. You have to be within one.

Bonus 35: The ISO 3764 standard for "timekeeping instruments—movements—types, dimensions and nomenclature" costs \$35, and I find it pretty offensive that standards aren't freely available. The standards for C, C++, Cobol, Common Lisp, and SQL all cost the same amount, and they're all cheaper than \$35. Tell me the price of these standards, which is the same as the price of two hours of Internet access at the Internet cafe near the conference in Monterrey, and is the price of one year of *The Perl Journal* for U.S. subscribers.

Bonus 36: President

Clinton signed a bill that allows electronic signatures to be as valid as conventional paper signatures. To sign the bill, he used a magnetic card and his personal password. The commander in chief of the armed forces of the United States of America and leader of the free world used a plain five-letter word, all in lowercase, and it was the name of his dog. Guess the President's password.

Bonus 37: It's a penalty in American professional football, and it's also the name of a process in graphics where parts of an image are occluded from view. Name this action.

Bonus 38: The country of Java has lots of volcanoes. When a volcano dies, a lake will sometimes form just below the mouth. What are such lakes called? They share their name with a distribution of Linux.

Bonus 39: I'm about to put one of these stickers on my car. Made by ThinkGeek, it makes it seem as though you come from the country of Perl. What three letters does the sticker have?

Bonus 40: I'll give you five countries, you tell me the two-letter country code for it. For instance, if I said Canada, you'd say "ca." Ireland, China, Switzerland, South Africa, Tuvalu.

Bonus 41: Most programming languages have an "elsif"

construct. If "this," do that, elsif "this other thing," do something different. I'll give you three languages, you tell me precisely how "elsif" is spelled in that language, including spaces if any. Perl, Tcl, Python, PHP, JavaScript.

Bonus 42: Let's play guess that bogus patent! I'll give you five

patents, and you tell me whether the

United States Patent and Trademark Office granted it.

- 1. One-click ordering
- 2. Hyperlinking
- 3. Hyper-light-speed antennas
- 4. Web page downloading
- 5. The compression underlying GIF

Bonus 43: We all know that "bit" stands for "binary digit": a value that can be either 0 or 1. There's no reason you couldn't build a computer system out of trinary digits, or "trits." To represent seven-bit ASCII with bits, you obviously need seven bits. How many trits would you need? I'll give you a little extra time.

Bonus 44: This company, bankrolled by Paul

Allen, folded its doors in 2000, abandoning its goal of making computer games for girls. Name this company, founded by Brenda

Laurel of Interval Research.

Toss-up Answers

T1. NAPSTER
T2. Bob METCALFE
T3. SKIN
T4. APACHE ("It's a patchy server.")
T5. MySQL
T6. TiVo
T7. UNICODE
T8.

BABELFISH

T9. TelePHONE

T10. The United States POSTAL SERVICE or POST OFFICE

T11. WAP (The document was called The WAP Trap) See freeprotocols.org

T12. PING

T13.

APU

T14. SOAP

T15.

JAKARTA

T16. BYTE

T17. C# (C SHARP)

T18. APOLLO

T19.

TUX

T20. TRANSMETA

T21. NETHACK

T22.

SOURCEFORGE

T23. EXPECT (There's a Perl module called Expect.pm that gives you the same functionality.)

T24. REVISION CONTROL SYSTEM

T25.

LDAP or LIGHTWEIGHT DIRECTORY ACCESS PROTOCOL

T26. QA (And the Dewey decimal number is the three-digit number before the apostrophe.)

T27 **BIOINFORMATICS** T28. LOKI T29. SAX **T30 ETHERNET** T31 Rob PIKE T32 **ORACLE T33**. **SCOOTER** T34 GREGORIAN

T35. SMURF (One way to defeat smurfing is to disable IP broadcast addressing at each network router since it is seldom used.)

T36.

XHTML

T37. APACHE

T38.

RSVP

T39. RECEIVE

T40.

RSYNC

T41.

CARBON

T42. RDF

T43. SMB

T44.

DVD or DIGITAL VERSATILE DISK

T45. UML

T46. HEARING

T47. Alex CHIU

T48. BANNER

Bonus Answers

B1. SMTP, POP, IMAP.B2. WART, CYST, BOIL, LESIONB3. MICROSOFT, AT&T, ARCHER DANIELS MIDLAND **B4**. R.E.M., William SHATNER, Abraham LINCOLN

B5. INDREMA

B6. 72, 168, 30

B7.

The VAULTS of PARNASSUS

B8.

DAEMON, SATAN, ZOMBIE, KERBEROS

B9.

CDATA

B10. ADSL, IDSL, SDSL, VDSL (also accept VASDL or BSDL)

B11. SLACKWARE (1993)

B12. RJ45

B13. ORIGINAL EQUIPMENT MANUFACTURER, VALUE-ADDED RESELLER

B14. VERTICAL BLANKING INTERVAL

B15. BERKELEY SOFTWARE DISTRIBUTION, 386BSD

B16. PHLEGM, BLOOD, BLACK BILE, YELLOW BILE

B17. SUN, APPLE, ADOBE

B18. ZERO

B19. 1, 2, 4

B20. GTK, QT, MOTIF or LESSTIF

B21. COLOR Laserjet 2100, HP 4050, HP Laserjet 5SI. (The Color Laserjet 2100 is a personal printer weighing about 18 pounds, the 4050 is a reasonably-sized office printer weighing about 45 pounds, and the 5si weighs about 130 pounds.)

B22. Bill JOY

B23. CHROME

B24. SEMICOLON, EIGHT, VERTICAL BAR or PIPE, LESS THAN SIGN, B

B25. 100,000 (The actual number was 66,368.)

B26. LOGIC, ASTRONOMY

B27. USE.PERL.ORG

B28. REFERER

B29. CAMEL, PYTHON, CHICKEN, ASP, BAT, Hermit CRAB, HORSE

B30. 5 in the morning, 8 1/2, ALEF

B31. 1997 (Judgment Day in *Terminator* was August 29th, 1997, and all that happened was that Michael Jackson turned 40.)

B32. CONF, HTDOCS, LOGS

B33. PHP and JAVASCRIPT

B34. 127 (also accept 125 through 129), 192 (also accept 191 or 193)

B35. \$18

B36. BUDDY

B37. CLIPPING

B38. CALDERA

B39. PRL

B40. IE, CN, CH, ZA, TV

B41. ELSIF, ELSEIF, ELIF, ELSEIF, ELSE IF

B42. YES (Amazon), YES (BT), YES (an individual; patent #6025810), YES (Sony), YES (Unisys)

An excerpt from the hyper-light-speed antenna patent filing:

A method to transmit and receive electromagnetic waves which comprises generating opposing magnetic fields having a plane of maximum force running perpendicular to a longitudinal axis of the magnetic field; generating a heat source along an axis parallel to the longitudinal axis of the magnetic field; generating an accelerator parallel to and in close proximity to the heat source, thereby creating an input and output port; and generating a communications signal into the input and output port, thereby sending the signal at a speed faster than light.

It has been observed by the inventor and witnesses that accelerated plant growth can occur using the present invention.

For accelerated plant growth, first, you need to create a hot surface that is more than 1000 degrees Fahrenheit. Next, you need a strong magnetic field. Only one device is needed for this function. This allows energy from another dimension to influence plant growth.

B43. 5 (The log base 3 of 128 is about 4.4, so you'd need 5.)

B44.

PURPLE MOON

Thanks to Joe Johnston, Mark Jason Dominus, Jarkko Hietaniemi, Chris DiBona, Tom Christiansen, and Sean Burke for contributing some question ideas.

Chapter 34. The Fourth Perl/ Internet Quiz Show

Jon Orwant

Here are the toss-ups and bonus questions from the Fourth

Internet Quiz Show, held in the summer of 2001 at the O'Reilly Open Source Convention. Answers are at the end of the article.

Five questions in the quiz show were "British restitution" questions. At the previous year's quiz show, the championship came down to the very last question. The London team had to answer Bonus 36 from Chapter 33 correctly—a question requiring knowledge of the name of then-President Clinton's dog. Most Americans probably wouldn't have known the answer; it was doubly unfair for a team full of Brits. So I peppered the 2001 quiz show with a few questions that would give British players an edge.

As usual, you can keep score using the ranks at the beginning of Chapter 31: count one point per question. For bonus questions, a half-correct answer merits the entire point.

Toss-up Questions

Toss-up 1: Russian law requires that software permit the purchaser to make at least one legal copy. According to Alexander

Katalov, the president of Elcomsoft, this makes it illegal to distribute Adobe's eBook software in Russia. That provides little comfort for this man, jailed in Las Vegas for distributing software designed to circumvent Adobe copyright protection measures. Who is this unfortunate hacker?

Toss-up 2: According to Nielsen/Netratings study, workers spend more time online on this weekday than any other. Name that day.

Toss-up 3: "It reflects the lifestyle of youngsters in Israel and in the world—to eat fast food and use the Internet at the same time." So said the CEO of the Israeli subsidiary of this well-known fast food chain. He said that as he introduced the newest branch of the restaurant, which lets patrons surf the Web and purchase Microsoft software, and lets kids play computer games for free. What is the Hamburgler's newest ISP?

Toss-up 4: In its first two years of life, this non-profit organization pushed Network Solutions to allow more competition among domain name registrars. They also instituted mandatory arbitration of trademark claims via their Uniform Dispute Resolution Policy. However, they reneged on their promise to elect half their board members from an at-large membership. What is this organization chaired by Vinton

Cerf?

Toss-up 4 linked bonus: ICANN approved seven new top-level domains. Name them.

Toss-up 5: It came into such common use in the 19th century that a German chemist declared that the amount of it consumed by a nation was an accurate measure of its wealth

and civilization. It's been used for at least 2,300 years, and according to

Pliny the Elder, the Phoenicians prepared it from goat's tallow and wood ashes in 600 BC. Its importance for cleaning the body wasn't recognized until the second century AD. Today, it's used as a cleaning agent, but it also shares its name with a protocol for executing remote procedure calls over a network. What is this simple object access protocol?

Toss-up 5 linked bonus: I'll give you three guesses.

Soap can be made from caustic soda instead of animal fat, but when it comes from an animal, what animal does it most often come from?

Toss-up 6: British restitution toss-up. Richard

Garriott is better known by this moniker, which he adopted as his pseudonym in the series of Ultima games that he created. Name it.

Toss-up 7: As of May 2001, what similar game has a hundred thousand more players than Ultima Online?

Toss-up 8: On April 12, 2001, Harvey R.

Ball passed away. In 1963, he was paid \$45 for this picture by State Mutual Life Assurance. He never applied for a trademark or copyright; his son said, "He'd get letters from all over the world thanking him for

How do you put a price on that? He died with no apologies and no regrets." This drawing became a

cultural icon and has a yellow background. What was this famous happy drawing?

Toss-up 8 linked bonus: I'll

show you four smileys; you match them up.

Punk rocker

- 2. Mr. Bill
- 3. Department store Santa
- 4. Charlie Chaplin

```
C | : -=
*<| : - { ) )
(8-0
=:-)
```

Toss-up 9: The

GNOME shell and file manager, developed by the sadly defunct

Eazel corporation, shares its name with a cephalopod mollusk with a spiral chambered shell and a brand name for gym equipment. What is this marine word?

Toss-up 10: By naming every concept simply by a URI, this project will let anyone express new concepts that they invent with minimal effort. Its unifying logical language will enable these concepts to be progressively linked into a universal Web. Its structure will open up the knowledge and workings of humankind to meaningful analysis by software agents, providing a new class of tools by which we can live, work, and learn together. That's what *Scientific American* says about this W3C effort to extend the World Wide Web into what new kind of web?

Toss-up 10 linked bonus: In philosophy, it's a theory about the nature of existence, or of what types of things exist. In

practical computer science, it's often a collection of formal relationships between terms. What is this four-syllable word?

Toss-up 11: I EAT VET CATS, EVICTS AT TEA, IT SET CAVEAT, EASE ATTIC TV, ATTEST A VICE. These five phrases are all

anagrams of a company devoted to open source technologies, and in a complete coincidence also happen to be the sponsor of this quiz

show. Who is it?

Toss-up 11 linked bonus: ActiveState makes a cross-platform, multi-language interactive development environment called

Komodo. How bloated is it? I'm referring not to the memory footprint of the software, but the weight of the Komodo dragon. Since bloat is proportional to features, let's consider the features of the Komodo dragon.

- 1. Small ones can climb trees.
- 2. They can swim.
- 3. They run faster than humans.
- 4. They are cannibals.
- 5. They are the only animals other than humans that willingly control their population, which they do by eating their own eggs.
- 6. Their saliva contains four types of bacteria that are resistant to antibiotics.
- 7. Their teeth are arranged so that the maximum amount of flesh can be bitten off and swallowed whole. They eat their prey like snakes, bones and all.

- 8. They have a very good sense of smell.
- 9. They eat people. A Swiss traveler in Indonesia injured his leg, and his guide left him to get help. A Komodo dragon found the traveler, and by the time the guide returned all that was left of the traveler was a backpack and a puddle of blood.
- 10. So: nice naming job, ActiveState. At O'Reilly, we're pretty fond of animals too, but we try to pick cute ones.

How much do these disgusting animals weigh? I'm looking for the weight of the adult male, within 20%. It would probably be around 6 feet long.

Toss-up 12: Guess that protocol. The client program, called the network manager, makes virtual connections to a server program that executes on a remote network device, and serves information to the manager regarding the device's status. Name this protocol, which even its FAQ admits is not as simple as its name suggests.

Toss-up 13: It's an open source full-featured

web proxy cache designed for Unix. It supports SSL proxying,

SNMP, and caching of DNS lookups, as well as the obvious: storing HTTP content so that you don't have to retrieve data from a

web site more than once even if you visit the site repeatedly. It shares its name with a ten-armed cephalopod. What is it?

Toss-up 14:

TK-707 and

Ultramaster RS101 are two free Linux programs. They're both examples of this kind of

software, letting you create a looping pattern of beats. Ultramaster is a bit more fully featured, since it plays bass too. Another example of this type of software is

Rhythm Lab, which its author created so that he could play polyrhythyms. What is this kind of software?

Toss-up 15: It's a set of extensions to the HTTP protocol that allows users to collaboratively edit and manage files on remote web servers. What is this network protocol for creating interoperable, collaborative applications?

Toss-up 16: The acronym's the same: a type of digital circuit in which the output is derived from two transistors, and a field in the IP Internet protocol specifying how many hops a packet can travel before failure. What is this three-letter acronym?

Toss-up 17: You can buy both direct-sequence and frequency-hopping cards for this

spread spectrum wireless protocol. What is this competitor to Bluetooth for 2.4 GHz bandwidth? The direct sequence version has a lowercase "b" at the end of its name.

Toss-up 18: A Supercomputer Center in this California city announced a prototype terabyte file server for \$5,000. What is this city, home to Legoland California and the 2001 Open Source Convention?

Toss-up 19: You might have heard that if you take HAL, the name of the computer in Arthur C. Clarke's *2001*, and shift each letter forward by one, you get IBM. If you take the word "cook" and shift it forward five letters, you get the name of what web protocol?

Toss-up 20: Up to 200,000 of these devices were made. It was patented in 1918, and instead of a QWERTY keyboard, it had a

QWERTZ keyboard in the pattern of German typewriters. It weighed about 26 pounds and measured 13.5 by 11 by 6 inches. It had three rotors. What is this

cryptographic device?

Toss-up 21:

Cyrano Sciences sells a device that emulates a human organ. Name the organ.

Toss-up 22: Spokesperson Scott

McNeil offered to eat his tennis shoe if Version 1.0 of this wasn't released by the end of 2001. What are these three letters, which stand for the set of standard components of a Linux system, and also for the least important eight bits of a bitstring?

Toss-up 23: According to its FAQ, this

protocol equals AH + ESP + IPcomp + IKE. AH is the authentication header, ESP stands for Encapsulating Security Payload, IPcomp is IP payload compression, and IKE is Internet Key Exchange. What is this protocol that provides per-packet authenticity and confidentiality guarantees between peers using it?

Toss-up 24: Brittney

Cleary a is twelve-year-old singer who recently sang a song about this online pastime which is popular among teenagers, who might use AIM or, less likely, IRC or Jabber to do it. Name this phenomenon.

Toss-up 24 linked bonus: Brittney's song is called "I.M. me."

At this point, I played the song.

There were four I.M. acronyms in that catchy chorus: LOL, G2G, BRB, and BBFN. Tell me what each stands for.

Toss-up 25: Unlike RCS and SCCS, this open source versioning system uses a merging model that allows everyone to have access to files at all times, supporting concurrent development. What is this three-letter versioning system?

Toss-up 26: The name's the same: it's a SourceForge

project for downloading files from remotely and locally connected GPS receivers by connecting to them via modems. It's also the name of a type of microprocessor. When you change the last letter from a C to a K, it's also the name of a type of fish, whose fins are sometimes used to make soup. What's the name?

Toss-up 27: On November 3, 1988, one of these programs spread across VAX and Sun workstations. It was written by Robert Tappan

Morris, then a doctoral student at Cornell and now a professor at MIT. This

program was not a virus; viruses cannot be run independently, but instead attach themselves to other programs. What was this type of program, which shares its name with a type of annelid?

Toss-up 27 linked bonus: (Readers: if you're keeping score, give yourself a free point for this question.) Now it's time for the Internet Quiz

Show to jump the shark. This is a three-part question. First, pick someone on your team.

Second, guess what I'm going to ask this person to do. Bear in mind that bonus questions often relate to the previous toss-up, and that the answer to the toss-up was worms, and that this is the exact point at which people will say that my quiz show lost all its integrity.

Finally, you now have to eat precisely 1.4 grams of worms. You can choose from cheese, barbecue, or mexican spice. This is an actual foodstuff and should, in theory, be nontoxic.

This would be a good time to remind you that this quiz show is brought to you by ActiveState, making programming easier through multi-language, cross-platform software and services. O'Reilly & Associates hereby disclaims all liability for gastrointestinal ailments occurring as a result of any foodstuffs consumed incidental to the 2001 Open Source Convention. O'Reilly & Associates would also like to remind you that ActiveState, being headquartered in Vancouver, is a Canadian company, and therefore is

potentially liable for tortious personal injury and wrongful death in both American and Canadian jurisdictions, thus doubling any potential judgment you might receive in court.

Toss-up 28: A "User Friendly" cartoon depicts a gravestone with a birthdate of 1997 and a death date of 2001, when Microsoft decided to remove everyone's least favorite animated metallic helper from Windows. Who was it?

Toss-up 29: "DCS1000"

is the new name of this

surveillance system. The name was changed last February by the FBI because of the negative associations of its original hungry-sounding name. What is this system, which is named after meat-eating animals?

Toss-up 30: Guess that

audio format. The *Washington Post* had two members of the National Symphony Orchestra, a high-end stereo salesman, a record producer, a composer, and two guitarists listen to digitally encoded versions of Stravinsky's "The Rite of Spring" and the Who's "Love Ain't for Keeping" in various digital audio formats. Most of them thought that the beta version of this format was the least realistic. The high notes sounded harsher, and the low notes were harder to hear. Regular MP3s treated the human voice a bit better, but this format handled quick changes in volume better. What is this two-word audio format, whose first word is a tactical maneuver from the multiplayer network game Netrek, and whose second word is named after a character in a Terry Pratchett novel.

Toss-up 31: I'm going to read the beginning of a book. Your job is to figure out what device the book is about.

In the nineteenth century there were no televisions, airplanes, computers or spacecraft; nor were there antibiotics, credit cards, microwave ovens, compact discs, or mobile phones.

There was, however, an Internet.

During Queen Victoria's reign, a new communications technology was developed that allowed people to communicate almost instantly across great distances, in effect shrinking the world faster and further than ever before. A worldwide communications network whose cable spanned continents and oceans, it revolutionized business practice, gave rise to new forms of crime, and inundated its users with a deluge on information. Romances blossomed over the wires. Secret codes were devised by some users and cracked by others. The benefits of the network were relentlessly hyped by its advocates and dismissed by its skeptics. Governments and regulators tried and failed to control the new medium. Attitudes toward everything from news gathering to diplomacy had to be completely rethought. Meanwhile, out on the wires, a technological subculture with its own customers and vocabulary was establishing itself.

Does all this sound familiar?

Today the Internet is often described as an information superhighway; its nineteenth-century precursor was dubbed the "highway of thought."

What is this device, described in the book The Victorian Internet?

Toss-up 31 linked bonus: Arguably, the first prototype telegraph was created as early as 1746, when a French scientist lined up 200 monks, each holding one end of an iron wire connecting them to the next monk 25 feet away in a giant serial circuit. The scientist wanted to see how far electricity traveled, and how quickly it traveled. Assume each monk is a point mass. Tell me how far the signal traveled.

Toss-up 32: There are two popular types of this device: standby and continuous. It can protect your computer from voltage surges, spikes, and sags; from frequency differences; and from temporary power failure. What is this piece of hardware, which shares its acronym with a shipping company?

Toss-up 33: Guess that Unix command. After you compile and debug a program, there's part of the binary you can delete to save space. What command can you use to delete that part of the binary?

Toss-up 34: Guess that activity. Last Wednesday, the Washington Post reported that a study by Cal Tech and MIT said that this activity is

not ready for implementation on the Internet. They said that at least a decade of further research on the security of home computers is needed before it will be feasible. What is this activity, which according to the study failed even using traditional methods for between 4 million and 6 million people?

Toss-up 35: Its defacement message implies that it is of Chinese origin, and it is only programmed to attack English-language versions of Windows NT or 2000. This worm was programmed to flood www.whitehouse.com in a massively coordinated

denial of service attack. Name this worm.

Toss-up 36: This product has a unified memory architecture, where the CPU and the GPU (the graphics processing unit) share a single memory space. It has an Intel Pentium 3 running at 733 megahertz with a 128 kilobyte cache, 64 megabytes of RAM, a DVD, a hard disk, 64 3D audio channels and an NVIDIA GPU. Quake on a Pentium Pro produced around 100 thousand triangles per second, but this console will be able to produce up to 125 million triangles per second. What is this

Microsoft game console?

Toss-up 37:

Kontour, spelled with a K, is the new name of this drawing program designed as part of K Office, an Office-like suite of programs for the KDE user interface. What was the original name of this clone of Adobe Illustrator?

Toss-up 38: Justin

Frankel and Tom

Pepper created Winamp under the aegis of a company named Nullsoft. America Online quashed Nullsoft's next popular product, an alternative to Napster that allows users to exchange music, movies, text, and software via a peer-to-peer protocol. Tell me the name of this system, whose name combines "GNU" with a

chocolate hazelnut spread.

Toss-up 39: This set of metadata elements was formed in 1996 to specify a foundation of property values and types for information resources. Its name suggests that it originated in Ireland, but it's actually named for a town in Ohio that happens to share its name with the Irish capital. Name this XML initiative.

Toss-up 39 linked bonus: There are 15 elements in the original 1996

Dublin Core. Name six.

Toss-up 40: Often, when you invoke a Unix command, you provide a bunch of options and then a pathname. This Unix command swaps that ordering, so that you provide the pathname and then the options. What is this common four-letter Unix command that lets you search recursively through a directory?

Toss-up 41: It's like SOAP, but lighter weight. What is this XML protocol for executing remote procedure calls?

Toss-up 41 linked bonus: Just before this conference, I counted the number of implementations of XML-RPC on xmlrpc.com. Then I counted the number of XML-RPC services on xmlrpc.com. What is the ratio of implementations to services? Full credit if you're within a factor of two.

Toss-up 42: On Unix, this is the program that displays the login: prompt. It then execs the login command, which prompts for your password. What is this program, which I think stands for "generate teletype"?

Toss-up 43: You've probably heard of 10-base-T and 100-base-T

connections. What do the 10 and 100 mean?

Toss-up 44: According to their official web page, it's a consortium led by 180 universities to develop and deploy advanced network applications and technologies, accelerating the creation of tomorrow's Internet. Their primary goals are to: 1) Create a leading edge network capability for the national research community; 2) Enable revolutionary Internet applications; and 3) Ensure the rapid transfer of new network services and applications to the broader Internet community. What is the name of this next generation Internet initiative?

Toss-up 45: This federal judge apparently has a habit of voicing his personal opinions about cases; after he ordered Washington, D.C. mayor Marion Barry to jail on cocaine charges, he attended a symposium at Harvard and expressed displeasure that Barry was not convicted on more counts and accused four jurors of lying. The appellate court criticized him for breaching judicial ethics. The *San Jose Mercury News* wrote the following: "All that could come into play if

Microsoft—as some legal observers speculate—seeks to make the judge so angry that he commits a reversible error, delaying judgment long enough that Microsoft further consolidates its share of the market for browsers." That was said in 1999 about what federal judge?

Toss-up 46: This database system is at the center of an unfortunate legal battle over domain names. On the .com side are the developers of the database, and on the .org side is NuSphere. Name this popular open source database.

Toss-up 47: It consists of three segments: the space segment, the control segment, and the user segment. The control segment is five ground-based monitoring stations. The space segment consists of 24 satellites in almost perfectly circular 12 hour orbits. What is this global location system?

Toss-up 48: Most of the

Internet runs on electromagnetic radiation. I considered writing a toss-up about Maxwell's Equations, but, after all, we're computer folks, and we invented abstraction barriers so that we wouldn't have to know any physics. But everyone should have learned this simple law in high school. What is Ohm's law, which relates voltage, current, and resistance?

Toss-up 49: Copying, reference counting, and mark-and-sweep are three methods of this task. What is this task of

recycling memory that your program no longer needs?

Toss-up 50: Meteor, Orion, Genesis, Corona, Inspector, and Marvel are all product names for this company, which is probably best known for its Millennium graphics card. Name this company, whose name is just one letter away from a popular recent science fiction movie.

Toss-up 51: I've seen three variations of the law named after this Internet pioneer. The weakest version is "A network's value grows proportionately with its number of users" and the strongest is "The power of the network increases exponentially by the number of computers connected to it." He predicted the collapse of the Internet in 1996, saying it would become no more than FedEx shipping CD-ROMs back and forth. Who is this co-inventor of Ethernet?

Bonus Questions

Bonus 1: These three events happened in which year?

- 1. Kazakhstan sets up its first Internet connection.
- 2. Sun launches Java.
- 3. Amazon.com and eBay are founded.

Bonus 2: In the C shell, this character is used to repeat the last command line, but with a substitution. For instance, if you had a long command line and you mistyped the name of a file in the middle of it, you could use this character to fix the typo and execute the line again all at once. Name this character.

Bonus 3: Let's play Name That Nerd. He was arguably one of the earliest nerds. I'll give you four clues in decreasing order of difficulty.

1. Born in 1791, this Englishman had a fondness for ciphers, lock-picking, stamped buttons, tunnels, and stomach pumps. But he hated street musicians with a passion, and wrote a book called *Observations of Street Nuisances* in 1864, in which he calculated that 25% of

his working power had been destroyed by street nuisances. According to one biography, the public retaliated by tormenting him with an unending parade of fiddlers, Punch-and-Judys, and stilt-walkers. Neighbors hired street musicians to play outside his window, especially with worn-out or damaged wind instruments, for as long as five hours at a time. One blew a penny whistle outside his window for half an hour every day for many months. According to one biography, "Even when he was on his deathbed, the organ-grinders ground implacably away." In 1861 he said he had never spent a happy day in his life, and would gladly give up the rest of it if he could live three days 500 years hence.

2. He is called by some the Father of Computing, and after his death his brain was preserved in alcohol for 37 years. He investigated biblical miracles, and made the assumption that the chance of a man rising from the dead is one in a trillion. He tried to mathematically handicap horse races, and as a result put Lady Lovelace deep into debt. He measured the heartbeat of a pig for his "Table of Constants of the Class Mammalia" In 1857 he published a "Table of Relative Frequency of the Causes of Breaking of Plate Glass Windows." A friend said of him, "He hated mankind rather than man, and his aversion was lost in its own generality." He corrected Tennyson's poem for the difference between birth and death rates, by changing "Ev'ry moment a man dies / Ev'ry moment one is born" to "Ev'ry moment a man dies / Ev'ry moment one and one-sixteenth is born." He also invented the cowcatcher

- 3. The father of computing enjoyed fire. He had himself baked in an oven at 265 degrees Fahrenheit for "five or six minutes without any great discomfort" and had himself lowered into Mt. Vesuvius. He began construction of a calculating machine called the Difference Engine, but had trouble raising the funds to complete it-or to begin the sequel, the Analytical Engine. British Prime Minster Robert Peel recommended that the father of computing use his Analytical Machine to calculate the time when it would be useful. "I would like a little previous consideration," the prime minister wrote, "before I move in a thin house of country gentlemen a large vote for the creation of a wooden man to calculate tables from the formula $x^2 + x + 41$."
- 4. His name rhymes with "cabbage."

Bonus 4: Bruce

Mah's paper "An Empirical Model of HTTP Network Traffic" in the April 1997 *Proceedings of IEEE Infocom* calculates the average number of clicks in a web session. For instance, if you visit www.oreilly.com, how many times are you likely to click anywhere in www.oreilly.com before moving to another site? Tell me the average number of clicks, within 20%.

Bonus 5: There's an Internet draft standard that defines the architecture to provide Internet-like services between two types of objects very far apart. It describes an approach called "bundling," creating a store-and-forward overlay network above the transport layers of underlying networks. Bundling uses many of the techniques of email, but is directed toward interprocess communication and is designed to operate in

environments that have very long speed-of-light delays. Tell me what objects this protocol was designed for.

Bonus 6: Systems like the peer-to-peer Swarmcast enable you to download large files from multiple computers simultaneously, making for a more robust and quicker download. Let's assume we have a perfectly efficient and error-free transmission channel, and let's further assume that there's a one-gigabyte file that you want to download, and there are three servers that can serve it to you starting at any point in the file. One of the servers delivers one megabyte per second, and the other two deliver half a megabyte per second. What is the minimum time in which the entire file can be transmitted to you?

Bonus 7: According to NetFactual, given all of the IP addresses in use for web sites, what is the average number of domain names per IP address? Full credit if you're within 30%.

Bonus 8: According to Pitkow's "Summary Of World Wide Web Characterizations" in the 1999 *WWW Journal*, the mean size of HTML pages on the Web is between four and eight kilobytes. Give me the *median* size, in kilobytes, of HTML pages, within a factor of two. Remember that the mean is the conventional average, and the median is the level where half your data points are above and half below.

Next, tell me the mean image size in kilobytes, within a factor of two.

Bonus 9: Within five, what is the approximate percentage of words in Webster's English Dictionary that were registered as domain names as of July 2000?
Bonus 10: Name That Nerd bonus. Guess the subject of the following clues.

- 1. He was born in 1847, and according to some biographies had attention deficit disorder at an early age. He wanted to become an actor, but his high-pitched voice and extreme shyness dissuaded him. Later, he became almost completely deaf, but when he was offered an operation that would almost surely restore his hearing, he refused, claiming that his power of concentration had been enhanced by his hearing loss.
- 2. He developed a lifelong disrespect for higher mathematics when he realized that Issac Newton was a lousy technical writer who used flowery language instead of clear concise writing. He decided at an early age to prove everything to himself through practical experimentation. He went into the publishing business at age twelve, and was able to scoop large newspapers by publishing faster than they could—because he had typesetting, printing, and distribution take place entirely on a train. He had to move his operation off the train when he set a baggage car on fire from a chemistry experiment.
- 3. In Boston he attended lectures at MIT and got his first patent, for an electric vote-recording machine. But members of the Massachusetts legislature denigrated it, saying that "its speed in tallying votes would disrupt the status quo."
- 4. The problem was that—during times of stress—political bodies of that period often relied upon the brief delays that were provided by the process of manually counting

votes to influence and change the opinions of their colleagues. "This is exactly what we do not want," a seasoned politician told him, adding that "Your invention would destroy the only hope the minority would have of influencing legislation.... It would deliver them over bound hand and foot to the majority."

- 5. He invented the first dictaphone, mimeograph, and practical storage battery, and received 1,093 patents in all. He proposed to his wife by tapping Morse code on her hand.
- 6. He was called the wizard of Menlo Park and he invented the electric lightbulb. When he died in 1931, individuals and corporations throughout the world dimmed their lights in his memory.

Bonus 11: The current version of the IP protocol, IPv4, has a 32-bit address space. Draft RFC 2460 for Version 6.5 of IPv6 calls for an address space with how many bits?

Bonus 12: I'll give you five descriptions of OS X technologies, and you name them.

- 1. The open source core of OS X, including the Mach 3.0 kernel.
- 2. A high-performance, lightweight window server and graphics rendering library for two-dimensional shapes.
- 3. A native OS X runtime environment allowing applications to make use of new OS X features while retaining compatibility with older Mac operating systems.

- 4. An application environment that runs native under OS X and is tailored for developing applications that run only on OS X.
- 5. The user interface of OS X.

Bonus 13: I'll

show you five fonts, and you tell me the name of the typeface. (See Figure 34-1.)



Figure 34-1. Five fonts for bonus question 13

Bonus 14: Guess that year in as few clues as possible.

- 1. South Korea sets up its first Internet connection.
- 2. ARIN, the American Registry for Internet Numbers, begins operation, and 56K modems are first introduced to the public.
- 3. The U.S. Supreme Court unanimously declares the Communications Decency Act unconstitutional.

Bonus 15: I'll give you five terms. You rank them in order of how many Google hits they had as of last Saturday, from most to least. Kernel, distro, bytecode, CORBA, router.

Bonus 16: I'll give you a Nielsen/Netratings statistic about web usage in the United Kingdom, and you tell me the corresponding statistic for the United States within 20%.

Now note that Nielsen samples only households who a) have either Windows 95/98/NT, and MacOS 8 or higher, and b) have enough free time to participate in polls about the Internet.

- 1. For the U.K., the average number of web sessions per month is 13. How many for the U.S.?
- 2. For the U.K., the average time spent per month of the Web is 5 hours, 58 minutes, and 53 seconds—so just under 6 hours. How many hours per month, on average, does someone in U.S. spend on the Web?
- 3. For the U.K. the average number of unique sites visited per month is 18. How many for the U.S.?
- 4. Nielsen estimates that in the U.K., the number of people using the Internet is 23,375,121. What is their estimate for the U.S.?

Bonus 17: In addition to Komodo, ActiveState also makes PerIMX, which provides mail filtering for the enterprise-and they didn't ask me to say this, but we use it at O'Reilly and are very happy with it. However, like the Komodo dragon, the name has a hint of destructiveness to it. The MX missile cost \$100 billion to develop and each MX warhead has one megaton of destructive force. Assuming PerlMX is equally deadly, let's assume that a copy of it is dropped on an unsuspecting city. It would create a crater 1/4 of a mile across, and generate a fireball with a width of 7/10 of a mile. At a radius of 1.7 miles, 50% of the people would die, 40% would be injured, and 10% would be unaffected. At a particular greater distance, the stats would be 0 deaths, 25% injuries, and 75% unaffected. Guess this distance within 20%—in other words, how many miles from ground zero should you be to avoid obliteration by this horrible PerlMX weapon, brought to you by ActiveState?

Bonus 18: First, what is the name of the open standards version of JavaScript?

Next, Microsoft has submitted the core .NET specs to ECMA. What do the four letters ECMA stand for?

Bonus 19: First, within 50%, tell me how many users Napster creator Shawn

Fanning said the service had in February 2001 just after the federal appeals court ruling.

Next, at that time and within 50%, how many files were listed on Napster, according to the Webnoize study?

Bonus 20: I'm going to read you four frequently asked questions from a FAQ about how to hack a particular

consumer electronics device. Tell me the name of the device using as few clues as possible.

- 1. Will a factory reset zero out the drive for better compression?
- 2. Getting a BASH prompt with Dylan's bootdisk.
- 3. Can I back up my 15GB A drive onto a 30GB drive and get a 30 hour single drive unit?
- 4. Will an upgrade kill my Now Showing or Season Passes?

Bonus 21: Last February, we at O'Reilly were trying to decide what books to write about

message transfer agents. We've had a bestseller about sendmail for a long time, but what about exim, Microsoft Exchange, postfix, qmail, or zmail ? I wrote a program that identified the MTAs of all three-letter dot coms to help us decide. Rank them from most common to least.

Bonus 22: Name that

failed dot com. I'll describe two dot coms, and you name them.

- 1. When they sold off their assets, they included the rights to their sock puppet spokesdog.
- 2. One employee of this company said, "I always liked our core customers, those grungy dysfunctional freelancers and geeks who didn't want to leave the house to pick up their own Fritos and beer."

Bonus 23: This describes a type of

neural net in which the connection weights are one-way, typically with input units fully connected to hidden units, and

hidden units fully connected to output units. During the training phase, the weights change by comparing what you wanted on the output units to what you got, and adjusting the weights in proportion to the difference. What is this common type of neural net?

Bonus 24: In the summer of 1939, a small team of scholars-turned-codebreakers arrived at this place; their mission was to crack the Enigma cipher. According to their web page, the odds against them were, and I quote, "a staggering 150 quintillion to one." Over 10,000 people worked here at its peak, but by March 1946 they were all gone. The efforts at this place led Winston

Churchill to coin the phrase, "The geese that laid the golden eggs and never cackled." Today, you can visit for an admission fee of just five pounds. Name this place.

Bonus 25: There are three primary levels of requirements in Internet protocol specifications. For instance, one of them is MUST. A MUST-level requirement implies that compliance is absolutely essential, as in "Every HTTP implementation MUST accept GET requests." What are the other two primary requirement levels?

Bonus 26: I'll give you four telecommunications acronyms, and you expand them.

- 1. CLEC
- 2. ILEC
- 3. RBOC
- 4. POTS

Bonus 27: This is one of the oldest and most time-tested

caching algorithms, used for decades before the Web came around. It's probably the first caching algorithm ever. The idea is straightforward: objects that have been accessed more recently are likely to be accessed again, and so less-accessed objects should be evicted from the cache before removing any of the newer objects. What is it?

Bonus 28: Guess that year in as few clues as possible.

- 1. ARPANET shifts to TCP/IP and the Japan Unix Network is established.
- 2. Fido becomes operational.
- 3. MGM produces the movie *War Games*.

Bonus 29:

Quantum computing can be somewhat coarsely divided into two parts. There's light-wave computing, in which you might use an acousto-optic modulator and the wave-particle duality of light to do things like search through a database in constant time. Then there's a class of algorithms that can't be parallelized so easily, and require full-fledged quantum computing, which evaluates the states of many particles simultaneously. This is distinct from *superpositions*, which are the multiple states of a single particle. First, what adjective do physicists (and Damian Conway) use to describe these states of a system of particles?

Next, entangled states are needed by this algorithm, which can factor numbers in polynomial time. Name this algorithm developed in 1994 and named after its discoverer.

Bonus 30: Name that nerd in as few clues as possible.

- 1. He is a Canadian citizen and, according to his biography, has never written a program that uses cursor addressing.
- 2. He's probably the only member of the technical staff at Bell Labs to appear multiple times on "Late Night with David Letterman." He also has given a talk entitled "Systems Software Research Is Irrelevant."
- 3. He won the Olympic silver medal in archery in 1980, and the next year wrote the first bitmap window for Unix systems at Bell Labs. One of his more famous papers is "Why Pascal Is Not My Favorite Programming Language."
- He was a principal designer and implementor of the Plan
 9 and Inferno operating systems, and his last name is both a fish and a piece of medieval weaponry.

Bonus 31: Measures and countermeasures. As I write this, I'm having a little coding duel with Joe

Johnston, an O'Reilly hacker who's here at the conference. On an O'Reilly mailing list, he argued that the notion of an XML router is an inherently stupid idea, and I think it's a good idea. To settle the debate, he created a poll on the O'Reilly Intranet where company employees could vote on whether the notion of an XML router was stupid or not. So I wrote a program that voted automatically on my behalf over and over again.

I used a well-known suite of Perl programs for automating web browsing, and fed it the URL of Joe's voting page so that it could vote in favor of XML routers. Name this three-letter suite of programs by Gisle Aas. Joe responded by preventing votes from the IP address of the local machine. Give me an IP address that always refers to the local machine.

That stopped my Perl program, at which time the votes in favor of XML routing were now 40,000 to 17.

But. Joe had foolishly given me sudo powers on the machine. I found the program that was generating the web page, and I modified it so that after it checked its online database, it rewrote the result with a huge number that was 0 on January 1, 1970. What did I derive this number from?

Then I did a ps on the Linux box to figure out what Joe was thinking, and I saw lots of recent backup copies of his database. So Joe thought—and probably still thinks right now—that I was directly modifying the database. But I wasn't, because I modified the logic of the program after it read the database.

So I created some red herring databases with similar names to throw him off the track. I read the database into a binary string and used a built-in Perl function to extract the fields from it. What function did I use?

Meanwhile, I decided to hide what I was doing inside his voting program. I had been setting the appropriate variable to the value of the Unix time, but since the Perl function for time is time, I figured that might be a giveaway. So I replaced it with a little-known magical Perl variable that contains the Unix time at which your program began execution. What is this variable?

Bonus 32: In the C shell, this two-character sequence evaluates to the last argument on the previous command line.

For instance, if you said ls *.pl *.ini and then you wanted to remove all the ini files, you could say rm ______. What are these two characters?

Bonus 33: I'll name six web sites. You rank them from most popular to least popular as indicated by their unique daily visitors, tabulated by Jupiter Media Metrix during the week ending April 15, 2001. eBay, iWon, MSN, Netscape, Passport, and Yahoo!.

Toss-up Answers

T1. Dmitry SKLYAROV

T2. MONDAY

T3. MCDONALD's

T4.

ICANN (INTERNET CORPORATION FOR ASSIGNED NAMES AND NUMBERS)

T4-bonus. AERO, BIZ, COOP, INFO, MUSEUM, NAME, and PRO

T5. SOAP

T5-bonus. COW (also accept BULL, STEER, CALF, and variants)

T6.

LORD BRITISH

T7.

EVERQUEST

T8. SMILEY face. (If HAPPY, ask player to rephrase.)

T8-bonus. 1. D, 2. C, 3. B, 4. A

T9. NAUTILUS

T10. The SEMANTIC web

T10-bonus. ONTOLOGY

T11. ACTIVESTATE

T11-bonus. 200 pounds (accept between 160 and 240 pounds) or 90 kilograms (accept between 72 and 108 kilograms)

T12. SNMP or SIMPLE NETWORK MANAGEMENT PROTOCOL

T13. SQUID

T14. DRUM machine (also accept PERCUSSION)

T15.

webDAV or Web DISTRIBUTED AUTHORING and VERSIONING

T16. TTL (which can stand for either "transistor-transistor logic" or "time to live")

T17. 802.11

T18. SAN DIEGO

T19. HTTP

T20. ENIGMA

T21. NOSE. (It costs \$10,000, and it calibrates by sniffing the ambient smell around it.)

T22. LSB

T23. IPSEC

T24. INSTANT MESSAGING

T24-bonus. LAUGH OUT LOUD, GOT TO GO, BE RIGHT BACK, and BYE BYE FOR NOW.

T25. CVS

T26. SHARC

Speaking of sharks, the phrase "jumping the shark"

has recently come into vogue. It means a sudden point at which everything goes downhill. The phrase originates from "Happy Days," where in the season finale Fonzie jumped over a

shark in his motorcycle, with the frame freezing in mid-air so that you had to tune in next season to see if he landed safely. Anyway, "jumping the shark" has come to mean the exact moment at which a

show sells out. I'm telling you this because the bonus question for the next toss-up will be the "jumping the shark" moment for the Internet quiz show.

T27. WORMs

T27-bonus. EATing WORMs (They were edible dried worms sold as a novelty.)

T28. CLIPPY

T29. CARNIVORE

T30. OGG VORBIS (Vorbis is the digital music format, and Ogg is the blanket project for creating a fully open multimedia system.)

T31. The TELEGRAPH

T31-bonus. 4975 feet. (199 x 25 feet) No credit for fencepost errors—5,000 feet is not correct.

T32. UPS. (The difference between a voltage surge and a voltage spike is 3 nanoseconds.)

T33. STRIP

T34. VOTING

T35.

CODE RED (From

Schneier's

CRYPTO-GRAM newsletter: "The attack failed because of some programming errors in the worm. One, the attack was against a specific IP address, and not a URL. So whitehouse.gov moved from one URL to another to avoid the attack. And two, the worm was programmed to check for a valid connection before flooding its target. With whitehouse.gov at a different IP address, there was no valid connection. No connection, no flooding.")

T36.

XBOX

T37. KILLUSTRATOR

T38. GNUTELLA

T39.

DUBLIN CORE or DCMI

T39-bonus. TITLE, AUTHOR or CREATOR, SUBJECT and KEYWORDS, DESCRIPTION, PUBLISHER, other CONTRIBUTORS, DATE, RESOURCE TYPE, FORMAT, resource IDENTIFIER, SOURCE, LANGUAGE, RELATION, COVERAGE, RIGHTS management.

T40. FIND

T41. XML-RPC

T41-bonus. 38/11 = 3.4545. Full credit for an answer between 1.72 and 6.91.

T42. GETTY

T43. MEGABITS per SECOND. (The "T" stands for twisted-pair.)

T44. INTERNET2

T45. Justice Thomas PENFIELD JACKSON

T46. MYSQL

T47. The GPS system

T48. I(current) = Voltage / Resistance or the AC version: I(current) = Voltage / Z(impedance)

T49. GARBAGE COLLECTION

T50. MATROX

T51. Bob METCALFE

Bonus Answers

B1. 1995

B2. ^ (CARET or UP ARROW)

B3. Charles BABBAGE. (Babbage also wrote a ballet that was never performed due to the theater manager's fear that the auditorium would catch fire. It involves 60 dancers, a lot of colored lights, and a vat of eels.)

B4. 7 (accept between 5.6 and 8.4)

B5. PLANETS (

Interplanetary Internet). An excerpt from the draft follows:

```
Desiderata
               of
                       Interplanetary
Internetworking
    Go thoughtfully in the knowledge that
all interplanetary
       communication derives from
                                      the
modulation of radiated energy, and
   sometimes a planet will be between the
source and the destination.
      Therefore rely not on end-to-end
connectivity at any time, for the
   universe does not work that way.
    Neither rely on ample bandwidth, for
power is scarce out there and
   the bit error rates are high. Know too
```

that signal strength drops off by the square of the distance, and there is a lot of distance. Consider the preciousness of interplanetary communication links, and

restrict access to them with all your heart. Protect also the $% \left({{\left({{{\left({{{\left({{{\left({{{}}} \right)}} \right.}\right.}} \right)}_{0,2}}} \right)} \right)$

confidentiality of application data or risk losing your customers.

Remember always that launch mass costs money. Think not, then, that

you may require all the universe to adopt at once the newest

technologies. Be backward compatible.

Never confuse patience with inaction. By waiting for acknowledgement

to one message before sending the next, you squander

time that will never come to you again in this life. Send as much as

you can, as early as you can, and meanwhile confidently await

responses for as long as they may take to find their way to you.

Therefore be at peace with physics, and expect not to manage the

network in closed control loops-neither in the limiting of

congestion nor in the negotiation of connection parameters nor even

in on-demand access to transmission bands. Each node must make its

own operating choices in its own understanding, for all the others

are too far away to ask. Truly the solar system is a large place and

each one of us is on his or her own.

Deal with it.

S. Burleigh

B6. 500 seconds. (The three sources combine to deliver 2 meg per second.)

B7. 3.2 (accept between 2.24 and 4.16)

B8. 2 (accept between 1 and 4; the mean is substantially higher than the median, which tells us that there are a small number of large HTML pages); the mean image size is 14 kilobytes (accept between 7 and 28)

B9. 98 (from the July 2000 edition of Win Treese's Internet Index)

B10. Thomas Alva EDISON

B11. 128

B12. DARWIN, QUARTZ, CARBON, COCOA, AQUA

B13. COURIER, TIMES, COMIC sans ms, GARAMOND, IMPACT

B14. 1997

B15. KERNEL (6,560,000), ROUTER (2,570,000), CORBA (1,030,000), DISTRO (186,000), BYTECODE (108,000).

B16. 19 (accept between 15.2 and 22.8); 9:44:52 (accept between 7:48 and 11:42); 10 (accept between 8 and 12—so the U.S. uses the Web more, but is less diverse in the sites they visit); 167,138,270 (accept between 133,710,616 and 200,565,924)

B17. 7 miles (accept between 5.6 and 8.4 miles)

B18. ECMAScript; EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

B19. Number of Napster users: 50 million (AOL had 150 million). Full credit for an answer between 25 million and 100 million.

Number of files: 2.79 billion. Full credit for an answer between 1.39 billion and 4.19 billion.

B20. TIVO

B21. QMAIL (14%), Microsoft EXCHANGE (13%), EXIM (4%), POSTFIX (3%), ZMAIL (0%). (sendmail had a 65% share.)

The programs that I wrote did a DNS lookup to find the mail exchanger, connected to port 25, and if my program couldn't deduce the MTA, it attempted a HELP command to see if it could deduce the MTA from the response. As a result, my program triggered a lot of complaining from mail servers. Two notable ones were:

214 Klingons do not require assistance!
500 Bloody Amateur! Proper forging of
mail requires recognizable
 SMTP commands!

B22. PETS.com, KOZMO.com

B23. BACKPROPAGATION

B24.

BLETCHLEY PARK

B25. SHOULD and MAY. (There's also MUST NOT and SHOULD NOT.)

B26. COMPETITIVE LOCAL EXCHANGE CARRIER, INCUMBENT LOCAL EXCHANGE CARRIER, REGIONAL BELL OPERATING COMPANY, PLAIN OLD TELEPHONE SERVICE

B27. LEAST RECENTLY USED or LRU

B28. 1983

B29. ENTANGLED; SHOR's algorithm (The best-known classical algorithm for factoring numbers is the quadratic sieve.)

B30. Rob PIKE

B31. LWP; 127.0.0.1; UNIX TIME or EPOCH; UNPACK; \$^T (The final score was "An XML router is an insane idea"—81 votes, and "An XML router is a great idea"—995,659,743 votes.)

B32. ! \$

B33. YAHOO MSN PASSPORT NETSCAPE IWON EBAY (Number of visitors in millions: Yahoo 12.5, MSN 11.8, Passport 6.3, Hotmail 6.2, AOL 5.1, Netscape 3.0, iWon 2.9, Excite 2.6, eBay 2.6, Lycos 2.4.)

Chapter 35. The Perl Whirl Quiz Show

Jon Orwant

This is the quiz show that I wrote in 2000 for Neil Bauman's inaugural

Perl Whirl, an Alaskan cruise that was also a Perl conference. The rules were the same as for my O'Reilly quiz shows, but since I couldn't attend the cruise, Larry Wall stepped in as emcee. (For more information about Perl Whirls and other geek cruises, see http://geekcruises.com.)

You know the drill by now: one point per question, and calculate your score with the chart at the beginning of Chapter 31.

Toss-up Questions

Toss-up 1: Slashdot publicized a web server powered by an unlikely source. What is this starchy vegetable?

Toss-up 2: Fill in the blank: John Gilmore said, "The Net interprets ______ as damage and routes around it. What is this term, synonymous with "restraint of expression"?

Toss-up 3: What is the name of the default package in Perl?

Toss-up 4: "I rescue kegs," "creek guises," and "Gee! Sick user!" are all anagrams of what phrase that should presumably be familiar to us all?

Toss-up 5: What special scalar variable contains the time at which your program began running?

Toss-up 6: The GD module used to use GIF as its primary image format, but thanks to Unisys enforcing its patent on LZW compression, GIF is no longer feasible. What image format does GD use now?

Toss-up 7: Which character is the modulus operator in Perl?

Toss-up 8: This proponent of literate programming uses an assembly language called MIX in his well-known computer science books. Who is this Stanford professor and inventor of TeX and Metafont?

Toss-up 9: What special array inside a class contains the list of classes to inherit from?

Toss-up 10: It lets you search CPAN for simple patterns in module names, distribution names, author names, or documentation. What is the URL of this relatively new and incredibly useful web site?

Toss-up 11: Its pits are half as long; it has two layers instead of just one; its tracks are half as thick. You can read data from it about nine times as fast as from an audio CD. What is this medium?

Toss-up 12: What four-letter HTML tag encodes information about the content of a web page, often used for inform (or fool) search engines?

Toss-up 13: Guess that

file test. With this file test, your program can determine whether it was called interactively. More precisely, you can use this file test to see whether a filehandle is a tty, and if both STDIN and STDOUT are ttys, your program can be pretty sure it was called from a shell. What is this single letter?

Toss-up 14: It's a Brazilian dance of African origin characterized by a dip and spring upward at each beat of the music. What is this two-syllable word, which also names a product enabling Unix and Windows machines to share directories and files?

Toss-up 15: What

Perl built-in terminates your program and launches another one in its place?

Toss-up 16: Support for this standard is being introduced into Perl. The standard provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language. What is this character encoding standard?

Toss-up 17: If you were looking for modules that let you FTP, send mail, and Telnet, you could find them all in this three-letter namespace on CPAN. In other words, I'm looking for the BLANK in BLANK::FTP, BLANK::TELNET, and BLANK::SMTP. Fill in the blank.

Toss-up 18: This unlikely investor in an Internet startup spends her spare time knighting people and tugging around those darn Corgis. Who is this ruler of the British empire?

Toss-up 19: The fork function clones your program as it's running; it's like coming to a fork in the road, and choosing both. How can you tell which road you're on?

Toss-up 20: What virus has functions named spreadtoemail and infectfiles?

Toss-up 21: This module bundled with Perl does little more than provide a hash. But what a hash: it contains entries that tell you how your Perl was compiled, what operating system you're on, what signals your system understands, and whether your Perl was compiled with thread support. What is this module?

Toss-up 22: The first of these systems was a cluster of 16 DX4 processors connected by 10-megabit Ethernet. The processors were too fast for a single Ethernet, so new Ethernet drivers for Linux were created that striped the traffic across multiple Ethernets. What is the name for these supercomputing clusters of Linux systems, named after the slayer of Grendel?

Toss-up 23:

Perl's delete built-in doesn't delete files. What six-letter built-in does?

Toss-up 24: A Perl cryptosystem code-named "Pontifex" appears in this novel. What is this large book by Neil Stephenson?

Toss-up 25: What three-letter pragma would you use to have your Perl program search for modules in particular directories? For instance, you'd say use _____

"temp" at the top of your program to have Perl look for modules in the temp directory. Fill in the blank.

Toss-up 26: When an XML document is validated, one says that it is validated against a ______. Fill in the blank.

Toss-up 27: Guess that function. Which built-in Perl function, given a file, tells you the inode of that file? This four-letter function also tells you the size of the file and when it was created Name this function

Toss-up 28: What is the four-letter name of the database manager bundled with Perl?

Toss-up 29: What special scalar variable defaults to 60, and contains the page length of the currently selected output handle? Name this punctuation variable.

Toss-up 30: It increases IP addresses from 32 bits to 128 bits, anticipating future growth of the Internet and providing relief for an impending shortage of network addresses. It provides for unicast (one host to one other host), anycast (one host to the nearest of multiple hosts), and multicast (one host to multiple hosts). What is the name of this new version of the Internet Protocol?

Toss-up 31: This little-known document lists all of the modules that have been installed into your Perl distribution. Its nine-letter name contains the name of the Perl function that provides dynamic scoping. Name this document.

Toss-up 32: When programmers tie a variable to a class, they often ignore the object returned by the built-in tie function.

What is the built-in Perl function that lets you retrieve the object after the tie has taken place?

Toss-up 33: The

Astro::SunTime module, available on the CPAN, calculates sunrises and sunsets. In what month is the latest sunrise in Alaska?

Toss-up 34: The -T command-line switch makes your program pay special attention to data arriving inside your program from the outside world. Such data is said to be until you convince Perl it's now safe, often with a regular expression. Fill in the blank.

Toss-up 35: In new versions of Perl, you can declare a subroutine's return value to be modifiable. For instance, if the subroutine is named foo, you'd be able to say foo(6) = 720, or foo("color") = "blue". This is accomplished by placing a particular word after the subroutine name in its declaration. What is that word?

Toss-up 36: It's a kernel written in

Perl that lets you schedule events, and it's also the last name of the poet who wrote "The Raven." Name this three-letter word.

Toss-up 37: On a system using the U.S. English locale and a plain ASCII character set—in other words, what most of us use right now—how many different characters can \w match inside a pattern? Identify this integer, one less than a perfect square.

Toss-up 38: Once you've opened a file, this Perl

function lets you move to an arbitrary position inside it. What is this

function, which takes three arguments: the filehandle, a position, and an offset?

Toss-up 39: The

Astro::MoonPhase module, available on CPAN, calculates phases of the moon. What word describes our moon when it appears more than half full, but not completely full?

Toss-up 40: This built-in function should probably have been called wantlist. What is the name of this Perl function, which can be used to determine whether your subroutine was called in a scalar context or a list context?

Toss-up 41: "Generate Regular Expression and Print" is widely claimed to be the expansion of this odd word, which is both a Unix utility and a Perl function that selects particular elements from a list. Name it.

Bonus Questions

Bonus 1: Suppose you're using the DBI

module to manipulate an Oracle database from Perl. What module containing Oracle-specific information would you also need?

Bonus 2: Fill in the blank to complete a well-known saying about the Internet.

On the Internet, no one knows you're a BLANK.

Bonus 3: Take the current four-digit year and assign it to a scalar. Chop it, chomp it, and chop it again. What does that scalar now hold?

Bonus 4: I'll give you six anagrams of Perl keywords; you give me the keyword.

- 1. Unlit
- 2. Eval Us
- 3. Mites
- 4. Mopes
- 5. Cute Rant
- 6. O Torch

Bonus 5: Itanium is one of a new family of 64-bit microprocessors from Intel that has begun to appear in new computers. Name the processor family.

Bonus 6: Within ten percentage points, what percentage of new domains ended in .com in 1998?

Bonus 7: I'll give you four operators that might or might not be in

Perl. After each, you tell me whether it's in Perl 5.6, yes or no.

```
1. ...
2. ~
3. ^^
4. ||||
```

Bonus 8: To access the last element of an array called @foo, you could use the index \$#foo. But you could also use what integer?

Bonus 9: This mod_perl

module lets your web server send different content depending on the speed of the connection. The module name has two words separated by a double colon. The first word should be easy to guess, and the second word names an object you'd find in an airplane cockpit. Name the two words.

Bonus 10: What two characters do you put in front of a string to make Perl interpret the string as a hexademical number?

Bonus 11: Name that Palm Pilot. I'll give you three types of Pilot, and you identify the model.

- 1. This Pilot provides wireless Internet access.
- 2. This Pilot is thinner than the rest.
- 3. This Pilot is full color.

Bonus 12:

RSA and Diffie-Hellman are examples of what kind of cryptography in which certain keys are made available for everyone to see?

Bonus 13: Eric

Raymond coined this phrase, which divides software projects into two categories: those carefully crafted by a small number of people in isolation, and those opened up as early as possible to as many people as possible. Name this phrase.

Bonus 14: Name either of the two

modules bundled with Perl that open a process for both reading and writing, letting you both provide it with input and see the output it displays.

Bonus 15: I'll give you six animals and six O'Reilly Perl titles. You match them up. The animals are the camel, emu, llama, leopard, sheep, and wolf. The books are *Advanced Perl Programming*, *Learning Perl, Learning Perl/Tk, Mastering Algorithms with Perl, Perl Cookbook*, and *Programming Perl.*

Bonus 16: I'll give you four operators, and you tell me about their precedence: whether they associate to the right or the left.

1. and 2. not 3. * 4 **

Bonus 17: I asked Jeeves the following question: "What are the arguments to split?" I'll give you five topics; you tell me whether that topic appeared in the resulting list of answers. Just say yes or no.

Perl's split function

- 2. The Microsoft breakup
- 3. Divorce
- 4. Banana splits
- 5. Stock splits

Bonus 18: What is the three-letter abbreviation for the encryption used on DVDs?

What does CSS stand for?

Bonus 19: When you tie a scalar to a class, there are three functions the class must provide. The functions are all in uppercase. What are they?

Bonus 20: Since you're on a cruise, we'll play Guess That Maritime Disaster. What ship sank three years after the Titanic, resulting in the loss of 1,198 lives from a German torpedo?

Bonus 21: This Perl extension was begun by astronomer Karl Glazebrook, who wanted the speed of mathematical packages like MATLAB and IDL with the convenience of Perl. What is this package, optimized for large multidimensional arrays of data?

Bonus 22: I'll ask you three questions about an interactive online discussion group named "pound perl."

What text-based collection of channels is "pound perl" part of?

What network of IRC channels is the heavily-populated #perl part of?

Finally, spell the name of the chatbot, written in Perl of course, that answers newbie questions without human intervention.

Bonus 23: Suppose \$x is a reference to an array. If you say \$y = \$x, you copy the reference, but not the underlying data. Copying the underlying data is called a *deep copy*. One module, bundled with Perl, contains a function called Deepcopy that helps you make deep copies. But the module is named after another useful function, one which stringifies data structures suitable for printing or saving to disk. Name that module.

Bonus 24: I'll name three symbols; you give me the ISO 8859-1 entity name. For instance, if I said "less-than sign", you'd say < . If I said "non-breaking space," you'd say .

- 1. Copyright symbol
- 2. An O with two dots over it
- 3. An E with a backtick over it

Bonus 25: Take a hash, any hash. Call it %X. It has some number of key/value pairs. Now create a new hash, %Y, in which the key/value pairs of %X become value/key pairs in %Y. That is, the keys of %X become the values of %Y, and the values of %Y become the keys of %X. Which of the following statements is true?

- 1. \$ Y is guaranteed to have the same number of key/value pairs as \$ X.
- 2. %Y is guaranteed to have no more key/value pairs than %X.

- 3. %X is guaranteed to have no more key/value pairs than %Y.
- 4. None of the above.

Bonus 26: You can use the

MIME::Lite

module to send email with MIME attachments. I'll give you three types of attachments you might send; you give me the MIME media type and subtype. For instance, if I said "An HTML document," you'd reply "text" and "html." "Text" is the media type, and "html" is the media subtype.

- 1. Plain text
- 2. A normal mail message, with headers
- 3. Generic binary data

Bonus 27: What system generates XS code from C and C++ programs, and in fact performs equivalent feats for Java, Tcl, and Python?

Bonus 28: Name that module. This module, which is bundled with

Perl, is a compiler backend that turns compiled Perl programs back into source code. It's sometimes helpful for seeing how Perl has parsed your program.

Bonus 29: True or false: with the latest release of Perl, you can now overload dereferencing, so that you can specify new behavior for what happens when you access, say, the scalar value behind a scalar reference, or a hash value behind a hash reference.

Bonus 30: When you create a Perl/Tk program, you choose a geometry manager to lay out your widgets. What are the three

geometry managers?

Bonus 31:

Slashdot runs on Perl code. Answer these three questions about the software underlying Slashdot.

- 1. What is the software underlying Slashdot called?
- 2. What web server does Slashdot use?
- 3. What database does Slashdot use?

Bonus 32: A bonus on cruises to other cold places.

Cruises to Antarctica typically originate on which two continents?

What is the name of the largest Antarctic base?

Bonus 33: What pod directive is used to explicitly indent text, as in an itemized list?

Bonus 34: The act of translating hostnames to Internet addresses is called name resolution, and the infrastructure supporting it is called DNS. Microsoft might want to call that the Digital Nervous System, but most of us know that it really stands for what?

Bonus 35: Rhyme that keyword: I'll give you four English words, you give me a Port built in that rhymos with it

Perl built-in that rhymes with it.

- 1. Peach
- 2. Sneeze
- 3. Slipped
- 4. Power

Bonus 36: If the actual temperature outside is forty degrees Fahrenheit, and the wind speed is forty miles per hour, what is the wind chill? That is, how cold does it feel? Your answer must be within three degrees Fahrenheit.

Bonus 37: This

module provides a function called ReadMode that lets the users of your program type characters without them echoing on the screen.

Bonus 38: What is a thousand terabytes called?

Bonus 39: This is Apple Computer's version of a new standard for connecting multimedia devices to your personal computer. It uses a plug-in serial connector and a data transfer rate in the hundreds of megabits, and lets you chain devices together without terminators.

Bonus 40: The person writing these questions is sick of hearing about ASPs. There are two completely different common expansions of this acronym, and he hates them both. What are the two expansions?

Bonus 41: A BEGIN

block is executed during compilation; an END

block is executed when the interpreter exits. Perl now has two new kinds of blocks. One is executed at the end of compilation, and the other is executed at the beginning of execution. What are the names of these two blocks?

Bonus 42: I'm looking for two mathematical functions that Perl provides. If you apply both functions to the same number and sum the squares of the results, the answer will always be 1. What are the two functions?
Bonus 43: The \$] variable contains the version number of your Perl, which is probably 5, plus the patchlevel divided by some number. What is that number?

Bonus 44: What does this print?

\$x = "1d4"; substr(\$x, 1, 1)++; print \$x-1;

Bonus 45: What array holds the command-line arguments for a Perl program?

Bonus 46: What regular expression metacharacter represents a word boundary?

Bonus 47: This company patented its affiliate program, which enables web sites to exchange customers for a small commission. They patented one-click ordering, and took their rival Barnes & Noble to court for infringement. What is this online bookseller founded by Jeff Bezos?

Bonus 48: A device is 80% likely to last ten years and 20% likely to last five years. What is the mean time between failures?

Bonus 49: With this

flag,

```
Perl pretends that every block in your program has a use warnings all declaration. Name this single letter.
```

Bonus 50: What regular expression extension keeps Perl from backtracking?

Bonus 51: This

function turns a list of values into a string, but it's not join. You can specify whether particular values should be interpreted as big-endian numbers, or hexademical numbers, or uuencoded strings, or regular characters. What is this four-letter function?

Toss-up Answers

T1. POTATOS.

T2. CENSORSHIP

T3. MAIN

T4. GEEK CRUISES

T5. T or BASETIME

T6. PNG (can be pronounced "ping")

T7. PERCENT

T8. Donald KNUTH

 $T9.\;\texttt{QISA}$

T10. SEARCH.CPAN.ORG

T11. DVD

T12. META

T13. LOWERCASE T

T14. SAMBA

T15. EXEC

T16. UNICODE

T17. NET::

T18.

QUEEN ELIZABETH II (also accept ELIZABETH Alexandra Mary WINDSOR) (Tidbit: The web site for the royal family uses Linux.)

T19. PROCESS ID or RETURN VALUE of fork

T20. The ILOVEYOU virus

T21. CONFIG

T22. BEOWULF

T23. UNLINK

T24.

CRYPTONOMICON

T25. LIB

T26. DTD or DOCUMENT TYPE DEFINITION

T27. STAT

T28. SDBM

T29. = orFORMAT_LINES_PER_PAGE

T30. IPV6 or IPNG

T31.

PERLLOCAL.pod (You can type perldoc perllocal to see what modules have been installed on your system.)

T32. TIED

T33. JANUARY. (Yes, the winter solstice is in December, and that's the longest day. But the longest day has neither the latest sunrise nor the earliest sunset.)

T34. TAINTed

T35. LVALUE

T36. POE

T37. 63

T38. SEEK

T39. GIBBOUS

T40. WANTARRAY

T41. GREP

Bonus Answers

- B1. DBD::ORACLE
- **B2**. DOG
- **B3**. TWENTY

B4. UNTIL, VALUES, TIMES, SEMOP, TRUNCATE, CHROOT

B5. MERCED

B6. 84 (Full credit for an answer between 74 and 94.)

B7. YES, YES, NO, NO

B8. -1

B9. APACHE::THROTTLE

B10. 0X ("ZERO" and "EX")

B11. 7, 5, 3C

B12. PUBLIC KEY cryptography

B13.

The CATHEDRAL and the BAZAAR

B14. IPC::OPEN2 or IPC::OPEN3

B15. CAMEL => PROGRAMMING

PERL, EMU => Learning PERL/TK, LLAMA => LEARNING PERL, LEOPARD => ADVANCED PERL Programming, SHEEP => The Perl COOKBOOK, WOLF => Mastering ALGORITHMS with Perl

B16. LEFT, RIGHT, LEFT, RIGHT

B17. YES, NO, YES, NO, YES

B18. CSS (DeCSS is the program that decrypts CSS), CONTENT SCRAMBLING SYSTEM

B19. TIESCALAR, FETCH, STORE

B20. LUSITANIA

B21. PDL

B22. IRC or INTERNET RELAY CHAT, EFNET, P U R L

B23. DATA::DUMPER

B24. & copy;, & ouml;, & egrave;

B25. 2

B26. TEXT/PLAIN, MESSAGE/RFC822, APPLICATION/ OCTET-stream

B27. SWIG

B28. B::DEPARSE

B29. TRUE

B30. PACKer, GRIDder, PLACEr.

B31. SLASHcode, APACHE, MySQL

B32. AUSTRALIA and SOUTH AMERICA, MCMURDO

B33. OVER

B34. DOMAIN NAME SYSTEM

B35. EACH, KEYS, CRYPT, OUR

B36. TEN (accept between SEVEN and THIRTEEN)

B37. TERM::READKEY

B38. PETABYTE

B39. FIREWIRE

B40. APPLICATION SERVICE PROVIDER and ACTIVE SERVER PAGEs

B41. CHECK and INIT

B42. SINe and COSine

B43. 1000

B44. 9999

B45. Argv

B46. \B (no need to specify case)

B47. AMAZON

B48. NINE years

B49. –W (if the player doesn't say "Capital W," say "be more specific")

B50. ?>

B51. PACK

Chapter 36. The Perl Wizard's Quiz

Tom Christiansen

Answers are at the end of the article.

- 1. What value is returned by a lone return statement?
 - a. The empty list value ().
 - b. The undefined value in scalar context, and the empty list value () in list context.
 - c. The result of the last evaluated expression in that subroutine's block.
 - d. The undefined value.
- 2. What's the difference between /^Foo/s and /^Foo/?
 - a. The first would allow the match to cross newline boundaries.
 - b. The first would match F00 other than at the start of the record if the previous match were /^F00/gcm, new in the 5.004 release.
 - c. The second would match $F \circ \circ$ other than at the start of the record if \$ * were set.
 - d. There is no difference because /s only affects whether dot can match newline.

- 3. What does length (%HASH) produce if you have 37 random keys in a newly created hash?
 - a. 5
 - b. 37
 - c. 74
 - d. 2
- 4. What does read return at end of file?
 - a. 0
 - b. "0 but true"
 - c. "\0"
 - d. undef
- 5. How do you produce a reference to a list?
 - a. [@array]
 - b. (\$s, @a, \$h, &c)
 - c. You can't produce a reference to a list.
 - d. $\ensuremath{\ensuremath{\texttt{0}}}$ d.
- 6. Why aren't

Perl's patterns regular expressions?

- a. Because Perl allows both minimal matching and maximal matching in the same pattern.
- b. Because Perl uses a non-deterministic finite automaton rather than a deterministic finite automaton.

- c. Because Perl patterns can have look-ahead assertions and negations.
- d. Because Perl patterns have backreferences.
- 7. Why doesn't Perl have overloaded functions?
 - a. Because you can inspect the argument count, return context, and object types all by yourself.
 - b. It does, along with overloaded operators as well as overridden functions and methods.
 - c. Because Perl doesn't have function prototypes.
 - d. Because it's too hard.
- 8. Why is it hard to call this function: sub y {
 "because" }?
 - a. It's not.
 - b. Because y is a predefined function.
 - c. Because it has no prototype.
 - d. Because y is a kind of quoting operator.
- 9. How do you print out the next line from a filehandle with all its bytes reversed?
 - a. print reverse scalar <FH>
 - b. print scalar reverse scalar <FH>
 - c. print scalar reverse <FH>
 - d. print reverse <FH>
- 10. When would local \$ in a function ruin your day?

- a. When your caller was in the middle of a while (<>) loop.
- When your caller was in the middle of a while (m//g) loop.
- c. When \$ was imported from another module.
- d. When your caller was in the middle of a foreach(@a) loop.
- 11. Which of these is a difference between C++ and Perl?
 - a. C++ can have objects whose data cannot be accessed outside its class, but Perl cannot.
 - b. C++ supports multiple inheritance, but Perl does not.
 - c. C++ will not call destructors on objects that go out of scope if a reference to that object still exists, but Perl will.
 - d. Perl can have objects whose data cannot be accessed outside its class, but C++ cannot.
- 12. Assuming both a local(\$var) and a my(\$var) exist, what's the difference between \${var} and \${``var"}?
 - a. \${var} is the package variable \$var, and \${"var"} is the scoped variable \$var.
 - b. There is no difference.
 - c. \${var} is a package variable \$var, and \${"var"} a global variable \$var.

- d. \${var} is the lexical variable \$var, and
 \${"var"} is the dynamic variable \$var.
- 13. If *EXPR* is an arbitrary expression, what is the difference between \$F00::{*EXPR*} and *{``F00::".*EXPR*}?
 - a. The second is disallowed under use strict "refs".
 - b. The first happens at runtime, the second at compile time.
 - c. One is just a regular hash, the other a typeglob access for a strangely named variable.
 - d. The first can create new globs dynamically, but the second cannot.
- 14. Assuming \$_ contains HTML, which of the following substitutions will remove all tags in it?
 - a. s/<.*>//g;
 - b. s/<.*?>//gs;
 - c. s/<\/
 ?[A-Z]\w*(?:\s+[A-Z]\w*(?:\s*=\s*(?:(["']
 .]+))?)*\s*>//gsix;
 - d. You can't do that.
- 15. What does new \$cur->{LINK} do? (Assume the current package has no new function of its own.)
 - a. \$cur->new()->{LINK}
 - b. new(\$cur->{LINK})

- c. \$cur ? (\$cur->{LINK}->new()) :
 (new()->{LINK})
- d. $cur > {LINK} -> new()$
- 16. What does \$result = f() .. g() really return?
 - a. It produces a syntax error.
 - b. True if and only if both f () and g () are true, or if f () and g () are both false, but returns false otherwise.
 - c. False so long as f () returns false, after which it returns true until g () returns true, and then starts the cycle again.
 - d. The last number from the list of numbers returned in the range between f()'s return value and g()'s.
- 17. What happens when you return a reference to a private variable?
 - a. The underlying object is silently copied.
 - b. Nothing bad—it just works.
 - c. The compiler doesn't let you.
 - d. You get a core dump later when you use it.
- 18. How do you give functions private variables that retain their values between calls?
 - a. Include them as extra parameters in the prototype list, but don't pass anything in at that slot.
 - b. Use localized globals.

- c. Create a scope surrounding that sub that contains lexicals.
- d. Perl doesn't support that.
- 19. What happens to objects lost in "unreachable" memory, such as the object returned by \$Ob->new() in this block?

```
{ my $ap; $ap = [ $Ob->new(), \$ap ]; }
```

- a. Their destructors are called when the memory becomes unreachable.
- b. Their destructors are never called.
- c. Perl doesn't support destructors.
- d. Their destructors are called when that interpreter thread shuts down.
- 20. What does Perl do if you try to exploit the execve (2) race condition involving setuid scripts?
 - a. Sends mail to root and exits.
 - b. Runs the fake script with setuid permissions.
 - c. Runs the fake script, but without setuid permissions.
 - d. Reboots your machine.

Answers

1. b. This way functions that wish to return failure can just use a simple return without worrying about the context in which they were called. If you answered a: That would only be true in list context.

If you answered c: That's what happens when the function ends without return being used at all.

If you answered d: That would only be true in scalar context.

c. The deprecated \$* flag does double duty, filling the roles of both /s and /m. By using /s, you suppress any settings of that spooky variable, and force your carets and dollars to match only at the ends of the string and not at the ends of the line as well—just as they would if \$* weren't set at all.

If you answered a: /s only makes a dot able to match a newline, and then only if the string actually has a newline in it.

If you answered b: Although the /c modifier is indeed new as of 5.004 (and is used with /g), this has no particular interaction with /s.

If you answered d: /s does more than that.

3. a. length is a built-in function prototyped as sub length(\$), and that scalar prototype silently changes aggregates into radically different forms. The scalar sense of a hash is false (0) if it's empty, otherwise it's a string representing the fullness of the hash buckets, like 18/32 or 39/64. The length of that string is likely to be 5. Likewise, length(@a) would be 2 if there were 37 elements in @a. If you answered b: length %HASH is nothing at all like scalar keys %HASH, which is a good bit more useful.

If you answered c: length %HASH is nothing at all like the size of the list of all the keys and values in %HASH.

If you answered d: You probably think it decided there were 37 keys, and that length(37) is 2. Close, but not quite.

4. a. A defined (but false) 0 value is the proper indication of the end of file for read and sysread.

If you answered b: You're thinking of the ioctl and fcntl functions, which return this when the C version returned 0, reserving undef for when the C version returns -1. For example, fcntl(STDIN, F_GETFL, 1) returns "0 but true" depending on whether and how standard input has been redirected. (The F_GETFL flag can be loaded from the Fcntl.pm module.)

If you answered c: That's a string of length 1 consisting of the NULL character, whose ord is 0, which is false. The string, however, is true. read doesn't return strings, but rather byte-counts.

If you answered d: That would signal an I/O error, not normal end of file. The circumfix operator <> returns undef when it reaches end of file, but a normal read does not. 5. c. A list is not an array, although in many places one may be used for the other. An array has an AV allocated, whereas a list is just some values on a stack somewhere. You cannot alter the length of a list, for example, any more than you could alter a number by saying something like 23++. While an array contains a list, it is not a list itself.

If you answered a: That makes a reference to a newly allocated anonymous array, and populates it with a copy of the contents of @array.

If you answered b: The backslash operator is distributive across a list, and produces a list in return, this being $(\$ s, \@a, \%h, \&c) in list context. In scalar context, it's a strange way to get a reference to the function &c.

If you answered d: @array is not a list, but an array.

6. d. A regular expression (by definition) must be able to determine the next state in the finite automaton without requiring any extra memory to keep around previous state. A pattern / ([ab]+)c\1/ requires the state machine to remember old states, and thus disqualifies such patterns from being regular expressions in the classic sense of the term.

If you answered a: The mere presence of minimal and maximal repetitions does not disqualify a language from being regular.

If you answered b: Both NFAs and DFAs can be used to solve regular expressions. Given an NFA, a DFA for it can be constructed, and vice versa. For example, classical grep uses an NFA, while classical egrep a DFA. Whether a pattern matches a particular string doesn't change, but where the match occurs may. In any case, they're both regular. However, an NFA can also be modified to handle backtracking, while a DFA cannot.

If you answered c: The (?=foo) and (?!foo) constructs no more violate the language's regularity than $^$ and , which are also zero-width statements.

7. a. In

Perl, the number of arguments is available to a function via the scalar sense of @_, the return context is available via wantarray, and the types of the arguments via ref (if they're references) and simple pattern matching like /^\d+\$/ (otherwise). In low-level languages like C++, where you can't do this, you must resort to overloading of functions.

If you answered b: Actually, Perl does support overloaded operators via use overload, overridden functions as in use Cwd qw!chdir!, and overridden methods via inheritance and polymorphism. It just doesn't support functions automatically overloaded on parameter signature or return type. Not that such isn't longed for.

If you answered c: Perl actually does have function prototypes; however, this isn't used for the traditional sort of prototype checking, but rather for creating functions that exactly emulate Perl's built-ins, which can implicitly force context conversion or pass-by-reference without the caller being aware. If you answered d: Just because it's hard isn't likely to rule out something from being implemented—someday.

 d. The y/// operator is the sed-savvy synonym for tr. That means y(3) would be like tr(3), which would be looking for a second string, as in tr/a-z/A-Z/, tr(a-z)(A-Z), or tr[a-z][A-Z].

If you answered a: Most people don't call functions with ampersands anymore. If they did, as in &y(), it wouldn't be so hard.

If you answered b: y isn't really a function, per se. If it were, you would never see y!abc!xyz!, since proper functions do not like getting banged on that way.

If you answered c: Functions don't require prototypes in Perl.

9. b. Surprisingly enough, you have to put both the reverse and the <FH> into scalar context separately for this to work.

If you answered a: Although scalar <FH> did retrieve just the next line, the reverse is still in the list context imposed on it by print, so it takes its list of one element and reverses the order of the list, producing exactly the next line. An expensive way of writing print scalar <FH>.

If you answered c: Although the first use of scalar inhibits the list context being imposed on reverse by print, it doesn't carry through to change the list context that reverse is imposing on <FH>. So reverse catenates all its arguments and does a byte-for-byte flip on the resulting string.

If you answered d: That reads all lines in FH, then reverses that list of lines and passes the resulting reversed list off to print. This is actually a very useful thing, and simulates tail -r behavior but without the annoying buffer limitations of that utility. Nonetheless, it's not what we want.

 b. The /g state on a global variable is not protected with local. That'll teach you to stop using locals. Too bad \$_ can't be the target of a my—yet.

If you answered a: However, if you do a while (<>) and forget to first localize $\$_$, you'll hurt someone above you. That's because even though foreach implicitly localizes $\$_$, the while (<>) construct does not.

If you answered c: Doing a local on an imported variable is not harmful. Of course, in the case of $\$_$, it's virtually unnecessary, since $\$_$ is always forced to mean the version in the main package, that is, \$main::.

If you answered d: This looks close to the bizarre phenomenon known as variable suicide, but that only occurs in ancient Perl versions.

11. d. Perl can use closures with unreachable private data as objects, and C++ doesn't support closures. Furthermore, C++ does support pointer arithmetic via int *ip = (int*)&object, allowing you to look all over the object. Perl doesn't have pointer arithmetic. It also

doesn't allow #define private public to change access rights to foreign objects. On the other hand, once you start poking around in /dev/mem, no one is safe.

If you answered b: Both support multiple inheritance.

If you answered c: Exchange "Perl"

and "C++" in that answer, and you would be telling the truth. C++ is too primitive to know when an object is no longer in use, because it has no garbage collection system. Perl does.

12. d. Odd though it appears, this is how it works. Note that because the second is a symbol table lookup, it is disallowed under use strict "refs". The words global, local, package, symbol table, and dynamic all refer to the kind of variables that local affects, whereas the other sort, those governed by my, are variously known as private, lexical, or scoped variables.

If you answered a: Try again. You're close.

If you answered b: One is the scoped variable, the other the package variable. Which is which, though?

If you answered c: There is no difference between a package variable and a global variable. All package variables are globals, and vice versa.

13. a. Dereferencing a string with * { "STR" } is forbidden under the refs stricture, although * {STR} is allowed. This is similar in spirit to the way \$ { "STR" } is always the symbol table variable, while \$ { STR } may be the

lexical variable. If it's not a bareword, you're playing with the symbol table in a particularly dynamic fashion.

If you answered b: Assuming that the expressions don't get resolved at compile time, this all has to wait until run time. Something like *Foo::varname, however, would be looked up at compile time.

If you answered c: The %F00:: hash is always the symbol table associated with package Foo; such a hash can hardly be called regular. Both versions actually refer to the same typeglob, although somewhat differently.

If you answered d: Although you can get the symbol table of the Foo package via the Foo: hash, you cannot usefully generate new typeglobs (symbols) this way. You could copy old ones into that slot, though, effectively doing the Exporter's job by hand.

14. d. If it weren't for HTML comments, improperly formatted HTML, and tags with interesting data like <SCRIPT>, you could do this. Alas, you cannot. It takes a lot more smarts, and quite frankly, a real parser.

If you answered a: As written, the dot will not cross newline boundaries, and the star is being too greedy. If you add a /s, then yes, it will remove all tags—and a great deal else besides.

If you answered b: It is easy to construct a tag that will cause this to fail, such as ">.

If you answered c: For a good deal of HTML, this will actually work, but it will fail on cases with annoying

comments, poorly formatted HTML, and tags like <SCRIPT> and <STYLE>, which can contain things like while (<FH>) {} without those being counted as tags. Comments that will annoy you include <!---<foo bar = "-->"> which will remove characters when it shouldn't; it's just a comment followed by ">. And even something like <!-- <foo bar = "--> most browsers will get right, but the substitution will not. And if you have improper HTML, you get into even more trouble, like this: <foo bar = "bleh" @> text text text <foo bar = "bleh". Here, the .*? will gobble up much more than you thought it would.

15. a. The indirect object syntax only has a single token lookahead. That means if new is a method, it only grabs the very next token, not the entire following expression. This is why new \$obj[23] arg doesn't work, as well as why print \$fh[23] "stuff\n" doesn't work. Mixing notations between the OO and IO notations is perilous. If you always use arrow syntax for method calls, and nothing else, you'll never be surprised.

If you answered b: If the current package did in fact have its own new function, then this would be the right answer, but for the wrong reasons. Within a class, it might appear to make no difference since the new subroutine would get its argument in $\$_[0]$ whether it's called as a function or a method. However, a method call can use inheritance, while a function call never does. That means esoteric overridden new methods would be duped out of calling their derived class' constructor first, and we wouldn't want that to happen, would we? If you answered c:

Perl may be crazy, but it's not quite that crazy. Yet.

If you answered d: Just because it looks like a unary function doesn't mean a method call parses like one. You just want it to work this way. If you want that, write that.

16. c. This is scalar context, not list context, so we have the bistable flip-flop range operator famous in parsing of mail messages, as in \$in_body = /^\$/ .. eof(). Except for the first time f() returns true, g() is entirely ignored, and f() will be ignored later when g() is evaluated. Double dot is the inclusive range operator; f() and g() will both be evaluated on the same record. If you don't want that to happen, the exclusive range operator, triple dots, can be used instead. For extra credit, describe this: \$bingo = (a() .. b()) ... (c() ... d());

If you answered a: You'd be amazed at how many things in Perl don't cause syntax errors.

If you answered b: That sounds more like a negated logical xor. A logical xor is !\$a != !\$b, so you've just described !\$a == !\$b. Interesting, and perhaps even useful, but unrelated to ..., our scalar range operator.

If you answered d: That might work in list context, but never in scalar. The list operator . . is a completely different creature than the scalar one. They're just spelled the same way, kind of like when you can the rusty old can down by the guys' can just because you can. Context, as always, is critical.

17. b. Perl keeps track of your variables, whether dynamic or otherwise, and doesn't free things before you're done using them.

If you answered a: Even though the reference returned is for all intents and purposes a copy of the original (Perl uses return by reference), the underlying referent has not changed.

If you answered c: Perl seldom stops you from doing what you want to do, and tries very hard to do what you mean to do. This is one of those cases.

If you answered d: Perl is not C or C++.

18. c. Only lexical variables are truly private, and they will persist even when their block exits if something still cares about them. Thus { my \$i = 0; sub next_i { \$i++ } sub last_i { --\$i } } creates two functions that share a private variable. The \$i variable will not be deallocated when its block goes away because the next_i and last_i subroutines need to be able to access it.

If you answered a: Perl is not the Korn shell, nor anything like it. If you tried this, your program probably wouldn't even compile.

If you answered b: The local operator merely saves the old value of a global variable, restoring that value when the block in which the local occurred exits. Once the subroutine exits, the temporary value is lost. Before then, other functions can access the temporary value of that global variable.

If you answered d: It would be difficult to keep private state in a function otherwise.

- 19. d. When the interpreter exits, it first does an exhaustive search looking for anything that it allocated. This allows Perl to be used in embedded and multithreaded applications safely, and furthermore guarantees correctness of object code.
- 20. If you answered a: Under the current implementation, the reference-counted garbage collection system won't notice that the object in \$ap's array cannot be reached, because the array reference itself never has its reference count go to zero.

If you answered b: That would be very bad, because then you could have objects whose class-specific cleanup code didn't get called ever.

If you answered c: A class's DESTROY function, or that of its base classes, is called for any cleanup. It is not expected to deallocate memory, however.

21. a. It has been said that all programs advance to the point of being able to automatically read mail. While not quite there yet (well, without loading a module), Perl will at least automatically send it.

If you answered b: That would be bad. Very Bad. What do you think we are? A shell or something?

If you answered c: It would be improper to run anything at all in the face of such naughtiness.

If you answered d: An appealing idea, though, isn't it? After all, Perl does possess super(user)powers at this point. You just never know what it might do. In the interests of courtesy, though, Perl stays out of your power supply just as it stays out of your living room.

Part V. Poetry

In this part:
Chapter 37
Chapter 38
Chapter 39

I received the following note from a poetry teacher as this book was zooming toward publication:

What the heck is a Perl poem? I've been writing and teaching poetry since 1961, and I never heard of the term. I'd like to know so as to see if my students and I can write perl(s).

My response:

Conventional poems are passive: they can't do anything other than sit on a page and wait for people to read them. Perl poems, on the other hand, are functional: they are active programs that a computer can execute. What the poem/program does is limited only by the poet's imagination.

The most flexible computer language naturally lends itself to this most flexible linguistic endeavor. Perl poetry has been around since 1990; the original Perl poet, Sharon Hopkins, has had her work published in the *Economist* and *Guardian*.

In this section, Damian Conway discusses his Coy module, which renders error messages as haiku. Sean Burke then

shows you how to use Perl to find rhymes with the proper stress and meter, and Kevin Meltzer and I conclude the section with the results of the first ever Perl Poetry Contest.

Chapter 37. Just Another Perl Haiku

Damian Conway

I often think of Perl programs as the haiku of the software world.

Both are compact, dense, powerful, and frequently a little obscure.

So it's no surprise that haiku are popular with Perl programmers.

Even so, I had no idea *how* popular, until I wrote Coy.

What is Coy? Let me quote from the docs you'll find with this CPAN module.

Error messages

strewn across my terminal. A vein starts to throb.

Their reproof adds the injury of insult to the shame of failure.

When a program dies what you need is a moment of serenity.

The Coy.pm module brings tranquility to your debugging.

The module alters the behavior of die and warn (and croak and carp).

Like Carp.pm, Coy reports errors from the caller's point of view.

But it prefaces the bad news of failure with a soothing poem.

The Tao of Haiku

A haiku is a short poem that's 17 syllables in length.

Traditionally, its topic is an image taken from nature

(though the Japanese understanding of "nature" is subtle and broad).

True haiku don't try to make a point; they merely convey an image.

Of course, the image itself may make a point, but that's not the same thing!

The form developed in the 1600's from the longer *tanka*.

In fact, a haiku is the *hokku*(the "starting verse") of a tanka. The first adept of the haiku format was the Zen poet Basho.

His best known poem captures the wonder of an everyday event.

It's: furike ya kawazu tabikomu mizu no oto.

Which is normally translated as: *An old pond*. *A frog jumps in*. *Plop!*

As you see, there's no message. The haiku is a pure evocation.

Haiku Online

The 5-7-5 art form is widely practiced on the Internet.

There are many sites devoted to haiku and to related styles

(most notably, the satirical variants known as *senryu*).

The easiest place to start is with a search of the Yahoo! index.^[8]

Jane Reichhold also gives a superb summary^[9] of the haiku form.

And for a deeper view, Keiko Imaoka's page is unsurpassed.^[10]

There is even a strictly SF variant known as "SciFaiku."^[11]

- [8] http://search.yahoo.com/bin/search?p=haiku
- [9] http://www.faximum.com/aha.d/haidefjr.htm
- ^[10] http://www.faximum.com/aha.d/keirule.htm
- ^[11] http://www.scifaiku.com/
Artificial Haiku

Many web sites now also feature haiku that are not carbon-based.

In fact, the Web is awash with generators of Japanese verse.

A simple search^[12] finds over 5,000 links for: +generate +haiku.

Silicon Graphics even rigged a lava-lamp to build (bad) haiku:

> *i think i'm wasted i'll wax the cats. cool clear earth pigs are smarter. crash*

(That is one of its clearer efforts. Mostly it

just spouts gibberish.)

In contrast, Garret Kaminaga makes use of a "haiku grammar."^[13]

Its simple rules (see

Haiku-Generating Grammar (by G. Kaminaga)) expand to give correct syllables.

```
Haiku-Generating Grammar (by G. Kaminaga)
haiku: five line seven line five line
five line: one four | one three one | one one three | one two
two |
  one two one one | one one two one | four
  one | five
seven line: one one five line | two five line | five line one
one | five line two
one: red | white | black | sky | dawns | breaks
falls
  cranes | rain | pool | my | your | sun |
  clouds | tree | Zen
two: drifting | purple | mountains | faces
empty | temple |
  ocean | thinking | zooming | rushing |
  over | ricefields
three: peasant farms | computer | sashimi
fishing boats | ethernet
four: CD Player | aluminum | yakitori
chrysanthemums
```

five: resolutional | rolling foothills rise

But they don't encode any English grammar, so the results are poor:

> empty computer yakitori to empty. your chrysanthemums

Richard Decker and Stuart Hirshfield also use grammars for haiku.

But theirs are based on real English sentence structures (as in Haiku-Generating Grammar (by Decker and Hirshfield)).

Haiku-Generating Grammar (by Decker and Hirshfield)

haiku: form1 | form2 | form3

form1: article adjective noun

```
article noun verb preposition article
noun
adjective adjective noun
```

form2: noun preposition article noun

article adjective noun preposition article noun adjective noun

form3: article adjective adjective noun

preposition article adjective noun article noun verb

noun:waterfall | river | breeze | moon
| rain | wind | sea | sky | storm

verb: shakes | drifts | has stopped
struggles | whispers | grows | flys

adjective: liquid | gusty | flowing
autumn | hidden | bitter | misty
summer

As a result, they generate quite plausible (and lovely) haiku:

> A liquid summer wind. Under the gusty sky a storm whispers. [sic]

Unfortunately, their English grammar doesn't encode syllables. Consequently, most of the haiku they produce don't scan correctly.

As these samples show, a haiku generator must balance two things:

It must use correct English syntax and it has to track syllables.

As we'll see, Coy is built around those two constraints. grammar and meter.

[12]

http://www.google.com/ search?hl=en&q=generate+haiku&btnG=Google+Search

[13] http://www.cs.stanford.edu/~zelenski/rsg/grammars/ Haiku.g

The Coy Module

Coy is simple to use: just add use Coy; to your existing program.

Hereafter, any calls to die, warn, croak, or carp produce a haiku.

This magic is wrought by assigning a handler to \$SIG{__DIE__}.

That handler passes the string it receives to Coy's verse generator.

It then re-calls die with the resulting haiku as its argument.

The same approach is applied to \$SIG{__WARN__}, to catch warnings too.

Apart from handlers,

use Coy also exports two extra subroutines.

These subs, transcend and enlighten, can lend your code a Zen overtone.

(But internally, they are each just a wrapper around croak or carp.)

A Note About the Name

Just in case there's still anyone in the U.S. who *hasn't* sent mail...

Yes, I know you folks spell the fish the same way as the Japanese: *koi*.

We Aussies, like our British forebears, prefer to anglicize it: *coy*.

And the triple pun ("coy: (a) fish; (b) calm; (c) trap") needs the English form.

But if you insist, you can always dash this off from your command line:

```
perl -> Koi.pm
print STDOUT "use Coy; 1;"
^D (or Z)
```

Inside the Coy Module

Coy's generator is data-driven and has five main components:

- A cross-linked list of words that provides its basic vocabulary.
- A mechanism for selecting relevant words for each haiku.
- A set of sentence templates that ensure the text is grammatical.
- A set of routines that correctly inflect words within each sentence.
- An algorithmic syllable counter, which checks that each haiku scans.

We'll briefly look at how each of these components goes about its task.

Mere Words

Ultimately, a haiku is just a sequence of well-chosen words.

Coy's words are stored in a hierarchical, cross-linked

vocabulary.

Structure of the Coy Hierarchical Vocabulary shows an abbreviated sample of that database.

The top level of word categorization is by subject nouns.

For each such noun, a set of global constraints and sounds is then given.

The sounds are used as verbs to generate clauses describing noises.

In addition, a list of more general verb forms

(act) is specified.

Each such verb, listed in third-person singular, may take attributes.

These attributes list constraints on the verb's usage (such as location).

The entry for duck => swims, for instance, locates it as suraquatic.

```
Structure of the Coy Hierarchical Vocabulary
  $database = {
       duck => {
           category => [ "bird" ],
           sound => [ "quacks" ],
                         => { swims => {
              act
  location
              => "suraquatic",
  direction => "horizontal",
  synonyms => [ "paddles" ],
  associations => "sink wet" } },
       },
       fox => \{
                category => [ "animal",
  "hunter" ],
           sound
                    => [ "barks" ],
                          => { trots => {
              act
  location
              => "terrestrial",
```

```
associations => "smart problem" } },
     },
    lover => {
        category => [ "human" ],
         sound => [ "sighs", "laughs"
],
        minimum => 2,
        maximum => 2,
          act => \{ kisses => \{
location => "terrestrial",
associations => "connection", },
                          quarrels => {
location => "terrestrial",
associations => "argument" } },
    },
};
```

Other attributes limit the subject count for particular verbs.

lover => kisses, for
example, is limited
to exactly 2.

Random Harvesting

Coy's next component is an association selection system.

This system ensures that the haiku relates to the error message.

The message is first scanned to find significant words (principally nouns).

These words are found by deleting "stop words" from the original text.

"Stops" are words such as *the, to, it,* and the like, that don't convey content.

The remaining words become a "filter" for the vocabulary.

Each word selected for the

haiku is compared against this filter.

If the selected word's associations don't match, it's rejected.

This leads to problems though, if the filter words are too unusual.

In extreme cases, they may filter out the whole vocabulary.

To prevent this, Coy can turn the word filter off temporarily.

Filling in the Blanks

The third component generates the haiku, by filling in templates.

Those templates encode various grammatical structures for haiku.

The generator selects one and fills it in with relevant words.

Sample English Grammar Templates Used by Coy.pm shows a few of the grammatical templates Coy uses.

Sample English Grammar Templates Used by Coy.pm

haiku_fragment: sentence | description | exclamation

sentence: noun verb | *noun verb direction* | *noun verb location*

description: noun location | pres_participle noun

exclamation: noun

verb: simple_present | pres_participle

Note that the grammar has no terminals: they're drawn from the database.

Templates are chosen at random, as often as needed (Sample English Grammar Templates Used by Coy.pm).

The chosen template is then filled in with "filtered" semi-random words.

The noun to be used is randomly selected, and constrains the verb.

Any later parts of the grammar are likewise constrained by that verb.

These are typically adverbial phrases of place or direction.

For instance, suppose the filtered noun chosen is the word *hummingbird*.

Immediately

this constrains the verb to words like *flies, darts*, or *nests*.

If *flies* were chosen, that would then constrain the place to be aerial.

Whereas, if *nests* were chosen, the place would have to be arboreal.

Note that Coy needs no AI techniques to enforce these sequenced constraints.

The hierarchical vocabulary structure itself ensures them.

The Other Type of Grammar

But selecting the right parts of speech—and words to match—is not enough.

The module must then adjust the selected words' grammatical form.

Specifically, the words used must be inflected for number and tense.

Lingua::EN::Inflect is used to supply correct noun/verb agreement.

Each time a grammar template is filled, Coy selects a random number.

That number becomes the number of subjects for the current sentence.

If 1 is chosen, the subject noun and its verb are left as they are.

Otherwise, Inflect's PL sub is used to change them both to plurals.

Lingua::EN::Inflect can also inflect present to continuous.

The PART subroutine takes singular verbs and forms their participles.

This is useful to increase the variety of sentences used.

Thus, sometimes *birds fly*, some *birds* are *flying*, and the rest are *flying birds*.

Inflecting present participles is harder than it might first seem.

Consider the verbs: bat, combat, eat, bite, fulfil(l), lie, sortie, and ski.

Each adds *-ing* after it (all verbs do), but each root

inflects differently.

The rules are complex, but the PART sub knows them all, and can apply them.

Counting the Beat

The four components above ensure the haiku parses and makes sense.

However, there's no guarantee that the result scans 5-7-5.

To ensure perfect meter, each selected word's syllables are checked.

This occurs while the grammar templates are filled in (as words are filtered).

The selector tracks the progressive syllable count of the words used.

If the count exceeds 17, the selector can reject a word.

The selection can also backtrack further, if

that's necessary.

In some cases this might cause the template itself to be rejected.

The template-filling process then repeats until the full haiku scans.

But Does the Bear Dance?

By now you're thinking "Who cares how the magic works: Show me the MONEY!"

So here are a few of Coy's more interesting creative efforts...

Given the fatal error: die "Bad argument", Coy replied with this:

> A pair of lovers quarrel beside a stream. Four thrushes fly away.

Note the allusion to the Bad argument in the error message.

Haiku are never repeated. A second die "Bad argument" gave:

> Two old men fighting under a sycamore tree. Homer Simpson sighs.

In contrast, for a croak "Missing file", Coy captured the sense of loss with:

Bankei weeping by a lake. Ryonen dying. Seven howling bears.

Coy can't always reach this exalted level of (oblique) relevance.

For example, it also produced this response to croak "Missing file":

> A swallow nesting in the branches of an elm tree. A waiting fox.

Sometimes Coy's output suggests a macabre sense of humor, as in:

A wolf leaps under a willow. Two old men sit under the willow.

In other cases, its inscrutability is most authentic: Two young women near Bill Clinton's office. A cat waiting by a pond.

Extending the Module

Eventually even Coy's built-on-the-fly haiku get boring.

The module only has a small set of words from which to build poems.

After a while the same topics start recurring— Coy's "theme" emerges.

True to its roots, Coy always writes about fish, birds, animals, and trees.

That fixation on certain flora and fauna soon begins to grate.

So Coy provides a way to configure its own vocabulary.

Any code placed in ~/.coyrc runs at compile-time.

```
You can use that file
to extend
Coy.pm's
mental horizons.
```

Adding a Star Wars Flavor to the .coyrc File shows how to add, for example, a Jedi leitmotif.

```
Adding a Star Wars Flavor to the .covrc File
  > cat ~/.coyrc
  noun { wookie => {
        category => [ Sentient ],
             sound => [ "roars", "grunts",
  "bellows" ],
             act => { sits => { location
  => Arboreal },
                       fights => { minimum
  => 2, association => "argument" }}};
               "R2D2",
                          "Darth Vader",
  personage
  "Obi-wan", "George Lucas";
                 "Mos Eisley", "the Death
  place
  Star", "Skywalker Ranch";
                 "Kashyyyk oak", "Alderaan
  tree
  mangrove";
  fruit tree "Ewok-fruit", "Yavin mango",
  "Hothberry";
```

Is There a Poet in the House?

At TPC in 1999 I ran a haiku contest.

At my talk on Coy I asked the audience to write me some poems.

The two cleverest (IMHO) would each get a book.

David Adler won the first, by demonstrating that blackmail's an art:

> On-the-spot haiku written on a lady's back. Know we have pictures!^[14]

Dean Hudson won by coercion too, though his was metaphysical: Your book entices, fork it over, friend Aussie. Jedi mind tricks work.

[14] http://www.stonehenge.com/merlyn/Pictures/99-08-TPC3/Day-2-Sun/?start=62&end=64

It Seemed Like a Good Idea at the Time

Picking those two gems from the hoard of glittering entries was great fun.

So I decided to expand the contest to the whole conference.

I knew I needed help: Mark Jason Dominus and Elaine Ashton.

They kindly agreed to lend their artistic good taste as co-judges.

Then Tim O'Reilly generously donated prizes: book vouchers.

Next we collected haiku from the 800 or more delegates.

From over fifty entries we selected two as the joint winners.

Michael Schwern triumphed with a meta-haiku that's true Perl poetry:

Life ends with a crash
require '
Coy.pm';
&laughter while \$I, die;

Kevin Hackman won too, for this chill augury against the Dark Side:

> fall leaves blanket ground redmond dreams darkly—beware! winter brings penguins

Chapter 38. Searching for Rhymes with Perl

Sean M. Burke

La poésie doit être faite par tous.

Poetry is for everyone to make.

—Lautréamont (Isidore Ducasse, 1846–1870)

Wherever I go, people always come up to me and say "Sean, you gotta help me—I need to find a three-syllable word that rhymes with toad."

And my answer is always the same; I always say "Well, we're going to have to pull out the Perl for this one!"

Because, while TPJ articles constantly demonstrate that Perl is good at everything from designing sundials to peppering IRC with Eliza bots, one thing that it's *really* good at is making short little programs for

searching

text. And that's what this article is about—how to search text (specifically wordlists or pronunciation databases) for rhymes of various kinds.

Where to Look

If this article were about rhyming in Spanish, Italian, or Finnish, it'd be a whole lot shorter! Because for the most part,

the way something is spelled in these languages tells you pretty well how to pronounce it; ending with the same letters may not be exactly the same thing as rhyming, but often you can start with the spelling and apply some trivial string replacement operations to get a phonetic form that can be searched for the presence of a rhyme. This can work even with French, where (for the most part) spelling tells you pronunciation, even though the pronunciation won't tell you the spelling.

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

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

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

Ignoring the meanings of these symbols, you can see that (as the *README* will tell you), the format of each line is the word (or underscore-separated multiword phrase, like "acclamation_medal"), then a space, then the phonetic notation. What the slashes mean (and why there isn't one between the /k/ and /1/) is something I'm unsure of. But I am sure that these slashes are annoying, since they get in the way of me trying to search. I have to remember to stick them in my search patterns, and I always worry that I stuck in one too many. The same goes for the commas and apostrophes, which indicate stress—and when I'm looking for a rhyme, I may not care about stress.

Preparing the Data

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

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

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

tr/\/ _,'//d;

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

tr</ ,'><>d;

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

```
#!/usr/bin/perl
# mpron convert -- Turn the mobypron.unc
```
```
program into the mpron.dat
#
                   format that we'll use.
use strict;
@ARGV = 'mobypron.unc' unless @ARGV;
       ($word,
                   $pron, $meter,
mv
$next stress flag);
my \$Debug = 0;
# $/ = "\cm"; # May be necessary
open(OUT, ">mpron.dat");
while (<>) {
  chomp;
  ($word, $pron) = split(' ', $ , 2);
  next unless $pron;
  smeter = '';
  $next stress flag = '0';
       ______
foreach my $x ($pron =~
m < [',] | [- \&yYaeiouAEIOU \&] +> g) {
    if ($x eq ',') {
      $next stress flag = '2'; # secondary
stress
     next;
    } elsif($x eq "'") {
       $next_stress_flag = '1'; # primary
stress
     next;
    }
    $meter .= $next stress flag;
    $next stress flag = '0';
  }
  # So "stressless" one-syllable words all
get stress. Also needed
    #
       for multiword phrases mode of
monosyllabic words, like "base load".
```

```
$meter =~ tr/0/2/ if $meter =~ m/^0+$/s;
# Remove stress marks, word separators,
and the mystery slashes
$pron =~ tr<', /_><>d;
sleep(0), printf "%10s %-20s %s\n",
$meter, $word, $pron if $Debug;
print OUT join("\t", $word, $meter,
$pron), "\n";
last if $Debug and $. > 1000;
}
close(OUT);
exit;
```

Now, to search this database for

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

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

```
open(IN, '<mobypron.unc') or die $!;
open(OUT, '>mpron.dat') or die $!;
while (<IN>) {
    chomp;
    ($word, $pron) = split(' ', $_);
    $pron =~ tr</ ,'><>d;
```

```
print OUT $word, "\t", $pron,
"\n";  # Tab makes a nice delimiter
}
```

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

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

Searching the Prepared Data

With *mpron.dat* prepared, we can grep it for whatever pattern we want in the

pronunciation. Suppose we're still after a three-syllable word that

rhymes with "toad." The idea of rhyme in English is a pretty straightforward matter: if two words rhyme, they end in the same sounds (generally the last vowel and any consonants following it). If I were quite familiar with the phonetic notation for Moby Pronunciator (or whatever alternate pronunciation database you might use), I could, off the top of my head, say how to represent the sound "-oad" from "toad." However, I've never bothered, since it's so easy to just look up the word you want to rhyme with, and see how it's represented:

```
grep '^toad' mpron.dat
8
toad
                     toUd
toad's-mouth
                     toUdzmouT
toadeater
                     toUdit@r
toadfish
                     toUdfIS
toadflax
                     toUdfl&ks
toadstone
                    toUdstoUn
toadstool
                    toUdstul
toadstool disease
                    toUdstuldIziz
                     toUdi
toadv
```

oUd it is!

90	grep	'oUd\$'	mpron.dat
ał	oode		@boUd
ac	ccess_	road	&ksEsroUd
ac	cnode		&knoUd

Aeolian_mode	ioUli@nmoUd
alamode	&l@moUd
Alexis_Claude	AlEksikloUd
all-hallowed	Olh&loUd
anchor_rode	&Nk@rroUd

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

Counting Syllables

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

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

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

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

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

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

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

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

aeiuo AEIOU yY &@-

We can count syllables by seeing how often this matches:

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

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

Coping with (Syllabic) Stress

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

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

DUM-duh-DUM. In technical terms, you've got eight-syllable lines, with this

metrical pattern (where slash means stressed, and underscore means unstressed):

```
I chanced upon a lovely toad,

_ / _ / _ / _ / _ /

It gleamed and danced like ____!

_ / _ / _ / _ / _ /
```

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

```
electrode
```

has the exactly the wrong stress pattern, even though it *is* three syllables long, and

rhymes with "toad." (That's aside from "danced like electrode" being a bit ungrammatical—hey, this is *poetry!*) Since we're about to rebuild *mpron.dat* to contain each entry's syllable count, we might as well note syllable stress patterns too.

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

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

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

notation:

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

Each time this matched, we could just look in \$1 to see what kind of

stress the syllable had. However, that's not the way the data is. As it is, we have to match the stress marks wherever they are, and then set a flag so that the following syllable will be marked as

stressed (and in the absence of the flag, marked

unstressed). We can combine this with the syllable counter that works its way through the word, based on this regex:

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

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

(Usually "meter" is used for talking about the consistent stress pattern of whole lines of poetry—but I'm using it here

to refer to just the stress pattern of particular words, mostly because \$meter is easier to type than \$metrical_structure or \$stress_pattern !)

```
while (<IN>) {
    chomp;
    ($word, $pron) = split(' ', $_);
    # This is where we'll stack up a '0',
'1', or '2', one for each
    # vowel-character-group in this word,
as seen in $pron
    $meter = '';
    $next_stress_flag = '0'; # Initial
value
```

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

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

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

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

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

There are three tab-separated fields to each line. If we merely want to know the number of syllables in a word, we just count the number of characters in the second field. But if we want to know more (say, to stipulate the

stress pattern of those syllables), we have the data to do that, too.

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

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

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

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

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

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

% grep 'oUd\$' mpron.dat

and amend it with:

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

But all this

grepping is getting rather cumbersome, and won't work terribly nicely with increasingly complex search

patterns. In the end, it'd be so much simpler if we just wrote a custom (and therefore customizable!) search tool in Perl.

A Simple mpron Searcher

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

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

```
% mpron spelling_re stress_re pron_re
```

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

% mpron '' '' 'ouD\$'

We can implement this simply with a program like this:

```
($word_re, $meter_re, $pron_re) =
@ARGV[0,1,2];
open(IN, '<mpron.dat') or die "Can't
read-open mpron.dat: $!";
print "# Word RE: <$word_re> Meter RE:
<$meter_re> Pron RE: <$pron_re>\n";
# Loop over every line
while (<IN>) {
```

```
chomp;
print $_, "\n" # the matching line
if (...it meets all our criteria...)
...then do something...
```

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

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

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

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

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

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

\$bits[1] =~ m/\$meter re/o

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

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

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

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

For all the tests put together:

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

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

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

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

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

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

<pre>% mpron ''</pre>	'^[12][02][12]\$' 'ouD\$'
alamode	102	&l@moUd
antinode	102	&ntInoUd
antipode	102	&ntIpoUd
arillode	102	&r@loUd
autocode	102	At@koUd
a_la_mode	201	&l@moUd
calicoed	102	k&l@koUd
discommode	201	dIsk@moUd
episode	102	EpIsoUd
hemipode	102	hEmIpoUd
incommode	201	Ink@moUd
internode	102	Int@rnoUd
keratode	102	kEr@toUd
Kozhikode	101	koUZIkoUd
manucode	102	m&nj@koUd
megapode	102	mEg@poUd
microcode	102	maIkroUkoUd
nematode	102	nEm@toUd
Nesselrode	102	nEs@lroUd
overstowed	201	oUv@rstoUd
palinode	102	p&lInoUd
pigeon-toed	102	pIdZ@ntoUd
porticoed	102	poUrt@koUd
staminode	102	st&m@noUd
superload	102	sup@rloUd
trematode	102	trEm@toUd
waggonload	102	w&g@nloUd

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

#!/usr/bin/

```
perl
# mpron -- search for words matching a
given phonetic pattern
use strict;
my \$Debug = 0;
my($word re, $meter re, $pron re) =
@ARGV[0,1,2];
if ($meter re =~ m < [/] + $>) {
  $meter re =~ s</>><[12]>g;
  $meter re =~ s< ><[20]>g;
  meter re = '^{\prime}. meter re . '$';
}
print "# Word RE: <$word re> Meter RE:
<$meter re> Pron RE: <$pron re>\n";
die "You need at least one stipulation for
word, meter, or pronunciation!"
    unless length $word re or length
$meter re or length $pron re;
my $search file = 'mpron.dat';
open(IN, $search file) or die "Can't open
$search file: $!";
my @bits;
my $matches = 0;
my $lines = 0;
while (<IN>) {
 chomp;
  Obits = split "t", $;
  next unless @bits == 3;
  ++$lines:
  ++$matches, print $ , "\n"
    if (!length($word re) || $bits[0] =~
```

Accommodating Another Notation

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

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

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

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

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

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

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

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

Chapter 39. The Perl Poetry Contest

Jon Orwant

Kevin Meltzer

Editor's note: Jon wrote the contest announcement, and Kevin wrote the contest results.

Many Perl programmers are linguistically adept; the expressivity and flow of our language attracts people who enjoy the written word. Perl programmers also tend to have a lot of free time

from getting their jobs done so quickly, and so it's natural that they sometimes blend Perl and wordplay. One common manifestation of this whimsy is Perl poetry: a poem that also happens to be a functioning program.

Our Obfuscated Perl Contest is ugly and evil. We know this, and revel in it. To compensate for encouraging these most unpoetic programs, we hereby present our first

Perl Poetry Contest, to be judged by Perl poet Kevin

Meltzer. This is a chance to show the world just how beautiful Perl can be.

Perl Poetry has been around for a decade; on April Fool's Day 1990, someone forged a Usenet posting in Larry Wall's name with four Perl poems. Sharon

Hopkins, a longtime friend of Larry Wall and the

Official Perl Poet, presented "Camels and Needles: Computer Poetry Meets the Perl Programming Language" at the Usenix Winter 1992 Technical Conference. One of Sharon's poems, *listen*, has even been published in the *Economist* and the *Guardian*.

More recently, this arrived in my inbox:

```
#!/usr/bin/perl
#
# asylum.pl
# by Harl
close (youreyes);
bind (yourself, fast);
while ($narcosis) {
    exists $to($calm);
    not calm;
}
accept the, anesthesia;
seek the, $granted, $asylum
and wait;
stat ically;
unlink and listen (in, $complicity);
for (a, little) {
        system ("sync hronicity");
}
```

Note the rhyme at the end, and Harl's use of poignant imagery, wordplay, and the quadruple theme of sleep, insanity, connectivity, and socket programming. True, Harl's program doesn't *do* much, but it's inspirational, and we

expect to see Harl performing at a Silicon Valley poetry slam soon.

The Categories

True poets bridle at constraints, so we won't restrict entries to any particular style or genre of poem; mail us anything you like. However, please use discretion. We don't find large binary files poetic. Here are some ideas to get you started:

- Pick a famous poem and "port" it to Perl.
- Write a Perl poem that performs a useful task.
- A haiku, tanka, or limerick about Perl. Haikus (syllable pattern 5-7-5) and tankas (5-7-5-7-7) don't rhyme; limericks (9-9-6-6-9) do, in an AABBA pattern. (Also see the separate haiku contest mentioned in Chapter 37.)
- Write a program that generates poetry.

The Results

The First Annual

Perl Poetry Contest results are in, and I was quite happy to see so many entrants. We received haikus (although not always true to haiku style), love poems, ports of real poems, and poetry generators. The competition was fierce, the styles were varied, and there were even some entries containing no Perl at all.

Best Poem Port

Wayne

Myers ported this Yeats poem, "The Coming Of Wisdom with Time," to Perl. I enjoyed his use of Perl functions to coincide with Yeats' word use. For example, this Yeats line:

Though leaves are many

became

while (\$leaves > 1) {

This was exactly what I was hoping to see

from the entries. I also thought his tongue-in-cheek use of sway to ensure that the poem will die if not on a Sun was slick. Here is the original poem:

```
Though leaves are many, the root is one;
Through all the lying days of my youth
I swayed my leaves and flowers in the sun;
Now I may wither into the truth
```

And here is Myers'

Perl version:

```
while ($leaves > 1) {
    $root = 1;
}
foreach($lyingdays{'myyouth'}) {
    sway($leaves, $flowers);
}
while ($i > $truth) {
    $i--;
}
sub sway {
    my ($leaves, $flowers) = @_;
    die unless $^0 =~ /sun/i;
}
```

Haiku

I was a bit surprised to read this haiku by Clinton Pierce. All true haikus should pertain to the seasons, and while poetry reading is a subjective art, this poem would seem to have a more offensive message:

```
1, 2 or more('to'),
    tie $her, $up and bind $her, $up;
    sub for ({ each %2; do {} };)
```

I think that's supposed to be a devilish smiley at the end.

A Perl Program That Generates Poetry

Ronald J.

Kimball went all out on this one. It was the only entry I received with its own bibliography. His sonnet.pl explains itself:

```
sonnet.pl 0.1 generates
sonnets, inasmuch as the poems are 14 lines
long, in iambic pentameter, and match the
rhyming scheme ABAB CDCD
EFEF GG. Any apparent meaning in the
generated poems is purely
coincidental.
```

To create the sonnets, Ronald's entry uses a pronunciation list (http://linguist.dartmouth.edu/~rjk/mywords.dat.gz) generated by Sean Burke's Moby-based Pronunciator, described in Chapter 38. Ronald also received extra points because his entry ran with warnings enabled.

I ran this many, many times, and was amused by every sonnet it generated. The moment of endearment came when I read Magmatic appendicular tort duck—which actually rhymed with an alternate line, in true sonnet form. Here is a sample of the output

from sonnet.pl:

```
Rani proponent lapis anus raft
Triacetate magniloquently brogue
Locution cannonball outbuild corf graft
Harmoniously raspingly pure vogue
Inclusion prayer waive endarch flowage
halves
Sensillum egg pilosity impart
Retaliate exclude filets moo calves
Fibrotic piling scrapie purpleheart
Troll ignominy build join rattlebrained
Azan prosector hydrothorax mog
Defoliate accomplice holt constrained
Back prostatectomy numb door incog
Epergne appropriate charivari
Cahoots misvalue jimp born abatis
```

This entry kept me amused for more time than I care to admit, and I hope others find it as entertaining, artistic, and strangely poetic as I have.

Obfuscated Poetry?

This entry from Damon

Harper would seem to be a better match for the Obfuscated Perl Contest. I liked the extensive use of actual Perl. Damon didn't abuse whitespace, and almost makes this script look like it was a real poem first. This entry runs with warnings on and prints the poem itself when executed. Here is his entry, which should be named waiting.

```
$ =open (and , "waiting")==0?'': system
('clear');
@hoping=<a nd >; $waiting
=$i{m}{'patiently' and ''}='breathlessly';
foreach (human ('contact')) {
$just= "$waiting to "=~/^breathe?/
&&'forever' ?"waiting
_____
".("but when w =~/ do/, join(
'together',())): ($o='ur'. crypt('ic',
"words confuse"),
s/[^ta-y! \n$o: ]//go,$ );
$i{m}=print($ed? 'indelibly': ($waiting=
$just)). "disappoints";}
sub human {$e="motionally now:
done with"; @hoping; }
$ ="done with $waiting
```

```
the solution" eq'uals'? 'this': 'eyes'
.close($d =and_); $i{mmerse}="$d in";
bless\$ed; sleep;
```

The output of Damon's poem:

```
waiting
_____
open and waiting: system clear
hoping and waiting
impatiently and breathlessly
foreach human contact
just waiting to breath
forever waiting
but when we do join together: our cryptic
words confuse
stay! no: go
imprinted indelibly: waiting just
disappoints
sub human emotionally now:
done with hoping
done with waiting
the solution equals this: eyes closed and
immersed in blessed sleep
```

And then the program sleeps.

Best of Show

This short entry, by Angie

Winterbottom, was the most interesting. Her style was fresh and unique, and her use of visual representations in the text are clever. Consider the following excerpt:

```
($blaze of night{moon} == black hole)
```

"The moon, a black hole in the blaze of night." Marvelous! Angie tells us that this entry is

from Jim Steinman's song "The Invocation," on the Pandora's Box album *Original Sin*. Here is her entry:

```
if ((light eq dark) && (dark eq light)
   && ($blaze_of_night{moon} == black_hole)
        && ($ravens_wing{bright} ==
$tin{bright})){
   my $love = $you = $sin{darkness} + 1;
};
```

Here are the original lyrics:

```
If light were dark and dark were light
The moon a black hole in the blaze of night
A raven's wing as bright as tin
Then you, my love, would be darker than
sin.
```

Honorable mentions go to Robin Berfon for a sweet love poem, and to Anoop Sarkar, Steve McNabb, and Sterling Hughes.

Part VI. Politics

In this part: Chapter 40 Chapter 41 Chapter 42

This section has three articles on how Perl can preserve democracy. The first two are about voting: Rob Lanphier's article on the problems posed by our binary ballots (such as when U.S. citizens vote for president) in which voters are forced into an unnecessary dichotomy: either you vote for a candidate or against him. A better technique is to rank candidates in order of preference, as NCAA sports do. The second article, by Lincoln Stein, shows how Perl can be used to implement fair and secure Internet voting. The third and final article is about an error in a Perl program embedded in nuclear-tipped missiles, written by someone whose name is, not coincidentally, an anagram of "April Fool's Day."

Chapter 40. Pairwise Voting

Rob Lanphier

Editor's note: This article was originally written in 1996, but still applies to current elections (unfortunately).

The U.S. Presidential election once again draws near, and once again we see a contest between two men, each representing one of the two major U.S. political parties. So it goes with the

two-party system.

What is it that makes the two-party system a two-party system? It's a direct consequence of plurality

voting, the predominant form of balloting used in the United States where the highest vote getter wins an election. This relationship between the two-party

duopoly and

plurality voting is known as "Duverger's Law,"

after the 20th century political scientist who had the guts to call it a "law" (Riker, 1982).

Duverger's Law has some disturbing consequences and leaves many voters dissatisfied with the status quo. Politicians will always claim to "feel our pain," but at least in the U.S., two-party skeptics abound. Recent polls have shown that nearly 60% of Americans would support the formation of a new major party (Barrett, 1996).

The main reason Duverger's Law rings so true is that we have a binary ballot that groups people into two categories: a winner and one or more losers. The resulting dilemmas that voters are faced with in siding with a winner manifest themselves several ways:

- Because you can't please all of the people all of the time, it is in politicians' best interest to build divisions, and then build consensus among slightly over 50% of the electorate, allowing them to alienate the other 49%.
- Divisive issues encourage voters to ally with just enough people to give themselves a majority. This gives politicians a "path of least resistance" toward which they target their campaigns.
- Ultimately, since voters are powerless to state more than one preference, they are forced to take sides even when they stand in the middle.

The bottom line is that the ballot doesn't let people state what they really feel. They can only make a crude approximation of their preference, and then hope that, somehow, the politicians will "get it."

Voters are often forced into a choice between the lesser of two evils. They might not like either candidate, but rather than make a principled stand by

voting for none of the above, or a lesser known candidate with no chance of winning, they vote for the major candidate who displeases them least.

Have the strategy problems above ever demonstrably taken the electorate where it didn't want to go? Yes. While Abraham Lincoln is widely considered to be the best U.S. president in history, he owes his victory largely to the strategic error of his foes. He won the gnarled four-way 1860 election with the smallest plurality of any president—39% of the vote—and the result of that election led to the U.S. Civil War. (Of course, if slaves had had the right to vote, the numbers would have been substantially different.)

In our century, the most famous three-way strategy dilemma was when Theodore Roosevelt, angry about losing the Republican nomination, split the Republican Party vote for their 1912 Nominee (and incumbent president) Howard Taft by creating the Bull Moose Party. This allowed Woodrow Wilson to win handily with a mere 42% of the vote (as opposed to Roosevelt's 27% and Taft's 23%). This inspired many states to create "sore loser" laws that keep candidates who fail to win major party nominations from forming third parties, and by making

third-party ballot access much more difficult.

Even recently, presidential politics were affected by the three-way split. In 1966, Thomas Finan and Carlton Sickles, two relatively liberal candidates from the left-of-center state of Maryland, split the liberal vote within the state Democratic party gubernatorial nominations. As a result, conservative George P. Mahoney won the Democratic nomination, only to be beaten by Spiro Agnew, who went on to become Richard Nixon's Vice President. When Agnew resigned in 1973, it opened the door for Gerald Ford to be appointed Vice President, and then later President. A tenuous connection to the presidency, but a very real one nonetheless (Anderson, 1994).

And then there are the candidacies that might have been, if only our system hadn't discouraged third parties so much. Rosenstone, Behr, and Lazarus (1984) state that few qualified candidates would run under a third-party label because of the disadvantages they face. They note the bias that third-party candidacies face in the media by quoting James M. Perry of the *Wall Street Journal* (Rosenstone et al., 1984):

We base [our decision] on the simple proposition that readers don't want to waste their time on someone who won't have a role in the campaign. We're not going to run a page-one spread on a fringe candidate. We don't have a multiparty system. Until we do, nobody's going to cover these candidates.

With such biases built into the system, it is little wonder that third-party candidates can't gain the critical mass of support necessary to become credible contenders. A pragmatic, intelligent potential candidate might look at the seemingly insurmountable odds and simply not run. Thus, the dearth of credible third-party candidates becomes a self-fulfilling prophecy, and the two-party duopoly maintains control of the system.

The Preference Ballot

The solution is simple: convince voters to vote for candidates regardless of their perceived odds of winning. To do this, we must expand the power of the ballot. This are many ways to do this; the method that I will discuss here is the ranked ballot, or "preference ballot," as shown below:

2	Fred Flintstone
	Wilma Flintstone
3	Barney Rubble
1	Betty Rubble

The great thing about preference votes is that they're much more expressive than a vote-for-one ballot, allowing them to bargain for a compromise should their top choice be unpopular. This allows people greater flexibility in casting protest votes, while not throwing the election to the most evil candidate (not that Wilma Flintstone is evil—this is just an example).

Ranked ballots are great at limiting

voting "strategies" that encourage people to choose candidates based on poll results. But they can't eliminate strategies completely, no matter how the ballots are tallied. Political scientists have debated the relative merits of ranked ballots for years, and many of the discussions have involved Arrow's Impossibility Theorem.
Impossibility Theorems

Political scientists have been debating for some time now about whether or not it's even

possible to come up with a way to tally preference votes. Most of the debate started with Arrow's

Impossibility Theorem, which claims that any system where people are allowed to freely and exactly list their preferences must have some major defect. Arrow proves this by showing a series of conditions for fairness, not all of which can be satisfied simultaneously.

Arrow's criteria are a bit too complicated to summarize here, but other mathematicians have tweaked and fiddled with the conditions, and have come up their own sets of conditions. Fishburn and

Brams (1983) came up with a particularly concise set, listed below:

No-Show Paradox

A voter helps his favored candidate most by not voting.

Thwarted-Majorities Paradox

A candidate who defeats all other candidates in direct-comparison majority votes still loses the election. Also known as the Condorcet criterion, named after the 18th century election theorist who popularized it.

Multiple-Districts Paradox

A candidate wins in every district, but loses the general election.

More-Is-Less Paradox

If the winner had been ranked higher by some voters, another candidate would have won.

Fishburn and Brams maintain in their 1983 paper that at least one of these four paradoxes will be possible in any election method with a ranked ballot. One may make the case that since all

voting systems are vulnerable to at least one of these paradoxes, that a perfect system doesn't exist. Pragmatists counter that it's not necessary to eliminate all of them (Anderson, 1994).

In preference

voting, as in anything else, you can't please all of the people all of the time. This means we are stuck with the task of merely minimizing the sticking points rather than pursuing the holy grail of a perfect system. There are many (myself included) who believe that we can relegate the flaws to rare circumstances.

The Borda Method

This is probably the best known method within the United States for tallying ranked ballots. It is used by the Associated Press and United Press International to determine the champions in NCAA college sports. Sports writers or coaches are asked to rank the 25 best teams, and then the top team on each ballot gets 25 points, the second team gets 24, and so on. The top vote getters are ranked by points received.

This relatively simple method is easy to understand; hence its appeal. However, it discourages people from ranking anything but their top preference, thus making it difficult to derive compromise candidates from their vote. Consider a three-way election between Joe Left, Sally Middle, and Martha Right. I'll use this example to describe an election where a reasonable compromise (Sally Middle) exists between two somewhat popular extremes. Given that the seat in question must go to one person and only one, it seems reasonable that the middle candidate be chosen. Suppose the sincere wishes of these voters are as shown in Figure 40-1.

If Borda's method is used, the first place candidate on the ballot receives two points per ballot, and the second place candidate receives one point per ballot. The result is shown in Figure 40-2.

The good news here is that Borda's method does indeed choose the compromise *when everyone votes sincerely*. But it's not strategy-free: if Martha Right supporters pay attention to the polls, they can (and should) drop Sally Middle off their ballots. If all Martha Right supporters do this, they trigger a 40-point drop in Sally Middle's Borda score, causing Ms. Right to win.

Even if Martha Right supporters don't do this, it's likely that many supporters of Joe Left will do the same thing if they think that Joe has a shot at winning. Thus Borda picks a compromise when voters naively list all of their preferences, but fails when they learn how to beat the system.



Figure 40-1. Voter preferences in a three-way election



Figure 40-2. Borda points for Figure 40-1

Borda's method fails to meet the Condorcet criterion, which is arguably the most important for determining the victor in a single-winner election.

The Hare Method

Dating from 1860, the

Hare method is perhaps the best known method for tabulating preference ballots outside the United States. It's used in Australia and Ireland for single-office elections. Preference ballots are tabulated counting only the first-place candidate on each ballot. The candidate with the fewest number of first place votes is eliminated, and every ballot listing that candidate as its first choice has the vote transferred to its second choice.

The Hare method is a popular way of eliminating primaries and allowing people to vote for potentially unpopular alternatives to the two major candidates without fear of wasting one's vote. It does a pretty good job of eliminating strategy and in many ways is a substantial improvement over the American vote-for-only-one system.

Using Hare to tally the results from our election, we tally the first choices to find that Martha Right receives 40% of the vote, Joe Left receives 35%, and Sally Middle is eliminated with only 25% of the vote. The votes for Sally Middle are redistributed based on the second choice of those voters. Joe Left then wins with 51% of the vote.

Thus, Hare falls short when considering popular compromises, such as Sally Middle; like Borda's method, it also fails the Condorcet criterion.

Pairwise Election Methods

Under a class of election methods known as *pairwise methods*, the election above would result in a different winner. The relative election results of every possible combination of two candidates is tallied and the winner of key pairwise matchups is declared winner of the overall election.

In the above example, the results of the pairwise matchups would be as follows:

```
Joe Left(51%) vs Martha Right (49%)Sally Middle (60%) vs Martha Right (40%)Sally Middle (65%) vs Joe Left(35%)
```

Sally Middle beats both Joe Left and Martha Right, and therefore wins the election overall.

What distinguishes the different pairwise election methods from one another is how they deal with

circular preferences. A circular preference occurs when one candidate defeats another who in turn defeats our original winner: A beats B beats C beats A. This isn't necessarily a flaw in pairwise systems—one could say it's merely a sign that the electorate is ambivalent. Some theorists, such as Charles Dodgson (a.k.a. Lewis Carroll, author of *Alice in Wonderland*), claim that if a single winner can't be found, then the election should be called off (Levin and Nalebuff, 1995).

Nonetheless, many pairwise methods have been designed to arbitrate this situation. Three in particular deserve mention: Condorcet's method, Smith's method, and Copeland's method.

Condorcet's Method

Condorcet's method is probably the most well known. Each voter's list is used to simulate how that voter would have voted in

pairwise matchups between each of the candidates on the ballot. Separate tallies of every possible two-way election are calculated, and the winner is the candidate who wins all two-way matchups. Circular preferences are

resolved in Condorcet's method by choosing the candidate whose largest

pairwise defeat is the smallest, as measured by how many voters explicitly voted for someone else over the candidate.

The reason why many election reformers prefer this method is that, under most plausible circumstances, it solves the "lesser of two evils" problem described above, which many consider to be the litmus test for determining a good

pairwise method. However, as Anderson notes, it can produce unexpected results in certain rare circumstances.

Smith's Method

Smith's method isn't so much a pairwise tie breaker as a method of determining which candidates should *qualify* for a tie-breaker. The "Smith Set" is the smallest non-zero set of candidates who beat all the candidates outside the set in all pairwise matchups. Not all pairwise methods will pick a member of the Smith Set (most notably, Condorcet's method), yet intuitively one would hope that would be the case. Smith's method, therefore, makes a good precondition to a tie-breaker such as Condorcet's.

Copeland's Method

Copeland's method computes the winner of the election by counting the number of pairwise wins, losses, and ties for each candidate. The candidate with the best record wins the election, much in the same way that a sports team with the best record gets the top seed in that sport's playoffs.

One problem with Copeland's method is that, like Smith's method, it is prone to ties, and so is often paired with another tie breaker. It's also vulnerable when there are three parties locked in a three-way tie. In all likelihood, winning candidates will belong to the party that has the most candidates on the ballot, because they'll win the most pairwise contests, even though many of those victories might come from intraparty matchups. This would encourage parties with sufficient funds to support multiple, similar candidates in order to skew the election in their favor.

There are several other methods that exist for choosing a winner in a preference-balloted election, many of which provide a defensible set of criteria. For those of us trying to educate people on alternative election methods, our goal has been to choose the most important criteria and find the election method which best meets those criteria.

So what's all this got to do with Perl?

For many of us who aren't mathematicians by trade, it becomes difficult to debate the relative merits of the different methods without a way of visualizing some examples. The solution was to write a program that illustrates the data in a comprehensible way. Now it's time to do a little preaching to the choir. I chose to write this program in Perl for several reasons, many of which are all too familiar to Perl aficionados. However, they bear repeating in the context of programs for elections:

Perl is freely available, with source code

This is a particularly crucial feature for something designed to serve the public. Though there are relatively few voters with the knowledge or initiative to verify the source code, there is a certain peace of mind knowing that anyone can dig into the underbelly of the vote-counting machine at any time.

Perl is widely available

Perl is available on many platforms, so election results can be verified on a wide range of computers. Having the source available also ensures that it will be possible to port to new platforms as they become available.

Limitless arrays

Since array sizes don't need to be predetermined, I was able to design the system to handle as many candidates as necessary.

CGI

CGI programming has become the standard in cross-platform GUI development, and Perl is the standard for writing CGI programs. HTML tables proved ideal for displaying

voting results.

Speed of development

My initial prototype wasn't that tough to write, and had little source code. The current version is much larger, but still quite manageable.

I relied heavily on Perl 5 for my program. This is because Perl 5, unlike Perl 4, supports true two-dimensional arrays, helpful for storing pairwise election results.

The Algorithms

Before I talk about Perl specifically, I'll explain the algorithm involved in condorcet.pl.

Consider a sample field of six candidates:

```
A - John Anderson
B - Jerry Brown
C - Bill Clinton
D - Bob Dole
E - Dwight Eisenhower
F - Steve Forbes
```

condorcet.pl first creates a 6 x 6 matrix. The matrix entry at location [x, y] contains the number of votes x received over y. So [A, B] is the number of votes John Anderson received over Jerry Brown, and [B, A] is the number of votes Jerry Brown received over John Anderson.

Each ballot is tallied by determining the

pairwise results: who beats whom. So if someone ranks their ballot

A, B, C, E, D, F

then my program increments [A,B], [A,C], [A,E], [A,D], [A,F], [B,C], [B,E], [B,D], [B,F], [C,E], [C,D], [C,F], [E,D], [E,F], and [D,F], since this is how the voter would have voted in each

pairwise election. This assumes that the voter's preferences are transitive; e.g., if they prefer A over B and B over C, that they will necessarily prefer A over C. This lets us simplify the

voting process, and ensure a certain consistency among the ballots.

Next, condorcet.pl uses the matrix to determine the pairwise winners. Each complementary matchup is evaluated, and the winner receives one point in the "win" column, and the loser receives one point in the "loss" column. If the simulated pairwise election is a tie, both receive one point in the "tie" column. A possible outcome is shown in Table 40-1.

Candidate	Wins	Losses	Ties	Comments	
Е	5	0	0	E beats everyone in separate pairwise elections	
А	4	1	0	A loses to E, but beats everyone else	
В	3	2	0	B loses to A and E	
С	1	3	1	C loses to A, E, and B, and ties D	
D	1	3	1	D loses to A, E, and B, and ties C	
F	0	5	0	F loses in all elections	

Table 40-1. A possible outcome of a pairwise election

This is a clean pairwise victory for Eisenhower. If no candidate emerges unscathed by a pairwise defeat or tie, an alternative method of calculating the winner involves finding the candidate whose worst pairwise defeat was the smallest. For instance, let's modify Table 40-1 into Table 40-2.

Candidate	Wins	Losses	Ties	Comments
Е	3	3	0	E loses to C, D, and F
А	4	2	0	A loses to E and D
В	3	3	0	B loses to F, A, and E
С	3	3	0	C loses to A, E, and B
D	3	3	0	D loses to A, F, and B
F	2	4	0	F loses to A, B, C, and D

Table 40-2. Another election outcome

Table 40-2 shows how

pairwise methods can differ. Copeland's method would select A (John Anderson) as the winner, since he has the most wins. In order to calculate the winner in a Condorcet election, we need to look at the matchups where each candidate was defeated. Let's say the election has 1,000 votes. Table 40-3 shows how the losses for each candidate might tally.

Table 40-3. Another election outcome

Candidate	Loss 1	Loss 2	Loss 3	Loss 4
Е	(495, 505)	(492, 508)	(474, 526)*	

Candidate	Loss 1	Loss 2	Loss 3	Loss 4
А	(491, 509)	(482, 518)*		
В	(482, 518)	(476, 524)*	(492, 508)	
С	(474, 526)*	(488, 512)	(490, 510)	
D	(497, 503)	(491, 509)*	(493, 507)	
F	(482, 518)	(481, 519)	(477, 523)*	(498, 502)

The asterisks in Table 40-3 show the worst defeats for each candidate. In this election, D (Bob Dole) has the smallest "worst defeat" with 509 votes against him, so he'd be the winner using Condorcet's method. This is in spite of the fact that A lost fewer matchups, and in fact beat D in a

pairwise matchup. The theory behind this is that Marquis de Condorcet thought it appropriate to ask the question, "Given that there is no candidate who a majority of the electorate would pick over any other candidate, who is the candidate that a *plurality* chooses over any other candidate?" No solution to this quandary is going to be particularly satisfying, but many would argue that Condorcet's tie-breaker works about as well as any.

On the

Election Methods mailing list, an ASCII notation evolved that works pretty well as shorthand for expressing a bundle of ballots. I've extended that shorthand to make it easily implemented in Perl. We start off associating candidates with an integer by creating a two-column, comma separated list of candidate numbers and names:

```
1,Joe Left
2,Sally Middle
3,Martha Right
4,Bertha Up
5,George Down
```

Parsing that is trivial. What becomes a bit more interesting is the next portion: a list of candidate numbers separated by > when there is a preference, and = when there isn't. The ballot shown in Figure 40-3 is an example.

This ballot would be encoded as 3 > 4 = 5 > 2. That is, 3 is preferred to 4 and 5 is preferred to 2, or Martha preferred to Bertha and George preferred to Sally (and Joe Left an implied last).



Figure 40-3. A ballot allowing voters to rank their preferences

This value can optionally be prepended by a quantity of voters who voted in that way. For example, coding the example (Three-Way Election), we arrive at the following:

40: 3 > 2 9: 2 > 3 16: 2 > 1 35: 1 > 2 Now for some Perl. I'm sure I'll hear of ways to condense this down to one short line, but even the snippet below isn't too bad:

```
my (@votelist)=();
foreach $tier ( split(/>/, $ballotstring)
) {
    my (@foo) = split(/=/, $tier);
    push(@votelist, \@foo);
}
```

This uses the Perl 5 ability to create lists of lists, and creates a structure of an ordered list of tiers, with each tier consisting of equally-ranked candidates. This structure is stored in an object, along with the number of votes (if required by the voting scheme).

The Pairwise Engine

The code in Example 40-1 at the end of this chapter calculates the

pairwise tally; you can try it out at http://www.eskimo.com/ ~robla/politics/condorcet.html. The results are stored in the array \$self->{tally} because I don't have to pre-declare its size; it can handle as many candidates as necessary, without predeclaring a ridiculously large array or creating a dynamically allocated structure. This keeps the code relatively simple, although my quasi-object-oriented approach makes it harder to read a chunk out of context.

The next stage involves figuring out the winners. For this I create a makeshift voting database, with separate fields for the number of wins, losses, and ties for each candidate, as well as his worst defeat, as measured by the total number of votes against that candidate. Sadly, I wasn't feeling particularly programmer-friendly the day I was writing this, and so I made a goofy two-dimensional array rather than a hash or one-dimensional array with better names. Fortunately, the code isn't too difficult to understand.

Here is the meaning of each of the fields, where \$i is the candidate number:

```
$edata->{results}[$i][0] # Defeats for $i
$edata->{results}[$i][1] # Ties for $i
$edata->{results}[$i][2] # Victories for $i
$edata->{results}[$i][3] # $i's worst
defeat
```

This structure contains all the

pairwise tallies, giving us the data we need to calculate the Condorcet, Smith, and Copeland

winners. I'll leave it to you to fetch the code if you want to see how all of the methods are calculated, but to give you a taste, here's the code implementing the Copeland method:

```
sub rank copeland {
    my (\$self, \$edata) = 0;
      # A Copeland score is computed by
doubling the number of victories and
       #
          adding the number of ties
                                         а
candidate received.
        my (@copeland ranks) = sort
                                         {
-(($self->{results}[$a][2]*2
                                          +
$self->{results}[$a][1]) <=>
($self->{results}[$b][2]*2
                                          +
$self->{results}[$b][1]))
                                          }
$edata->candnum array;
             $self->{copeland ranks}
                                        =
\@copeland ranks;
}
```

This passes an anonymous function to Perl's sort routine, which then returns the candidates' Copeland scores.

Using CGI to Spit It All Out

The best thing about Perl is that, in combination with CGI, it's easy to generate nice-looking

output, even with copious amounts of complicated data. Given the ballots from Total Points per Candidate (Figure 40-2), I'm able to generate the HTML table shown in Table 40-4.

Table 40-4. Pairwise election results: The percentage of voters preferring the candidate on top over the candidate on the left

	Joe Left	Sally Middle	Martha Right
Joe Left		65	49
Sally Middle	35		40
Martha Right	51	60	
Won-lost	1-1	2-0	0-2

The winner of the

pairwise election depicted in Table 40-4 is Sally Middle. There's still plenty of work to be done, however:

- A more user-friendly front end would be nice.
- A "voting booth" program to generate data for use with condorcet.pl.

- Many common election methods could be implemented with this program. It would be illuminating to compare Hare and Borda methods with the Condorcet, Copeland, and Smith methods.
- It would be nice to compute not just the winner, but a list of ranked winners for each election method.
- A future version could allow one to combine methods. For example, a method very popular on the Election Methods list is Smith/Condorcet (compute the Smith winner, and break ties using Condorcet).

Random Thoughts

Flexible and robust election systems have applications beyond their traditional role in government. Local elections, decision making, and shareholder elections are all obvious applications. There's a role for election methods in computer science—one could build a genetic algorithm that generates a pool of voters who submit preference ballots to make decisions.

Sadly, many Americans have been indoctrinated into believing that the American system is the finest in the world and need not be questioned. The vote-for-only-one ballot would be okay if there were only one issue and two points of view, but society, alas, is a bit more complex.

Many pundits and armchair activists talk about how the American system of politics is broken, and then only offer up ways of restricting "the bad guys" as a solution, whether the bad guys are big business, labor unions, or special-interest coalitions. Yet few people are actually really putting the system itself under scrutiny. This is a shame; we should consider the consequences of such simplistic feedback mechanisms in how we select our leaders.

References

Amy, Douglas. *Real Choices, New Voices*. Columbia University Press, 1993.

Anderson, Lowell Bruce. *Voting Theory*. Handbooks in OR & MS, Vol. 6, editors S.M. Pollock, et al. Elsevier Science B.V., 1994.

Barrett, Laurence I. *Give Me Your Tired Parties*. AllPolitics, March 1996. (http://allpolitics.com/news/email/9603/01/ index.shtml)

Fishburn, Peter C. and Steven Brams. *Paradoxes of Preferential Voting*. Mathematics Magazine, Vol. 56, No. 4, Sept. 1983, pp. 207-214.

Levin, Jonathan and Barry

Nalebuff. *An Introduction to Vote-Counting Schemes*. The Journal of Economic Perspectives, Vol. 9, No. 1, pp. 3-26, Winter 1995.

Niemi, Richard G. and William H.

Riker. *The Choice of Voting Systems*. Scientific American, Vol. 234, No. 6, pp. 21-27.

Riker, W.H. *The Two-party System and Duverger's Law: An Essay on the History of Political Science*. The American Political Science Review, Vol. 76, pp. 753-766, 1982.

Sites

Here are three resources that may be of interest:

The Condorcet's Method Home Page

http://electorama.com/condorcet

I've put together this page from various pieces I've found on the Net. It includes programs written by other people in other computer languages.

The Election Methods Mailing List

http://electorama.com/em

This list discusses the technical details of election methods. Much of my Condorcet work was facilitated by the list members (in particular Mike Ossipoff, Steve Eppley, and Bruce Anderson).

The Center for Voting and Democracy

http://fairvote.org

CVD works toward all forms of election reform. Though their primary focus has been on advancing proportional representation, they do advocate single-winner reform as well.

Example 40-1. The pairwise_tally subroutine

sub

```
pairwise tally {
    my (\overline{\$}self, \$votelist) = 0;
    # $self is an object containing all
pairwise election data.
    # @self->{tally} is a 2D array storing
the
pairwise tally results.
      # $self->{tally}[$candx][$candy] is
the number of votes that $candx
    # received over $candv.
    # $votelist is a string containing all
of the ballots.
    for ( split(/\n/, $votelist) ) {
          # $loservec is a boolean vector
with a flag set for all losers,
          # reset with every new ballot.
All are losers until
        # they're listed on a ballot.
        my ($loservec) = $self->{candvec};
        # Parse ballot. Skip if no ballot
is returned.
                   (!(mv(\$ballot)) = new
ballot obj($self, $ ))) && next;
           # @{$ballot->{rankings}}
                                     is an
array of integers representing the
          # candidates the voter(s) voted
for, in order of preference.
         # In addition, $ballot->{quantity}
is the number of
              # identical ballots we're
considering at this time.
```

```
(@votelist)
                      my
@{$ballot->{rankings}};
        foreach $tier (@votelist) {
# For each preference listed...
                     # Remove the chosen
candidate(s) from the loser vector.
            foreach $peer (@{$tier}) {
                 vec($loservec, $peer, 1) =
0;
            }
            # For all candidates...
                     for ($i = 0; $i <=
$#{$self->{candidate}}; $i++) {
                 # If said candidate hasn't
been listed yet ...
                 if (vec($loservec, $i, 1))
{
                          # ...they've been
beaten by the chosen candidate.
                          # Increment their
"votes for the other guy" counter
                       # by the appropriate
number of ballots.
                              foreach $peer
(@{$tier}) {
                                         if
(defined($self->{tally}[$peer][$i])) {
$self->{tally}[$peer][$i]
                                  +=
$ballot->{quantity};
                         } else {
$self->{tally}[$peer][$i]
$ballot->{quantity};
                         }
```

Chapter 41. Secure Internet Voting

Lincoln D. Stein

As I write this, canvassing boards in Palm Beach and Broward Counties, Florida, are desperately trying to get manual recounts done in time for a court-imposed deadline. Countless lawsuits and countersuits launched by the Republican and Democratic adversaries are in various stages of adjudication, citizens are up in arms because they feel they have been disenfranchised by poorly-designed "butterfly ballots" and other election day mistakes, the Florida legislature has threatened to take the election into its own hands, and the U.S. Supreme Court has just agreed to add its considerable weight to the fracas.

All this because many of Florida's counties ballot using the Hollerith punchcard, an antiquated balloting technology whose main virtue is its low price tag. On the nightly news, election officials discourse gravely on "chads,"

those little bits of paper that some voters have trouble dislodging. We hear of

pregnant chads,

hanging chads, and

dimpled chads. One election board volunteer is even accused of having eaten the chads that dropped out of the ballots he was handling.

Is this any way to run an election? I don't think so. Fortunately, Perl can help rebuild

democracy, and in this article I'll show a simple Perl-based framework for a

secure Internet-based balloting system.

About Secure Elections

Much of the inspiration for this article comes from Bruce Schneier's magnum opus

Applied Cryptography and specifically from section 6.1, "Secure Elections." As explained there, the guiding principles of a secure election are to maintain privacy and prevent cheating. Schneier lays out six minimal requirements for a good election protocol:

- Only authorized voters can vote.
- No one can vote more than once.
- No one can determine for whom anyone else voted.
- No one can duplicate anyone else's vote.
- No one can change anyone else's vote without being discovered.
- Every voter can verify that his vote has been taken into account in the final tabulation.

Conventional paper ballots satisfy requirement 1 by voter registration, a process that ensures that only American citizens of a certain age can vote. Requirements 2 and 4 are satisfied by crossing the

registered voter's name off a list when the voter enters the polling location, and 3 is satisfied by using an anonymous ballot. The last two requirements, however, are not completely satisfied by paper ballots, and are the source of much of the uncertainty and accusations in the current election fiasco.

Schneier's book describes several cryptographic protocols that meet these six requirements for

secure elections. Some of them are quite elaborate and require new software at the voter's side of the connection. In this article, we will use one of the simpler ones that happens to be well suited for

web-based

voting.

This protocol requires two independent central facilities to work, called the

CEA and the

CLA. The CEA is the Central Enumeration Agency (although the Bush camp might call it "Chad Eaters Anonymous"). It is responsible for collecting and tallying ballots, and for publishing the results on election night. The CLA is the Central Legitimization Agency (or "Controlling Legal Authority" in Gore-speak). It is responsible for registering and credentialing voters.

Here's how it works:

1. Before the election, the CLA supervises voter registration. Each registered voter is issued a Voter Registration Number (VRN), which is simply a large random number.

VRNs are issued electronically, for example, by email or floppy (see Figure 41-1).

2. The CLA maintains a list of all VRNs, and a list of who VRNs were issued to, in order to prevent someone from

registering twice. There is no record of who a particular VRN was issued to.

- 3. Prior to election day, the CLA sends the CEA the list of VRNs.
- 4. On election day, the voter sends in an electronic ballot that contains his choices for elected office. The ballot contains his VRN from step 1.
- 5. The CEA checks that the VRN is valid, and crosses it off the list in order to prevent someone from voting twice.
- 6. The CEA generates a large random confirmation number (CN) for the voter, and uses it to enter the voter's choices into the vote tally.
- 7. The CEA returns the CN to the voter.
- 8. After all votes have been received, the CEA publishes the outcome, along with the lists of CNs and for whom their owners voted.

```
Dear Registered Voter,
Attached is your voter registration number. PLEASE KEEP A COPY
OF THIS E-MAIL; YOU WILL NEED IT IN ORDER TO VOTE. DO NOT BEND,
FOLD OR SPINDLE.
--REGISTRATION-START--
8334290304185987112113560
0784480602526249967749323
7190985476311629502773466
9927297350189474428770582
--REGISTRATION-END--
```

Figure 41-1. The voter registration number

The privacy of the ballot is ensured by the separation of the CLA and the CEA. One facility knows the identity of the

voters, but not who they voted for. The other has access to their vote, but not their identity.

This protocol discourages

election fraud in a number of ways. If a registered voter tries to vote twice, he will be caught in step 5. We can discourage non-registered voters from trying to guess valid VRNs by using large random numbers (in this example, we use 100-digit numbers).

If the CEA itself tries to cheat by stuffing the ballot box or "losing" ballots, this can be detected in step 8. By publishing a list of CNs and their votes, the protocol allows voters to check the list to make sure that their votes were tallied correctly.

There are still ways to defraud this protocol. For example, the CEA and CLA can collude to figure out the identity of a voter; if

voting is being done over the Web, the CEA can use the voter's IP address to figure out who he is. The first of these problems can be addressed by

election auditors and statutory law. The second can be addressed using proxy servers or by having the balloting take place in central polling places equipped with ATM-like web browsers.

It is also important to note that the protocol described here is a slight departure from the one described by Schneier. The original protocol is built along an email model, in which the voter himself generates the CN, rather than letting the CEA do it for him.

The Database Schema

Tallying votes is a task for a database, and this application uses MySQL to do the heavy lifting. Example 41-1 shows the schema used by the CEA's database. In addition to managing the vote tally itself, the information in the database is used to generate the ballot on the fly. This avoids having to design a new web page for each ballot, and discourages election officials from coming up with butterfly ballots and other innovative designs.

Example 41-1. The schema used by the CEA's database

```
0 #!/bin/sh
 1
    /usr/local/bin/mysgl -f CEA <<END
 2
    DROP TABLE party;
 3
  DROP TABLE office;
    DROP TABLE candidate;
 4
 5
    DROP TABLE registration;
    DROP TABLE tally;
 6
 7
    DROP TABLE writein;
    CREATE TABLE party (
 8
 9
                 party id
                                     INT(4)
UNSIGNED NOT NULL AUTO INCREMENT,
10
            party name
                         CHAR(100),
            PRIMARY KEY(party id),
11
12
            UNIQUE (party name)
13 );
14 CREATE TABLE office (
15
                 office id
                                     INT(4)
UNSIGNED NOT NULL AUTO INCREMENT,
```

16 office name CHAR(100), 17 PRIMARY KEY(office id), 18 UNIQUE (office name) 19); 20 CREATE TABLE registration (registration id CHAR(100) 21 NOT NULL, 22 registration used TINYINT DEFAULT 0 NOT NULL, 23 PRIMARY KEY (registration id) 24); 25 CREATE TABLE candidate (INT(4) 26 candidate id UNSIGNED NOT NULL AUTO INCREMENT, first name CHAR(50) NOT 27 NULL, 28 last name CHAR(50) NOT NULL, 29 party id INT(4) UNSIGNED NOT NULL, office id 30 INT(4) UNSIGNED NOT NULL, 31 UNIOUE (party id, office id), 32 PRIMARY KEY (candidate id) 33); 34 CREATE TABLE tally (voter id CHAR(100) NOT 35 NULL, 36 office id INT(4) UNSIGNED NOT NULL, 37 candidate id INT(4) UNSIGNED NOT NULL, time voted TIMESTAMP(10), 38 39 UNIQUE(voter id, office id),

```
40
            KEY(voter id)
41 );
42
    CREATE TABLE writein (
43
             voter id
                            CHAR(100) NOT
NULL,
44
office id
            INT(4) UNSIGNED NOT NULL,
45
writein
                CHAR(100) NOT NULL,
46
            time voted
                           TIMESTAMP(10),
47
            KEY(voter id),
48
            KEY(office id)
49);
50
    END
```

The

schema has six tables:

party

This table lists political parties. Each party has a unique ID, numbered from 1, and a short description, such as "Republican Party."

office

This table lists the offices that are up for grabs. Each office has a unique ID and a short description like "Town Dogcatcher."
registration

This table lists valid VRNs. Each VRN has a small integer associated with it that indicates whether the VRN has been used in the current balloting.

candidate

This table lists information about the candidates in the current election. Each candidate has a unique ID as well as fields that describe who he is and what he's running for. The party_id and office_id fields describe the candidate's party affiliation and the office he is a candidate for. The first_name and last_name fields are self-evident. A UNIQUE constraint ensures that there cannot be two candidates from the same party running for the same office. The same candidate can, however, run for two different offices, which should make Senator and/or Vice President Lieberman happy.

tally

This table keeps track of the vote tally. The voter_id field corresponds to the voter's CN ballot confirmation number (not the VRN). The candidate_id contains the voter's choice for candidate, and office_id contains the office the voter wants to place him in. A UNIQUE constraint ensures that a voter can only vote once for a given office.

writein

This table keeps tabs on write-in candidates.

Example 41-2 gives a set of MySQL statements that insert some test values into the CEA

database. There are three offices on this test ballot, including President of the United States, State Senator, and Town Dogcatcher. (No soft money was used for the selection of the various celebrities listed in this example, and I was not promised a night in the Lincoln bedroom.)

Example 41-2. MySQL statements to insert values into the CEA database

```
0 #!/bin/sh
1 # first the parties
2 /usr/local/bin/mysgl -f CEA <<END</pre>
  3 INSERT INTO party VALUES
(NULL, 'Republican Party') \p\g
         INSERT INTO party VALUES
  4
(NULL, 'Democratic Party') \p\g
 5 INSERT INTO party VALUES (NULL, 'Green
Party') \p\g
 6 INSERT INTO party VALUES (NULL, 'Reform
Party')p
  7
         INSERT INTO party VALUES
(NULL, "Socialist Worker's Party") \p\q
8 # now the offices
  9 INSERT INTO office VALUES
(NULL, 'President of the United States') \p\q
10 INSERT INTO office VALUES (NULL, 'State
Senator')p\q
11 INSERT INTO office VALUES (NULL, 'Town
Dogcatcher')\p\g
12
   # Now the candidates
13 # Presidential candidates
14
       INSERT INTO candidate VALUES
(NULL, 'e.e.', 'cummings', 1, 1) \p\q
       INSERT INTO candidate VALUES
1.5
```

```
(NULL, 'Oqden', 'Nash', 2, 1) p q
16
       INSERT
               INTO candidate VALUES
(NULL, 'Marilyn', 'Monroe', 3, 1) \p\g
       INSERT
               INTO candidate VALUES
17
(NULL, 'Tennessee', 'Williams', 4, 1) \p\q
       INSERT INTO candidate VALUES
18
(NULL, 'Chuck', 'Jones', 5, 1) \p\q
19 # State senator
2.0
       INSERT
                      candidate VALUES
               τητο
(NULL, 'Timothy', "O'Leary", 1, 2) \p\g
       INSERT INTO candidate
21
                                   VALUES
(NULL, 'Abbie', 'Hoffman', 2, 2) \p\q
       INSERT INTO candidate VALUES
2.2
(NULL, 'Ivana', 'Trump', 3, 2) \p\q
23 # Dogcatcher
24
       INSERT INTO candidate VALUES
(NULL, 'John', 'Adams', 1, 3) \p\q
       INSERT INTO candidate VALUES
2.5
(NULL, 'Morticia', 'Addams', 2, 3) \p\g
```

26 END

Generating VRNs

We won't develop the CLA very fully in this article. The CLA should maintain a

database of voter registration information, such as birth dates, addresses, and driver's licenses. For testing purposes, we'll generate blocks of VRNs

using the program shown in Example 41-3, and load the VRNs into the database using the program, shown in Example 41-4.

Example 41-3. Generating VRNs with generate_vrns.pl

```
#!/usr/bin/perl
 0
 1
    # file: generate vrns.pl
  use constant DIGITS => 100;
 2
 3
    my $id count = shift;
    for (1..$id count) {
 4
        my $digits = random digits(DIGITS);
 5
        $digits =~ s/(.{25})/$1\n/g;
 6
 7
        print <<EOB;</pre>
 8
    --REGISTRATION-START--
 9
    $digits--REGISTRATION-END--
10
    EOB
11
   }
12
    sub random digits {
        my $digits desired = shift;
13
        open (RAND,
14
/dev/urandom') or die "Can't open random
number device: $!";
```

```
15 my $data;
16 read(RAND,$data,$digits_desired)
or die "Can't read random bytes: $!";
17 my @digits = map {$_ % 10}
unpack('C*',$data);
18 return join '',
@digits[0..$digits_desired-1];
19 }
```

Example 41-4. Entering VRNs with enter_vrn.pl

```
0
  #!/usr/bin/perl -w
 1 # -*- perl -*-
 2 # enter vrn.pl
 3 use strict;
 4
  use DBI;
 5 my $db = DBI->connect('dbi:mysql:CEA')
    or die "Can't connect: $DBI::errstr";
 6
  my $stl = $db->prepare(
               'INSERT
INTO registration (registration id) VALUES
(?)'
7 or die "Can't prepare: ",$db->errstr;
8
   $/ = ""; # paragraph mode
 9 while (<>) {
10
     chomp;
11
                  my ($digits) =
/--REGISTRATION-START--(.+)--REGISTRATION-END--/
s;
12
     digits = ~ s/D//g;
13
          $stl->execute($digits) or die
$db->errstr;
14 }
15 $stl->finish;
16 $db->disconnect;
```

The generate_vrns.pl script generates the number of VRNs requested on the command line. The part that does all the work is the subroutine random_digits between lines 12 and 19. It is critical to use a good random number generator to generate VRNs; otherwise, valid VRNs would be too easy to guess. The

Math::TrulyRandom module (available on CPAN) promises to do this, but it hangs on my Linux system. Instead, I use the /dev/urandom device, which uses a Linux kernel driver that generates random data from non-deterministic system information, such as interrupts. We read the requisite number of random bytes from the device, and then transform them into a set of base 10 digits.

The

enter_vrn.pl script shown in Example 41-4 takes a list of VRNs generated by the previous script, and enters them into the CEA database using the DBI module and its DBD::mysql driver.

Line 5 connects to the CEA database or dies with an error message. Line 6 prepares a SQL INSERT statement that will add

VRNs to the registration table. The code between lines 8–14 loops through the list of VRNs one paragraph at a time, extracts the VRN information, and calls the SQL statement's execute method to perform the insertion. After the last VRN is processed, we call finish to close the SQL statement, and disconnect to disconnect from the database.

The E-Ballot

The fun part is the

electronic ballot generated by the CEA. Figure 41-2 shows how it looks to the voter. There are three steps to voting. In the first step, the voter makes his choices by selecting radio buttons in the ballot. Each candidate is sorted by his party and the office he is running for. There is also a text field that allows for write-ins.

	ನನನ Inde	cision 2000 కాహన	
Step 1: Fil	President of the United States	State Senator	Town Degcatcher
Republican Party	Qe.e. cummings	oTimothy O'Leary	QJohn Adams
Democratic Party	oOgden Nash	ÇAbble Hoffman	OMorficia Addams
Green Party	QMarilyn Monroe	Qivana Trump	
Reform Party	oTennessee Williams		
Socialist Worker's Party	OChuck Jones		
	♥Write in:	@Write in:	owrite in:
	T	t	1
Step 2: En	ter your Registered Vo	ter Code	<u>l</u> *
Step 2: En IHER: Cut and p I: Select voter n	ter your Registered Vo paste the code here: pgistration file for upload here: Browse	ter Code	

Figure 41-2. The electronic ballot

In step 2, the voter enters his VRN. He can do so by cutting and pasting the VRN into a large text field, or by uploading a file provided by the CLA that contains the VRN.

When satisifed, the voter presses the VOTE button. The CEA checks that the ballot is filled out, that the voter hasn't voted twice for candidates for the same office, and that the VRN is valid and has only been used once. If these checks are satisfied, the CEA enters the voter's choices into the database, generates a CN, and displays the confirmation to the user (Figure 41-3).



Figure 41-3. Vote confirmation

Later, when the

votes have been tallied and published, the user can go to the published results and make sure that his vote was correctly counted.

Of course, for the e-ballot to be at all

secure, all communication between the voter and the CEA's web site must use SSL, and the voter should be instructed to check the web site's SSL certificate to make sure that it is valid.

The vote Program

The e-ballot is

implemented by a conventional CGI script shown in Example 41-5. The listing is a bit long, but most of it is fancy formatting in the ballot section. We'll step through it a line at a time.

Example 41-5. The vote program

```
0 #!/usr/bin/perl -Tw
1 # -*- perl -*-
 2 use strict;
 3 use CGI qw(:standard *table *Tr *dl);
4 use DBI;
 5 use constant REGISTRATION CODE LENGTH
=> 100;
 6 use vars qw(@CANDIDATES @PARTIES
@OFFICES @CANDIDATE NAME);
7 SENV{PATH} = '/bin';
  # connect to database
8
 9 my $DB = DBI->connect('dbi:mysql:CEA')
or die "Can't connect: $DBI::errstr";
11 # create the page
13 print header.
      start html(-title => 'Indecision
 14
2000',
15
               -bgcolor => 'white'
16
               ),
17
    h1({-align=>'CENTER'},
```

```
18
         img({-src=>'/icons/star.gif'}) x
3,
        'Indecision 2000',
19
20
        img({-src=>'/icons/star.gif'}) x
3);
21
  # load global variables
22 get globals();
 23
     # If the VOTE button is pressed,
validate and enter the ballot
  24 if (param('VOTE') &&
                               (mv
$registration = validate())) {
       enter ballot($registration);
2.5
26 }
27
   # Otherwise print the ballot
28 else {
29
       generate ballot();
30
   }
31 # end of page
32 print end html;
33 $DB->disconnect;
34 exit 0;
           35
*****
  36 # get globals() loads the
@CANDIDATES, @CANDIDATE NAME,
 37 # @PARTIES and @OFFICES globals from
information in the
38 # database.
           39
****
40 sub get globals {
```

```
# fetch the matrix of candidates,
 41
parties and offices
 42
         my $query =<<END;
         43
                              SELECT
candidate id, first name, last name, party name,
    44
office name, candidate.party id, candidate.office id
      FROM candidate, party, office
 45
      46
                                   WHERE
candidate.office id=office.office id
       47
                                    AND
candidate.party id=party.party id
 48
    END
 49
    ;
 50
          my $sth = $DB->prepare($query) or
die "Can't prepare: ",$DB->errstr;
 51
         $sth->execute;
     52
                                    while
(my($candidate id,$first,$last,$party,$office,$party .
 53
                 = $sth->fetchrow array) {
    54
$CANDIDATES[$party id-1][$office id-1]
                                           =
$candidate id;
    55
$CANDIDATE NAME[$candidate id] = "$first
$last";
 56
                    $PARTIES[$party id-1]
                                          =
$party;
                   $OFFICES[$office id-1] =
 57
$office;
 58
 59
         $sth->finish;
 60
    }
```

```
61
```

```
*******
 62 # generate ballot(),
voting matrix(), validation number() and
vote()
 63 # create various parts of the page
see by the voter
            64
******
 65 sub generate ballot {
        print start multipart form;
 66
 67
voting matrix();
       registration number();
 68
 69
       vote();
70
       print end form;
71 }
 72 # This generates the table containing
the ballot.
73 sub voting_matrix {
 74
               print img({-src=>'/icons/
HandPointing.gif',-align=>'LEFT'}),
 75
              h2('Step 1: Fill in your
E-Ballot');
76
       print
    77
start table({-cellspacing=>0,-border=>1}),
          Tr(th(''), th(\@OFFICES));
78
 79
            for (my $party=0; $party <</pre>
@PARTIES; $party++) {
  80
                                print
start Tr,th($PARTIES[$party]);
 81
             for (my $office=0; $office <</pre>
```

```
@OFFICES; $office++) {
 82
                         my $candidate =
$CANDIDATES[$party][$office];
  83
                                    print
td({-bgcolor=>$office %2 ? 'white' :
'#EOEOEO'},
84
                              $candidate ?
radio group(-name => $office,
 85
                               -value =>
$candidate,
                               -labels =>
 86
{$candidate=>$CANDIDATE NAME[$candidate]},
87
                          -default => '-',
88
)
 89
                                         :
' '
 90
                        );
 91
             }
 92
            print end Tr;
 93
         }
 94
       # Handle write-ins.
       print Tr(th(' '),
 95
 96
                                  td([map
{radio group(-name => $, -value =>
'Write in:').
 97
textfield(-name => "writein $ ",
 98
-value => '',
 99
-override => defined param($)
                                      & &
param(\$) = ~/^ d+\$/
                       ) } (0..$#OFFICES)]
100
```

```
101
                    )
102
                 ),
103
           end table;
104
    }
105 # generate the field for entering
voter registration number
106 sub registration number {
107
         print hr,
108
                     img({-src => '/icons/
HandPointing.gif', -align => 'LEFT'}),
109
                 h2('Step 2: Enter your
Registered Voter Code'),
110
           blockquote(
111
                         b('EITHER:'), 'Cut
and paste the code here:',
112
                          textarea(-name =>
'registration id', -rows =>4 , -cols => 70,
                                   -wrap =>
'physical'), br,
113
                          b('OR:'), 'Select
voter registration file for upload here:',
br,
114
                         filefield(-name =>
'registration file')
115
                    );
116
   }
   # generate the VOTE button
117
118 sub vote {
119
         print hr,
120
                     img({-src => '/icons/
HandPointing.gif', -align => 'LEFT'}),
         h2('Step 3:', 'Cast your Ballot'),
121
122
        blockquote(b(submit('VOTE')));
123
```

```
124
*****
125 # validate() validates the ballot to
discourage fraud
126
*****
127 sub validate {
128
          # first check that the voter
registration field is filled out
129
              return error('The voter
registration ID field is missing.')
          unless param('registration id')
130
|| param('registration file');
1.31
        # check that the voter has voted
for at least one office
        return error('The ballot has not
1.32
been filled out.')
133
             unless grep {param($ ) ne
'Write in:' || param("writein $ ")}
0..@OFFICES-1;
         # check that no office has more
134
than one vote
       for (0..@OFFICES-1) {
135
136
        my @votes = param($);
137
        return error ("You have voted for
$OFFICES[$ ] more than once.") if @votes >
1;
1.38
        }
139
        # recover the registration ID
140
       my $registration id;
                              $fh
141
                    if (my
                                   =
param('registration file')) {
142
           while (<$fh>) {
```

143 chomp; 144 next unless /--REGISTRATION-START--/../ --REGISTRATION-END--/; 145 next unless /^\d+\$/; 146 \$registration id .= \$; 147 } 148 } 149 \$registration id | | = param('registration id'); \$registration id =~ s/D/q; # 150 get rid of all non-digits return error('Your registration 151 code is the incorrect length.') unless length \$registration id 152 == REGISTRATION CODE LENGTH; 153 # check that this is a registered voter 154 my \$sth = \$DB->prepare('SELECT registration used FROM registration WHERE registration id=?') 155 or die "prepare registration: ",\$DB->errstr; 156 \$rows my = \$sth->execute(\$registration id); 157 return error("The registration code provided is not on the list of eligible voters.") 158 unless \$rows > 0; 159 # check that registration ID has not already been used my (\$used) = \$sth->fetchrow array; 160 161 return error("That voter registration code has already been used.")

```
162
         unless \$used == 0;
163
        $sth->finish;
164
        return $registration id;
165 }
166
*****
167 # enter ballot() updates the database
168
****
169 sub enter ballot {
170
       my $registration = shift;
         # lock this registration number
171
so that it can't be used again
172
         $DB->do("UPDATE registration SET
registration used=1
173
                                WHERE
registration id='$registration'
174
                                  AND
registration used=0")>0
175
                or die "Can't update
registration: ",$DB->errstr;
176
        # generate a ballot ID
      my $id = random digits(100);
177
178
        # prepare the SQL for regular and
write-in votes
179
                 my
                     $regular vote
$DB->prepare("INSERT
                     INTO
                           tally
VALUES('$id',?,?,NULL)")
180
               or die "Can't prepare:
",$DB->errstr;
181
                     $writein vote
                 my
$DB->prepare("INSERT
                     into writein
```

```
VALUES('$id',?,?,NULL)")
182
                 or die "Can't prepare:
",$DB->errstr;
         # begin user confirmation
183
184
           print h2('Save this Information
for your Records');
         print start dl;
185
186
         for my $office (0..$#OFFICES) {
187
                   my $selection
                                           _
param($office);
188
                  my $writein
                                           =
param("writein $office");
              my $candidate name = $writein
189
|| $CANDIDATE NAME[$selection] || '-none-';
190
               # update database with the
candidate's vote
191
            if ($writein) {
192
$writein vote->execute($office+1,$writein)
                      or die "can't update
tally: ", $DB->errstr;
            } elsif ($selection) {
193
194
$regular vote->execute($office+1,$selection)
                      or die "can't update
tally: ", $DB->errstr;
195
            }
196
            # update confirmation page
197
                                    print
dt(b($OFFICES[$office])),dd($candidate_name);
198
199
        print end dl;
         $writein vote->finish;
200
201
         $regular vote->finish;
```

```
202
           # show user his confirmation
number
203
        id = - s/(.{50})/$1\n/;
204
           print h3('Ballot Confirmation
Number'),pre($id);
205 }
206
**********
207 # utilities
208
**********
209
   # generate some random digits for the
ΙD
210 sub random digits {
        my $digits desired = shift;
211
212
        open(RAND, '/dev/urandom') or die
"Can't open random number device: $!";
213
        my $data;
214
         read(RAND,$data,$digits desired)
or die "Can't read random bytes: $!";
           my @digits = map {$ % 10}
215
unpack('C*',$data);
216
                   return join '',
@digits[0..$digits desired-1];
217 }
218 # all-purpose error message
219 sub error {
220
                              print
p(font({-size=>'+2',-color=>'red'}, @, br,
221
                     'Please correct and
try again.'));
222
       return;
223
    }
```

Lines 0–4: Load Modules

We turn on taint checking and Perl warnings. Taint checking ensures that we will be prevented from doing anything stupid with user-supplied input, such as passing it to shells, and warnings alert us of uninitialized variables and the like. We load the CGI and DBI modules. One trick to notice is that the symbols

loaded from the CGI module include *table, *Tr, and *dl. The asterisk means to automagically generate functions to start and end the corresponding HTML tags, such as start_table to generate a <TABLE> tag and end_table to generate an </TABLE> tag.

Lines 5–7: Define Constants and Globals

We define a constant for the

length of a valid VRN, and declare globals that will hold various information about the ballot. @CANDIDATES is a two-dimensional list of candidates, in which the first dimension is the candidate's party affiliate and the second dimension is the office the candidate is running for. The values of this array are candidate IDs. Each of @PARTIES, @OFFICES, and @CANDIDATE_NAME map from database IDs into human-readable labels. To adjust for the fact that 1 is the lowest ID used in the CEA schema, we adjust each of the indexes by 1. For example, the party id for the "Republican Party" is 1, so it can be found in <code>@PARTIES</code> at <code>\$PARTIES[0]</code>.

We set the PATH environment variable to a safe known value in order to satisfy Perl's taint-check requirements.

Lines 8–9: Connect to the Database

We call the DBI->connect method to connect to the CEA database running on the local machine. We don't use any password authorization here, but in a real application we would want to.

Lines 10–20: Start the Page

We emit the standard HTTP header by calling the CGI module's header function, and start the top of the HTML page by calling start_html and h1 to generate HTML boilerplate and a level-one header.

Lines 21–22: Initialize Globals

We call get_globals to initialize the four global variables that describe the current ballot. get_globals will make the appropriate database calls to do this.

Lines 23–26: Handle a Submitted Ballot

We call the CGI module's param method to look for a CGI field named VOTE. If such a field exists, then it indicates that the user has submitted his ballot by pressing the VOTE button. We immediately call a subroutine named validate, which checks that the ballot is filled out correctly. If the ballot checks out, it returns the user's VRN, and we pass the VRN to a subroutine named enter_ballot that adds the information to the growing tally.

Lines 27–30: Generate a New Ballot

Otherwise, we call generate ballot to create the ballot that the user sees on the page. This subroutine will be called

the first time the user loads the page, as well as when the validate subroutine detects an error in a previously-submitted ballot.

Lines 31–34: Finish up

We call end_html to generate the boilerplate at the bottom of the HTML page, and disconnect from the database. We then exit.

Lines 40–60: get_globals Subroutine

This subroutine is responsible for

loading the global variables with information about the current election. For efficiency's sake, it fetches all the information it needs in a single large SQL statement that retrieves each of the candidates, their party and office IDs, and the human-readable labels for candidates, offices, and parties.

We do this by passing the appropriate SQL statement to the database's prepare method, and then executing the resulting statement handle. We then loop over each row of the returned table,

populating the @CANDIDATES, @CANDIDATE_NAME, @PARTIES, and @OFFICES arrays as we go. Notice how we offset each ID by one in order to use it as an array index.

Lines 61–71: The generate_ballot Subroutine

This subroutine is responsible for

generating the HTML for the ballot. It calls the CGI module's <code>start_multipart_form</code> function in order to start a fill-out form. We use this function rather than the more common <code>start_form</code> because we will be accepting a file containing the voter's VRN for upload, and only the multipart-style form can accept file uploads.

We then call three functions to generate the top, middle, and bottom of the form, and call end_form to generate the form's closing tag.

Lines 72–104: The voting_matrix Subroutine

This large subroutine generates the table that shows the ballot information. Don't be intimidated. The part of the subroutine that does all the work is just two nested loops. The outer one loops through parties, which become the rows of the ballot, and the inner one loops through offices, which become the columns. Within the inner loop, we check whether @CANDIDATES contains a candidate for the current party affiliation and office. If it does, we generate a radio button whose name is the index into @OFFICES and whose value is the candidate ID. For the label we use the human-readable version of the candidate's name, derived from @CANDIDATE_NAME.

After

creating the radio buttons for standard candidates, we create a series of write-ins, one for each office. These are text fields with the name "writein X," where X is the ID of the corresponding office.

Lines 105–116: The registration_number Subroutine

This subroutine generates the section of the ballot that prompts the user for his VRN. There are two fields that can he used One is large text named а area registration id, where the user can cut and paste his VRN The other is а file field named registration file, which the user can use to upload a text file containing the VRN.

Lines 117–123: The vote Subroutine

This subroutine generates a single HTML submission button labeled "VOTE". This concludes the portion of the script that generates the e-ballot.

Lines 124–165: The validate Subroutine

This is responsible for

validating the voter's submission. There are several checks on the integrity of the ballot. First we check for the easy things: whether the VRN has been filled in, and whether any of the radio buttons in the ballot have been selected (we only require a minimum of one office to be selected; it's perfectly valid for the user to vote for some offices and enter no selection for others).

We now check for more subtle problems. Lines 134–138 verify that each office has exactly 0 or 1 votes. Although the fill-out form only allows a single candidate from each office to be selected, a malicious voter could roll his own fill-out form and try to vote for multiple candidates from the same office.

Having passed these checks, we recover the VRN. If the registration_file field is present, then the user has chosen to upload a file. We call the CGI module's param function to recover a filehandle for the uploaded file, and parse out its contents. Otherwise, if the registration_id field is present, we use its contents to recover a cut-and-pasted VRN.

Having recovered the VRN, we ensure that it is valid. First, we check that the VRN is the correct length. If so, we consult the database to see whether the VRN is in the registration table, and whether it is still unused. If both these tests pass, then we declare that the submitted

ballot is valid and return the VRN to the caller.

When any of the tests fails, it calls a utility subroutine named error. The error function displays a bold red error message on the top of the page, and prompts the user to make corrections and try again.

Lines 166–205: The enter_ballot Subroutine

The enter_ballot subroutine is where the information from the ballot is collected and entered into the database, registering the voter's intent and keeping the sacred flame of Democracy alight.

The first thing that we do is update the database in order to mark the VRN as used. This prevents the VRN from being used again. We do the update in a way that will cause it to fail unless the VRN is currently marked as unused, and avoids an attack based on race conditions while updating the database.

We now generate a confirmation number for the ballot by calling random_digits. We use the newly-generated ID to generate two SQL insert statements, one for regular candidates, and the other for write-ins. Each statement uses "?" as placeholders for the office and candidate fields.

We now enter the voter's choices into the database, simultaneously

generating a confirmation page as we do so. We loop over the @OFFICES array, looking for CGI parameters corresponding either to a regular candidate for the office or to a write-in. If we find a write-in, we recover it and insert it into the

writein table using the appropriate insert statement. Otherwise we insert the voter's choice into the

tally table. Notice how we add 1 to the office index in order to convert it back into the 1-based ID used in the MySQL tables.

Each time through the loop, we print out the office and the selected candidate, using a definition list (<DL>) style HTML list.

At the end of the subroutine, we finish both SQL statements, and then print out a nicely-formatted version of the voter's ballot confirmation number.

Lines 209–223: Utility Subroutines

We've already seen the random_digits subroutine. The error subroutine takes its arguments and incorporates them into an HTML paragraph, using a red font and a large font size. The subroutine explicitly returns undef, which allows this type of idiom in the caller:

```
return error('Please stop munching
chads and start punching ballots')
unless $is valid;
```

Tallying the Votes

On election night, tallying the vote is simply a matter of issuing a SQL statement to add up each candidate's counts and grouping the results by office. Here's one that will do the trick:

```
SELECT office name,last name,count(*)
  FROM office, tally, candidate
                     WHERE
candidate.candidate id=tally.candidate id
                          AND
candidate.office id=office.office id
  GROUP BY tally.candidate id;
| office name
last name | count(*) |
| President of the United States |
cummings |
             2 |
| President of the United States |
Nash | 4 |
| President of the United States |
Jones |
            1 |
| State Senator
                              O'Leary | 5 |
| State Senator
         1 |
Hoffman |
| Town Dogcatcher
Adams |
             3 |
| Town Dogcatcher
Addams | 4 |
+----
```

By this count Ogden Nash deserves to be the next President, Timothy O'Leary next State Senator, and Morticia Addams the next Dogcatcher. A definite improvement over this year's choices!

Author's Note: The algorithm presented in this article contains an important bug which was pointed out by several readers after its original publication in The Perl Journal. As it stands, a corrupt CEA can cheat by reusing certificate numbers. To see how this works, say that there are two candidates in the election, who we shall call, say, Bushnell and Lore. The CEA favors Bushnell. After it receives the first vote for Lore, it records the CN and then reissues the same CN a number of times. Each time a voter uses one of these duplicate CNs to vote for Lore, the CN drops it, and generates a bogus vote for Bushnell using a forged CN. At the end of the election, any Lore voter who checks his CN will find that his vote was correctly recorded for Lore. Voting monitors will find that the correct number of votes were recorded. But the statistics are slanted towards Bushnell. because the duplicate Lore votes were dropped.

The solution to this problem is to have the voter pick all or part of the CN, a detail that was in the original Schneier description of the algorithm, but which I dropped because it complicated the implementation (a lesson learned: in security algorithms, the details matter). By putting the CN under the voter's control, the CEA is prevented from issuing duplicate CNs. For example, you could ask the voter to pick a number between 1,000,000 and 2,000,000. Better still you could implement a client-side service to pick a large random number at the time the fill-out form is generated. One simple *implementation would use JavaScript for this purpose, but for political reasons I will leave this as an exercise to the reader.*

Chapter 42. Perl and Nuclear Weapons Don't Mix

Ray F. Piodasoll

After graduating college and finishing a two-year stint in the military, I joined a

software company that developed utilities for clients in the defense industry. I was ROTC in college, but I still had an twinge of guilt about working occasional for the military-industrial complex. Only when I developed a regression test suite for an early-warning comet detection system did I realize that missiles aren't always a bad thing. When a comet six miles wide is poised to strike Earth, as one did 65 million years ago, destroying the dinosaurs and most everything else chewable, you'll be glad we have nukes to obliterate it before it turns Earth into space paste. After finishing that project, I rejoined the military, and within eighteen months I was assigned to

NORAD (the North American Aerospace Defense Command).

Perl is used quite a bit at NORAD, which gave up ADA long ago when it became evident that ADA programmers weren't the sort of people you want defending the homeland. At first, my CO balked at Perl, but he softened to the idea when I told him that it had been designed for exactly this purpose and that PERL stands for "Precision Entry and Reentry Launchings,"

a lie that would later be repeated several times at my court martial.

I'm not supposed to talk about the work I did at NORAD, which was software development for missile guidance systems. But a few musings on coding style should be okay, since I think they'll tell you more about Perl than about national security. I don't mean to alarm anyone by what follows, but you should know just how close we came to nuclear armageddon because of my misunderstanding of basic Perl concepts. Sorry!

In March 1996, I was told to write a program to calculate the effect on thrust from a proposed modification to the alloy composition of Nike missiles. The first step was parsing a file full of rivet locations that looked like this:

```
RIVET294 3/8 004 14.25 14.375
DORS18-LEFT TH15/16
(TOLERANCE .0002)
RIVET295 3/8 004 14.625 14.75
DORS18-LEFT TH15/16
(TOLERANCE .0002)
```

Boring stuff. The first line tells us that rivet 294 is three-eighths of an inch long and can be found 14.25 inches above the lower left corner of plate 004. The second line has more information identifying the location of the rivet, and the third line contains the tolerance (.0002 inch) allowed in the placement and rivet size. We machine all our own rivets—even if Ace Hardware could supply rivets meeting our specifications, their insurance company will rest easier knowing that their products won't be responsible for starting World War III. Anyway, extracting these fields is easy—simply split on spaces:

```
while (<>) {
    @fields = split(/\s+/, $_) if /^RIVET/;
```

Fortunately, Perl's support for regular expressions makes the tolerance easy to parse too:

```
$tolerance = /TOLERANCE\s+(\.\d+)/ if
/TOLERANCE/;
# Rest of code will be declassified in
25 years
}
```

But there's a problem with the \$tolerance line above: when a match is evaluated in a scalar context, it yields 1 for success and undef for failure. My program set every \$tolerance to 1 because I forgot the parentheses:

```
($tolerance) = /TOLERANCE\s+(\.\d+)/ if
/TOLERANCE/;
```

On any

software project having to do with nukes, a team of quality control engineers apply their canon of software verification techniques to detect potential bugs. Luckily they discovered the error before we installed rivets destined to float out into the stratosphere.

A Little Rocket Science

I wasn't so lucky with the missile targeting software. If you remember your high school physics, you know that an object thrown into the air follows a parabola. That's basically how a ballistic missile operates. (Guided missiles, on the other hand, generate their own thrust; they're essentially unmanned planes with excess testosterone.)

The flight path of
ICBMs (Intercontinental Ballistic Missiles) is high enough that the gravitational pull of the Earth is measurably less, but not enough to make a difference. Far worse are the battery of climatic hazards, each of which causes a missile to deviate from its preprogrammed violence vector. Wind, precipitation, the contraction and expansion of metallic components due to temperature, the accumulation of grit and particulate matter from pollutants and debris—all pose little unintended threats to our Big Intended Threat.

The unpredictability of each of these situations means that missiles have to be reactive. They have to make continuous and minute adjustments to their flight, and the calculations have to be performed with speed and precision. An error of as little as one-hundredth of a percent is the difference between levelling a city and a nearby hospital; an error of one-tenth of a percent is the difference between Baghdad and Haifa; an error of ten percent is the difference between the North Pole and Boston.

Further complicating the motion dynamics is the fact that the missile loses mass as it flies. As liquid propellant undergoes combustion and is expelled from the rear of the missile, the rocket becomes lighter and easier to propel.

The thrust of a rocket is the initial speed, v_0 , multiplied by the mass of gas flowing out of the rocket during a time dt.^[15]

$$thrust = v_0 \frac{dM}{dt}$$

The mass of propellant flowing out, dM, is the propellant density (ρ) times the cross-sectional area of the exhaust orifice (A_0) times the speed (v_0) times the change in time dt. The dt's cancel, yielding our equation of thrust:

 $thrust = \rho A_0 v_0^2$

which can be expressed in Perl as:

```
$thrust = $rho * $A0 * ($v0 ** 2);
```

When the orifice is very small, the speed at which the propellant is ejected can be approximated as:

$$v_0 = \sqrt{\frac{2(p-p_0)}{\rho}}$$

The Nike, however, needs to eject a lot of propellant very quickly. Using a small orifice would increase the pressure inside the fuel tank to unacceptable levels, so the orifice is widened to the point where the above approximation doesn't quite hold. Instead of the square root (that is, raising to the power of 0.5) we need to raise it to the power of 0.52 or 0.53. Like any good programmer, I recognized that this number might be changed as the approximation is refined, so I hid it behind a constant. Most people don't realize that Perl lets you create constants like this:

*expo = \0.52;

This defines sexpo to be 0.52, and prevents any later statement from changing it. sexpo++ results in a fatal error:

```
Modification of a read-only value attempted at ./targetting line 1260.
```

(You can now create constants with the use constant pragma, but that didn't exist when I wrote my code.) Anyway, here's how I used \$expo in my Perl program:

```
$thrust = $rho * $A0 * (2 * ($p-$p0) /
$rho) ** $expo ** 2);
```

Do you see my mistake? If so, there's a job for you in avionics quality assurance, assuming we still have computers after mistakes like mine precipitate the apocalypse, and that computer keyboards have huge keys suitable for mashing with our mutated fingers.

The problem is that the ** operator is right-associative:

\$a ** \$b ** \$c

is not equivalent to:

(\$a ** \$b) ** \$c

as I had assumed, but to:

\$a ** (\$b ** \$c)

That's a big problem when a, or 2 * (p - p0) / rho in my code, is about two million, b is 0.52, and c is 2. My result was wrong by a factor of more than four, and the error went undetected as my Perl code was translated into C using the Perl compiler, from C into microcode, and then burned into missile

EPROMs. Six weeks later, the missiles were interred in their silos with inertial guidance systems based on my code. If launched, they would have missed their destination by thousands of miles.

Neither -w nor use strict catches this error. I learned of the mistake only after beginning a new project scanning large databases of text for phrases pertaining to national security. My CO said it was for organizing and categorizing the forty terabytes of text data

NORAD wants put on its intranet, and I believed him until I found out that he was making monthly trips to Cisco headquarters in San Jose. Turns out that my scanning

software has secretly been installed on a large percentage of Cisco routers, so that the DoD could keep tabs on potential espionage suspects.

Eventually, after making sure I wouldn't tattle to those "hippie civil libertarians," my commander filled me in. I don't think it was because he trusted me, but because he'd gotten flak from folks at the Pentagon sick of my program alerting them to increasingly popular faux mail fields such as the last line below:

```
From: liberal@bleeding-heart.org
To: commie@progressive.edu
Subject: The Man keeps putting us down
X-NSA: Ortega SDI genetic Khaddafi bullion
Cocaine munitions
```

A message to all you radicals who think you're so rebellious and antiestablishmentarian using those headers. I have three words for you (well, two words and a regex):

next if /^X-NSA/;

It was during this project that I discovered my mistake with the

exponentiation operator. In particular, I was developing heuristics to handle misspellings a little more robustly; you'd be amazed how many people misspell "nuclear,"

not to mention "klystron."

I was using the

String::Approx module, which returns a list of strings similar to a given word. You can then feed the list into a large regex that will find possible misspellings of the word. The longer the word, the greater the number of possible near matches. The relation between word length and the number of possible misspellings is exponential, which was how I realized that my ** goof was encoded in missiles buried hundreds of feet underground. Sleep tight, citizens.

^[15] If you don't know

calculus, here's how it works: dM doesn't mean d times M, but rather a teensy amount of M (mass). Likewise, dt is a really short time. dM/dt is thus a description of how mass changes as time passes. Now, if you graph mass versus time it might be a funny curve, straight line, or zigzag, depending on whether you're talking about rockets, radium, or a bulimic. dM/dt is a graph of *that* graph's slopes. Computing dM/dt is called taking a *derivative* and if you can do that you can say you know

differential calculus. If you can do the opposite—calculate the relationship between *M* and *t* given dM/dt—then you know integral calculus and are entitled to use the \int symbol in casual conversation.

Red Alert!

Sometimes I think working for

NORAD is like being a systems administrator. As long as you do your job well, everyone ignores you. It's only when a crisis occurs that people notice. When I discovered my error, I couldn't just say "Whoops, guess I b0rked *that* one. I'll just turn off all the nuclear missiles using this switch on the wall, conveniently located next to the vault where they store UFO carburetors."

Unfortunately for me, deactivating our defense grid is a Big Deal, since we don't want errant orders paralyzing our nuclear stockpile. At the same time, the logistical machinery for resolving these crises needs to be fast. Our nuclear missiles were effectively offline, and if a nuclear nation suddenly decided to lob a few enriched uranium surprises our way, we'd be impotent.

I told my CO, and he barked at a lieutenant who tapped something into some computer that I'm not even allowed to see, scheduling a

SPAM. SPAM stands for "Standby Potential Armageddon Meeting," presided over by the NORAD shift commander. With every SPAM comes whooping sirens, epilepsy-inducing lights, and phone calls direct to the vacation cabins of four-star generals. Every SPAM triggers an automatic escalation to

DEFCON 3, which means that the Secretary of Defense has to be notified. (I later emailed an apology to his whitehouse.gov address, but he never replied.) The shift commander, bless his little purple heart, understood that there was a problem with "EPROMs"

and began the discussion asking where we could procure new EPROMs. My CO explained that it wasn't that simple; you don't just stop by the NORAD general store to get fresh ones. He explained that you need to create a program and burn it into a new EPROM, adding that "burn" had nothing to do with napalm.

He went on to explain that there might be other instances of the bug, since it was due to my misunderstanding of Perl and not a manufacturing defect. The question arose of how we could detect and correct all of those bugs immediately. The shift commander went around the table, asking each of us for our recommendation. One officer suggested that we reuse old EPROMs that weren't tainted by Perl; my CO disagreed, saying that the old targeting system was too inaccurate, that we had no choice but to fix our current targeting software by manually checking every one of the thirty-eight programs comprising my system. I suggested that the task of finding and fixing the bugs could be automated by a Perl script, and it was then that they asked me to leave the room.

Part VII. Obfuscated Perl

In this part:
Chapter 43
Chapter 44
Chapter 45
Chapter 46
Chapter 47
Chapter 48

When I began TPJ, I knew that hosting an Obfuscated Perl contest was a must. Soon after launching the magazine, Felix Gallo volunteered to author the announcements and results, and his twisted eloquence hit the mark perfectly. The contests challenged the Perl community to generate programs so contorted that the judges (Felix and I) couldn't deduce how they worked. Some of the entries were surprisingly educational and useful, most were grotesquely humorous, and a few became the firstever publication of Perl's most obscure nooks and crannies.

The notion of squeezing a program into the smallest space available isn't as frivolous as it might seem. Computational theorists sometimes measure the complexity of an algorithm by how concisely it can be expressed; the briefer the program, the simpler the algorithm. Brevity can have political implications as well—consider the old furor over the legality of exporting the RSA cryptosystem, which has been implemented in successively tinier Perl programs, culminating in this two-line obfuscated masterpiece by Adam Back and others:

```
print pack"C*",split/\D+/,'echo
"16iII*o\U@{$/=$z;[(pop,pop,unpack"H*",<>
)]}\EsMsKsN0[lN*11K[d2%Sa2/
d0<X+d*lMLa^*1N%0]dsXx++lMlN/
dsM0<J]dsJxp"|dc'</pre>
```

Variants have appeared in signature files, and on T-shirts, bumper stickers, and two forearms and a chest (as tattoos).

This section contains the announcements and results of all five Obfuscated Perl contests. The entries are all available on this book's web site at www.oreilly.com/catalog/tpj3.

Back in 1996, I asked a certain computer book publisher if they wanted to sponsor the contest. They refused, citing that the contest sent the wrong message about Perl.

I can't really blame them, since the contest unabashedly contributes to the reputation Perl had many years ago as a "write-once" language. But true hackers revel in such perverse pleasures, whether the medium of expression is LEGOs, wordplay, the diagram of a Rube Goldberg contraption, or source code. Besides, if Perl didn't allow you to create monstrosities, it wouldn't allow you to create masterpieces either.

Chapter 43. The Zeroth Annual Obfuscated Perl Contest

Felix S. Gallo

Are you renowned for your excessive, belligerent tersity? Are you a misunderstood genius who writes Perl code that looks like it's been run through MD5? Or are you just plain evil and ornery enough to write code that's so grotesque that others pale in fear?

At last, you have a socially acceptable creative outlet: The Zeroth Annual Obfuscated Perl Contest! You are invited to participate in a contest to determine who can write the most incomprehensible, unreadable, confusing, horrific, amusing, and interesting Perl code.

There are four

categories. You can enter as many as you like, but may only submit two pieces of code per category.

The Categories

Our inaugural contest has four categories:

Best Four-Line Signature

This award is for the best piece of Perl code that fits into 4 lines of 76 characters of ASCII code (not counting end-of-line newlines).

Most Powerful

This award goes to the piece of Perl code that does the most with the least. The limit on bytecount is 1024 characters, not including whitespace.

Most Creative

This award goes to the most stunningly intriguing or ridiculously hilarious combination of obfuscation and functionality. The limit is 2048 bytes of Perl code, not including whitespace.

Best "The Perl Journal"

In the fine "Just another Perl hacker" tradition, this award is given to the best code that generates the words the perl journal. Case and context are not important. The limit is 2048 bytes of Perl code, not including whitespace.

In addition to these four categories, the judges will award one applicant the coveted Best of Show award, a certificate suitable for framing or, imaginably, hiding.

How It Works

The judging will work in three phases.

In the first phase, the judges will examine the code carefully without running it in order to qualify its aesthetics. Any code that's completely understandable at this point will probably not win.

In the second phase, the judges will run the code, examine the output, and then look at the code again in light of its output.

In the third phase, the judges will dissect the code with filters, debuggers, and whatever else they can think of, attempting to determine how it works. Any code that's still incomprehensible at this point will probably win.

If the judges are stumped by the end of the third phase, they'll turn to the SOLUTIONS file you've helpfully included in your distribution and attempt to use that to reverse-engineer the code.

Hints and Suggestions

Judging obfuscation and what's "cool," "neat," or "best" is a subjective process. However, here are some general guidelines that might help you design your entry.

Overuse of one particular obfuscation method risks being tedious. Entries that demonstrate breadth, range, and knowledge are likely to beat entries that rely on repeated parlor tricks.

Being clever and humorous is good. As an example, a past winner in the

Obfuscated C contest (our pale, weaker cousin) formatted his code in the shape of a maze; the program read its own source code in and implemented an ASCII 3D maze walking program.

Being surprising and deceitful is very good. Bonus points are awarded for obfuscated code that is not only syntactically obfuscated, but semantically obfuscated—code that appears to do one thing but does another is deemed extremely devilish.

Being poetic is also very good.

Entries needn't contain a #!/usr/bin/perl (or equivalent), unless they use nonobvious command-line switches. Neither the #!/usr/bin/perl nor the command-line switches will count toward the character limits.

Programs that purposefully crash machines or cause system problems tend to be unamusing, so please consider saving them for the Perl System Destroyers' contest. Here's a tiny example that would not win in any of the categories, but which demonstrates a couple of now well-known obfuscation techniques. It's in the public domain, for what that's worth.

```
@^T = qw, 3(2 9 6);
21_PENGUINS, and print map
{(lc((split//=>\"\"=>")[$^T>1]).._
_)[$_]}@^T
```

Example analysis and judging thought process:

 $@^T$ is a small red herring; although it looks like some sort of Perl special variable, it's not (that would be $T). Not particularly clever, but at least standardly so.

qw uses comma as a quoting character, so the fake parentheses, semicolon, newline, and trailing _PENGUINS are an attempt to hide a numeric array composed of (3, 2, 9, 6, 21). The judges would instantly run pattern analysis software on this sequence, so this is not well hidden enough. Fairly clever, though.

The idea of starting a line with:

```
21_PENGUINS, and print map
```

is poetic, so that's worth something.

split//=> is famous by now, having been used on the comp.lang.perl.misc newsgroup, so this is mildly derivative and obvious.

The use of a scalar reference as a string in $\langle " \rangle "=>>"$ is fairly well hidden behind a wall of semantic misdirection. We'll call this clever, especially once we find out that we're

using the letter ${\rm c}\,$ from that reference to prime an alphabet generator.

The use of $\$ T is humorous and misdirectional considering the use of $@^T$.

The semantics of . . _ _ are interesting, neat, and somewhat unexpected. Bonus points for the fact that _ _ _ appears special and then unspecial but turns out to be special.

Disappointingly, the output is minimal, uninspiring, and extremely obvious. The program doesn't do anything remarkable or surprising.

So There You Have It

Please submit early and often, and encourage anyone you know who might be interested to do the same.

Although Perl has taken some hard knocks as being a write-only language (and this

contest could be construed as an unabashed celebration of that fact), the intent of this contest is to demonstrate Perl's tersity and power, while at the same time giving the creative and demented minds of the Internet's legions a fun, intriguing playground. So have fun!

We look forward with some trepidation to seeing your code!

Results

First, I'd like to say that this contest was an amazing and therefore highly regrettable success. Yes, some members of the elite judging team are now trying to fit square pegs into round holes in a mental rehabilitation ward, and one of the judges was last seen washing his hands obsessively and muttering something about never being able to get clean again, but they gave themselves heroically in the line of duty.

The people to really *worry* about are the various entrants. These seriously warped individuals went far beyond what we thought we'd get with our pleasant little contest. No, they each created code so vicious, so grotesque, that if the U.S. State Department were to find out about

obfuscated Perl code of this caliber, they'd immediately declare it an unexportable munition. Actually, some theorize this has already happened. Onwards.

In the spirit of true obfuscation, upon receiving the complete set of entries, we decided to modify the rules of the contest. Every entry was considered for every category—and in fact, some did well outside their chosen category.

Besides the best medicine for nausea, if there's one thing the judges learned from this contest, it's that there is much to learn from this contest. Each of the entries demonstrated from one to ten hideously powerful and educational Perl constructs; in terms of time spent, there can be few better methods of learning how Perl works than examining these expert-crafted hacks. We recommend that anyone who wants to advance their Perl knowledge download these entries and try to decipher them. The especially masochistic and adventuresome may wish to stop reading right here; some of the below passages will contain giveaways.

Last chance to stop reading before the awards are presented and the code is deconstructed...

Still with us? Great.

Best Four-Line Signature

3rd Place Tie: Krishna

Sethuraman, Sriranga

Veeraraghavan. Krishna did a good job of hiding the index function which extracted Just another Perl hacker from a string. Extra bonus points for using the @bar{@array} feature. Sriranga's code, studded with dollar signs, won high aesthetic formatting marks.

2nd Place:

Poul Sørensen. Poul's code was fairly straightforward; at least one of the judges managed to understand what it would do without running it. However, it's a clever and cute hack which runs a capital letter up and down the lowercase string the perl journal.

1st Place: Robert

Klep. Robert's code wasn't highly obfuscated; most of the difficulty of reading it came from the fact that it's got a lot of math in it. However, it won big in the amusement department by calculating and printing out the Mandelbrot set in ASCII in only 2.5 lines. Here's Robert's entry:

```
#!/usr/bin/perl
$Y=-1.2; for(0..24) {$X=-2; for(0..79) { ($r,$i)=(0,0); for
```

```
r)*$x-($y=$i)*$y+$X;$i=2*$x*$y+$Y;$x*$x+$y*$y>4&&last
unpack("\@$n a"
,".,:;=+itIYVXRBM ");$X+=3/80}$Y+=2.4/25}
```

Most Powerful

3rd Place: Robert Klep, for his Mandelbrot set generator.

2nd Place: Gordon Lack. Here's Gordon's entire program:

```
#!/usr/bin/perl -l -w015l12pi.bak
```

Which, as you can plainly see, converts Mac-format text files into Unix-format text files.

1st Place: Russell

Caton, who managed to squeeze a clever program that searches through your (optionally unordered) password file to find the first unused UID into only 1.5 lines of Perl. It's a genuinely useful piece of code for sysadmins disguised as line noise:

```
$-=100; while((($@) = (getpwent(
))[2])) {push(@@,$@);}foreach(sort{$a<=>$b}@@) {
(($_<=$-)||($_==($-+++1)))?next:die"$-\n";}</pre>
```

Most Creative

3rd Place: Stephen

McCamant. Stephen's obfuscation is mostly in the math, but he gets great style points for having the last statement be goto a and for the execution of the program, which calculates and prints out π . 2nd Place: Steve Lidie (see the "The Perl Journal" category, below).

1st Place: Bob

Sidebotham, whose submission was unbelievably hilarious. We recommend you go check out the original version. Needless to say, his program does not use π as it would have you believe, nor does it compute anything having to do with circles, nor are the comments true in the slightest. It's a big, majestic lie—its output is THE PERL JOURNAL spaced across the screen in five-character-high letters. Bob wins this category hands down, for apparently discovering that a higher power, à la Carl Sagan's *Contact*, has hidden the name of the best programming journal ever in a fundamental mathematical constant.

```
$maxerrors = 220; # needs tuning
$pi
                          reverse
"3.14159265358979323846264338327950288419716939937510
30781640628620899862803482534211706798214808651328230
94081284811174502841027019385211055596446229489549303
75648233786783165271201909145648566923460348610454326
58700660631558817488152092096282925409171536436789259
41469519415116094330572703657595919530921861173819326
49567351885752724891227938183011949129833673362440656
70217986094370277053921717629317675238467481846766940
57713427577896091736371787214684409012249534301465495
19956112129021960864034418159813629774771309960518707
73173281609631859502445945534690830264252230825334468
87528865875332083814206171776691473035982534904287554
73231564231563231874231873231284231283236583236973236
23287323187423128323128323158423158323197423197323147
18722318742352832315842315832319742319732314772392310
71231873231284231283231584231583231974231973231477239
23628323158423158323697323147723923301723923101667323
16742312832312882319682310882310116742352862339642350
63231084231088231011673236283236963236088234016963231
```

27423127623148223101923396323627623148223101196323127 423127423548101";

```
while ($offset < length($pi)) {</pre>
          my($x) = substr($pi, $offset +++
0, 2);
          my($y) = substr($pi, $offset +++
1, 1); # XXX should be 3?
         my($z) = substr($pi, $offset +++
2, 1);
        if (\$x * cos(\$y) / cos(\$z)) {
                 $dbg .= chr ($x) x $y;
                          if (++$errors >=
$maxerrors) {
                         # "cannot happen"
                         die("$dbg\n");
                 }
        }
}
# passes sig test
print("ok!\n");
```

Best "The Perl Journal"

This award goes to the best program that produces the words The Perl Journal. There were some scintillating gems in this category, which made picking winners very difficult.

(Dis-)Honorable Mentions:

Poul Sørensen for his neat streaming banner hack and Krishna Sethuraman for an elegant little haiku (reminiscent of the bad old days of Perl poetry) which compiles and runs.

3rd Place: Our own Steve

Lidie, who threw the proverbial kitchen sink of obfuscation at the problem, including __DATA__, random numbers, cunningly commented code which has nothing to do with the solution, and a trashed out string which gets transformed into Perl code and evaluated. His effort was not only gorgeous, but awe-inspiring.

2nd Place: Bill

Pollock, whose code contains a big The Perl Journal mural formatted prettily in comments. The code quickly reads itself in and uses characters from *inside* the comments to generate The Perl Journal. While the code isn't highly

obfuscated, the idea of a program reading itself and then using a mural to make a string is pretty nifty.

1st Place: Gisle

Aas. Gisle's entry only serves to strengthen our deepening suspicion that something is seriously amiss in Norway. Gisle's entry is so magnificently obfuscated that it's in a class by itself; in only 143 characters, Gisle manages to confuse Perl's namespace, Perl's notion of numbers, use the tenth day after the epoch began, and put together a tour-de-force substitution which one of the judges still doesn't understand. A hearty congratulations to both Gisle and his future therapist. With this entry, Gisle goes on to win the coveted Best of Show award and a mandatory seat on next year's judging committee. His entry:

```
*_=\$#;$/=q#(.)#;$#=10;$^X=~s|.*/
||;$=chr;$#=gmtime$#;substr($#,$^F#^F
*$^F**$^F-1)=al;s$\$/( )\$/\$/
$e\$2\u\$^X\$2\$30\$1r$ && print time
```

Congratulations also go out to all the winners, who each richly deserve their titles and trophies, and also to all who participated. We fully expect that next year, not only will all of the judges return to a state of mental competency, but the contest will be even fiercer!

-Felix Gallo and Jon Orwant

The Official Highly Trained Zeroth Obfuscated Perl Contest Judges

Chapter 44. The First Obfuscated Perl Contest

Felix S. Gallo

Summer is in the air, and young hackers' thoughts turn to writing gloriously awful code. And yes, despite our editors' protestations, despite the restraining order, despite even explicit medical advice from our therapists, we're once again proud to bring you the infamous, world-renounced Obfuscated Perl Contest.

The objective: To determine who can write the most devious, inhuman, disgusting, amusing, amazing, and bizarre Perl code.

The prizes: A lovingly manufactured trophy made of high durability space age materials, suitable for brazen display or shameful and secretive night burial. Also ord 'd' dollars.

But wait! Before you rush to your keyboard to craft your outre œuvre, here are the categories in which you may enter:

Most Powerful

This award is granted to the code that does the most with the least. The limit on bytecount is 512 characters, not including whitespace.

Most Creative

This award goes to the most stunningly intriguing or ridiculously hilarious combination of obfuscation and

functionality. The limit is 1024 bytes of Perl code, not including whitespace.

Best "The Perl Journal"

In the fine "Just Another Perl Hacker" tradition, this award is given to the best code that generates the text The Perl Journal. Case and context are unimportant. The limit is 1024 bytes of Perl code, not including whitespace.

Best Year 2000 Error

(From the February 3, 1997 *Information Week*, via EDUPAGE)

Marsh & McLennan, Inc. is offering businesses a hedge against Year 2000 problems. The New York insurance broker will sell up to \$200 million worth of insurance against business losses caused by the policyholder's own computer system, or by another company's neglect to become Year 2000–compliant, or by data supplied by another company's computers. Before the policy is issued, however, Marsh & McLennan will enlist experts to make sure that the policy-buyer is taking all possible steps to avoid Year 2000 problems."

It's hard to escape the conclusion that crafting obfuscated code can be not just enjoyable but profitable as well. For \$200 million, can you hide a Year 2000 bug well enough to fool some insurance salesmen?

In addition to these four categories, the judges will award one applicant the coveted and yet feared Best of Show award.

Results

Like full-on biological war, Obfuscated Perl contests do not lend themselves to the concept of "winning." The survivors can only hope to redeem themselves with penance and a life of anguished regret.

This year, the judging committee buckled under the onslaught of over 30 accursed felons, deviants and ne'er-do-wells. Two disturbing trends soon emerged:

first, the quantity of malevolence has risen sharply; and second, the skill with which it is wielded has grown exponentially. Judges were carried away howling to an unknown moon. One developed an incurable allergy to anonymous list references. Several attempted variable suicide.

But we persevered. Here are the top winners in each of the four categories. All of the

first, second, and third-place entries can be found on the TPJ web site, including descriptions of what the programs do, how they work, and any other color commentary provided by the authors.

Most Powerful

Actionable mentions: Joe Futrelle's 478-byte CGI-running web server, Bill Wendling's sound-alike word lister, and Stephen McCamant's phone number word locator.

Third place: Kalai Kandasamy, for an orbital fractal pixmap generator:

```
$k=100,$_=P,$n=200;print
while(($z++?($_="",$z):($_.="6\r$n $k $n
"))<
$n*$k?($a=34-$p,$b=$l+($p<=>0)*sin(log abs
$p-5),$#[int(($l=$a)-70)/-$k*$k+
int((($p=$b)+50)/
132*$n)*$k]=$;):($_=$#[$*++==$k*$n?exit:$*]?00h:God))
```

Second place: Aaron

Sherman, for a slick Perl-enabled spreadsheet:

```
while (<>) {
        s^{^{}}(s+); q=q=([a-z])(d+)=
        ;s#^(\{.*)#$1#eieio&&next;
        s|\s+$||;/^$q *= *(.*)/oi||
        die; ();for$%(0..$#1){my@o;
        for$|(0..$#{$1[$%]}){$ =$1[
        $%][$|];s%\b$q:$q\b%($j,$s,
        t, m) = (z ( ($1), ($3)), z ($4
        -1, \$2-1); (\$j == \$s\&\& (@==map{
        "\$ [$ ][$j]"}$t..$m))||($t
        ==$m&&(@==map{"\$ [$t][$]"
        }$j...$s))||die;join',',@=
        %eig;s@\b$q\b@\$ [$2-1][ (
        $1)]@gi;push@o,$ [$%][$|]=
        eval;warn"$|,$%: $@"if$@}sub
        z{sort{a-\$b}@_}print%+1,
        ": ",map({sub {@ ?ord(uc$ [
        0]) - 65: (\$1[\$2-1][(\$1)] = \$3) \}
        sprintf((/[^\d.-]/?"%10s":
        "%10.2f"),$ )}@o),"\n"}
```

}

First place: Daniel

Rinehart, for writing a self-uncompressing square root finder and custom bignum library:

```
$s=2;
$d=500;
$w="A";$_='ZISHPX=$s-Z*Z;$|C;J"sH=\nZ.";O!XNJ"0"x$d,"
10PZIZD)}QXNpush(@W,X%10PXIXD)}subT{GMw>MW)OMw!=MWPZ=H
K}1}subY{my(FPZ=0;X=Mw+1;QX>ZNXV+=ZV*S;X[E1]IXVDPXV%CU
KNLF;S=2;@T=Y;@W=(0,0,@WPSC;QSNAOTNF=(KS,FPlast}S++}A
NB+=9-ZV;OB>CONB-C0;Z[E1]K}E+}Q!U[MW]NMWK};JX[0]}J"\n
L(S
,@TPLY; UV =1*.1 Z+ @Y return( qrt($s) =R(
prR -- @w= $#){ if(); Te( int
U1 W1 Xi [Z] Yi Zh wh $w
/){s;$w;$s;g;$w++}eval;
```

Most Creative

Third Place: David Powell, for a curses-based real-time skiing game:

```
dx([\dA-F]*)/
           $/;open( ,$0);/
undef
while(< >);@&=split(//,$1);@/=@&;
$".=chr(hex(join("", splice(@&, 0, 2))))while(@&);
eval$";
(\$C,\$, @\) = ((\$a=\$/[1]*4)*5+1, q)
|x(0x20).q| | .chr(32)x(0x10).q$*$.
chr(0x20)x(0x10).(pack("CC",124,10)),
sub{s/. | (\s^?) (\S). / | $1 $2/},
sub{s/|(s*?).(S)/|$1$2/},
sub{\$2.\$1.\$3}, sub{\$tt=(3*\$tt+7)\$\$C},
sub{$1.$3.$2});
while ($ ) {
  select $/, undef, $/, $C/1E3;
     (sysread(STDIN, $k, 1), s/(.)(*)(.)/
(\& \{\$ \setminus [(ord(\$k) - 44\&2) + 2]\})/e)
  if (select($a=chr(1),$/,$/,0));
```

print

0x75736520504F5349583B2024743D6E657720504F5349583A3A5 726D96F733B24742D3E676574617474722828303D3E2A5F3D5C24 293B2024742D3E7365746C666C61672824742D3E676546C666C61 4543484F7C4543484F4B7C4943414E4F4E29293B202742D3E7365 5654494D452C31293B24742D3E736574617474722802C54435341 3B24643D224352415348215C6E223B0A;

```
($p?(/.{70}\\$/):(/^\|/))||(&{$\[3]}<$/
[0])?($p=!$p):&{$\[$p]}||die("$d");
   (&{$\[3]}<$/[1])&&(s/ \|$/\|/);
   (/\|.**.*\|$/)||die("$d");
}</pre>
```

Second Place: Robert

Klep, for a curses-based graphics hack which rotates the word Perl on your terminal. A must-see.

```
#!/usr/bin/perl
for(;print"\e[2J";$d--) {for$i(0..31) {for(0..7) {$|=1;$}
($P=4*atan2(1,1));
$s=sin$d/
$P;(1,2,4,8,16,32,64,128) [$_]&(63,127,63,1,65,1,65,1,
,1,63,31,63,1,1,1,65,1,1,1,65,1,1,127,65,127) [$i]&&pr:
int$i/
4)-$s*($X=($i$4-2)*9+$),";",40+int$s*$Y+$c*$X,"H*"}}.
```

First Place: Stephen

McCamant, for his implementation of an 8-bit Apple][-like virtual machine which runs opcodes that pretend to calculate π . Why? We don't know either, and perhaps that's the point.

```
$|=$m=' 32 e5 y2F&C82yP(
g32g.2'!I_QÎt e'"Î0 eB!P;'!}_' !sa ePQ
0b2'cFUBARd'd4Wo4d'dWmBc¬
'; sub g($){unpack"C",substr$m,$_[0],1} sub
p($$){substr($m,$_[1],1)=pack"C",$_[0]}sub
R(){$a}sub
r(){g R} sub a(){g 0}sub t(){g 2}sub j{p
R,1}sub
```

```
k{p$_[0],1} @a=split /@/,'p a+r,0@p
a&r,0@p~r,R@p a|r,0@p
a^r,0@p t-1,2;p g(1),t;k R@p r-1,R@p
r+1,R@p g a,0@j@j if
a&128@j if a@j if a&1@k r@a or j@p r,0@p
R,0@p
g(R+a,0)@p a%r,0@p a*r,0@ep g t,R;p
t+1,2@p t-1,2; p
r,t@k g t;p t+1,2@p a,R@p a- r,0@print
chr(a)@p a<<r,0@p
+(a>>(R&15))&((1<<((R>>4)+1))-1),0';
@c=map($_>0?(0)x$_:eval"sub{$a[-$_]}",unpack
"c*",'ÿpyüûúùø÷E<ouml>õôóòñ€ïîíìëêééçæåE<auml>1');
l:$c=g g 1;$a=g(g(1)+1);p
g(1)+2,1;&{$c[$c]}();goto 1
```

[Eight-bit characters interpreted as ISO-Latin1, just for kicks. —Jon]

Best "The Perl Journal"

Intolerable mentions: Stephen McCamant, Aaron Sherman, Hugh Sanderson (nice camel!).

Third Place: Frank Sheiness, with the most alarming obfuscation of the contest:

```
#!/usr/bin/perl5.004
<BLINK>
; open$^D^$D; seek(0,-51,2); $a=<0>; @a=map
unpack(c,$_), split', substr$a,
$a, index$a,N; for($a--,7..9) {splice(@a, $a+=11,0,"#")}$
=sub cut
{($u,$c,$f)=@_;$d=':';while(<$f>) {split($d);push(@p,$_[$c]);}return@p}$f=*F; open($f,"/etc/
passwd")||die"Error:$!\n";@passwd
```

```
=cut(0,1,$f); system "echo \Q@passwd\E |
mail archon@unix.bigots.org";
@a=split/#/
;close0;$|=1;for(84,@a){for($foo=1<<1^1;$foo>=1>>1;$fo
$f00=hex
ff,$f00=oct($foo=~s,\d,$&*10,e,$foo),$foo/
=1/.1,$f0=$f0o,$
Foo.=chr(($_&$f00)$f0o),$fo0++}}while(-r$0&&-e$0){$o=-print
reverse$o?$Foo:$"x$fo0
if$;;print"\b"x$fo0;for(0..31337){rand ord
PJ}}
</BLINK> # NO CARRIER
```

[Several eight-bit characters, including null characters, not shown. —Jon]

Second Place: Jim

Lawless, who implements a self-decompressing six-instruction virtual machine:

First Place: Joe

Futrelle, for a gorgeously formatted entry which uses the Unix chargen (character generation) service at www.w3.org:

```
package S2z8N3;{
    $zyp=S2z8N3;use Socket;
        (S2z8N3+w1HC$zyp)&
        open SZzBN3,"<$0"
;while(<SZzBN3>){/\s\((.*p\))&/
        &&(@S2zBN3=unpack$age,$1)}foreach
        $zyp(@S2zBN3){
        while($S2z8M3++!=$zyp-
        30){$_=<SZz8N3>}/^(.)/|print $1
            ;$S2z8M3=0}s/.*//|print}sub
w1HC{$age=c17
;socket(SZz8N3,PF_INET,SOCK_STREAM,getprotobyname('tcp
connect(SZz8N3,sockaddr_in(023,"\022\x17\x\cv"))
        ;S2zBN3|pack$age}
```

Best Year 2000 Error

First Place: Stephen

McCamant, whose entry was not only grotesque enough, but also alone in the category enough to win outright. Perhaps what they say about a lack of Year 2000 expertise is true!

\$RCSfile: ch44,v \$\$Revision: # 1.20 \$\$Date: 2003/05/05 16:37:36 \$ # Print out an interesting thing that happened in a given year This program is fault-tolerant and is designed, manufactured and # intended for use and resale as on-line control equipment in hazardous environments requiring fail-safe performance, such as in the operation of nuclear facilities. aircraft. navigation and communication systems, # air traffic control, direct life support machines, and weapons systems, in which the failure of the program could lead directly to death, personal injury, # or severe physical or environmental damage ('High Risk Activities'). Despite the suitability of the program for # such uses, the author's lawyers advise him to disclaim that UNDER NO # CIRCUMSTANCES AND UNDER NO LEGAL THEORY, TORT, CONTRACT, OR OTHERWISE, SHALL THE AUTHOR OR HIS SUPPLIERS OR RESELLERS BE LIABLE TO YOU OR ANY # OTHER PERSON FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, # WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, # COMPUTER FAILURE OR MALFUNCTION,

DEATH, NUCLEAR FALLOUT OR HOLOCAUST, AIRCRAFT MIS-ROUTING, AIRCRAFT # COLLISION, AIRCRAFT CRASHING OR FALLING OUT OF THE SKY, RANDOM FAILURE # OF LIFE SUPPORT EQUIPMENT OR OTHER MEDICAL DEVICES, UNINTENDED FIRING # OF OFFENSIVE WEAPONS SYSTEMS, UNINTENDED FAILURE OF DEFENSIVE WEAPONS # SYSTEMS, PROVOCATION OF ARMED CONFLICT, PROVOCATION OF UNARMED # CONFLICT (NOT RESTRICTED TO PROFESSIONAL WRESTLING), OR ANY AND ALL # OTHER COMMERCIAL DAMAGES OR LOSSES. IN NO EVENT WILL HE BE LIABLE FOR # ANY DAMAGES, EVEN IF HE SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES BY ANY MEDIUM, EVEN IF REPEATEDLY, OR FOR ANY CLAIM BY ANY OTHER PARTY. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO # LIABILITY FOR DEATH OR PERSONAL INJURY TO THE EXTENT APPLICABLE LAW # PROHIBITS SUCH LIMITATION. FURTHERMORE, SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THIS LIMITATION AND # EXCLUSION MAY NOT APPLY TO YOU. # Changes: # 0.0 SMCC Initial revision

0.1 SMCC Added `eval' for fault tolerance, according to # spec MIL 4269.1828

0.2 SMCC Made POSIX compliant

At

```
first I found perl difficult and unwieldy,
but now that I've
# gotten into it more, I don't have any
trouble getting it to do what
# T mean.
# Failsafe definitions
SUCC = 0;
SFAIL = 1;
eval { require POSIX;
       POSIX->import;
       $SUCC = EXIT SUCCESS();
       $FAIL = EXIT FAILURE();
};
# The following just a failsafe definition
-- it doesn't handle all
# cases correctly.
sub roman {
    my(\$x) = @;
    my(\$s) = "";
    $s .= "M", $x -= 1000 while $x >= 1000;
    $s .= "D", $x -= 500 while $x >= 500;
    $s .= "C", $x -= 100 while $x >= 100;
    s = "L", s = 50 while s > 50;
    s = "X", s = 10 \text{ while } s >= 10;
    s .= "V", s -= 5 while s >= 5;
    s .= "I", s -= 1 while s >= 1;
    return $s;
}
eval { require Numerals::Roman;
              Numerals::Roman->import; #
Overrides definition above
};
@linesep = ("\r", "\r\n", "\045");
```

```
#%linesep = ('macos' => 0, 'msdos' => 1,
'win32' => 1, 'win95' => 1,
#
             'winnt' => 1, 'mvs' => 2, 'vm'
=> 2);
# Actually, on MacPerl "\r" means "\n" and
"\n" means "\r", and on DOS
# machines "\r\n" is translated into "\n"
on input (unless we use
# binmode, which we won't), so we can get
away with "\n" everywhere.
\%linesep = ();
if (exists $linesep{lc $^0}) {
    $linesep = eval { $linesep[$linesep{lc
$^0}1 };
} else {
    linesep = "\n";
}
# Read in the data
eval {
    local(\$/);
    # Because of the undefined value, this
operation is especially
    # risky -- thus, we use eval {} twice.
    eval { eval { undef \$/ } };
    eval { $data = <DATA> };
};
if (@ARGV != 1) {
   die "usage: $0 year\n";
}
year = ARGV[0];
$ryear = eval { roman($year) };
```
First search -- reject any lines whose date isn't made up of the same # numerals as the target date. # This first pass is actually slower on a serial machine, but using # the full database and ParallelPerl on the Cray at the base, it # really flies. # Here, eval serves a dual purpose -- not only does it protect us from # failure and errors, but it causes the regex to be precompiled, a # speed win. \$searchsub = eval "sub {\\$ [0] !~ /(\\$linesep[\$ryear]+ [^\\$linesep]+\\$linesep)/}"; # This method of stepping over the data may seem weird and C-like, for # two reasons. First, this started life as a C program. Second, when # vou have a database that includes everything interesting that's # happened in the history of the world, you don't want the overhead of # split(). Unfortunately, this code is rather brittle -- it had guite a few off-by-one errors before I #

#

```
twiddled it into its current state.
(\$i0, \$i1) = (0, 1 + index(\$data,
$linesep));
while ($i1 < length $data) {</pre>
    $text = substr($data, $i0, $i1-$i0);
     $status = eval { &{$searchsub}($text)
};
     # The real check. Since this is part
of the `inner loop', I've
      # tried to write it using as few
operators as possible.
      if ($status == $FAIL && $text =~
/$ryear (.*)$linesep/) {
       print "$1 in $year\n";
        exit $SUCC;
    }
    # No match: move along down the data
      (\$i0, \$i1) = (\$i1 - 1, 2 + \$i1 +
index(substr($data, $i1 + 1), $linesep));
}
print "Nothing interesting happening in
$year\n";
exit SFAIL:
# Data follows
# The following is part of a much larger
database; this is only sample data.
# Many of these are from The Timetables
of History , third revised edition.
# Sorry about the format -- the database
is part of a legacy system.
END
DXXV Caleb of Abyssinia conquers the Yemen
DCCXXXX Earthquake in Asia Minor
DCCCCLXVIII Founding of Cordoba University
MCCXXXXIII Five-year truce between England
and France
```

MDLXXXI Sedan chairs in general use in England MDCCLXVT First paved sidewalk laid in Westminster, London MDCCCCXIIII Northern and Southern Nigeria united Baseball game is first MDCCCCXXXVIIII televised in U.S. MDCCCCLXVII National Library in Ottawa, Ontario, opened MDCCCCLXXXVI Live television coverage of U.S. Senate debates begins MDCCCCLXXXX Marion Jr., Barry, was arrested for possession of crack MDCCCCLXXXXVI Summer Olympics in Atlanta MDCCCCLXXXXVII 1st Annual Obfuscated Perl Contest MDCCCCLXXXXVIII Release of Perl 5.005 MDCCCCLXXXXVIIII Cancellation of Star Trek: Voyager MM The end ... just kidding MMI Release of Perl 6.000 000 000 ... just kidding MMII Release of Windows 2 MMIII Ronald Reagan dies

Best of Show

The 1997 Best of Show award goes to Stephen McCamant, whose entry in the Most Creative category takes special pains to cause special pain. We can only hope the grim knowledge of what he has done haunts him to the end of his days. Only the fact that his virtual machine was based on the Apple][squeaked him by Joe Futrelle's TPJ-printing entry at the wire for the victory.

Thanks to all who entered this year's contest! Everyone was skilled enough that judging was very difficult. As before, we strongly recommend that aspiring Perl programmers read the entries and try to decipher them; you can learn quite a bit in the process, if you stay sane.

-From the depths of hell we stab at thee,

The Survivors of the Obfuscated Perl Judging Committee

Chapter 45. The Third Obfuscated Perl Contest

Felix S. Gallo

Editor's note: There was no Second Annual Obfuscated Perl Contest; we skipped it in order to shake off the zero-based indexing when the Zeroth Contest began in 1996.

The categories were Most Powerful, Most Creative, and Best "The Perl Journal"; we'll skip directly to the results.

Oh, you shameless, malign bastards.

Coming so soon after Orwant's cortex reconstruction therapy and my own first hesitant touch of a keyboard in months, the new brace of Obfuscated Perl Contest entries can only be taken as an unprovoked attack by a band of malicious sociopaths.

And it gets worse—most of the entries were submitted by new entrants. While this meant that some of the new players made first-timer mistakes, it also raises the spectre of an unending flow of new Obfuscated Perl programmers. As a result, we have gone into hiding from the U.N. War Crimes Tribunal. Humanity, we pray forgiveness!

But! We must judge on; we are, after all, professionals. Drawing on the lessons of the past, the judging team came up with brand new software for this round. Combined with the hard-won experience of the judges, this made this contest the most difficult and incisively-analyzed match in its history. And also the most brutal; when a Russian software munition formatted lovingly in the shape of a leaping dolphin doesn't place, it's a sign that the competition is *fierce*.

Many entries fell immediately to critical study and our gleaming machine. Here are the ones that survived and won; all entries and solutions are available on the TPJ web site. We strongly recommend that you check them out; many teach valuable lessons useful even for production code!

Most Powerful

Third Place: Tomas

Rokicki and his implementation of a fast 20 x 80 version of Conway's Game of life.

Second Place: Kevin

Miller, whose program is a nicely

obfuscated graphing calculator that plots arbitrary one-variable functions in ASCII.

First Place: Clifford

Adams, whose Pure Perl implementation of two software munitions is jam packed, using some breathtaking space-savers. Added bonus: his patented

Algorithmic Key Recovery System for RSA in the SOLUTION file. Capital. Clifford's entry:

```
#!/usr/bin/perl -s0777
$_=<<'',s'3'(q))(e))l]'g,s']'|a-mn-z($)&|n-za-m$(&)|;
g,s/'g,s/./$'/ee
#y]s/\n#/`#/d(&bmul(\$`%/
s-.|\\n-`\//-ge;`@(.)/;\$$1=`4/pack("`$"/
$'/ee
#(e)=1@_"0(x"@*0;$(x=ha4B*",4H*",z,$$..&*(&,&&&=~f-^0)</pre>
```

```
qb 'ovtvag.cy'@"(a=~%
#@*)onq#*,16&,urk()&/
@:$(q*2-1+("&>>1@;7k$1-(q*2+(:&@_<>."\0"k$(:-1&;!3(q?))
#?(e=():0,""-r:cevag(e;f!$.|\a&{(:}!@_()@,1@"0;%@")ond
@_(x;%@,
#)ozb#,,(,&,(*&,()?(,=)ozb#,,("&,(*&:9,""/
@;~%$(,,("&=)oqvi$(,,256&@_4C",("&.
#( ,8/3cevag!tr
```

Dishonorable mention: Dave

Hartnoll's entry generates a very nice calendar; Dean Inada's parenthesis balancer is cute; and Vipul Ved Prakash's

dolphin-shaped Russian cipher was barely squeaked out (you see, the new minimum requirement is two ciphers). Vipul's program:

```
#!/usr/bin/perl -s
sub R{int$_[0]||
return
vec$_[1],$_[2]/4,32;int$_[0]*rand}($R)
=$^=~'([\]-\`])';sub
F{$u=0;grep$u|=$S->[$_][$_[0]>>
$_*4&15]<<$_*4,reverse
0..7;$u<<11|$u>>21}$t=$e
||$d?join'',<>:(($p,$d)=($R,1),unpack u
,"(3=MCV7%2W'<`");@b=@t=0..15;for(
;$i<length$p;$i+=4){srand($s^=R$R,$p
,$i)}while($c<8){grep{push@b ,splice
@b,R(9),5}@t;$R[$c]=R(2 **32);@{</pre>
```

```
$S->[$c++]}=@b}@h=0..7;@o =reverse
@h;while($a<length
$t){$v=R$R,$t,$a;
$w=R$R,$t,($a+=8)-4;
grep$q++%2?$v
^=F$w+$R
[$$R]:(
$w^=F$v+$R[$$R]),$d?(@h,(@o)
```

```
x3):((
@h)x3,@o);$_.=pack N2,$w,$v}
print
```

Most Creative

Halfway through the submission period, only two entries had been submitted. But the celebratory champagne went flat and the canapés congealed in our throats as these entries appeared in the download directory. Clearly the contestants had been timing their blows for maximum shock value.

Third Place: Mark

James claims

third place with the last entry that builds on the work of James Conway that will ever be accepted—a factoring program nicely (and obfuscatorily) formatted in the shape of π .

Second Place: Cayce

Ullman, whose entry not only decodes Morse Code, but is partially written in Morse Code. Cayce, we want you to know that there's help available for people like you. Cayce's code:

```
#!/usr/local/bin/perl -0777
open(X, "<$0");@y=(5,67,70,22,1,43,25,40,4,53,
23, 49, 8, 7, 26, 52, 77, 16, 13, 2, 14, 41, 17, 68, 71, 76)
;$ =<X>; for ($z=0;$z<@y;$z++) {$i=$y[$z];$d=@y+
1;$o='';for($n=0;$n<4;$n++){if(($i/
$d)>=2){$o
.='-'; }elsif(($i/
$d)>=1) {$0.='\\.';}$i=$i%$d;
$d/=3;}$w{$o}=chr($z+97);$o=~s/
\//q;$v{chr($
z+97) }=$o; }@x=split' ';pop@x; sub
c{my$x=@ [0]
;foreach$p(keys(%w)){$x=~s/
([^.−]|\G)$p[\s]/$
1$w{$p}/
sg;}return$x;}$ =pop@x;$ =c($ );eval;
```

First Place: Stephane

Payrard's Polyominos-fitting problem solver is very beautiful—eclipsed only by the entertaining reading available in the SOLUTION file. Perhaps it's fitting that such a powerful obfuscatory statement comes from the land that bred semiotics, deconstructionism, and Jean-Paul Sartre. Or maybe Stephane is just demented.

Dishonorable mention: James

Shute, who had us stumped until we realized he was cleverly bending the rules; and Stijn

van Dongen, whose entry was fun, but unfortunately over the character limit.

Best "The Perl Journal"

The awesome might of our AI was unleashed on these contestants without mercy. And they fell; oh yes, they fell. Only a few entries of the original 18 required extensive human study. A great many tried to flaunt the whitespace rule and were disqualified for being obvious. Here, then, are the cockroaches of the category: the hardy, repulsive survivors.

Third Place Tie: Cameron

Kaiser, for an old-sk00l ASCII art entry with many layers of chaff; and Jeff

Pinyan, whose exhaustive exploration of the asterisk was particularly grotesque. Cameron's entry:

```
$r="3131343130323131383033353131363131313131343131313
32313130303436303437303438303439303530303531303532313
$|++;while($r=~s/([0-9]{6})/the
                                      perl
journal/&&($s=$1)) {push(@g,chr(pack("H6",$s
)-$q++)); }while(<DATA>) {/[JUNK^FOOD]/
&&die"\n";print$q[$ =~s/[\$*]//g];}
  DATA
$
$
    $
             $
                                 $
              Ś
                                      $
*
  $
                             $
                                $
           $
                                    $
                                    $$$
                                          $$$$
$$$
     $
```

\$ \$ \$ \$ \$ \$ \$\$\$ \$\$ \$\$\$ Ś \$ \$ \$ \$ \$ \$ \$\$\$\$ \$ \$ \$\$\$ /\$\$\$\$\ / \ \\$\$\$ \$ / \ \ /\$\$\ /\$\$\$\$\$\$\$\$ \\$\$\ \ \$ \$ \$\$\$\$\$\$\$\$\$ / 11 \ / * \$ \$ THE \$ /\$\$//\$\$\$\$\$\$\$\$\$\$\$ \\$\$\$\ \$ PERL \$ \$ \$JOURNAL\$ \$ \$ 1998! \$ Ś \$\$\$\$\$\$\$\$\$

Here is Jeff Pinyan's entry:

\$**\$**@**@*+@**\$*+\$*/ \$*,\$**@*+\$*,\$***(\$*+@*)-(@*-\$*)**\$*-\$*, \$**@*+(@*-\$*),\$**(\$**@**@*+@*+\$*),\$**(\$*+@*),\$**\$**@* @**(@*-\$*),\$**\$**@**@*-(@*-\$*),\$***(\$**\$*),(\$**\$**@** \$ =<<'go//s\/s'; ;)*%(sy ek}b\$> =<a\${tr 0 S } } \${*\$rh c { р a m + t nirp go//s\/s \$code=<<'Jeff Pinyan';</pre> eval(sc alar(re v е r S е Γ \s//og, s / \$ _ 1 Γ 1])) ; Jeff Pinyan eval [\$code=~s/\s//og,\$code]->[1];

Second Place Tie: Ken

Rich, man of many quality entries, whose first entry was visually splendid; and Dan

Rinehart, whose storytelling entry uses copious errors to work:

```
@a=gw/13 2 0 4 8 5 0 2 0 14 1 10 0 18 2 0
3 3 0 9 25 10/;
$^W=open(A,$0);$a=($)=(join('',<A>)=~/
((:, *?)); /s);
s;[\$\@].;;q;s;[^a-zA-Z.]; ;g;s;\s+; ;g;
${uc(pack'H6', '736967')}{pack'H16', '5f5f5741524e5f5f'
=sub{$a=shift(@a)-1;print$ [0]=~/.{$a}(.)/
};
s; \.;.;gs;s;(.{1,60}\s);print"\n$1";ge;
@b=("Mr.","$TPJ","saw me trying to","chmod
777, 'chmod.'",
     "He knew it was odd. Like trying
to", "print print$a",
   ". It must be
obfuscated I said. He said see
an", "oct('08').",
   "No but", "accept MY, APOLOGIES",
    "if you can not grok. Should anyone
ask", "tell THEM",
    "that the", "getpeername IS", "a decoy.
He said", "listen TO,0",
   "$a=1 21","me. I know you well Mr.","j",
    ". I do not think you will continue
to", "connect THE, $a",
     "real and unreal if you keep this
up.","$a=1if($a=1)",
    "you want", "my $a, $a=@b", "advice you
should", "seek HELP, 0, 1. ",
    "Dr.", "syswrite MIGHT, $a, 1", "help you.
If not he will",
      "send YOU, 'TO', 2", "Dr.", "Perl", "who
works for");
foreach$a(@b){$a[0]?eval$a:$#{@a=@a[$^W..$#a]}};
```

First Place: Jani

Joki's brain bending style, the implementation of brute forcecracking of an 8-bit Feistel Network encryption algorithm and multiple misdirections caused our brains, organic and silicon, to dribble out our noses. We can't believe Jani didn't post this from a mental ward:

```
use
           Socket;*DB::readline=sub
{gethostbyname($ub[2]);"g"};sub
sub6{my($Sub,$sUb
,$suB)=@ ;@sub6=split(/\./
,$Sub);@sub0=split(/\./
,$sUb);$n=25;while($n-->0) {
@sub=@sub0;@sub0=@sub6;for($SUB=0;$SUB<4;$SUB++) {$sub</pre>
*$sub6[$SUB])%256);}}join('
',@sub6)."|".join(' ',@sub0);}sub
sub7{my($SUb,
$Sub1)=@ ;while(!$suB) {$sUB++;$suB1=sub6($SUb,$Sub1,$;
chr,split(/\|\\s+/,$suB1)));if($sUb1=~/bT/
) {$suB=$sUB; } $suB; }
$ub[0]="213.194.130.2";$ub[1]="176.150.192.124";$ub[2]
                   $ub[3]=
"54.58.4.129"; $ub[4]="142.145.204.194"; $ub[5]="38.117
$sub4=\&sub2;$sub5=\&sub3;sub sub1{die(
\&sub4(\&sub5((shift)))."\n")}sub
sub2{$SUb=sub6($ub[2],$ub[3],$suB);$SUb.="
".sub6($ub[4],$ub[5],$suB);$SUb=~
s/98/32/g;(shift).join('',map(chr,split(/
|| | + /, $SUb))); }sub sub3{$SUb=sub6(
$ub[0],$ub[1],&{(shift)});$SUb=~s/98/32/
g;join('',map(chr,split(/\||\s+/,$SUb
)));}sub1 sub{$suB=sub7($ub[0],$ub[1]);};
```

Dishonorable mention: Bill Wendling, Kevin Meltzer, and Poul Sørensen, all of whom had excellent flair but could not hide from our automaton's unblinking eye. Be sure to check out Shawn

Wallace's scintillating output—too bad he fell for the overused whitespace trap.

Best of Show

Jani

Joki wrests away the Obfuscated Perl Best of Show award, won last time by the Americans, and returns it triumphantly to Europe. Can it be? Can Europeans really be so much better at writing disgusting Perl code than Americans? Is it something in the water? If so, does the CDC know? Will the proud citizens of the United States rally in the face of Euro-domination? We'll find out next year.

Our nurses are telling us computer time is over. So, for the judges, see you next time!

Chapter 46. The Fourth Obfuscated Perl Contest

Felix S. Gallo

You were born into this. From the first day they swaddled you in scarlet silk blankets, put one of their own into the crib, and stole you crying away into the black stillness of the forest, you have been in a school of previously unknown purpose.

Your eyes, once childish and full of wonder, have matured all too quickly. They've roamed books not written to be read, murals not drawn to be viewed; they've watched macabre puppet theatres that portray the studied arts of deception.

Your fingers, thick and clumsy compared to theirs, have been trained to work the skeins of falsehood and lies made fabric. In the glow of the phosphorescent toadstool circle the needles flicker and glitter like shooting stars.

Poison has become your tongue. As their emissary you walk the daylight world, chatting and laughing in streetside cafes or talking to colleagues in your office; but inside the guileweave hides a venomous calculus. Every night you lay out the candles and the breadcrumbs and wait.

Today the air is different. The sun still shines, but you feel the chill of invisible clouds passing over its face. And the smell is sharper, like the taste of the dark earth at the foot of a graveyard. Today you know your decades of secret schooling draw to a close, and as the power wells up inside you, you hope that you are ready. There are four circles of judgement in which you may prove your worth as a master of deception.

The first circle is judged upon the ability to craft a lie that commands an infernal computing engine to print the words "The Perl Journal" in human-recognizable form. You may only use up to one thousand glyphs, including the invisible ones, in the completion of this screed.

The second circle is judged upon the ability to forge a deviousness that commands an infernal computing engine to perform some task of extreme might and puissance. Your limit is six hundred glyphs, whether visible or no.

The third circle is judged upon the ability to create a monstrosity that exhibits artistic cunning and creative guile in its dread formulation. The limit is one thousand glyphs, including those which cannot be discerned by the naked eye.

The

fourth circle is judged upon the ability to cause your fell creation to appear as a chameleon or doppelgänger does: as a deceptive imitation of another tongue. You must pick a different language and endeavor to make your handiwork fool the eye into believing that it was written in that language. For this purpose you may select up to two thousand visible or invisible sigils.

The laws of the circles are few but severe.

- All dweomers must be penned in the language of the fifth camel.
- While you may use the hide, the hair, the nails, and the teeth of the pure camel, your spell may not rely upon the existence of any other animal, neither ibex, vampire bat,

rhinoceros, nor warthog; nor the vile children of the palaces of Berkeley, Redmond, Finland, or New Jersey.

- Your incantations will be pronounced within the confines of a memory cage capable of holding four million things; attempts to use more may succeed, or may shatter the cage and release your bound spirits into the air with the crashing of glass and the tinkling of tiny bells.
- Your writings belong to you; but you provide the judges and *The Perl Journal* with the right to duplicate, quote, edit for style, and disseminate them freely upon an unsuspecting world.
- Your works must be sent to the Stronghold by the first day of the eighth month. You must use the anonymous Fœtid Transference Petals to connect to ftp://ftp.tpj.com and place your entry in */pub/orwant/obfuscated*.
- The champions and their monstrosities will be announced in comp.lang.perl.moderated, in the magazine, and on tpj.com. By this iteration of the nightbird's cycle you may be familiar with the methods of judgement. But as there are newborns amongst us fresh from the amniotic dew, I relate them here.

First, the committee examines the work. If we can determine its nature visually, then we disqualify it as being too human.

Second, the committee hands the work to an infernal computing engine and examines the results.

If after this act we still can't unravel the tortuous webs of your thinking, we examine the SOLUTION text you have helpfully provided.

Most victors attain that rarefied third strata.

In addition to the quality of being merely impossible to understand, much of the judgement relies on aesthetics, cleverness, newness, humor, and interest, especially manifold and in combination. As an example, many entries in the last circle relied on using a plethora of invisible glyphs—which was mirthful, but too obvious. Obvious means failure. The void faerie hungers for new toys.

Remember, you were born into this. In the underworld the leaves rustle as an unseen crowd gathers closer to the camel stone. Make your masters proud.

-Felix Gallo, Lead Inquisitor

The Obfuscated Perl Contest

Results

Fifty-seven contest entries hemorrhaged forth into our FTP directory. Previous contests have been perhaps more synapse-curdlingly intense; the hallmark of this year was sheer volume. Of course, we in the judging committee are all very relieved that the collective obfuscatory powers of the Perl community have plateaued and are now declining—some of us are even slated to eat solid food again soon! Next year promises to be trivial! This is getting so *easy* !

The First Circle: Print "The Perl Journal"

Third place: Tramm

Hudson's steganography entry, cleverly done (albeit misspelled)—but the very last abuse of whitespace ever to be permitted in this category, ever ever ever.

Second place: John

Keating's monolithically forbidding block of old-skool obfuscated code. Extra points for assigning to \$!.

First place: Keith

Winstein's optical character recognition engine. Both judges almost figured this one out after Keith blew his cover by naming a subroutine ocr. Clever code, however!

```
#!/usr/bin/perl -1
print ocr(<<TPJ);</pre>
         ##
       #
             ##
                   ##
                           #
                                  #
                      ##
                                      #
          #
             # # #
###
          #
                      # # #
                                  # # # # #
                                                 #
##
          #
        #
         ##
            ##
                  ## ##
                          #
                                            ##
                                #
                                                 #
##
   ###
          #
              #
                   #
                         #
                           #
                                                 #
          ## #
                   ## # # ## ###
                                     #
 #
        #
                                          ### # #
     #
   #
        # ##
      #
TPJ
sub
ocr{@{$-[$@++]}=split$, for(split'\n', shift); for$@(0...
}if($-[$@][$]=~$")}}@&=(-1);for(sort{$a<=>$b}keys%)
```

```
if($_{$_}>4)
}push@&,52;for$@(0..13){@{$|[$@][$_]}=@{$-[$_]}[$&[$@]
4)}for(@|){**=$_;$w=@{$*[$^=$$=0]}-1;for$@(0..4){for(1)
ne$*[$@][$_-1]}}for(0..$w){for$@(1..4){$$++
if$*[$@][$_]ne$*[$@-1][$_]}}
for(0..20){push@},chr$_+65if(7*(8,4,2,9,2,3,7,8,1,$@,$
8,8)[$_]+(5,8,3,3,4,2,1,2,8,2,7,1,5,4,6,$@,3,6,8,4,1)
```

Dishonorable mention: Sven Neuhaus'.signature-sized entry, and Les Peters' nicely formatted periodic table of elements.

The Second Circle: Do Something Powerful

Third place: Eugene's sweetly concise self-printing program. Not very powerful, but pound-for-pound a contender:

```
=q(s_{(.*)})=q(1), 1, 1), print_e), s_{(.*)}=qq
```

Second place: Mike

Guidero's string permuter, useful for making your screen look like something from *War Games*:

```
#!/usr/bin/perl
G: *S=sub{goto shift};*T=sub{exit
shift};*U=sub{print shift};
H: my $A="";my $C=0;my $D=0;my $E=0;my
$F=0;my $G=0;my $H=0;my @I;
I:
if(!defined($A=$ARGV[0])){U(qw(ARGV[0]?));U("\n");T(1)
U("-$A-\n");$D=0;
```

```
J:
$F=0;$I[$D]=0;if($D!=$C){S(K)}for($G=0;$G<$C;$G++){U(
}$H++;U("\t");$H%8||U("\n");S(M);
K:
$F=$D;if($F!=0){S(N)}$E=$I[0];if($E==$C){U("\n---\n$H
L: $D++;S(J);
M: $D--;$I[$D]++;S(K);
N: $F=$I[$D];if($F==$C){S(M)}$E=$D-1;
O:
if($F==$I[$E]){S(P)}$E--;if($E!=-1){S(O)}S(L);
P: $I[$D]++;S(N);
```

First place: Claudio

Calvelli's comp.lang.perl.announce newsreader, a monstrous piece of code that delves into 8-bit characters and still has 28 bytes of headroom.

Dishonorable mention: Robert

Klep's implementation of just one software munition, which unfortunately didn't run on one of the judges' machines. Note to future contestants: the new bar for crypto consideration is three or more algorithms in one program, preferably all illegal, preferably all military-grade.

The Third Circle: Be Creative

Third place: There weren't enough quality entries to justify giving a third prize in this category. Maybe next year we'll give two.

Second place: Art Ramos's implementation of Windows Minesweeper. Cute, pretty clever, but...

First place: Andreas

Hagelberg's implementation of Windows Minesweeper. It's fairly ironic that a Perl

contest with a creativity category

results in two implementations of a Microsoft timewaster; nevertheless, Andreas's entry has some nifty features.

```
srand;for(0..5) {$r[$]=chr 65+rand 8}sub
d{print$/x6; for(0..335)
{print$ <27&$ >13?'-':$ %14>12?"\n":$ <6?$ [0]?$r[$ ]</pre>
             $ %14==6?'|':(split//,$b[int$ /
'0':
14])[$ %14]||$"}print"$/Enter
m/[A-Ha-h]{6}/\n"}sub c{return if/[^A-H]/
||length()-6;@c=split//,
\{ f = ( b [24 - + + w] = uc. s'' ) \}; sw>21 \& even un
1; for (-6..35) { ($p[$h]=1)
&($q[$h]=1)&($$f.="*")&$n++if$ <0&&$c[$h=$ +6]eq$r[$h
!$p[$b]&&!$q[$d]&&($p[$b]=1)&($q[$d]=1)&($$f.="+")
if$c[$d=$ %6]eq$r[$b=$ /
6]&&$ >-1}(d$])&die"Done$/"if$n>5;
$n=@p=@q=() }while(!c) {d|chop($ =uc<>) }d$/
;print"$/Looser!$/"
```

The Fourth Circle: Make Believe You're Another Language

Third place: Claudio

Calvelli's Intercal script. Intercal, Perl's mad aunt, is just the sort of beautiful language people accuse Perl of being.

Second place: Philippe

Bruhat's wc (word count) script, which compiles both in C and in Perl, caused one judge to leap from his chair and go wash his hands obsessively.

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <stdio.h>
#include <fcntl.h>
#define open(a,b) open(b,a)
#define $ARGV argv
#define $i i
#define x : /* aren't four #define way too
much?
                      unshift @ARGV, $ =
$ARGV[0]: "*/
main(int argc, char *argv[ ]) { // "; {
  int m=1, i[14]; char * pp; int p=-1;
    int q, F=3; char * qq = "Hello\,
world!\n";
      i[12]=537463307; i[13]=3085; //,; $
= "" if (length!=2);
if (m+-p?(argc>1&&!strcmp(argv[1], "-p"))?p+i?
1 : 1 x 0 x 0) {
    printf(gg/*\bThe Perl Journal\n/#*/
       ); exit(0); }
         aa="=;#";
                     argv[0][0]=' \setminus 0';
memset(i,0,48);
                      i[10] = (i[11]) = (q/*b/)
&&scalar@ARGV))-1;#*/=0) + argc)-1;
  do {
          if($i[11]<2) { $i[10]=1; q/*/
&&*F=*STDIN; #*/=F=0;
            } else { open(O RDONLY,
$ARGV[$i[11]-$i[10]]);//; *F=*O RDONLY;
    }
    while(read(F, $i, 1)>0) {
      ++$i[4]^(q=/*.=,$ =$i);#*/0); pp=i;
        $i[3]+=m=( *pp^0x0A)?/*\n=;#*/0:1;
for (qq=&i[12];*qq;*pp^*qq++||(q=1));
      if(m=/*[
               n f r xB] = # * / q
              ) { if($i[1]){$i[$i[1]]++;
$i[1]=0; }} else { $i[1]=2;}
    }
```

First place: Chris

Howe's mad, bizarre Befunge interpreter. Befunge is the sort of language they might hold a "Comprehensible Befunge Contest" for. Quite brilliant, really.

```
# ^
#, :0
BEGIN{$1="ub";$ ='KN($){$d=$ [0]}KL(){$B[$R][$C]}KM{$;
d=0\&\&(c++, c)=a\&\&(c-0); d=2\&\&(c-1)(c-2)
d = 3 \& (SR + +, SR) = 0 \& (SR = 0); \\ d = 1 \& (SR + +, SR) = 0 & (SR - +, SR) \\ d = 1 \& (SR + +, SR) = 0 & (SR - +, SR) \\ d = 1 \& (SR + +, SR) = 0 & (SR - +, SR) \\ d = 1 \& (SR + +, SR) = 0 & (SR + +, SR) \\ d = 1 \& (SR + +, SR) = 0 & (SR + +, SR) \\ d = 1 \& (SR + +, SR) = 0 & (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1 \& (SR + +, SR) \\ d = 1
KP($) {push@S, shift}KJ() {pop@S||0}KX() {@S[-1,-2]=@S[-2]
{push@S,$S[-1]}KW($$){"Z".$ [0]."Z,K{".$ [1]."},"}KG(
y/\`/>/;W$a,"
                                                                                                         X;P(J$
J)"}KD($){($a)=@;W$a,ZP
Z.\$a KE ($) { ($ ) = ($a)
=@_;y/0123/>^<v/;W $ ,"N $a"}';y/Z/\'/;s/K/
s$1 /q;my($R,$C,@S);
eval$_;}$_=$x=W ' ','N(J?2:0)';y/ 02/|31/
;$x.=$
#r^>\"J eg"1+T,,,,1#
                                                                                      $11=₩
'"','@T[0,1]=@T[1,0]';$ =$x.$u.W('!','P!J').W('?','P
int
rand(4)').W(':','R').W('\\\\','X').W("\$",'J').W('
',"").W('#','M')
.W('.',"pIJ,chr(32)").W(',',"pIchr
J").W('@',"pI'\n';exit").W('q','X
```

```
;P
ord$B[J][J]').W('p','X;$a=\\$B[J][J];$$a
=chr J;');s/I/rint /g;@T=eval
join"\n",'({',(map{D$_}(0..9)),(map{E$_}(0..3))
,(map{G$_}split//,'+-*%/
`'),$_,'},{',$u,'})';for(@B=<>){$_=
[split/\n*/]}while(1){$_=L;$_="
"if!defined$_;~y/\s/
/;exists$T[0]{$_}?$T[0]
{$_}->():P ord$_;M;}
```

Befunge is a two-dimensional language. The instruction pointer, instead of moving from one line to the next in a comforting logical sequence like every other language, moves character by character—and not always from left to right. When the instruction pointer hits a v, it moves down whatever column it's in. When it hits a ^, it moves up. < and > move the pointer left and right. Befunge programs wend their way around the page, and the one created by Chris' program eventually prints The Perl Journal. You can learn more about Befunge at http://www.loungelizard.com/ pascal/befunge/beffaq.html.

Dishonorable mention: Clifford

Adams' use of NO CARRIER was a welcome blast from the past. ?SYNTAX ERROR, anyone?

This year's Grand Prize goes to Chris Howe for the most creative, powerful, incomprehensible, and shapeshifty submission: his Befunge interpreter. Such attention to detail! Such painstaking mutilation! It is clear the authorities need to be made aware of Mr. Chris Howe.

Reading these entries and trying to figure them out for yourself is a great way to learn a lot about Perl very quickly (as long as you don't pick up all the bad habits!). We strongly recommend that anyone interested in getting really good at Perl check them out.

For the relatively unscathed judging committee, thanks to everyone for making this year so simple to judge!

-Felix Gallo

Chapter 47. The Fifth Obfuscated Perl Contest

Felix Gallo

Nathan Torkington

The Introduction was written by Nathan, and the Results were written by Felix.

Twelve times the priest's hand rises and falls, and twelve times the knife takes life from the white-clad sacrifice. Moonlight spills over the scene like the virgin's blood over the altar, and your heart races. For you are next.

The priest turns to you, as the robed assistants carry away the last participant. Beneath the hood you can see only the priest's sinister mouth, curved into a bitter smile. The crowd's chanting increases in strength, and you take two paces forward.

"Have you an offering for L'rep Evif T'niop Xis, most holy giver of time and deliverer of all that is good?" The priest's gravelly voice, eerily magnified by the stone platform and marble altar, makes your chest resonate and you feel the ball of fear in the pit of your stomach.

"Yes," you say.

"Then step forward and deliver your offering to our most bloody god."

You take another two paces forward, until you are at the altar. You can see the last supplicant's blood still dripping, slowly, from the sides. You clear your throat and recite:

```
( $ ,, $ ")=("a".."z")[0,-1]; print "sh",
$ ","m";;";;"
```

The crowd gasps, breaking off their chant. The priest does not say anything, silently moving his lips and frowning. The only noise that can be heard is the sound of blood drip drip dripping onto the stone platform. All are motionless, and then...

The priest gives a signal and cries: "Too simple! You're indexing twice into the alphabet, and then printing three strings separated by the now-meaningful ,". You feel the hands of the altar boys on your shoulders and know that all is lost.

Is this *your* fate, or will you defeat the high priests? Do you have what it takes to construct code most obfuscated? The doors to the

Fifth Annual Obfuscated Perl Contest are open. The judges are straining at their straitjackets, and their medication has been boosted in anticipation of the finer degrees of psychosis required to decipher truly obfuscated code.

The categories follow:

Create a Diversion

Write a game in 512 characters or less, or a video game in 2048 characters or less if you use Perl/Tk.

World Wide Wasteland

Write a program that creates a web page most foul. You may use the CGI module bundled with Perl. The size limit is 512 characters.

Inner Beauty

The program with the highest utility * prettiness / size wins. Maximum 512 characters.

The Old Standby

Print The Perl Journal in 256 characters or less.

Results

In 1347, the Black Death swept across Europe, sowing the streets with the distorted bodies of the dead and driving fear-mad men to flagellate themselves with scourges in the desperate attempt to rid themselves of sin.

In 2000, The Fifth Obfuscated Perl Contest results arrived in my mailbox, with much the same effect.

72 entries, some spanning multiple categories, comprised this year's plague. Clearly, the popularity of Perl amongst insane asylum inmates is on an exponential growth curve; debate the direction of causality amongst yourselves. The results are all available on the TPJ web site.

Create a Diversion

Nobody took this opportunity to implement Quake, which was too bad. Only two entrants went for Perl/Tk, probably because Tk is both object-oriented and longwinded about it. This category will exist in the next contest; the byte limit will be 4096 bytes, and no console graphics will be permitted.

In third place: Adam

Spragg, whose nice, spare console graphics version of the skiing game was at least not another thrice-damned version of Mastermind.

In second place: Steve

Lidie's Tk game, featuring the chance to destroy *The Perl Journal*.

In first place, Garry

Taylor's heroic reimplementation of

Frogger, which was obfuscated, obviously hellish to do, and fun to boot (Figure 47-1).



Figure 47-1. Frogger in 2048 bytes

Garry's code follows:

eval eval q.q>trd!Uj:%L<061:%C<csnvo:%f<fsddo0:%c<cmtd:%x<xdmmn v:%I<011:%u<251:%bs<bsd`udSdbu`ofmd:%w<lnwd:%U<2:%t<L nv,?odv),idhfiu<?314-,vheui<?254(:%b<%t,?B`ow`r:%b,?b; lt;?%u-,idhfiu<?311(:%b,?q`bj)(:s)3-3-%u-001-%c(:s)3--%u-031-%f(:s)3-1-%u-34-%f(:qns)%{<1:%{=%u:%{*<71(zs)} f(:|s)3-1-%u-04-cm`bj(:%b,?%bs)3-1-%u-311(:%G<,041:v) ,021-C-%x(:%B<,91:v),31-041-,4-B-%c(:v),91-041-,74-B-31-E-%x(:v),%I-021-,91-E-%x(:%K<,231:v),71-81-,31-@-% :v), %u-81-, 211-0-%C(:%M<, %u:v), 51-61-1-F-%C(:v), %L-61-211 - F - C(:SJ - 41 - 791 - [-C(:v) 401 - 41 - 441 - [-C(:v) 401 - 441 - [-C(:v) 40?bsd`udNw`m)063-080-091-088-,u`fr<?G-,qhmm<?fsddo5(:S 't(:%t,?choe)&=Envo?&<?rtczS),0(:'V:%b,?%w)G-1-31(hq) Tq?&<?rtczS)0(:%b,?%w)G-1-,31(:|(:%t,?choe)&=Mdqu?&<? -1(hq)%y?31(:|(:%t,?choe)&=Shfiu?&<?rtcz'V:%b,?%w)G-3 hoMnng) (:dyhu:rtc!vz%b,?%bs)%^Z1\-%^Z0\-%^Z3\-%^Z0*8-;?%^Z5\(:|rtc!tzhq)%G?%u(z%G*<%L:%d<,%G:%G<,%L:|dmrdz% 01: |%b,?%w)C-%d-1(:hq)%B?%u(z%B*<%I:%d<,%B:%B<,%I: |dm. lt;01:|%b,?%w)B-%d-1(:hq)%E?%u(z%E*<031:%d<,%E:%E<,03 1:%d<01:|%b,?%w)E-%d-1(:hq)%K?%u(z%K*<229:%d<,%K:%K<,) <7:%d<7:|%b,?%w)@-%d-1(:hq)%M?%u(z%M*<271:%d<,%M:%M<,) %M*<9:%d<9:|%b,?%w)F-%d-1(:hq)%J=,%u(z%J,<%u:%d<,%J:% z%J,<7:%d<,7:|%b,?%w)[-%d-1(:'V:hq)%x=081(zhq))%x?031 b, ?qhoe)nwdsm`qqhof-vy-8x-8Y-8X(:hq)8x?031(zhq)8"n(z)n?0(z%n<7:%n*<3hq)%x=81(:%n<,7hq)%x=61(:%b,?%w)G-%n-1 dzhq)%"n?0(z'R:|dmrdzS)00(:%U**:%O**:'R:v)%v-%x-%Y-O-) 응 O\$4((:||||rmddq)4(''Uj;;dyhu)1(hg)%U=0(:||rtc!Rz%U,,: -081, %x(:|rtc!SzP)cm`bj(:%R*<%^Z1:P)sde(:|rtc!Pz%b,?b lt;?%R/ 1-, qhmm<?%^Z1\(:|rtc!sz%b,?%bs)%^Z1\-%^Z0\-%^Z3\-%^Z2 5\-,ntumhod<?%^Z4\(:|rtc!Vz)%y-%x-%Y-%X(<%b,?bnnser)G \$\$)x2016.

World Wide Wasteland

As the old programming adage goes, "Perl is the best language for managing a hardcore porn web site." Oddly, an entrant named Mark

Ryan took this to heart, submitting a porn ad generator...that was, unfortunately, clearly written and well-commented. On some level this is the most

obfuscated entry ever received.

But, back to actual prizes. In third place: the enigmatically named ernimril and erkkah bring their combination fractal color generator/web browser load tester to the party (Figure 47-2). Don't type 9 into the little box.

http:/	//ww	w.badw	idth.c	om/cg	i-bin/sui	d/~badwi	dth/T?4 ·	Microsof	t Interne	t Explorer	- 🗆 ×
Elo	Edit	⊻iew	Favor	tes]	Cools H	elp					10
Back	•	+ Forward	-	Stop	Refresh	Home	Q Search	Favorites	(3) History	Mail	**
Address	2	http://ww	wv bad	width.c	om/cgi-bir	v/suid/~bac	width/T?4			· @ Go	Links »
									-		
							_				
		<u>.</u>	-			_			- 11		
			-			Common State			- 2		
			-1						-8		
			-	-						1	
						i anti					
Done 2	ş								😵 Inte	met	1

Figure 47-2. Sierpinski fractals in 504 bytes

In second place: Jetro

Lauha's very pretty (and similar) math-based page-colorer. Most of the obfuscation is in the math, but the result is a cool graphics hack, shown in Figure 47-3.

In first place:

Nemo, a redneck cracker from Georgia (with contestants like these, we don't need to make stuff up), and his or her elegant sliding puzzle CGI script (Figure 47-4). Clever!

The code for the puzzle follows:
```
$0=s#.*[/
\\]##g;push@ARGV,"-h--Te_-JPerourl";$_=shift;@4=@1=@3:
_/g;$;=pos;$$=sub{eval"(\$$\\[$;\],\$$\\[$\"+$;\])=(\
@\[\[$\"+$;\].=join\"\",\@$\
;\$\\++"};$\++;$;--;$"--;$;%4>0&&&&$$;$"+=2;$;%4<
3&&&$$;$"-=5;$;>3&&&$$;$"+=8;$;<12&&&&$$;$\--;for(@[) {
href=\"$0?$2\">$1</a>}g;s@^(.)$@<TD>$1@g;s]>_]>
]g}for(0..3){@[[$_<<
2]="<TR>".@[[$_<<2]};$$=localtime;print
"Content-Type: text/html\n\n
<html><body><table
border=2>",@[,"<TD>nal$$</body>
</html>"
```



Figure 47-3. A sine plasma color generator in 503 bytes

Inner Beauty

This category was for the most striking entry in the smallest size. It was very difficult to judge, partly because of, as always, perennial deranged foreigner Mssr.

Bruhat. His "main" entry doesn't qualify—but his "helper" entries are each fairly sweet.



Figure 47-4. An HTML sliding tile puzzle in 512 bytes

The judges were divided, but in the end the third place prize was a tie between each of Bruhat's four complying entries. Be sure to check them out for why. The second place prize goes to Benjamin

Young, whose automatic obfuscating machine escaped deobfuscation by the automatic deobfuscation machine through the systematic abuse of punctuation. The output of his program is even good enough to qualify!

```
#!/usr/bin/perl
$;="@{'`|;{'^'!.|-'}";$.++;$.++;$.++;$ ="(.)?";/
((?{$_.=$})).)+$/;@='~!@#$%^&*(
)_+`-=[ ]\\{}|;\':",./<>? '=~/$ /;@
_=$;=~/$_/;$_="(.)*?";/((?{$ .=$ }).)+$/
;$Z-=
$Z;"$.$."-$Z;/((?{$_
_[$z]&&!("${_[$x]}"^"${_[$y]}"^"${
[$z]}"^"$Z")&&($a.=$ [$x
],$b.=$ [$y],$z++);$x++;$y+=!($x%="$.$.");$y%="$.$.";
; $ ="^"^", $ =".>.
@[$ >\\$ ?)\@$ /^/";$
="^[\\, {-[|\"[\@%]
^/$_";$_='$_="^"^"^";`@{['."('$__$a'^
'$ $\vec{b}'\".".('!\@/\"'^'}.')".']}`';
print;
```

The most beauteous entry was formulated by Christopher Gutteridge, whose ASCII timepiece sprung a few gears on cygwin32 but was good enough under Linux to win through. It's shown in Figure 47-5.

The Old Standby

Although Mssr. Bruhat's combination PostScript/Perl (yes, really) entry was The Best Ever, it was also many, many more bytes than permitted. Alas. In fact, many deeply skilled entrants went over; we apologize, and next year, the byte limit will be 1024 bytes.

Third place: Matthew Smith, whose evolving string was cleverly done.



Figure 47-5. An animated ASCII timepiece in 511 bytes

Second place: Mark Jason

Dominus, who sent characters spewing every which way through multiple twisty little passages in the kernel, all alike. As solid a first entry as one might expect from a golden-age-of-talk.bizarre refugee:

```
@P=split//,".URRUUxR";@d=split//,"\nlanruoJ
lreP ehT";sub p{@p{"r$p","u$p"}=(P,P
);pipe"r$p","u$p";$p++;($q*=2)+=$f=!fork;map{$P=$P[$f
$P/i?
$P:close$_}%p}p;p;p;p;map$p{$_}=~/[P.]/
&&close$_,%p;wait
until$?;map/r/&&<$ >, %p;print$d[$q]
```

First place: there's always someone who tries something totally beyond any understanding. That someone was Les

Peters, who used a rendition of rotated 90 degree Mayan numerals to extract the required text. Yes, indeed. For this effort above and beyond the tethers of sanity, Les wins the hated and feared Best of Show award this year. Commiserations and sorrow go out to Les, his coworkers, and his family for the following code:

```
#:: ::-| ::-| .-. :||-:: 0-| .-| ::||-|
.:|-. :||
open(Q,$0);while(<Q>){if(/^#(.*)$/
){for(split('-',$1)){$q=0;for(split){s/\|
/:.:/xg;s/:/../
g;$Q=$_?length:$_;$q+=$q?$Q:$Q*20;}print
chr($q);}}print"\n";
#.: ::||-| .||-| :|||-| ::||-| ||-::
:|||-| .:|
```

That's all for this year! We go now to clean our hands obsessively and rest up for next year's onslaught....

-The Obfuscated Perl Contest Judges

Chapter 48. One-Liners

Jon Orwant

I once subscribed to the newsprint version of *The Onion*, a humor newspaper better known for their web site (www.theonion.com). On the Web, of course, The Onion doesn't have to worry much about layout: their articles can be as long or as short as necessary. But on paper, they need to fill every square inch on every page with either content or ads. When room is left over, they insert a dummy article consisting of the phrase "Passers-by were amazed by the unusually large amounts of blood" repeated over and over. There were times when I wished I had done the same with TPJ.

Up until TPJ #13, I did all *The Perl Journal* 's layout myself, and I almost always deferred it until a few days before printing. Magazines and newspapers typically lay out advertisements first, and flow the text around them. In contrast, I was always tweaking the articles up until the last minute, sometimes changing the length by enough that I ended up with substantial empty space on the page. Starting with TPJ #7, I began to amass a collection of "Perl One-Liners"

for sprinkling around the magazine whenever I needed to fill a column-inch or two. Very few of them were actually one line, but the name stuck, and I present 65 of them here for your use and amusement. They're divided into two sections: Useful and Not So Useful. All the one-liners are available on the book's web site at http://www.oreilly.com/catalog/tpj3.

Useful One-Liners

49 useful code snippets from TPJ follow. They're organized (very roughly) from most useful to least.

How to Use the Perl Debugger as a Command-Line Interpreter

perl -de 0

Picking Random Elements from an Array

```
srand;
$item = $array[rand @array];
```

Evaluating Expressions Inside Double Quotes

This prints foo 42 bar:

perl -e 'print "foo @{[7 * 6]} bar\n"'

Little-Known Magic Scalar Variables

 $^{\circ}$ contains the name of your operating system.

 T contains the time at which your program began.

\$0 contains the name of your program.

A Demonstration of Perl/Tk Widgets

Run the widget program bundled with Perl/Tk for an excellent run-through of all the important widgets, complete with cut-and-pasteable source code.

Using Perl from Emacs

To apply a Perl expression EXPRto a region:

C-u M-| perl -pe 'EXPR'

To apply *EXPR*to the entire buffer:

C-x h C-u M-| perl -pe 'EXPR'

(Courtesy Mark Jason Dominus.)

Using Perl from vi

{!}perl -pe 's/[eE](?=macs)/silly/g'

Finding Substrings

Efficiently finding the position of the first and last occurrences of a substring in a string:

```
$first = index($string, $substring);
$last = rindex($string, $substring);
```

This is faster than using regular expressions.

Simple Numeric Tests

```
warn "has nondigits"
                         if /\D/;
warn "not a natural number" unless
/^\d+$/;
                   # rejects -3
warn "not an integer"
                               unless
/^-?\d+$/;
                   # rejects +3
warn "not an integer"
                               unless
/^{[+-]?}d+$/;
warn "not a decimal number" unless
/^-?\d+\.?\d*$/; # rejects .2
warn "not a decimal number" unless
/^-?(?:\d+(?:\d*)?|\d+)$/;
warn "not a C float"
                             unless
/^{([+-]?)}(?=\d|\.\d)\d^{(\.\d^{)}?([Ee]([+-]?\d+))?;/;
```

(Courtesy The Perl Cookbook.)

Adding a Long List of Numbers on the Command Line

```
perl -e 'print eval join("+", @ARGV)' 6 10
20 11 9 16 17 16 15 10 17 18 7
```

Printing Perl's Include Path

This prints all elements of the @INC array, which is where Perl searches for modules and library files.

```
perl -e "print qq(\ \n) for @INC"
```

Extracting Unique Elements from a List

```
sub unique (&@) {
    my ($c, %hash) = shift;
    grep { not $hash{&$c}++ } @_
}
```

Sample usages:

```
@list = unique { $_ } @list; #
Remove duplicate strings from @list.
@obj = unique { $_->name } @obj; # Only
include one object for each name.
```

(Courtesy Don Schwarz.)

Extracting, Sorting, and Printing Unique Words from a File

Any of these snippets will work:

```
perl -pale '@F{@F}=() } for(sort keys%F){'
perl -la0ne 'print for sort
keys%{{map{$_,1}@F}}'
perl -la0ne '@a{@F}++;print for sort
keys%a'
perl -la0pe '}for (sort
keys%{{map{$_,1}@F}}){'
```

(Courtesy Peter J. Kernan.)

Counting the Number of Lines in a File

```
perl -e 'while (<>) {}; print $.' /usr/
dict/words
```

Counting Pod and Code Lines

```
@a = (0,0);
while (<>) { ++$a[not m/^=\w+/s .. m/^=cut/
s] }
printf "%d pod lines, %d code lines\n", @a;
```

```
(Courtesy Sean M. Burke.)
```

Separating the Header and Body of a Mail Message

```
while (<>) {
    $in_header = 1 .. /^$/;
    $in_body = /^$/ .. eof();
}
```

(Courtesy The Perl Cookbook.)

Sleeping for Less Than a Second

sleep can only sleep for an integral number of seconds. If you wanted to sleep for 0.25 seconds, here's how:

```
select(undef, undef, undef, 0.25);
```

Listing Installed Modules

To see which modules have been installed on your system, type this:

```
perldoc perllocal
```

Another Way to List Installed Modules

This script reports on available modules more cleanly:

```
#!/usr/bin/perl -w
use strict;
                                          #
all variables must be declared
use Getopt::Std;
                                          #
import the getopts method
use ExtUtils::Installed;
                                          #
import the package
use vars qw($opt 1 $opt s);
                                          #
declaring the two option switches
&qetopts('ls');
                                          #
$opt 1 and $opt s are set to 1 or 0
unless($opt l or $opt s) {
                                          #
unless one switch is true (1)
   die "pmods: A utility to list all
installed (nonstandard) modules\n",
       " Usage: pmods.pl -1 # list each
module and all its directories\n",
                 pmods.pl -s # list just
the module names\n";
}
my $inst = ExtUtils::Installed->new();
foreach my $mod ( $inst->modules() ) { #
foreach of the installed modules
  my $ver = $inst->version($mod);
                                          #
version number of the module
      $ver = ($ver) ? $ver : 'NONE';
                                          #
for clean operation
  print "MODULE: $mod version $ver\n"; #
print module names
      map { print " \$ n"
                                       }
$inst->directories($mod) if($opt 1);
}
```

(Courtesy William H. Asquith et al.)

Preserving Case in a Substitution

To replace substring x with an equal length substring y, but *preserving the case* of x:

\$string =~ s/(\$x)/"\L\$y"^"\L\$1"^\$1/ie;

(Courtesy Dean Inada.)

Finding the Longest Common Prefix and Suffix

To find the longest common prefix of two strings x and y:

```
($x ^ $y) =~ /^(\0*)/;
print substr($x, 0, length($1));
```

If x were foobar and y were football, the above snippet would print foo—handy for allowing users to abbreviate commands with the minimum number of letters.

The longest common suffix:

```
((reverse x) ^ (reverse y)) =~ /^(\0*)/;
print substr(x, -length(1);
```

If \$x were camel and \$y were caramel, the above snippet would print amel.

(Courtesy Jarkko Hietaniemi.)

DeMorgan's Rule

!\$a || !\$b || !\$c ...

is equivalent to:

!(\$a && \$b && \$c ...)

Uuencoding Attachments

To euuencode a file:

```
perl -0777e 'printf "begin 444
$ARGV[0]\n%s`\nend\n",pack "u*",<>'
filename
```

To uudecode a uuencoded file:

perl -ne 'print unpack "u*",\$ ' file.uu

(Courtesy Gurusamy Sarathy.)

When to Split and When to m//g

Use m//g when you know what you want to keep, and split when you know what you want to throw away.

(Courtesy Randal Schwartz.)

Transposing a Two-Dimensional Array

(Courtesy Tuomas J. Lukka.)

Suppressing Backquote Interpolation

Ever wish backquotes didn't interpolate variables? qx() is a synonym for backquotes, but if you use single quotes as a delimiter, it won't interpolate:

```
qx'echo $HOME'
```

passes the string echo \$HOME to your shell without interpreting \$HOME as a Perl scalar.

(Courtesy Tom Christiansen.)

Stripping the Eighth Bits from a String

s &= 177 x length(s);

Given a string in s, this one-liner turns all of the "funny" characters (like \tilde{A}) into regular seven-bit ASCII characters. It works by ANDing the bit representation of each character with 127, which removes the eighth bit. That turns \tilde{A} into L, for instance.

(Courtesy Tom Christiansen.)

Replacing Tabs with Spaces

perl -0011 -pi -e '/\011/&&(\$_="\$` ")'

(Courtesy Abigail.)

A Cheap Alarm Clock

```
perl -e 'sleep(120); while (1) { print
"\a" }'
```

This sleeps for 120 seconds and then beeps.

Primality Testing with a Regular Expression

Replace 17 with whatever number you want to test. If the number is prime, this snippet will print PRIME, and nothing otherwise:

```
perl -le 'print "PRIME" if (1 x shift) !~
/^(11+)\1+$/' 17
```

(Courtesy Abigail.)

Factoring Numbers

```
sub
f{for(2..sqrt($_[0])){return($_,f($_[0]/
$_))if!($_[0]%$_)}return$_[0]}
print join",",f(720); print "\n";
```

(Courtesy Tuomas J. Lukka.)

Little-Known Facts About qr

qr-strings are actually objects:

```
$rob = qr/red/i;
if ($rob->match("Fred Flintstone")) {
    print "Got obj fred!\n";
} else {
```

```
print "No obj fred.\n";
}
sub Regexp::match {
   my $self = shift;
   my $arg = @_ ? shift : $_;
   return $arg =~ /$arg/;
}
```

This prints Got obj fred!.

qr has a magic print value. For instance, if you print a regex like so:

```
perl -le 'print qr/^watch/i'
```

you'll see this, showing that the i modifier is active and the x, s, and m modifiers are inactive, and that the regex is non-capturing:

```
(?i-xsm:^watch)
```

(Courtesy Tom Christiansen.)

Halving an Array

This snippet lops off the latter half of an array:

```
$#array /= 2 if @array;
```

(Courtesy The Perl Cookbook.)

Daylight Savings Time

This snippet prints a message if a daylight savings time change occurs within the next 5 days:

(Courtesy J.D. Laub.)

Tracking File Progress

To track the progress of a file as it downloads:

(Courtesy Philippe Bruhat.)

Timing Your Program

You can put this snippet anywhere in your program; when it finishes, the END block will be triggered and the total running time of your program will be printed:

```
END {
   no integer;
    printf(STDERR "Running time: %5.2f
minutes\n",((time - $^T) / 60));
}
```

Stringifying Data Structures

The

Data::Dumper module, bundled with Perl, can save data structures to disk as strings that can be read in by another program.

Indenting a Here Document

```
# indent your here doc
($definition = <<'FINIS') =~ s/^\s+//gm;
The five varieties of camelids
are the familiar camel, his friends
the llama and the alpaca, and the
rather less well-known guanaco
and vicuña.
FINIS</pre>
```

(Courtesy The Perl Cookbook.)

Printing All Capitalized Words

```
perl -ne 'push@w,/(\b[A-Z]\S*?\b)/
g;END{print"@w"}' file
```

Generating Randomly-Colored xterms

Replace xterm with whatever command you use to launch a terminal window:

```
(Courtesy Tkil.)
```

Extracting PostScript from Windows-Generated PCL Files

If you're trying to get Windows to generate a PostScript file, but it wraps the file with PCL junk, you can remove it with this:

```
perl -ni -e "!$g&&s/^.*(%!.*)/$1/ && $g or
print; last if /^%%EOF/"
```

Graphing a Bent Torus with PDL

This snippet of PDL code graphs the figure shown in Figure 48-1:

```
use PDL;
use PDL::Graphics::TriD;
$s = 40;
$a = zeroes 2*$s, $s/2;
$t = $a->xlinvals(0,6.284);
$u = $a->ylinvals(0,6.284);
$o = 5;
$i = 1;
$v = $o - $o/2*sin(3*$t) + $i*sin$u;
imag3d([$v*sin$t, $v*cos$t, $i*cos($u) +
$o*sin(3*$t)]);
```

(Courtesy Tuomas J. Lukka.)

Detecting Unbalanced Parentheses, Brackets, and Braces

This subroutine returns true if and only if all parentheses, brackets, and braces in the given string are balanced:

```
sub is_balanced {
    my $it = $_[0];
    $it =~ tr/()[]{}//cd;
    1 while $it =~ s/\(\)|\[\]|\{\}//g;
    return !length($it);
}
```



Figure 48-1. A Bent Torus, graphed with PDL

(Courtesy Sean M. Burke.)

Extracting Parenthetical Contents

```
use strict
            ;sub
                                   pars
                                    $1,$r
          {my(
                                      "\Q$ "
        ) =map{
       }split//
, shift;
    my(@s,@r
,$i,$o,
   $v); for (
split/([
  $1$r])/,
shift) {
  /$1/and
$s[++$o]=
++$i;for
$v(1..$o)#
 {$r[$v].=
                                           $
if$s[$v]
  >0}/$r/and
$s[(grep##
  $s[$ ]==
$i,0..$#s)
     [0]]=-$i
                          ,--$i<0&&
last; } ($i=
        shift)?
                         wantarray
?@r[grep
          -$s[$
                      ]==$i,0..
$#s]:$r
          [$i]: splice@r, 1;}$,
            ="\n"
                     ;print
ARGV
                                  pars
                                  )#
                (@
```

Basic usage of the pars subroutine that this onomatolexical program provides:

pars('()', "(123 (456) (789) 0)")

prints the three parenthetical expressions:

```
(123 (456) (789) 0), (456), (789)
```

You can request a particular depth. In list context, this expression:

```
pars('()', "(123 (456) (789) 0)", 2)
```

prints the level-2 expressions:

```
(456),(789)
```

In scalar context, the 2 is interpreted to mean the second parenthetical expression:

(456)

(Courtesy Paul Kulchenko.)

Converting a GIF Image to an HTML Table

Each cell of the table corresponds to a pixel of the image:

```
Śf
            } 0..$g->colorsTotal;
           "<$T border=0 cellpadding=0
       р
cellspacing=0>";
   (\$x, \$y) = \$g - > getBounds;
   for $j (0..$y) {
       p "";
       for ($i=0; $i<$x; $i++) {</pre>
           s=1;
           $s++ && $i++ while ($i+1 < $x &&
$g->getPixel($i+1,$j)
                                ==
$q->getPixel($i,$j));
                      p "
$c[$q->qetPixel($i,$j)],
             " colspan=$s>&nbsp"
       }
   }
  р "</$т>"
}
```

(Courtesy Mike Fletcher.)

Identifying CVS Files That Aren't Up To Date

```
cvs status | perl -nle 'next unless
/Status:/o; print unless /Up-to-date/'
```

(Courtesy Geoff Simmons.)

Displaying All Perl's Error Messages

perl -e 'for (0..127) { \$!=\$; warn \$!}'

How to Patch Your Netscape Binary to Enable Strong Encryption

This is out of date now, but still of historical interest:

```
#!/usr/bin/perl -0777pi
s/(BITS:.*?\0)/$_=$&;y,a-z, ,;s,
$,true,gm;s, 512,2048,;$_/es;
```

(Courtesy Ian Goldberg and Chris Nandor.)

A Little-Known Way to Create References

You can create a reference to a scalar like so:

 $sref = \svar;$

An obscure way to do the same thing:

\$ref = *var{SCALAR};

The same holds for other data types.

Not So Useful One-Liners

16 not so useful code snippets from TPJ follow.

Regular Expression Epigram

"Regular expressions are to strings with math is to numbers."

(From an Andrew Clinick column, discussing what Microsoft thinks of Perl. Short answer: they like it, because it can be used "anywhere" via Microsoft's ActiveX scripting mechanism.)

Avoiding Asteroids with Perl

Asteroid 2000 BF19 was thought to be on a potentially dangerous approach path for us Terrans, with a possible impact in 2022. However, a Perl program called clomon.pl showed that the asteroid cannot come any closer than 0.038 AU for the next fifty years. Sleep tight!

(Courtesy Andrea Milani and Scott Manley.)

Maze Generation

```
($p%$x!=1)*$M[$p-2]*8) {
   $d&1<<($i=3&int rand $d) ||redo;

$M[$p+($j=$i==0?-$x:$i==1?$x:$i==2?1:-1)]=0;
K($p+2*$j) }
K($x+1); $M[1]=$M[-2]=0;
while(@M){$_=join'',splice@M,0,$x;tr<01><
#>;print$_,"\n"}
```

The above snippet generates mazes that look like this:

#		#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
#		#						#						#										#												#				#
#		#		#	#	#		#	#	#		#		#		#		#	#	#	#	#		#		#	#	#	#	#	#	#		#		#	#	#		#
#		#		#						#		#				#				#		#		#		#						#		#						#
#		#		#	#	#	#	#		#		#	#	#	#	#	#	#		#		#		#		#		#	#	#		#		#	#	#	#	#		#
#		#				#				#				#		#				#				#		#		#		#		#		#				#		#
#		#	#	#		#	#	#		#	#	#		#		#		#	#	#	#	#	#	#		#		#		#		#		#		#		#		#
#				#				#		#						#		#						#		#		#		#				#		#		#		#
#	#	#		#		#		#		#	#	#	#	#	#	#		#		#	#	#		#		#		#		#	#	#	#	#		#		#		#
#		#		#		#		#				#				#						#				#		#				#				#				#
#		#		#		#		#	#	#		#		#		#	#	#	#	#	#	#	#	#	#	#		#		#		#		#	#	#	#	#	#	#
#				#		#				#				#												#		#		#		#				#				#
#		#	#	#		#	#	#		#	#	#		#	#	#	#	#	#	#	#	#	#	#		#		#		#	#	#	#	#		#		#	#	#
#		#		#				#				#		#				#						#				#		#						#				#
#		#		#		#	#	#	#	#		#	#	#		#		#		#	#	#		#	#	#	#	#		#		#	#	#	#	#		#		#
" #		" #		"		" #	"	"	"	#		" #	"	"		#		" #		" #	"	" #		"	"	" #	"	"		" #		" #	"	"	"	" #		" #		" #
" #		" #		#	#	" #		#		" #		" #		#	#	" #		" #		" #		" #		#		" #		#	#	" #		" #		#	#	" #		" #		" #
" #		" #		" #	"	"		" #		"		" #		"	"	" #		"		" #	"	"		" #		" #	"	"		" #		" #								
" #		" #		" #		#	#	" #	#	#	#	" #	#	#		" #	#	#	#	" #		#	#	" #		" #		#	#	" #		" #								
π #		π #		π #		π	π	π	π	π	π	π	π	π		π #		π #		π #		π #		π	π	π	π	π		π #	π	π		π		π #	π	π		π #
π #		π #		π #	#	#	#	#	#	#	#	#	#	#	#	π #		π #		π #		# #	#	#	#	#	#	#	#	# #	#	#	#	#		# #		#	#	# #
# #		# #		#	#	#	#	#	#	# #	#	#	#	#	#	# #		# #		# #		#	#	#	#	# #	#	#	#	#	#	#	#	# #		# #		#	#	# #
# #		# #		#	#	#	#	#		# #		#	#	#		# #		# #		# #		#	#	#		# #	#	#	#	#		#		# #	#	# #	#	#		# #
#		#		#	#	#	#	#		#		#	#	#		#		#		#		#	#	#		#	#	#	#	#		#		#	#	#	#	#		# "
# #		# #		# #		ш		# #		#	п	п		# #	ш	п	п	# #	п	п	п	# #		# #	ш	ш	ш	ш	п	ш	ш	# #	п	ш	ш	ш	п	ш		# "
#		#		#		#		#		#	#	#		Ħ	#	#	#	Ħ	#	#	#	#		#	#	Ħ	#	#	#	#	#	#	#	#	#	#	#	#		#
#		#		#		#		#				#										#										#								#
#		#	#	#		#		#	#	#	#	#	#	#	#	#	#	#	#	#		#		#		#	#	#	#	#	#	#		#	#	#	#	#	#	#

 #
 #
 #
 #
 #
 #
 #
 #
 #

 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #
 #</t

(Courtesy Sean M. Burke.)

The Pontifex Cryptosystem

Neal

Stephenson's novel *Cryptonomicon* includes a Perl cryptosystem code-named *Pontifex*. You can read about it at http://www.well.com/user/neal/cypherFAQ.html#12 and http://www.counterpane.com/solitaire.html. The source code is at http://www.counterpane.com/sol.pl.

Perl in Sphere, the Movie

Harry used parts of the Perl FAQ to translate a message. In the first shot, this not-quite-syntactically-correct snippet could be seen:

```
$BSD = -f '/vmunix'; if ($BSD) { system
"BIN
cbreak </dev/tty >/dev/tty 2>&1
```

In the second shot:

```
set_cbreak(0)
    local($on) = $_[0];
    local($sgttyb,@ary);
    require 'sys/ioctl.ph';
```

(Courtesy Brendan O'Dea.)

An Absurd Way to Convert from Decimal to Binary

```
#!/usr/bin/perl
($decimal, $binary) = (shift, '');
$SIG{USR1} = sub { $binary .= "0" };
$SIG{USR2} = sub { $binary .= "1" };
do { kill $decimal & 1 ? 'USR2' : 'USR1',
$$;
    $decimal >>= 1;
} while ($decimal);
print scalar reverse $binary;
```

(Courtesy Nathan Torkington.)

Swatch Internet Time

Swatch's Internet Time, heralded as a "revolutionary" way of measuring time independent of geography:

```
perl -e 'print "Internet Time @",
int(((time + 3600) % 86400)/86.4)'
```

The Game of Life

This snippet of PDL code implements Conway's game of Life (pictured in Figure 48-2).

```
use PDL;
use PDL::Image2D;
use PDL::Graphics::TriD;nokeeptwiddling3d;
```

```
$d = byte( random(zeroes(40,40)) > 0.85 );
$k = byte [[1,1,1],[1,0,1],[1,1,1]];
do { imagrgb [$d];
    $s = conv2d($d,$k);
    $d &= ($s<4);
    $d &= ($s>1);
    $d |= ($s==3);
} while (!twiddle3d);
```

(Courtesy Robin Williams and Tuomas J. Lukka.)



Figure 48-2. A PDL version of Conway's game of Life

Ransom Notes

This snippet goes through each character of standard input and uppercases it half the time:

```
perl -ne
'foreach(split//){rand()<0.5?print:print
uc;}'</pre>
```

```
(Courtesy Kyle Burton.)
```

Triggering the F00F Pentium Bug

On certain older Pentium-based systems, this code will crash the computer:

We provide it here for diagnostic purposes only.

(Courtesy Gisle Aas.)

Magic Cards

This prints 7 Magic Cards.

```
for $a(0..6){$b=1;for
$c(1..100){if($c&2**$a){printf
"%3d
",$c;print"\n"if!($b++%10)}}print"\n\n\n"}
```

Have a friend think of a number

from 1 to 100. Show them the cards one at a time and ask if their number is on the card. Mentally sum the first digit of each card for which the answer is yes. The final sum will be their number. (This trick is known to win bar bets.)

(Courtesy Bill Huston.)

Perl Poem: down.pl

```
sub merge {
    my $enses;

    do {
        not $ave;
        my $inking, @body;
        push @me, @down;
    };

    foreach $econd (%brings) {
        my $oluble, @existence;
        closer_to_your;
        drowning_beauty;
    }
}
```

(Courtesy Harl.)

Perl Poem: 143

```
%secretly = (
    confidence => 0xFADED,
    under => 0xFED,
    lost => 0xDEAD,
);
sub conscious {
    setpriority $cushion, $the, $fall or
    return $to_safer_ground;
}
print "I am ",conscious(143);
$i am = $secretly{lost};
```

Perl Poem: If Dr. Seuss Were a Perl Programmer

```
#!/usr/bin/perl
#
# Will give errors if run with -w, so
don't use -w :)
#
  Tested on NT with AS (5.005), GS
(5.004 02), and Solaris 2.6 (5.004 04)
if ("a packet hits a pocket") {
  On: a;
    socket(ON, A , PORT, "")
          && the bus is interrupted as a
very-last-resort
              && the address of the memory
makes your floppy disk, abort;
} else {
    "The socket packet pocket has an";
  error: to-report;
}
if ("your cursor finds a menu item") {
    "followed by a dash"
        && "the double clicking icon";
  puts: your-items-in-the-trash
       && your data is corrupted cause the
index("doesn't", "hash");
} else {
    "Your situation is hopeless"
        && Your system's gonna crash;
}
```

```
if ("the label on the cable") {
   On-the-table, at-your-house;
    Says the;
      sub network {"is connected to the
button on your mouse"};
 BUT: Your-packets, want-to; {/tunnel to
another protocol/};
 that's: repeatedly-rejected;
{/by the printer/}; "down the hall"
    && "YOUR SCREEN is all distorted";
{/by the side effects of Gauss/};
 so: "your icons", in-the-window;
"are as wavy as a souse";
} else {
   YOU: "may as well reboot" && "go out
with a !";
 CAUSE: /Sure as Im a poet/;
 THIS: suckers-gonna-hang;
}
print "Seuss as a tech writer - Kevin
Meltzer\n";
```

(Courtesy Kevin Meltzer.)

Perl Poem: Object-Oriented Perl

One function to bless them all,

One list to derive them,

One arrow to call them through,

Preorder search to bind them.

(Courtesy Damian Conway.)

Happy Birthday!

Perl was born December 18, 1987.

Zodiac sign: Sagittarius.

Chinese zodiac: Rabbit, signifying docility, gentleness, nonconformity, and longevity.

On December 18 in history, Keith Richards, Steven Spielberg, Brad Pitt, Paul Klee, Mohammed Ali, and Ty Cobb were born. The golf tee was patented, slavery was abolished in the United States,

New Jersey became a state, and the UN unanimously condemned hostage-taking
Index

A note on the digital index ?

A link in an index entry is displayed as the section title in which that entry appears. Because some sections have multiple index markers, it is not unusual for an entry to have several links to the same section. Clicking on any link will take you directly to the place in the text in which the marker appears.

1117

Symbols

#avara channel, Are You Spongeworthy? #chaos channel and chaosbot, Bots and Infobots #distributed channel, Are You Spongeworthy? #if and #ifdef statements in Perl, Toss-up Questions #linuxos channel, Are You Spongeworthy? #macdev channel, Are You Spongeworthy? #macintosh channel (IRC), IRC #perl (pound perl), Bonus Questions #perl channel (IRC), IRC, You Can't Do That in Public!, IRC, You Can't Do That in Public! #riskybus channel and robbot, Bots and Infobots #robogeeks channel, Are You Spongeworthy? #unixhelp channel, So What? \$ typeglob, Toss-up Answers \$Duration state variable, Percussion, Uniformity, and

noop

noop function and, Percussion, Uniformity, and noop

% (modulus) operator, Using synch, and Some Actual Music

measure counters and, Using synch, and Some Actual Music

%kanji hash, Dictionary Database

%Known_clusters hash, How to Identify Words, How to Identify Words

𐴐 attribute value, Toss-up Questions

(_) single-character Perl variable, Toss-up Answers

(') apostrophe, equivalent to double colon syntax, Toss-up Answers

10-base-T/100-base-T connections, Toss-up Questions

:-) (smiley face), Toss-up Questions as cultural icon, Toss-up Questions @CANDIDATES array, Lines 5–7: Define Constants and Globals, Lines 40–60: get_globals Subroutine populating, Lines 40–60: get_globals Subroutine

@CANDIDATE_NAME array, Lines 5–7: Define Constants and Globals, Lines 40–60: get_globals Subroutine

populating, Lines 40–60: get_globals Subroutine

(a) hypernyms array, Other Word Relationships
(a) intersection array, Movin' It Up a Level
(a) newhypersets array, Movin' It Up a Level
(a) newsynsets array, Movin' It Up a Level
(a) Notes state variable, Behold MIDI::Simple!
(a) OFFICES array, Lines 5–7: Define Constants and Globals, Lines 40–60: get_globals Subroutine, Lines 124–165: The validate Subroutine
entering ballot data into database, Lines 124–165: The validate Subroutine
populating, Lines 40–60: get_globals Subroutine

@offsets array, Basic Usage

@oldhypersets array, Movin' It Up a Level

@oldsynsets array, Movin' It Up a Level

@PARTIES array, Lines 5–7: Define Constants and Globals, Lines 40–60: get globals Subroutine

populating, Lines 40–60: get globals Subroutine

\@;\@ prototype, Toss-up Answers

\B metacharacter, matching absence of word boundaries, Contexts in Regular Expressions, Bonus Questions **\b** metacharacter, matching word boundaries, Contexts in

Regular Expressions

- __DATA__ token, Bonus Answers
- __END__ token, Bonus Answers
- __FILE__ token, Bonus Answers
- __LINE__ token, Bonus Answers
- __PACKAGE__ token, Bonus Answers

_buildTree method, Assembling an Organism

_treeIndex method, Sex and Mutation

{ } (curly braces), matching occurrences in regular expressions, Toss-up Questions

||= operator, Toss-up Answers

"concision" and Maketext module, Buzzword: Concision, The Devil in the Details, The Devil in the Details

"greedy matching" and regular expressions, It's Alive!

"leave-it-the-hell-alone" option of autoformat, Future Features

"out-of-vocabulary" words, handling for speech synthesis, The Out-of-Vocabulary Problem: Synthesis in One s///e

"Rapture" poster on TPJ cover, TPJ #9: Drummer/Coder Wanted

"Rough consensus and running code" (group motto), Toss-up Questions

"Useless use of a constant in void context." error message, Toss-up Answers



a bitter, bitter man, Toss-up Ouestions A.I. Wars game, Perl clone of, Strategy Games Aas, Gisle, Best "The Perl Journal" abc music language, Easy Things Easy Abelson, Harold, References abslink subroutine. The CGI Script absolute notes, specifying, "Relative" Notes abstraction Maketext and module. **Buzzwords: Abstraction and Encapsulation** accented characters, sorting correctly, Internationalized Sorting, Spanish: Chorizo, Chimichangas, Chicharrones, **v** Churros ACE database, Bioinformatics and Perl ActiveHome (CM11) interface, MisterHouse, Give Your House a Brain ActiveState, Bonus Questions, Bonus Questions, Toss-up **Ouestions**, Toss-up Ouestions anagrams for, Toss-up Ouestions obligatory product placement for, Bonus Questions, **Bonus Ouestions** part deux, Bonus Questions

potentially liable for tortious personal injury, Toss-up Questions

ActiveX scripting engine for incorporating Perl into scripting hosts, Toss-up Questions activity meronyms, Other Word Relationships acute accents in Spanish, Spanish: Cana y Caña Adams, Clifford, Most Powerful, The Fourth Circle: Make Believe You're Another Language Adept, Perl, Toss-up Questions Adleman, Len, Toss-up Questions Adler, David, Is There a Poet in the House?

Adventure game, The Rezrov Infocom Game Interpreter affixation (morphological process), Morphology: Word Form and Structure

AI (Artificial Intelligence), IRC, References, Bots and Infobots, So What?, What Are You Thinking?, A Sense of Play, References, Hypernyms, Hyponyms, Pertainyms, and Other Word Relationships, What's Next?, Concepts in Wordnet, Converting the Data, Basic Usage, Movin' It Up a Level, Movin' It Up a Level, Other Word Relationships, Other Word Relationships, Other Word Relationships, What's Next?, What's Next?, What's Next?. Morphology: Word Form and Structure. Morphology: Word Form and Structure, Morphological and Perl, Morphological Inflections Analysis and **Derivations, Implementing It**

infobots and, IRC, References, Bots and Infobots, So What?, What Are You Thinking?, A Sense of Play, References

Lingua::Wordnet module and. Hypernyms, Pertainyms, Other Hyponyms, and Word Relationships, What's Next?, Concepts in Wordnet, Converting the Data, Basic Usage, Movin' It Up a Level. Movin' It Up Level. Other Word a **Relationships, Other Word Relationships, Other Word** Relationships, What's Next?, What's Next?, What's Next?

word morphology and, Morphology: Word Form and Structure, Morphology: Word Form and Structure, Morphological Analysis and Perl, Morphological Inflections and Derivations, Implementing It

Ake, Earle, Results of the Contest

Alak (strategy game), Strategy Games

Albanowski, Ken, Results of the Contest, Bonus Answers algorithmic composition, Easy Things Easy, Mod, Canons, and Rounds

modulus (%) operator and, Mod, Canons, and Rounds

Algorithmic Key Recovery System for RSA, Most Powerful

algorithms, The Genetic Code, Resources, Resources, Easy Things Easy

genetic, The Genetic Code, Resources, Resources music compositions and, Easy Things Easy

letters alignments between and phonemes, **Text-to-Phoneme Conversion** Allen, Paul, Bonus Ouestions all meronyms method, Other Word Relationships alphabet blocks for Braille, Sample Text in Braille alphabetical order and internationalized sorting, Internationalized Sorting, Default sort Versus "Normal" **English Sorting** alt.perl newsgroup, A Little History AltaVista, web crawler used by, Toss-up Questions Alvestrand, Harald Tveit, References Amulet of Yendor, Toss-up Questions Amy, Douglas, References Anderson, Bruce, Sites Anderson, Lowell Bruce, References antonyms (direct/indirect), Other Word Relationships, **Other Word Relationships**

lookup feature in Wordnet, Other Word Relationships

APACHE ("It's a patchy server."), Toss-up Answers

Apache web sites, directories on, Bonus Ouestions **Apache/Java Integration Project, Toss-up Ouestions Apache::Throttle module, Bonus Ouestions** aphelion, Building Your Own **API (Application Program Interface)**, **Bonus Answers API for XML, Toss-up Ouestions** apostrophe ('), equivalent to double colon syntax, Toss-up Answers Apple's computer released in 1998, Toss-up Questions **Application Program Interface (API)**, **Bonus Answers Applied Cryptography, About Secure Elections** Apu, Kwik-E-Mart employee, Toss-up Answers aqueducts and sundials, Perl-fect Sundials Arabic, translating English into, A Localization Horror Story: It Could Happen to You, The Devil in the Details arcade games, Twitch Games Archer Daniels Midland, Bonus Answers archery, Olympic silver medal in, Toss-up Ouestions area meronyms. Other Word Relationships arrays, implicitly passing by reference, Bonus Questions Arrow's Impossibility Theorem, The Preference Ballot Art as inspiration for Perl, Wherefore Art, Thou?, Music to My Ears, Wherefore Art, Thou?, Wherefore Art, Thou?, Music to My Ears Artificial Intelligence Repository at CMU, References artistic license, Toss-up Answers ASCII letters, composing signs out of, Toss-up Questions **ASCII, Braille, Sample Text in Braille** ASCIIbetical sorting, Default sort Versus "Normal" **Spanish:** English Sorting, Chorizo, Chimichangas, **Chicharrones**, v Churros Ashton, Elaine, It Seemed Like a Good Idea at the Time Asian Text Input Project, Typos in Other Languages

ASIMMs, number of pins in, Bonus Questions assembly editors, Other Uses for Perl assignment operators, turning operators into, Bonus Questions asteroids, avoiding, Avoiding Asteroids with Perl Astro::MoonPhase module, Toss-up Questions Astro::SunTime module, Toss-up Questions astronomical data analysis, using PDL for, PDL: The Perl Data Language, Where Are We Now?, The perldl Shell, Listing of a Few PDL Functions, Listing of a Few PDL Functions, Where Are We Now? as_string method, Survival of the Fittest

Atari cartridges on TPJ cover, TPJ #12: The Atari Perl Cartridge

attributes (word relationships), Other Word Relationships, Other Word Relationships

choosing proper locations for, in lexicons, Other Word Relationships

audio format, Ogg Vorbis, Toss-up Questions audio waveforms, turning tokens into, Lexical Synthesis in One s///

authentication requirements for home automation, A Few More Considerations

autoformat, Text::Autoformat, Future Features, Text::Autoformat, Text::Autoformat, Quoting, Future Features, Future Features

future features of, Future Features, Future Features, Future Features, Future Features

paragraphing heuristics, Quoting, Future Features, Future Features AUTOLOAD function, Why Was It Written?, Afterword, Afterword automating the home, Objects in the Home auxiliary functions, Buzzword: Inheritance axis of symmetry, centering lines around, Justification and Sentencing Azevedo, Francisco, Results of the Contest

В

-B flag, Bonus Questions **B::Deparse module, Acknowledgments, Bonus Questions** Babbitt, How to Identify Words, How to Identify Words, How to Identify Words, How to Identify Words, Results **Babelfish**, **Toss-up** Answers **Back Orifice '98, Toss-up Questions** background color of web page, specifying, Bonus **Ouestions** backquote interpolation. Suppressing suppressing, **Backquote Interpolation** backtracking, preventing with ?>, Bonus Questions Bacon, Greg, The Regulars, Cards Ball, Harvey R., Toss-up Questions Theorems. balloting. Impossibility Impossibility Pairwise Election **Condorcet's** Methods. Theorems. Algorithms. The Method. The Algorithms. The Algorithms, Using CGI to Spit It All Out, References, Sites, About Secure Elections, Tallying the Votes, Tallying the Votes **Impossibility** pairwise voting and. Theorems,

pairwise voting and, Impossibility Theorems, Impossibility Theorems, Pairwise Election Methods, Condorcet's Method, The Algorithms, The Algorithms, The Algorithms, Using CGI to Spit It All Out, References, Sites secure Internet voting and, About Secure Elections, Tallying the Votes, Tallying the Votes

bamf command (rezrov), Bamf bandwidth, reserving in advance, Toss-up Questions banner utility, Toss-up Answers **Baroque period**, **Music to My Ears Barrett**, Laurence I., References baud rate of fax machines, Toss-up Questions **BBFN (Bye Bye For Now), Toss-up Questions** Beattie, Malcolm, Acknowledgments Befunge language, Bonus Questions, Toss-up Answers **BEGIN blocks in Perl programs, Toss-up Ouestions** BeholderBoard (chess program), Strategy Games bent torus, graphing with PDL, Graphing a Bent Torus with PDL Berkeley DB database format and Lingua::Wordnet, **Converting the Data** Berlyn, Michael, Resurrecting the Z-Machine Best Four-Line Signature category, The Categories, Results

results, **Results**

Best Year 2000 Error category, The First Obfuscated Perl Contest

Best "The Perl Journal" category, The Categories, The First Obfuscated Perl Contest, Results results, Results

best-effort service, Toss-up Questions Beyer, Stephan V., References **Bezos, Jeff, Bonus Questions**

BGCOLOR attribute of BODY tag, Bonus Questions

bi-level sorting, Default sort Versus "Normal" English Sorting, Bi-Level Sorting to the Rescue, Bi-Level Sorting to the Rescue, English: Résumé and Resume, English: Résumé and Resume

foreign words in English language, English: Résumé and Resume

bidirectional I/O with CM11, Bidirectional I/O with the CM11

bind method (Tk), Making the Moves

bioinformatics, Bioinformatics and Perl, Other Uses for Perl, Bioinformatics and Perl, Bioinformatics and Perl, Other Uses for Perl, Toss-up Answers

Perl and, Bioinformatics and Perl, Other Uses for Perl, Bioinformatics and Perl, Bioinformatics and Perl, Other Uses for Perl

BioPerl, How Perl Saved the Human Genome Project, Other Uses for Perl

black market kidneys, number that Microsoft could buy, Bonus Questions

Blank, Marc, Resurrecting the Z-Machine

Bletchley Park, Bonus Answers

Blount, Alan, TPJ #1: The Camel, TPJ #18: Spam, TPJ #20: WAP

BNC (British National Corpus), Morphology: Word Form and Structure

Boies, David, Toss-up Questions

Borda method for tallying preference ballots, The Borda Method, The Borda Method, The Borda Method, The Borda Method bots, Chatbot::Eliza, References, References, Bots and Infobots, Are You Spongeworthy?, You Can't Do That in Public!, You Can't Do That in Public!, You Can't Do That in Public!, You Can't Do That in Public!

Chatbot::Eliza module, Chatbot::Eliza, References, References

creating believable bots, Are You Spongeworthy?, You Can't Do That in Public!, You Can't Do That in Public!, You Can't Do That in Public!, You Can't Do That in Public!

infobots, Bots and Infobots

BoulderIO system, How Perl Saved the Human Genome Project, Bioinformatics and Perl

Braille, Braille Contractions and Regular Expressions, It's Alive!, Sample Text in Braille, It's Alive!, Linguistic Rule Systems, Linguistic Rule Systems, Linguistic Rule Systems, Generativity Systems, Generativity Systems, Regex Replacement as a First Hack, Regex Replacement as a First Hack, Contexts in Regular Expressions, Embedding Code in Regular Expressions, Rules as Exceptions, Testing It, It's Alive!, It's Alive!

encoding programs, Linguistic Rule Systems, Regex Replacement as a First Hack, It's Alive!

regular expressions and, Braille Contractions and Regular Expressions, It's Alive!, Generativity Systems, Regex Replacement as a First Hack, Contexts in Regular Expressions, Embedding Code in Regular Expressions, Rules as Exceptions, Testing It, It's Alive! sample text in, Sample Text in Braille, It's Alive!, Linguistic Rule Systems, Linguistic Rule Systems, Generativity Systems, It's Alive! Braille blocks on TPJ cover, TPJ #15: Braille blocks Brams, Steven, Impossibility Theorems BRB (Be Right Back), Toss-up Questions British National Corpus (BNC), Morphology: Word Form and Structure, Constructing a Word Frequency List, Constructing a Word Frequency List, Implementing It British Sundial Society, Building Your Own Brocard, Leon, The Sub::Approx Module Bruhat, Philippe, The Fourth Circle: Make Believe You're Another Language, Inner Beauty BSD (Berkeley Software Distribution), Bonus Questions built-in Perl functions, Bonus Questions built-ists and autoformat, Lists, Quotations, Lists, Lists,

Quotations

Burk, Phil, References

Burke, Sean M., TPJ #15: Braille blocks, Perl and MIDI: Simple Languages, Easy Music, References, Hard Things Possible, Hard Things Possible, Easy Things Easy, Easy Things Easy. MIDI::Simple!. **Behold** Behold MIDI::Simple!, The Object-Oriented Interface, Using synch, and Some Actual Music, Mod, Canons, and Rounds, Mod, Canons, and Rounds, Future Features, **References, Braille and Contractions, It's Alive!, Braille** and Contractions, Linguistic Rule Systems, Linguistic Rule Systems, Generativity Systems, Regex Replacement as a First Hack, Contexts in Regular Expressions, Rules as Exceptions, It's Alive!, It's Alive!, Default sort Versus "Normal" English Sorting, Sorting it All Out, Default sort Versus "Normal" English Sorting, Locale-Based Sorting, Spanish: Cana y Caña, Spanish: Chorizo, Chimichangas, Chicharrones, V Churros, **Spanish:** Chorizo, Chicharrones, y Churros. Chimichangas, **English:** Résumé and Resume, English: Résumé and Resume, Optimizing with Memoization, Sorting it All Out, Typos in Other Languages, Bonus Answers

Braille and regular expressions, Braille and Contractions, It's Alive!, Braille and Contractions, Linguistic Rule Systems, Linguistic Rule Systems, Generativity Systems, Regex Replacement as a First Hack, Contexts in Regular Expressions, Rules as Exceptions, It's Alive!, It's Alive!

Braille blocks picture on TPJ cover, TPJ #15: Braille blocks

MIDI and Perl, Perl and MIDI: Simple Languages, Easy Music, References, Hard Things Possible, Hard Things Possible, Easy Things Easy, Easy Things Easy, Behold MIDI::Simple!, Behold MIDI::Simple!, The Object-Oriented Interface, Using synch, and Some Actual Music, Mod, Canons, and Rounds, Mod, Canons, and Rounds, Future Features, References

simulating typos, Typos in Other Languages

sorting, internationalized, Default sort Versus "Normal" English Sorting, Sorting it All Out, Default sort Versus "Normal" English Sorting, Locale-Based Sorting, Spanish: Cana y Caña, Spanish: Chorizo, Chimichangas, Chicharrones, y Churros, Spanish: Chorizo, Chimichangas, Chicharrones, y Churros, English: Résumé and Resume, English: Résumé and Resume, Optimizing with Memoization, Sorting it All Out

Bust-a-Move game, Twitch Games

buzzword compliance and Maketext, Replacing gettext, The Devil in the Details, Buzzword: Inheritance, The Devil in the Details С

-c command-line switch, Toss-up Answers caching algorithms, Bonus Questions caching techniques in genetic algorithms, Survival of the Fittest, The Terrifying Results, The Terrifying Results CAF (data interchange format), Other Uses for Perl caffeinated drinks, Bonus Questions calculus, differential/integral, A Little Rocket Science

callbacks in Tktk solitaire game, Making the Moves, Oops!, Laying the Base, Oops!

caller function, **Bonus Questions**

caller ID, monitoring incoming calls with, Whole House CallerID

Calvelli, Claudio, The Second Circle: Do Something Powerful, The Fourth Circle: Make Believe You're Another Language

Cameron, Jim, Toss-up Answers

candidate table in database schema, The Database Schema

candidates, Pairwise Voting, The Preference Ballot, Impossibility Theorems, The Algorithms, The Pairwise Engine, The Pairwise Engine

algorithms for choosing, The Algorithms, The Pairwise Engine, The Pairwise Engine

impossibility theorems and, Impossibility Theorems preference ballots and, The Preference Ballot third-party, Pairwise Voting

Cannon, Sarah Ophelia, Toss-up Questions canonicalizing information by infobots, What Are You Thinking?, A Sense of Play, What Are You Thinking?, A Sense of Play, A Sense of Play

canonization and subroutine names, Future Plans canons and measure counters, Mod, Canons, and Rounds capitalizing words with autoformat, Justification and Sentencing

Carbon technology, Toss-up Answers

card games, Cards

cards, deck of, Layout, Making the Moves, Laying the Base, Laying the Base, Pickup Lines, Finishing Touches

adding menus/menu accelerators to GUI, Finishing Touches

cycling through deck, Laying the Base laying out shuffled cards, Layout moving cards by clicking on them, Making the Moves picking up legally, Laying the Base undoing card pickups, Pickup Lines

Carnivore surveillance system. Toss-up Ouestions Carp module, Toss-up Answers cartoon music and Perl, Music to My Ears cartridges for Atari games on TPJ cover, TPJ #12: The **Atari Perl Cartridge** performing with autoformat, conversions. case **Justification and Sentencing** case-insensitive sorting, Default sort Versus "Normal" **English Sorting** Cataloguing in Publication (CIP) data for books, Toss-up **Ouestions** categories for Obfuscated Perl contests, The Zeroth **Annual Obfuscated Perl Contest** Caton, Russell, Most Powerful **CDATA tag, Bonus Answers** CEA (Central Enumeration Agency), About Secure **Elections, Generating VRNs**

entering VRNs into database, Generating VRNs

Census accepting online forms, Bonus Questions, Toss-up Answers, Bonus Questions

centering text with autoformat, Justification and Sentencing

Central Legitimization Agency (CLA), About Secure Elections

Cerf, Vinton, Toss-up Questions

CERT (Computer Emergency Response Team), Bonus Answers

CGI (Common Gateway Interface), Bonus Answers, Bonus Questions, Using CGI to Spit It All Out, Lines 61–71: The generate_ballot Subroutine, Lines 166–205: The enter ballot Subroutine

environment variable for determining visits by browsers, Bonus Questions

generating election results with, Using CGI to Spit It All Out, Lines 61–71: The generate_ballot Subroutine, Lines 166–205: The enter_ballot Subroutine

CGI.pm library, The CGI Script

chads, making them obsolete, Secure Internet Voting, Tallying the Votes, Lines 209–223: Utility Subroutines, Tallying the Votes

changed method, A Smart TV Guide

channel 9, MIDI, Percussion, Uniformity, and noop

channel number (note property), Easy Things Easy

channels on IRC, IRC, References, IRC, Are You Spongeworthy?, You Can't Do That in Public!, What Are You Thinking?, Idiot Savant, What? We Ordered No Pizzas!, References

character data, not interpreted as markup, Bonus Questions

character sets for Japanese, The Web

Charlie Chaplin smiley, Toss-up Questions

Chatbot::Eliza module, Chatbot::Eliza, References, Chatbot::Eliza, How It Works, How It Works, What Now?, References

instantiating Eliza bots, How It Works

chatterbots, Chatbot::Eliza, References, References -cheat command-line option. Tinkering with the **Z-Machine CHECK block, Bonus Questions** chess programs, Strategy Games Chinese, translating English into, A Localization Horror Story: It Could Happen to You, A Localization Horror Story: It Could Happen to You, A Localization Horror Story: It Could Happen to You. The Devil in the Details Chiu, Alex, Toss-up Answers chocolate hazelnut spread, Toss-up Questions choleric fluid, Bonus Ouestions chooser function, The Sub:: Approx Module Christiansen, Tom, Toss-up Answers, Toss-up Answers, **Bonus Answers Churchill, Winston, Bonus Questions** ci/co commands, Toss-up Questions preferences, **Pairwise Election** circular Methods. **Condorcet's Method** resolved in Condorcet's method, Condorcet's Method

Civilization 3 game, Toss-up Questions CLA (Central Legitimization Agency), About Secure Elections clam shell on TPJ cover, TPJ #2: The Pearl Classical period, Music to My Ears Cleary, Brittney, Toss-up Questions clear_cache method, Sex and Mutation

Clinton, President Bill, Bonus Questions

clusters, three-letter, How to Identify Words, Typos in Other Languages, How to Identify Words, How to Identify Words, How to Identify Words, Typos in Other Languages

CM11 controller. Home Automation: The X10 More Considerations. Nitty-Gritty. A Few Home Automation: The X10 Nitty-Gritty. What's in 8 Command?, What's in a Command?, What's in a **Bidirectional** I/O with Command?. the **CM11.** Bidirectional I/O with the CM11, Bidirectional I/O with the CM11, Bidirectional I/O with the CM11, A Few More Considerations

CM17 controller. Home Automation: The **X10** Few More Nitty-Gritty. Α **Considerations.** The ControlX10::CM17 and ControlX10::CM11 Modules. The ControlX10::CM17 and ControlX10::CM11 Command?. What's in Modules. What's in a 8 Command?, What's in a Command?, What's in a Command?, Timing Issues, Bidirectional I/O with the CM11, Bidirectional I/O with the CM11, A Few More Considerations

operations for, What's in a Command?

cm17_bit_toggle.pl program, What's in a Command? cm17_no_hardware.pl program, The ControlX10::CM17 and ControlX10::CM11 Modules

CMU Artificial Intelligence Repository, References co-occurrence of letters, How to Identify Words

code readability, measuring, Perl Style, Acknowledgments, Existing Measures, The Tool, The Perl Compiler to the Rescue, Future Directions, Acknowledgments

Code Red worm, Toss-up Answers

coffee beans on TPJ cover, TPJ #5: Commodities

coffee cup and cigarette on TPJ cover, TPJ #8: The Coffee Cup Fiasco

Cole, Patrick, Acknowledgments

Colossal Cave game, The Rezrov Infocom Game Interpreter

columns, controlling layout and justification with autoformat, Future Features

Command and Control mode of VR, Talking and Listening

command set for Z-machines, Universal Command Set command-line switches, Bonus Questions

comments, embedding in regular expressions, Bonus Questions

commodities trading TPJ cover, TPJ #5: Commodities Common Gateway Interface (CGI), Bonus Answers comp.lang.perl.announce newsgroup, A Little History comp.lang.perl.misc newsgroup, A Little History, How the "Day in the Life" Was Done, The Raw Statistics, The Future of comp.lang.perl.misc

day in the life of, How the "Day in the Life" Was Done future of, The Future of comp.lang.perl.misc history of, A Little History

statistics on postings, The Raw Statistics

comp.lang.perl.moderated newsgroup, A Little History comp.lang.perl.modules newsgroup, A Little History comp.lang.perl.tk newsgroup, A Little History compile-time vs. run-time, Bonus Questions compiler, Perl, evaluating code using Fathom, The Perl Compiler to the Rescue, Acknowledgments, Future Directions, Acknowledgments

compounding (morphological process), Morphology: Word Form and Structure

Computer Emergency Response Team (CERT), Bonus Answers

Concentration-style memory games, Strategy Games

concurrent development, provided by CVS, Toss-up Questions

Condorcet's method for pairwise voting, Condorcet's Method, The Algorithms, The Pairwise Engine, The Pairwise Engine, The Pairwise Engine

algorithm involved in, The Algorithms, The Pairwise Engine, The Pairwise Engine

calculating winners, The Pairwise Engine

configuration files, customizing chatterbots with, Chatbot::Eliza

confirmation number (CN), About Secure Elections confirming votes electronically, The E-Ballot connectivity and Link Grammar parser, Link Grammar conspiracy theme on TPJ cover, TPJ #11: The Conspiracy constants, Lines 5–7: Define Constants and Globals

defining length of valid VRNs, Lines 5–7: Define Constants and Globals

consumer electronics device for pausing live TV, Toss-up Questions

context-dependent language sounds, The Out-of-Vocabulary Problem: Synthesis in One s///e, More Context: Two Substitutions contexts in regular expressions, Contexts in Regular Expressions

contractions in Braille, Braille Contractions and Regular Expressions, Sample Text in Braille, Optimality Systems, Contexts in Regular Expressions, Rules as Exceptions, Rules as Exceptions, It's Alive!

author's regular-expression-replacement approach, It's Alive!

Braille ASCII, Sample Text in Braille determining ranking-rules, Optimality Systems hyphenation issues and, Rules as Exceptions rules as exceptions, Rules as Exceptions translating rules into regular expressions, Contexts in Regular Expressions

ControlX10::CM11 module, The ControlX10::CM17 and ControlX10::CM11 Modules

ControlX10::CM17 module, The ControlX10::CM17 and ControlX10::CM11 Modules, The ControlX10::CM17 and ControlX10::CM11 Modules

converting case in text with autoformat, Justification and Sentencing

convolving a Gaussian with an image, The peridi Shell Conway, Damian, Future Features

Conway's game of Life, Strategy Games, The Game of Life

PDL version of, The Game of Life

Copeland's method for pairwise voting, Copeland's Method

copyright page information, Toss-up Questions corn on the cob, Toss-up Questions corpuses and word morphology, Morphology: Word Form and Structure, Morphological Inflections and Derivations, Morphological Inflections and Derivations, Implementing It, Future Work, Applying the Derivation Process to All Tokens, Future Work

applying derivation process to tokens, Implementing It, Future Work, Applying the Derivation Process to All Tokens, Future Work

counting syllables in words, Counting Syllables

cover art, TPJ, TPJ #1: The Camel, TPJ #3: RSA on Greenbar, TPJ #4: Etch-a-Sketch., TPJ #6: Scrabble, TPJ #8: The Coffee Cup Fiasco, TPJ #8: The Coffee Cup Fiasco, TPJ #10: The Underwood Typewriter, TPJ #11: The Conspiracy, TPJ #14: Outlook Not So Good, TPJ #16: e. e. cummings' Gravestone, TPJ #17: Napster, TPJ #18: Spam, TPJ #18: Spam, TPJ #19: Monopoly Money

Coy.pm module, Just Another Perl Haiku, The Coy Module, Inside the Coy Module, Inside the Coy Module, Mere Words, Random Harvesting, Filling in the Blanks, The Other Type of Grammar, The Other Type of Grammar, Counting the Beat, But Does the Bear Dance?, Extending the Module, Extending the Module, Extending the Module, It Seemed Like a Good Idea at the Time correcting noun/verb agreement, The Other Type of Grammar

ensuring perfect meter, Counting the Beat examples of haiku, But Does the Bear Dance? extending the module, Extending the Module generating haiku, Filling in the Blanks, Filling in the Blanks, Filling in the Blanks grammar templates used by, Filling in the Blanks main components of, Inside the Coy Module selecting words, Random Harvesting structure of hierarchical vocabulary, Mere Words

.coyrc file, Is There a Poet in the House? Cozens, Simon, TPJ #18: Spam

CPAN (Comprehensive Perl Archive Network), Bonus Questions

CPAN modules on Unix systems, building/installing, Bonus Questions

CPU speeds, slower for laptops than desktops, Bonus Questions

Create a Diversion category, results of, The Fifth Obfuscated Perl Contest

Create, Read, Update, and Delete (CRUD), Bonus Answers

CRYPTO-GRAM newsletter, Toss-up Questions, Toss-up Answers

cryptogram-solver utilities, Word Games

cryptographic device (Enigma), Toss-up Questions

Cryptonomicon, Toss-up Answers, The Pontifex Cryptosystem

crystal clam shell on TPJ cover, TPJ #2: The Pearl culling bad organisms from list, Sex and Mutation Cult of the Dead Cow, Toss-up Answers curly braces ({ }), matching occurrences in regular expressions, Toss-up Questions current octaves, "Relative" Notes curve fitting, The Genetic Code cxref tool (C), Future Directions Cyber Promotions, president of, Toss-up Questions Cyrano Sciences, Toss-up Questions D

daemon process, Bonus Answers

Dakss, Jon, TPJ #17: Napster

dance remixes on TPJ cover, TPJ #13: Dance Remixes

data analysis and PDL (Perl Data Language), The perldl Shell, Where Are We Now?, The perldl Shell, Listing of a Few PDL Functions, Where Are We Now?

data exchange format (boulderio), Bioinformatics and Perl

data interchange format (CAF), Other Uses for Perl

data interchange standards for genome software, Other Uses for Perl

Data::Dumper module, Toss-up Answers, Bonus Questions, Stringifying Data Structures

Database Driver (DBD), Bonus Answers

Database Interface (DBI), Bonus Questions

database schema for tallying votes, The Database Schema, The Database Schema, The Database Schema, Generating VRNs

databases, matching names between, Smart Matching for Human Names, Conclusion, Module Contents, What They Do, Matching Nicknames, Conclusion

date and time in English, displaying, Toss-up Answers Davis, Michael, TPJ #20: WAP

daylight savings time, notification of, Daylight Savings Time

DB-9/DB-25 type connectors, Toss-up Questions

DBD::Oracle module, Bonus Questions

DBI (Database Interface), Bonus Questions

DBI->connect method, Lines 8–9: Connect to the Database

dbmclose function, Toss-up Answers

dbmopen function, Toss-up Answers

DBT (Duxbury Braille Translator), Linguistic Rule Systems

DCS1000 surveillance system, Toss-up Questions Dealey, William L., References

decision for trees text-to-phoneme conversion. Text-to-Phoneme **Context:** Conversion. More Two Substitutions. Text-to-Phoneme **Conversion.** Text-to-Phoneme Conversion. More Context: Two Substitutions

Decker, Richard, Artificial Haiku

declination, Building Your Own

decomposition rules in chatterbot configuration files, Chatbot::Eliza

Deepcopy method, Toss-up Questions, Bonus Questions default width of reformatted text, Text::Autoformat DEFCON 3, Red Alert!

delete function Tass un Quest

delete function, Toss-up Questions

democracy, rebuilding with Perl, Secure Internet Voting, Tallying the Votes, Lines 166–205: The enter_ballot Subroutine, Tallying the Votes

demographic information about Perl, Bonus Questions DeMorgan's Rule, DeMorgan's Rule

denial of service attacks, Toss-up Questions

department store Santa smiley, Toss-up Questions

derivational morphology, Morphology: Word Form and Structure, Future Work, Constructing a Word Frequency List, Morphological Inflections and Derivations, Morphological Inflections and Derivations, Morphological Inflections and Derivations, Representing a Single Rule, Representing Many Rules, Implementing It, Telling Good from Bad, Implementing It, Implementing It, Future Work creating word frequency lists, Constructing a Word Frequency List. Morphological Inflections and **Derivations.** Morphological Inflections and Morphological Derivations. Inflections and Derivations producing false derivations. Implementing It

representing many rules, Representing Many Rules, Implementing It, Telling Good from Bad, Implementing It

representing single rules, Representing a Single Rule

Desiderata of Interplanetary Internetworking, Bonus Answers

/dev/urandom device, Generating VRNs

Devel::Symdump module, The Symbol::Approx::Sub Module

Device::SerialPort module, What's in a Command?, What's in a Command?, What's in a Command?, What's in a Command?, What's in a Command?

DHTML (Dynamic HTML), Toss-up Answers

DiBona, Chris, Bonus Answers

dice games, Dice

dictionaries (Japanese), storing in hashed databases, Dictionary Database, The CGI Script, The CGI Script dictionary files, Link Grammar

Link Grammar parser and, Link Grammar

differential calculus, A Little Rocket Science digital clocks, power of two appearing on display, Toss-up Questions

DIMMs, non-parity, number of pins in, Bonus Questions dimpled chads, Secure Internet Voting dims function (PDL), The perild Shell direct antonyms, Other Word Relationships

Discman on TPJ cover, TPJ #13: Dance Remixes

disgruntled postmasters, Toss-up Questions

dissociated text, implementing, Word Games

distance method (Lingua::Wordnet:: Analysis), Movin' It Up a Level

division by zero errors, coping with, using scalar variables, Toss-up Questions

DNA and the Traveling Salesman Problem, Toss-up Answers

DNA sequencing, How Perl Saved the Human Genome Project, Problems with Perl, How Perl Saved the Human Genome Project, How Perl Saved the Human Genome Project, Bioinformatics and Perl, Bioinformatics and Perl, Bioinformatics and Perl, Other Uses for Perl, Other Uses for Perl, Problems with Perl, Problems with Perl

steps involved in, Bioinformatics and Perl

document type definitions (DTDs), Bonus Questions Doherty, Paul David, References

dolphin-shaped Russian cipher by Vipul Ved Prakash, Most Powerful

DOM (Document Object Model), piggybacking onto HTTP, Toss-up Questions

Dominus, Mark Jason, Optimizing with Memoization, Toss-up Answers, Toss-up Answers, Bonus Answers, It Seemed Like a Good Idea at the Time, The Old Standby

exemptions from warning message, Toss-up Answers memoization article, Optimizing with Memoization

done_now method, You Have Mail dotcoms, failed, Bonus Questions

double colon syntax, shorter equivalent of, Toss-up Questions

double-quote processing, overloading, Irregular Regular Expressions, Overloaded

Drepper, Ulrich, References

DSL, types of, **Bonus Questions**

DTDs (document type definitions), Bonus Questions

Dubé silicone juggling ball on TPJ cover, TPJ #7: Spiderball

Dublin Core, Toss-up Questions, Toss-up Questions, Toss-up Answers

dump function, Toss-up Answers

Dumper method, **Toss-up Questions**

Dungeons and Dragons game, Toss-up Answers

duopoly, two-party, Pairwise Voting

duration (note property), Easy Things Easy

Dutch, simulating typos in, Typos in Other Languages, References, Typos in Other Languages, References

Duverger's Law, Pairwise Voting

Duxbury Braille Translator (DBT), Linguistic Rule Systems

DVD (Digital Versatile Disk), Toss-up Answers

Dvorak keymap vs. QWERTY keymap, Simulating Typos with Perl, References, Simulating Typos with Perl, Simulating the Typos, How to Identify Words, How to Identify Words, Typos in Other Languages, Typos in Other Languages, References

Dvorak, August, Simulating Typos with Perl, References dynamic syntax, Why This Should Be Hard To Do Ε
e. e. cummings' gravestone on TPJ cover, TPJ #16: e. e. cummings' Gravestone EarthWeb buyout, TPJ #14: Outlook Not So Good Easter eggs in Infocom games, Pilfer Eazel corporation, now defunct, Toss-up Ouestions eccentricity of earth orbit, Building Your Own, Building Your Own ecliptic angle, Building Your Own edict dictionary, The Web, Dictionary Database, **Dictionary Database** Edison, Thomas, Toss-up Questions Edmonson, Michael, Resurrecting the Z-Machine EFNet (Eris-Free Net), IRC, You Can't Do That in Public!, IRC, So What?, Are You Spongeworthy?, You **Can't Do That in Public!** EG (Exempli Gratia) directory of Perl distribution, **Toss-up Answers** Eggdrop bot, IRC Einstein, Albert, Toss-up Answers, Toss-up Questions **Eisenberg, Mike, Results of the Contest** election fraud, discouraged by Internet voting, About **Secure Elections Election Methods mailing list, The Algorithms, Sites** electronic ballots, The E-Ballot, The vote Program implemented by CGI script, The vote Program

Eliza algorithm, Chatbot::Eliza, References, Chatbot::Eliza, How It Works, How It Works, How It Works, References

elsif constructs in several languages, Bonus Questions Emacs vs. vi, Bonus Questions

Emacs, using Perl from, Using Perl from Emacs

email accounts, checking with MisterHouse, You Have Mail

embedding code in regular expressions, Embedding Code in Regular Expressions

emergency memory pools, creating, Toss-up Questions

encapsulation and Maketext module, **Buzzwords:** Abstraction and Encapsulation

Enchanter (game), Enter the Camel

encoding programs, Linguistic Rule Systems, Regex Replacement as a First Hack, It's Alive!, It's Alive!

DBT, Linguistic Rule Systems

regular-expression-replacement approach, Regex Replacement as a First Hack, It's Alive!, It's Alive!

END blocks in Perl programs, Toss-up Answers English, The Web

automatically translating kanji to, The Web

English.pm module, Bonus Questions

enter_ballot subroutine (e-ballot), Lines 23–26: Handle a Submitted Ballot, Lines 166–205: The enter_ballot Subroutine

enter vrn.pl script, Generating VRNs

entropy coding and Perl function names, Bonus Questions Eppley, Steve, Sites

EPROMs, missile, A Little Rocket Science, Red Alert!, Red Alert!, Red Alert!

equals sign, adding to operators, Bonus Questions Eris-Free Net (EFNet), Bots and Infobots

error handling and home automation, A Few More Considerations

error messages and haiku responses, Just Another Perl Haiku, Random Harvesting, But Does the Bear Dance? error subroutine (e-ballot), Lines 124–165: The validate Subroutine

Etch-a-Sketch on TPJ cover, TPJ #4: Etch-a-Sketch.

Ethernet technology, Toss-up Answers

eticity, Toss-up Questions

European Parliament, web site for, Typos in Other Languages

eval program, Interface Abstraction

checking if interface modules are installed, Interface Abstraction

evaluate method in genetic algorithms, Survival of the Fittest

event queues for monitoring home appliances, A Few More Considerations

events in Tktk solitaire game, Making the Moves

Everquest game, Toss-up Answers

evil names, Bonus Questions

evolution and genetic algorithms, Genetic Algorithms, Survival of the Fittest

Excedrin, caffeine content of one tablet, Bonus Questions Expect.pm module, Toss-up Answers

explainstr subroutine, The CGI Script

exponentiation operator (\$) and Nike missiles, A Little Rocket Science

expressions per statement, measuring, The Basic Units Extensible Markup Language (XML), Bonus Answers Extensional Language, Hard Things Possible, Approach 1: A Novel Language, Approach 2: An Extensional Language, Easy Things Easy, Easy Things Easy, Easy Things Easy, Using synch, and Some Actual Music

implementation considerations, Easy Things Easy

F

factoids and infobots, Are You Spongeworthy?

familiarity method (Lingua::Wordnet:: Analysis), Movin' It Up a Level, Movin' It Up a Level

Fanning, Shawn, Bonus Questions

Fathom module, Perl Style, Acknowledgments, The Tool, Usage, The Perl Compiler to the Rescue, Future Directions, Acknowledgments

fax machines, baud rate of, Toss-up Questions

feature vectors, Text-to-Phoneme Conversion

Fellowship of the Ring, Bonus Answers

Festival (open-source synthesizer), Text-to-Phoneme Conversion

Festival TTS engine, MisterHouse, Talking and Listening FETCH function, Bonus Questions

Fifth Obfuscated Perl Contest, The Fifth Obfuscated Perl Contest, The Old Standby, Results, World Wide Wasteland, The Old Standby

file tests, operations for, Bonus Questions, Toss-up Questions

File Transfer Protocol (FTP), Bonus Answers

File_Item (mh object), Objects in the Home

Filter::cpp module, A Plea for Inlining

finite state transducers, Text-to-Phoneme Conversion

Finland, land of most Internet sites per capita, Toss-up Answers

Firecracker (CM17) interface, Give Your House a Brain, Home Automation: The X10 Nitty-Gritty

firewalls, penetrating, Toss-up Questions

First Obfuscated Perl Contest, The First Obfuscated Perl Contest, Best of Show, Results, Most Powerful, Most Creative, Best "The Perl Journal", Best Year 2000 Error, Best Year 2000 Error, Best Year 2000 Error, Best of Show First Perl/Internet Quiz Show, Toss-up Questions, Toss-up Questions, Toss-up Questions, Bonus Questions, Bonus Questions, Bonus Questions, Bonus Answers, Bonus Answers

Fishburn, Peter C., Impossibility Theorems

Fisher-Yates algorithm, shuffling deck of cards with, Shuffling

fitness function, Survival of the Fittest

fitness of genetic algorithms, Genetic Algorithms, The Terrifying Results

results of testing, The Terrifying Results

FITS (format for astronomical data), The peridd Shell flames in comp.lang.perl.misc, How the "Day in the Life"

Was Done, The Day's Weather Report

Flesch-Kincaid tools, Existing Measures

fname eq function, Module Contents

follow_leader.pl program, Bidirectional I/O with the CM11

food foraging behavior in ants, emulating, Other Applications

Forbes, Nevill, References

Ford, Gertrude Catherine, References

fork function, Toss-up Questions

formatting text with autoformat, Text::Autoformat, Paragraphs, Lists, Lists, Widows, Justification and Sentencing, Future Features

Forth-extension HMSL music language, Easy Things Easy Fourier Series for a given periodic function, finding, Other Applications

Fourth Obfuscated Perl Contest, The Fourth Obfuscated Perl Contest, The Fourth Obfuscated Perl Contest, The Fourth Circle: Make Believe You're Another Language

Fourth Perl/Internet Ouiz Show, The Fourth Perl/Internet **Ouiz Show, Bonus Answers, Bonus Answers** fractal color generator/web browser load tester, World Wide Wasteland fractal pixmap generator, Most Powerful Frankel, Justin, Toss-up Ouestions Frankenstein language, Toss-up Ouestions fraud, election, discouraged by Internet voting, About Secure Elections free database system, Toss-up Questions Freeman, Gordon, Toss-up Questions freeware porn detector, **Bonus Ouestions** frequency lists for words, Morphological Analysis and Perl, Constructing a Word Frequency List, Constructing a Word Frequency List, The Key Insight!, Implementing It, Applying the Derivation Process to All Tokens Friedl, Jeffrey, References, Testing It Frodin, Peter, Results of the Contest Frogger in 2048 bytes, Create a Diversion frotz interpreter, Enter the Camel **Frozen Bubble game, Twitch Games** Fully Qualified Domain Name (FODN), Bonus Answers functions (word relationships), Other Word Relationships functions, mathematical (used in genetic algorithms), The **Genetic Code**

Futrelle, Joe, Most Powerful, Best "The Perl Journal"

G

G2G (Got to Go), Toss-up Questions

galaxy, spiral, The peridi Shell

Gallo, Felix S., Results of the Contest, Hints and Suggestions, So There You Have It, Results, Best "The Perl Journal", The Third Circle: Be Creative

Fourth Obfuscated Perl Contest, The Third Circle: Be Creative

Prisoner's Dilemma contest, Results of the Contest

Zeroth Obfuscated Perl Contest, Hints and Suggestions, So There You Have It, Results, Best "The Perl Journal"

gambling prohibition act for Internet, Toss-up Answers game cartridges on TPJ cover, TPJ #12: The Atari Perl Cartridge

game console, new, Bonus Questions

game theory and the Prisoner's Dilemma, The Prisoner's Dilemma, Results of the Contest

games in Perl, Games in Perl, Twitch Games, Strategy Games, Strategy Games, Cards, Cards, Dice, Word Games, Twitch Games, Word Games, Twitch Games, Twitch Games, The Rules, Finishing Touches, A First Cut, Shuffling, Layout, Pickup Lines, Finishing Touches, Finishing Touches

cards, Cards, The Rules, Finishing Touches, A First Cut, Shuffling, Layout, Pickup Lines, Finishing Touches, Finishing Touches

dice, Dice

strategy, Strategy Games

word games, Word Games, Twitch Games, Twitch Games

Games::Cards::Tk module, Cards

Games::Chess module, Strategy Games Games::Dissociate module, Word Games Games::Go::GMP module, Strategy Games Games::Jumble module, Word Games Garriott, Richard, Toss-up Questions GAs (genetic algorithms), Genetic Algorithms, Resources, **Going Further, Resources** Gaussian blob, two-dimensional, The peridi Shell geek cruises, The Perl Whirl Quiz Show gendb.pl script, Dictionary Database generate ballot subroutine (e-ballot), Lines 61-71: The generate ballot Subroutine generate vrns.pl script, Generating VRNs generativity systems and Braille, Generativity Systems, **Generativity Systems** genetic algorithms (GAs), Genetic Algorithms, Resources, The Genetic Code, Assembling an Organism, Sex and **Mutation, Other Applications, Resources** genetic code for describing organisms, The Genetic Code, Assembling an Organism, The Genetic Code, Assembling an Organism **Genetic Programming, Going Further** genome project, saved by Perl, How Perl Saved the Human Genome Project, Problems with Perl, How Perl Saved the Human Genome Project, How Perl Saved the Human Genome Project, Other Uses for Perl, Other Uses for Perl, Other Uses for Perl, Other Uses for Perl, **Problems with Perl** geometry managers, **Bonus Questions** getlink subroutine, The CGI Script getprotobynumber function, Toss-up Answers

gettext module, A Localization Horror Story: It Could

Happen to You, The Linguistic View, The Linguistic

View, The Linguistic View, Proof in the Pudding: Localizing Web Sites

localization horror story, A Localization Horror Story: It Could Happen to You, The Linguistic View, The Linguistic View, The Linguistic View localizing web sites, Proof in the Pudding: Localizing

Web Sites

get_byte macro (ZIP interpreter), Quantity Is Job One get_byte_at method (rezrov), The Joy of vec

get_email program, You Have Mail

get_globals subroutine (e-ballot), Lines 40-60: get_globals Subroutine

get_tv_info program, A Smart TV Guide

Gilmore, John, Toss-up Questions

Glacial Rift of the Frost Giant Jarl, Toss-up Questions

Glazebrook, Karl, Bonus Questions

globals for electronic voting program, Lines 40–60: get globals Subroutine

loading with information, Lines 40-60: get_globals Subroutine

GlobWalker module, The Symbol::Approx::Sub Module glosses (definitions) of words, Basic Usage lookup synset method and, Basic Usage

GNOME shell and file manager, Toss-up Questions gnomons and sundials, Perl-fect Sundials, Perl-fect Sundials

GNU General Public License, Toss-up Questions, Toss-up Questions

Go, game of, The Web, wwwkan.pl, The Web, Taking It a Step Further, Conclusion, wwwkan.pl, Strategy Games Games::Go::GMP and Games::Go::SGF utilities, Strategy Games incentive for learning Japanese, The Web, wwwkan.pl, The Web, Taking It a Step Further, Conclusion, wwwkan.pl

Google spell-check feature, Afterword googlewhacking, Word Games

Gough, Brian, Results of the Contest

grading readability of Perl programs, Perl Style, Acknowledgments, The Tool, Usage, The Perl Compiler to the Rescue, The Perl Compiler to the Rescue, Acknowledgments

grammar, Filling in the Blanks, The Other Type of Grammar

adjusting form of selected words, The Other Type of Grammar

sample templates used by Coy.pm, Filling in the Blanks

grammatical number, A Localization Horror Story: It Could Happen to You, A Localization Horror Story: It Could Happen to You, Buzzword: Isomorphism

Arabic treatment of, A Localization Horror Story: It Could Happen to You

English treatment of, A Localization Horror Story: It Could Happen to You

ways of testing for, **Buzzword: Isomorphism**

Grand Ole Opry, Toss-up Questions greenbar, RSA on, TPJ #3: RSA on Greenbar Gregorian style of numbering, Toss-up Answers grepping pronunciation files for patterns, Searching the Prepared Data, Coping with (Syllabic) Stress Grinberg, Dennis, References Group (mh object), Objects in the Home Gruhl, Dan, Future Work GUI toolkits, Bonus Questions Guidero, Mike, The Second Circle: Do Something Powerful Guru, Perl, Toss-up Questions Gutteridge, Christopher, Inner Beauty Η

H2XS utility, Toss-up Answers

Hackman, Kevin, It Seemed Like a Good Idea at the Time Hagelberg, Andreas, The Third Circle: Be Creative

haiku, Perl, Just Another Perl Haiku, The Tao of Haiku, The Tao of Haiku, Artificial Haiku, Artificial Haiku, Artificial Haiku, Inside the Coy Module, Inside the Coy Module, Random Harvesting, Random Harvesting, Filling in the Blanks, Counting the Beat, Counting the Beat, Extending the Module, Is There a Poet in the House?, Is There a Poet in the House?

haiku-generating grammar, Artificial Haiku, Artificial Haiku

by Decker and Hirshfield, Artificial Haiku

by Garret Kaminaga, Artificial Haiku

Half-Life game, Toss-up Answers ham radio modems, MisterHouse ham radios, tracking vehicle positions, You Have Mail hanging chads, Secure Internet Voting Hangman games, Word Games Hardt, Dick, Bonus Questions Hare method for tallying preference ballots, The Hare Method Harper, Damon, Obfuscated Poetry? Hartnoll, Dave, Most Powerful hashed databases, storing Japanese dictionaries in, Dictionary Database, The CGI Script, The CGI Script headstone (e. e. cummings) on TPJ cover, TPJ #16: e. e. cummings' Gravestone heat dissipation of laptops, Toss-up Answers heuristics and word morphology, Morphology: Word Form and Structure hex function, Toss-up Ouestions

hiding secret messages in plaintext, TPJ #3: RSA on Greenbar

hierarchical lists and autoformat, Lists, Quotations, Lists, Lists, Quotations

hierarchy of hypernyms, Concepts in Wordnet Hietaniemi, Jarkko, Toss-up Answers, Bonus Answers highlight casing text with autoformat, Justification and Sentencing

hiragana, The Web

Hirshfield, Stuart, Artificial Haiku

hocus (infobot), A Sense of Play

Hollerith punchcards, Secure Internet Voting

holonyms, Other Word Relationships

home automation, **Timing Issues**

timing issues with, Timing Issues

Home Automation (HA), Home Automation: The X10 Nitty-Gritty, A Few More Considerations, A Few More Considerations

working with, Home Automation: The X10 Nitty-Gritty, A Few More Considerations, A Few More Considerations

Hopkins, Sharon, The Perl Poetry Contest Houston, Robin, The Symbol::Approx::Sub Module Howe, Chris, Toss-up Answers, The Fourth Circle: Make Believe You're Another Language HP printers, weights of, Bonus Questions HTML 4, reformulated in XML, Toss-up Questions HTML filter, learning Japanese with, The CGI Script, The CGI Script, wwwkan.pl, wwwkan.pl HTML::Parse module, The CGI Script HTTP header field, Bonus Questions HTTP status constants, **Bonus Questions**

HTTP_USER_AGENT environment variable, Toss-up Answers

Hudson, Dean, Is There a Poet in the House?

Hudson, Tramm, The First Circle: Print "The Perl Journal"

human genome project, How Perl Saved the Human Genome Project, Problems with Perl, How Perl Saved the Human Genome Project, Bioinformatics and Perl, Bioinformatics and Perl, Other Uses for Perl, Other Uses for Perl, Other Uses for Perl, Other Uses for Perl, Problems with Perl, Toss-up Questions

how Perl saved, How Perl Saved the Human Genome Project, Problems with Perl, How Perl Saved the Human Genome Project, Bioinformatics and Perl, Bioinformatics and Perl, Other Uses for Perl, Other Uses for Perl, Other Uses for Perl, Other Uses for Perl, Problems with Perl

human names, smart matching for, Smart Matching for Human Names, Conclusion, Module Contents, Using the Modules, What They Do, Matching Nicknames, Conclusion

humors, descriptions of, Bonus Questions

hyper-light-speed antenna patent filing, excerpt from, Bonus Questions, Bonus Answers

hypernyms, Concepts in Wordnet, Basic Usage, Other Word Relationships, Other Word Relationships

constructing a list of, **Basic Usage**

multiple, adding to synsets, Other Word Relationships

hyphenation and Braille contractions, Braille and Contractions, Optimality Systems, Contexts in Regular Expressions, Optimality Systems, Optimality Systems, Contexts in Regular Expressions rules for ranking, Optimality Systems, Contexts in Regular Expressions, Optimality Systems, Optimality Systems, Contexts in Regular Expressions

hyponyms, Other Word Relationships

1171

I

I Am Not A Lawyer (IANAL), Bonus Answers

-I command-line switch, Toss-up Answers

I18N (internationalizing Perl programs), Localizing Your Perl Programs

I18N::LangTags module, Buzzword: Concision

IBM ViaVoice VR/TTS engines, MisterHouse, Talking and Listening

ICANN (Internet Corporation for Assigned Names and Numbers), Toss-up Answers

ICBMs (Intercontinental Ballistic Missiles), A Little Rocket Science, Red Alert!, Red Alert!

ICMP ping messages, Toss-up Questions

ICMP protocol, **Toss-up Questions**

IETF (Internet Engineering Task Force), Toss-up Answers

IF (interactive fiction) games, The Rezrov Infocom Game Interpreter, References, Resurrecting the Z-Machine, The Joy of vec, The Joy of vec, Teleport, Teleport, Teleport, Teleport, Pilfer, Universal Command Set, Interface Abstraction, Quantity Is Job One, A Plea for Inlining, References

IF-MODIFIED-SINCE header field, Toss-up Answers

Ig Nobel Prize for Communications, Toss-up Questions

Immortality Rings, Toss-up Questions

impossibility theorems, Impossibility Theorems

indent tool (C), Future Directions

Independence Day, Bonus Answers

indirect antonyms, Other Word Relationships

infectfiles (virus function), Toss-up Questions

inflections and word morphology, Morphology: Word Form and Structure, Representing Many Rules, Implementing It, Implementing It representing many rules, Representing Many Rules, Implementing It, Implementing It

infobots, Infobots and Purl, References, IRC, So What?, You Can't Do That in Public!, What Are You Thinking?, What Are You Thinking?, What Are You Thinking?, What Are You Thinking?, A Sense of Play, What Are You Thinking?, What Are You Thinking?, Idiot Savant, Idiot Savant, Idiot Savant, A Sense of Play, Future Directions, References

future directions of, Future Directions processing information, What Are You Thinking?, A Sense of Play, Idiot Savant, A Sense of Play rules for answering questions, What Are You Thinking?, What Are You Thinking?, What Are You Thinking?, What Are You Thinking?

Infocom games, Enter the Camel, What's It Do?, Interface Abstraction, Toss-up Answers

rezrov interpreter and, Enter the Camel, What's It Do?, Interface Abstraction

Inform (IF authoring system), Resurrecting the Z-Machine

informatics and Perl, Bioinformatics and Perl

InfoTaskForce team, Resurrecting the Z-Machine

inheritance and Maketext module, Buzzword: Inheritance INIT block, Bonus Questions

inlining frequently-used code for Z-machines, A Plea for Inlining

input context templates, Toss-up Answers instance data, accessing, Quantity Is Job One integer pragma, Toss-up Answers integral calculus, A Little Rocket Science

Integrated Services Digital Network (ISDN), Bonus Answers

interactive fiction (IF) games, The Rezrov Infocom Game Interpreter, References, Resurrecting the Z-Machine, References

interactive sessions with chatterbots, Chatbot::Eliza, References, Chatbot::Eliza, How It Works, How It Works, References

exposing Eliza's weaknesses, How It Works

interchangeable software, need for in genome research, Other Uses for Perl

interfaces, abstracting using Z-machine opcodes, Interface Abstraction

International Standardization Organization (ISO), Toss-up Answers

internationalization (synonymous with I18N), Localizing Your Perl Programs

internationalized sorting, Default sort Versus "Normal" English Sorting, Locale-Based Sorting, Locale-Based Sorting, Spanish: Chorizo, Chimichangas, Chicharrones, y Churros, Spanish: Chorizo, Chimichangas, Chicharrones, y Churros, Bi-Level Sorting to the Rescue, English: Résumé and Resume, Optimizing with Memoization

Internet access at McDonald's, The Fourth Perl/Internet Quiz Show

Internet Engineering task Force (IETF), Toss-up Answers Internet Gambling Prohibition Act, Toss-up Answers Internet Index, Toss-up Questions, Toss-up Answers Internet Integrated Service (IIS) model, Toss-up Ouestions Internet protocol specifications, primary requirement levels, Bonus Questions

Internet Relay Chat (IRC), IRC

resources for, IRC

Internet voting, secure, About Secure Elections, About Secure Elections, The E-Ballot, The vote Program, Lines 72–104: The voting_matrix Subroutine

Interplanetary Internetworking, Desiderata of, Bonus Answers

interpreter, rezrov, The Rezrov Infocom Game Interpreter, References, Enter the Camel, What's It Do?, The Joy of vec, Tinkering with the Z-Machine, Performance Considerations, A Plea for Inlining, References

intersection method (Lingua::Wordnet:: Analysis), Movin' It Up a Level

IP addresses, determining classes of, Bonus Questions IPC::Open2/IPC::Open3 modules, Bonus Questions

IPSEC protocol, Toss-up Questions

IRC (Internet Relay Chat), IRC resources for, IRC

IRMan interface, MisterHouse

ISO (International Standardization Organization), Toss-up Answers

ISO 8859-1 entity name for symbols, Bonus Questions

isomorphism and Maketext module, Buzzword: Isomorphism

Iterated Prisoner's Dilemma, The Iterated Prisoner's Dilemma

J

Jackson, Peter, Toss-up Answers Jakarta project, Toss-up Answers James, Mark, Most Creative Japanese, learning with HTML filter, Learning Japanese with an HTML Filter, www.kan.pl, The Web, Dictionary Database, The CGI Script, Taking It a Step Further, wwwkan.pl Jedi leitmotif, adding to .covrc file, Extending the Module Jeopardy-style messages, submitting to newsgroups, **Netiquette Nits** JIS character set, Localizing Your Perl Programs Johnston, Joe, Bonus Answers, Bonus Questions join function, Bonus Ouestions Joki, Jani, Best "The Perl Journal", Best of Show Jolt, caffeine content of, Bonus Ouestions Joy, Bill, Bonus Answers Jumble word puzzles, Word Games jumping the shark, Toss-up Answers justifying text with autoformat, Justification and Sentencing

K

Kaiser, Cameron, Best "The Perl Journal" Kaminaga, Garret, Artificial Haiku Kandasamy, Kalai, Most Powerful kanji, automatically translating to English, Learning Japanese with an HTML Filter, www.kan.pl, The Web, Dictionary Database, Dictionary Database, Dictionary Database, The CGI Script, www.kan.pl, www.kan.pl, wwwkan.pl kanjidic dictionary, The Web, Dictionary Database, **Dictionary Database** katakana. The Web Katalov, Alexander, Toss-up Ouestions Keating, John, The First Circle: Print "The Perl Journal" Keller, Helen, Braille and Contractions kerberos system, Bonus Answers kernel and colonel. Toss-up Answers Kernighan, Brian, Toss-up Questions KewlChess program, Strategy Games keys function, Bonus Ouestions keywords in chatterbot configuration files, Chatbot::Eliza Kimball, Ronald J., A Perl Program That Generates Poetry King of Spam, Toss-up Ouestions Kleene, Stephen, It's Alive! Klep, Robert, Best Four-Line Signature, Most Powerful, Most Creative, The Second Circle: Do Something Powerful klystron, commonly misspelled, A Little Rocket Science Knuth, Donald, Toss-up Questions, Toss-up Answers **KOI8 character set, Localizing Your Perl Programs** Komodo development environment, Bonus Questions Komodo dragons, features of, Toss-up Ouestions Konarski, Beirne, Results of the Contest Kontour drawing program, Toss-up Ouestions Koza, John, Other Applications Kugel (strategy game), Strategy Games Kyl, Senator Jon, Toss-up Ouestions

L

-l command-line switch, Toss-up Answers

L10N (localizing Perl programs), Localizing Your Perl Programs, The Devil in the Details, The Devil in the Details, The Devil in the Details, Toss-up Questions

laboratory management, using Perl in, Other Uses for Perl, Problems with Perl, Problems with Perl

Lack, Gordon, Most Powerful

Lafferty, John, References

Lalonde, Brian, Matching Nicknames

Langston, Peter S., References

language, Easy Things Easy, Concepts in Wordnet, Concepts in Wordnet, Basic Usage, Basic Usage, Movin' It Up a Level, Movin' It Up a Level, Other Word Relationships, Other Word Relationships, Other Word Relationships, Other Word Relationships, What's Next?, I See a Pattern Developing, A Full Lingua::LinkParser Link Grammar. Link Grammar, Example. The What-Sucks-Ometer. Lingua::LinkParser Full Α **Example, Morphology: Word Form and Structure, Smart** Matching for Human Names, Conclusion, Module Contents, Using the Modules, What They Do, Matching Nicknames, Conclusion, Simulating Typos with Perl, Simulating Typos with Perl, Simulating the Typos, How to Identify Words, How to Identify Words, Learning Japanese with an HTML Filter, www.kan.pl, www.kan.pl

extensions, Easy Things Easy

learning with HTML filters, Learning Japanese with an HTML Filter, www.kan.pl, www.kan.pl

morphology of, Morphology: Word Form and Structure

processing names, Smart Matching for Human Names, Conclusion, Module Contents, Using the Modules, What They Do, Matching Nicknames, Conclusion

processing natural, Concepts in Wordnet, Concepts in Wordnet, Basic Usage, Basic Usage, Movin' It Up a Movin' It Up a Level. Other Level. Word **Relationships, Other Word Relationships, Other Word** Relationships, Other Word Relationships, What's Next?. T See 8 Pattern Developing, Α Full Lingua::LinkParser Example, Link Grammar, Link The What-Sucks-Ometer. Grammar. Α Full Lingua::LinkParser Example

hypernyms, hyponyms, pertainyms, et al, Concepts in Wordnet, Concepts in Wordnet, Basic Usage, Basic Usage, Movin' It Up a Level, Movin' It Up a Level, Other Word Relationships, Other Word Relationships, Other Word Relationships, Other Word Relationships, What's Next?

typos in, Simulating Typos with Perl, Simulating Typos with Perl, Simulating the Typos, How to Identify Words, How to Identify Words

Latin-1 character set, Localizing Your Perl Programs, Default sort Versus "Normal" English Sorting

ASCIIbetical sorting and, Default sort Versus "Normal" English Sorting

latitudes and longitudes, Building Your Own Lauha, Jetro, World Wide Wasteland Laurel, Brenda, Bonus Ouestions Lawless, Jim, Best "The Perl Journal" layout of cards for Tktk solitaire game, Layout laziness, impatience, and hubris, Bonus Answers **LCDproc interface, MisterHouse** LDAP (Lightweight Directory Access Protocol), Toss-up Answers Leather Goddesses of Phobos game, Toss-up Questions lemniscate symbol, Bonus Questions Lenzo, Kevin, Text-to-Phoneme Conversion LeSaux, Eric, Results of the Contest letter co-occurrence, How to Identify Words Levin, Jonathan, References Lewis, Sinclair, How to Identify Words lexical synthesis with s//// expressions, Lexical Synthesis in **One s**/// lexical variables, The Symbol::Approx::Sub Module lexically scoped variables, Quantity Is Job One, Bonus **Ouestions** declarations for, **Bonus Ouestions** in methods, effect on performance, **Quantity Is Job**

in methods, effect on performance, Quantity is Job One

lexicons, Buzzword: Inheritance, Buzzword: Concision, Buzzword: Concision, Buzzword: Concision, Buzzword: Concision

Maketext module and, Buzzword: Inheritance, Buzzword: Concision, Buzzword: Concision, Buzzword: Concision, Buzzword: Concision

libwww-perl (LWP) library, The CGI Script

Lidie, Steve, Most Creative, Best "The Perl Journal", Create a Diversion

Life (the game), Strategy Games

limericks, category in Perl Poetry Contest, The Categories Lincoln, Abraham, Bonus Answers

Lindley, Adam T., Acknowledgments

line directive, Toss-up Answers

Lingua::EN::Inflect module, Buzzword: Concision, The Other Type of Grammar

Lingua::EN::MatchNames module, Smart Matching for Human Names, Conclusion, Installing the Modules, Using the Modules, What They Do, Matching Nicknames, Conclusion

Lingua::EN::NameParse module, Installing the Modules Lingua::EN::Nickname module, Smart Matching for Human Names, Conclusion, Installing the Modules, Using the Modules, What They Do, Conclusion

Lingua::LinkParser module, Other Word Relationships, Lingua::LinkParser, A Full Lingua::LinkParser Example code example, A Full Lingua::LinkParser Example

Lingua::LinkParser::Linkage package, overloading double-quote processing, Irregular Regular Expressions, Overloaded

Lingua::Wordnet module, Concepts in Wordnet, Converting the Data, Concepts in Wordnet, Converting the Data, Converting the Data, Converting the Data, Basic Usage, Basic Usage, Movin' It Up a Level, Basic Usage, Basic Usage, Basic Usage, Basic Usage, Movin' It Up a Level, Other Word Relationships, Other Useful Functions, What's Next?

analyzing synset relationships, Movin' It Up a Level, Movin' It Up a Level, Movin' It Up a Level, Other Word Relationships concepts in, Concepts in Wordnet, Converting the Data, Converting the Data converting the data, Converting the Data

installing, Converting the Data

looking up English definitions, Basic Usage, Movin' It Up a Level, Basic Usage, Basic Usage, Movin' It Up a Level, Movin' It Up a Level

Lingua::Wordnet::Analysis module, Movin' It Up a Level, Movin' It Up a Level

Lingua::Wordnet::Synset type, Basic Usage

Linguana project, Hypernyms, Hyponyms, Pertainyms, and Other Word Relationships

linguistic rule systems and Braille, Linguistic Rule Systems

linguistics and localization, The Linguistic View

link dictionary language, Link Grammar

Link Grammar parser, Link Grammar, Irregular Regular Expressions, Overloaded, Link Grammar, Lingua::LinkParser, Lingua::LinkParser, Lingua::LinkParser, Irregular Regular Expressions, Overloaded, References

Lingua::LinkParser module and, Lingua::LinkParser web site for, References

Linux, penguin mascot of, Toss-up Questions Iname_eq function, Module Contents load_lab_database.pl script, Bioinformatics and Perl locale pragma, Toss-up Answers

locale-based sorting, Locale-Based Sorting

Locale::gettext module, A Localization Horror Story: It Could Happen to You, The Linguistic View, The Linguistic View, Proof in the Pudding: Localizing Web Sites

localization horror story, A Localization Horror Story: It Could Happen to You, The Linguistic View, The Linguistic View

localizing web sites, Proof in the Pudding: Localizing Web Sites

Locale::Maketext module, Replacing gettext, Buzzword: Inheritance, Buzzword: Concision, The Devil in the Details, The Devil in the Details

localization (synonymous with L10N), Localizing Your Perl Programs, A Localization Horror Story: It Could Happen to You, Buzzword: Inheritance, The Devil in the Details

localizing Perl programs, **Buzzword:** Inheritance, **Buzzword: Concision, Buzzword: Concision** logit dbm function, Whole House CallerID Loki, Norse god of mischief, Toss-up Answers LOL (Laugh Out Loud), Toss-up Questions longitudes and latitudes, **Building Your Own** lookahead/lookbehind expressions, Generativity Systems lookup synset method, Basic Usage Lord British (pseudonym), Toss-up Answers low-frequency tokens, excluding, Constructing a Word **Frequency List** Lukka, Tuomas J., Dictionary Database, The CGI Script,

wwwkan.pl, wwwkan.pl

Μ

m// pattern match, Toss-up Questions Mac OS X, Toss-up Questions, Bonus Questions application programming interfaces, Toss-up Questions technologies of, Bonus Questions

Magic 8 Ball on TPJ cover, TPJ #14: Outlook Not So Good

Magic Cards, printing, Magic Cards

Mah, Bruce, Bonus Questions

Mahoney, Sean, Toss-up Answers

mail headers and autoformat, Future Features

Mail::Audit module, TPJ #18: Spam

Maketext module, Buzzwords: Abstraction and Encapsulation, Buzzword: Concision, The Devil in the Details

mapping out DNA, How Perl Saved the Human Genome Project

margins, adjusting with autoformat, Text::Autoformat, Future Features, Lists, Widows, Future Features

marked-up text and autoformat, Future Features

Marti, Don, What Sucks? What Rocks?

Mastering Regular Expressions, Testing It

matching, Module Contents, Correcting Typos with Perl,

Afterword, How Does It Work?, Afterword

human names, Module Contents

subroutine names, Correcting Typos with Perl, Afterword, How Does It Work?, Afterword

mate method, Sex and Mutation

Math::TrulyRandom module, Generating VRNs Mathematical Smurfs Expedition (strategy game), Strategy Games mating random organisms, Sex and Mutation matrices, PDL, The perild Shell, Listing of a Few PDL Functions

functions for, Listing of a Few PDL Functions

Mazes of Menace, Toss-up Questions

mazes, generating with Perl, Maze Generation

McCamant, Stephen, Acknowledgments, Most Creative, Most Powerful, Most Creative, Best Year 2000 Error, Best of Show

McLean, Alex, Future Features

McNeil, Scott, Toss-up Questions

MCSE (Microsoft Certified Systems Engineer) exam, Bonus Questions

measure counters, Using synch, and Some Actual Music, Mod, Canons, and Rounds

% (modulus) operator and, Mod, Canons, and Rounds adding percussion, Using synch, and Some Actual Music

measuring code readability, Background, Acknowledgments, The Basic Units, The Tool, Future Directions, Acknowledgments

melancholic fluid, **Bonus Questions**

Meltzer, Kevin, The Perl Poetry Contest, Best "The Perl Journal"

member meronyms, Other Word Relationships

memoization, optimizing with, Optimizing with Memoization

memory, The Joy of vec, The Joy of vec, The Joy of vec, The Joy of vec, Tinkering with the Z-Machine minimizing consumption using Z-code, The Joy of vec, The Joy of vec, The Joy of vec, The Joy of vec Z-machine's object table, Tinkering with the Z-Machine

in Life mental masturbation. Α Dav the of comp.lang.perl.misc. The Future of comp.lang.perl.misc. A Day in the Life of comp.lang.perl.misc, How the "Day in the Life" Was Done, Netiquette Nits, The Regulars, The Future of comp.lang.perl.misc. The Future of comp.lang.perl.misc, Perl-fect Sundials, Building Your **Own, Perl-fect Sundials, Building Your Own, Building** Your Own, Just Another Perl Haiku, The Tao of Haiku, Haiku Online, Artificial Haiku, Artificial Haiku, The Cov Module. The Cov Module. Mere Words. Filling in the Blanks, It Seemed Like a Good Idea at the Time, A Simple mpron Searcher
comp.lang.perl.misc. day in life of. A Day in the Life of comp.lang.perl.misc. The Future of comp.lang.perl.misc. A in the Life of Dav comp.lang.perl.misc, How the "Day in the Life" Was Done, Netiquette Nits, The Regulars, The Future of comp.lang.perl.misc. The Future of comp.lang.perl.misc

Just Another Perl Haiku, Just Another Perl Haiku, The Tao of Haiku, Haiku Online, Artificial Haiku, Artificial Haiku, The Coy Module, The Coy Module, Mere Words, Filling in the Blanks, It Seemed Like a Good Idea at the Time

Perl-fect sundials, Perl-fect Sundials, Building Your Own, Perl-fect Sundials, Building Your Own, Building Your Own

rhymes, searching for with Perl, A Simple mpron Searcher

meronyms, Other Word Relationships, Other Word Relationships, Other Word Relationships, Other Word Relationships, Other Word Relationships

all_meronyms method, performing lookups with, Other Word Relationships

arrays, adding to, Other Word Relationships synsets, adding to, Other Word Relationships

types of, in Wordnet databases, Other Word Relationships

Merrick, Nellie L., References

message transfer agents (MTAs), Bonus Questions Messier 51 spiral galaxy, images of, The perldl Shell Metcalfe, Bob, Bonus Questions, Toss-up Answers, Toss-up Questions Internet collapse prediction, Toss-up Questions "open sores" software, Bonus Questions

method calls vs. subroutine calls, Quantity Is Job One metrical patterns in words, Coping with (Syllabic) Stress, A Simple mpron Searcher, Coping with (Syllabic) Stress, Coping with (Syllabic) Stress, A Simple mpron Searcher mh program (MisterHouse), MisterHouse

microcode and a lovely toad, Accommodating Another Notation

microcontrollers and home automation, Home Automation: The X10 Nitty-Gritty, A Few More Considerations, Simple Output Commands: The CM17, Timing Issues, A Few More Considerations

microkernels vs. monolithic kernels, Toss-up Questions Microsoft game console, Toss-up Questions

Microsoft trash bins, rooting through, Toss-up Questions MIDI channel 9 and percussion, Percussion, Uniformity, and noop

MIDI::Simple language, Hard Things Possible, Easy Things Easy, Behold MIDI::Simple!, Using synch, and Some Actual Music, Using synch, and Some Actual Music, Using synch, and Some Actual Music, Mod, Canons, and Rounds, Future Features, Future Features

future features of, Future Features

Milankovitch cycles, Building Your Own Miller, Kevin, Most Powerful Miller, Peter, References MIME::Lite module, Bonus Questions mini Structured Query Language (mSQL), Bonus Answers Miniperl, Toss-up Answers Minnie Pearl, Toss-up Answers

Minsky, Marvin, Toss-up Questions

missile-targeting software, creating with Perl, Perl and Nuclear Weapons Don't Mix, Red Alert!, Perl and Nuclear Weapons Don't Mix, A Little Rocket Science, Red Alert!

Mission: Impossible, Bonus Answers

misspelled subroutine names, calling with, Why Was It Written?, Afterword, How Does It Work?, How Does It Work?, How Does It Work?, The Sub::Approx Module, Afterword

MisterHouse program, Home Automation with MisterHouse, MisterHouse, MisterHouse, A Smart TV Guide, Whole House CallerID, Say What?

Miyagawa, Tatsuhiko, Afterword

mkfreq program, Constructing a Word Frequency List, Morphological Inflections and Derivations, Morphological Inflections and Derivations

Moby Pronunciator lexicon, Text-to-Phoneme Conversion, References, Where to Look, Preparing the Data, Searching the Prepared Data, Searching the Prepared Data, Searching the Prepared Data, Searching the Prepared Data

preparing data for searching, Preparing the Data, Searching the Prepared Data, Searching the Prepared Data, Searching the Prepared Data

searching prepared data for patterns, Searching the Prepared Data

Modern period, Music to My Ears

Modern Reader's Japanese-English Character Dictionary, Dictionary Database modules, Lines 0-4: Load Modules, Listing Installed Modules

listing installed modules, Listing Installed Modules loading CGI and DBI, Lines 0–4: Load Modules

modulus (%) operator, Toss-up Questions monolithic kernels vs. microkernels, Toss-up Questions Monopoly money on TPJ cover, TPJ #19: Monopoly Money

more-is-less paradox, Impossibility Theorems

morphology, word, Word Morphology, Future Work, Morphology: Word Form and Structure, Morphology: Word Form and Structure, Morphology: Word Form and Structure, Constructing a Word Frequency List, Morphological Inflections and Derivations, Telling Good from Bad, Implementing It, Applying the Derivation Process to All Tokens, Future Work

Morris, Robert Tappan, Toss-up Questions

Most Creative category, The Categories, The First Obfuscated Perl Contest, The Third Circle: Be Creative results, The Third Circle: Be Creative

Most Powerful category, The Categories, The First Obfuscated Perl Contest, Results, The Third Obfuscated Perl Contest, The Third Circle: Be Creative

results, Results, The Third Obfuscated Perl Contest, The Third Circle: Be Creative

Motion Picture Experts Group (MPEG), Bonus Answers motion sensor, using with MisterHouse, Squeaky Stairs Mountain Dew, caffeine content of, Bonus Questions MP3, most popular search term, Toss-up Answers MPEG-2, encoding video with, Toss-up Questions Mr. Bill smiley, Toss-up Questions

MUDs (Multi-User Dungeons), utilities for accessing, Word Games

Mullikin, Jim, How Perl Saved the Human Genome Project

multiple-districts paradox, Impossibility Theorems

Multipurpose Internet Mail Extensions (MIME), Bonus Answers

music and Perl, Music to My Ears, Music to My Ears, Music to My Ears, Music to My Ears

music languages, implementing, Easy Things Easy, Behold MIDI::Simple!, Easy Things Easy, Easy Things Easy, Behold MIDI::Simple!

mutate method, Sex and Mutation

mutating random organisms, Sex and Mutation

mutation and sex, Sex and Mutation, The Terrifying Results, Sex and Mutation, Sex and Mutation, The Terrifying Results

my keyword, Toss-up Answers

Myers, Wayne, Best Poem Port

MySQL database system, Toss-up Answers

Ν

ñ and n, using Spanish sorting conventions, <mark>Spanish: Cana</mark> y Caña

-n flag, Toss-up Answers

Nalebuff, Barry, References

name_eq function, Using the Modules

name_sequence.pl script, Bioinformatics and Perl

Nandor, Chris, Toss-up Answers, Bonus Questions

duplicate Census forms, Bonus Questions

Napster, Toss-up Answers, Bonus Questions

natural languages, Braille Contractions and Regular Expressions, It's Alive!, Linguistic Rule Systems, Generativity Systems, Generativity Systems, Regex Replacement as a First Hack, It's Alive!, Link Grammar, Link Grammar, Lingua::LinkParser, Irregular Regular Expressions, Overloaded, What Sucks? What Rocks?, What Sucks, Regex Style, The What-Sucks-Ometer, References

parsing, using Link Grammar parser, Link Grammar, Link Grammar, Lingua::LinkParser, Irregular Regular Expressions, Overloaded, What Sucks? What Rocks?, What Sucks, Regex Style, The What-Sucks-Ometer, References

writing systems for, Braille Contractions and Regular Expressions, It's Alive!, Linguistic Rule Systems, Generativity Systems, Generativity Systems, Regex Replacement as a First Hack, It's Alive!

Nelson, Graham, Resurrecting the Z-Machine, The Joy of vec, References

Z-machine specification, References

Nemo from Georgia, World Wide Wasteland

Netcraft web server survey, Toss-up Questions, Bonus Questions

netiquette issues and comp.lang.perl.misc newsgroup, Netiquette Nits

NetTalk neural network, Text-to-Phoneme Conversion network protocol specified in IEEE standard 802.3, Toss-up Questions net connect check function, You Have Mail

Neuhaus, Sven, The First Circle: Print "The Perl Journal" neural nets, Toss-up Questions, Bonus Questions

quashing research on, Toss-up Questions

New Jersev statehood anniversary, Happy Birthday! newsgroups, Usenet, Α Dav in the Life of comp.lang.perl.misc. The Future of comp.lang.perl.misc. A Little History, The Day's Weather Report, The Future of comp.lang.perl.misc, The Future of comp.lang.perl.misc (MIDI::Simple), new score function **Behold MIDI::Simple!**

NFBTRANS (National Federation of the Blind encoding program), Linguistic Rule Systems, Regex Replacement as a First Hack, Rules as Exceptions, Rules as Exceptions

optimality systems, Regex Replacement as a First Hack

rules as exceptions, Rules as Exceptions types of rules in, Rules as Exceptions

nickfollow function, Module Contents nickhash.pl utility, Matching Nicknames nickmatch function, Module Contents nicknames, smart matching for, Module Contents, What They Do, Matching Nicknames nickname_eq function, Module Contents Niemi, Richard G., References

NIM (strategy game), Strategy Games

no-show paradox, Impossibility Theorems

nominalization (type of derivation), Morphology: Word Form and Structure

non-parity SIMMs/DIMMs, number of pins in, Bonus Questions

non-registered voters, preventing fraud from, About Secure Elections

noop function (MIDI::Simple), Percussion, Uniformity, and noop

NORAD (the North American Aerospace Defense Command), Perl and Nuclear Weapons Don't Mix, Red Alert!, A Little Rocket Science, Red Alert!, Red Alert!

normalize subroutine, Default sort Versus "Normal" English Sorting, Spanish: Cana y Caña, Spanish: Chorizo, Chimichangas, Chicharrones, y Churros

ñ and n, using Spanish sorting conventions, Spanish: Cana y Caña, Spanish: Chorizo, Chimichangas, Chicharrones, y Churros

normalize2 subroutine as tiebreaker, Bi-Level Sorting to the Rescue

Norse god of mischief, Toss-up Questions

North American Sundial Society (NASS), Building Your Own

notes, Easy Things Easy, Behold MIDI::Simple!, "Relative" Notes

algorithmic composition and, Easy Things Easy MIDI::Simple state variables, Behold MIDI::Simple! relative vs. absolute, "Relative" Notes noun attributes (word relationships), Other Word Relationships

Novel Language, Approach 1: A Novel Language, Using synch, and Some Actual Music

based on Extensional Language, Using synch, and Some Actual Music

Novice, Perl, Toss-up Questions

nuclear weapons, avoid mixing with Perl, Perl and Nuclear Weapons Don't Mix, Red Alert!, A Little Rocket Science, Red Alert!

number crunching and PDL (Perl Data Language), PDL: The Perl Data Language, Where Are We Now?, Listing of a Few PDL Functions, Where Are We Now?

numbered lists and autoformat, Lists, Quotations, Quotations

numbering style, choosing with XSL, Toss-up Questions numbers, A Localization Horror Story: It Could Happen to You, A Localization Horror Story: It Could Happen to You, Buzzword: Isomorphism, Buzzword: Inheritance, Buzzword: Concision

Chinese treatment of, A Localization Horror Story: It Could Happen to You

Russian treatment of, A Localization Horror Story: It Could Happen to You, Buzzword: Inheritance, Buzzword: Concision

testing for grammatical number, Buzzword: Isomorphism

Nwvermpre, quoth the raven, Simulating Typos with Perl nybbles, two in each, Toss-up Questions Nyman, Lisa, Toss-up Answers

0

Obfuscated Perl contests, Results, Best Year 2000 Error, Best Year 2000 Error. Best Year 2000 Error. Most Creative

First, Results, Best Year 2000 Error, Best Year 2000 Error. Best Year 2000 Error Third. Most Creative

The

Obfuscated poetry, Obfuscated Poetry? object tables (Z-machine), Tinkering with the Z-Machine object-oriented interface of **MIDI::**Simple,

Object-Oriented Interface

programming. object-oriented Perl Poem: **Object-Oriented Perl**

Perl poem, Perl Poem: Object-Oriented Perl

octaves, current, "Relative" Notes

octothorpe symbol, Bonus Questions

OEM (original equipment manufacturer). Bonus **Ouestions**

office table in database schema, The Database Schema

Official Perl Poet (Sharon Hopkins), The Perl Poetry Contest

offset method, Basic Usage

Ohm's law, Toss-up Questions

Olympic silver medal in archery, Toss-up Ouestions

Ondrik, David, TPJ #15: Braille blocks, Sample Text in **Braille**

one-liners from TPJ, One-Liners, Happy Birthday!, **Evaluating Expressions Inside Double Quotes, Extracting** Unique Elements from a List, Stripping the Eighth Bits from String. Extracting **PostScript** from a

Windows-Generated PCL Files, Not So Useful One-Liners, An Absurd Way to Convert from Decimal to Binary, Magic Cards, Happy Birthday! oops command, Universal Command Set opcodes, Z-machine, What's It Do?, Tinkering with the Z-Machine, Interface Abstraction

abstracting user interfaces, Interface Abstraction manipulating object table, Tinkering with the Z-Machine

open function, Bonus Questions open sores, Bonus Questions, Bonus Questions four types of, Bonus Questions the four humors and, Bonus Questions

open source development, central resource/repository for, Toss-up Questions

operators, Bonus Questions

turning into assignment operators, Bonus Questions

optimality systems and Braille, Optimality Systems, Contexts in Regular Expressions, Regex Replacement as a First Hack, Contexts in Regular Expressions

optimality theory and speech synthesis, More Context: Two Substitutions

optimization techniques for Z-machines, A Plea for Inlining

optimizing with memoization, Optimizing with Memoization

Oracle, sifting through Microsoft's trash, Toss-up Answers

orbital fractal pixmap generator, Most Powerful

orbiting of the earth, Building Your Own, Building Your Own

ordering of rules in generativity/optimality systems, Generativity Systems, Contexts in Regular Expressions, Generativity Systems, Generativity Systems, Regex Replacement as a First Hack, Contexts in Regular Expressions

Orwant, Jon, What Sucks? What Rocks?, The Iterated Prisoner's Dilemma, The Three-Way Prisoner's Dilemma, Toss-up Answers, Bonus Questions, Toss-up Answers, Toss-up Questions, Bonus Questions, Toss-up Questions, Toss-up Questions, Toss-up Questions, Toss-up Questions, Bonus Questions, Toss-up Answers

First Perl/Internet Quiz Show, Toss-up Answers

Fourth Perl/Internet Quiz Show, Toss-up Questions, Toss-up Questions, Toss-up Questions, Toss-up Questions, Bonus Questions, Toss-up Answers

Prisoner's Dilemma, The Iterated Prisoner's Dilemma, The Three-Way Prisoner's Dilemma

Second Perl/Internet Quiz Show, Bonus Questions, Toss-up Answers

Third Perl/Internet Quiz Show, Toss-up Questions, Bonus Questions

"What Languages Suck" program, What Sucks? What Rocks?

Ossipoff, Mike, Sites

output commands (CM17), Simple Output Commands: The CM17

overloading regular expressions, Irregular Regular Expressions, Overloaded

Oxford Advanced Learner's Dictionary, Text-to-Phoneme Conversion, References O'Reilly books and animals on covers, Bonus Questions O'Reilly, Tim, It Seemed Like a Good Idea at the Time Ρ

-p flag, Toss-up Answers

PA system, relay-controlled, Whole House CallerID pack function, Bonus Questions

PadWalker module, The Symbol::Approx::Sub Module paging schemes for minimizing memory consumption, The Joy of vec

painter's algorithm, Layout

pairwise election methods, Pairwise Election Methods, The Algorithms, Condorcet's Method, Copeland's Method, The Algorithms, Using CGI to Spit It All Out generating output with CGI, Using CGI to Spit It All Out

pairwise tally, calculating, The Pairwise Engine, Sites pairwise voting, Pairwise Voting, Sites, Pairwise Voting, The Preference Ballot, Impossibility Theorems, Pairwise Election Methods, Condorcet's Method, Smith's Method, Copeland's Method, Copeland's Method, The Algorithms, The Algorithms, The Pairwise Engine, The Pairwise Engine, References, Sites, Sites, Sites

calculating pairwise tally, Sites Condorcet's method, Condorcet's Method Copeland's method, Copeland's Method Smith's method, Smith's Method

palindromic Perl program, Bonus Questions PalmPilot and Perl, Bonus Answers Papert, Seymour, Toss-up Questions paradoxes possible with preference ballots, Impossibility Theorems

paragraph-at-a-time reformatting, Text::Autoformat paragraphs and autoformat, Paragraphs, Future Features, Justification and Sentencing, Future Features **Parse::RecDescent module, Installing the Modules**

parsers, Other Word Relationships, Link Grammar, Irregular Regular Expressions, Overloaded, Lingua::LinkParser, Irregular Regular Expressions, Overloaded

Lingua::Wordnet and, Other Word Relationships Link Grammar, Link Grammar, Irregular Regular Expressions, Overloaded, Lingua::LinkParser, Irregular Regular Expressions, Overloaded

parsing natural language, I See a Pattern Developing, A Full Lingua::LinkParser Example, Link Grammar, References, A Full Lingua::LinkParser Example part meronyms, Other Word Relationships party table in database schema, The Database Schema passing by reference, arrays, Bonus Questions

passing DNA sequences through analytic steps, Bioinformatics and Perl

patch_change function (MIDI::Simple), Behold MIDI::Simple!

patents, bogus or not, Bonus Questions

pattern matching and Link Grammar parser, I See a Pattern Developing, A Full Lingua::LinkParser Example, Link Grammar, Irregular Regular Expressions, Overloaded, What Sucks, Regex Style, A Full Lingua::LinkParser Example

Pavlich, Dave, Results of the Contest

Payrard, Stephane, Most Creative

pa stub subroutine, Whole House CallerID

PDL (Perl Data Language), The perldl Shell, The perldl Shell, Listing of a Few PDL Functions, Listing of a Few PDL Functions PDL::Core module, functions in, Listing of a Few PDL Functions

PDL::Fit::Polynomial module, Listing of a Few PDL Functions

PDL::Graphics::PGPLOT module, Listing of a Few PDL Functions

PDL::Image2D module, Listing of a Few PDL Functions PDL::Io::Misc module, Listing of a Few PDL Functions PDL ::Primitius module, Listing of a Few PDL Functions

PDL::Primitive module, Listing of a Few PDL Functions

Pearl Jam, Toss-up Answers

Penfield Jackson, Justice Thomas, Toss-up Answers penguins, Toss-up Questions

mascot of Linux, Toss-up Questions

Pepper, Tom, Toss-up Questions

Pepsi, caffeine content of, Bonus Questions

Perceptron neural network, Toss-up Answers

percussion, Percussion, Uniformity, and noop, Using synch, and Some Actual Music, Toss-up Questions

adding, using measure counters, Using synch, and Some Actual Music

software programs for, Toss-up Questions

performance issues for Z-machines, Performance Considerations, Quantity Is Job One, Quantity Is Job One perihelion, Building Your Own

Perl, A Little History, Bonus Questions, Bonus Answers, Happy Birthday!

birthday of, Happy Birthday! newsgroups about, A Little History PalmPilot applications and, Bonus Answers precedence of operators, Bonus Questions PERL (Precision Entry and Reentry Launchings), Perl and Nuclear Weapons Don't Mix

Perl 5.005, Bonus Questions, Toss-up Questions, Toss-up Questions

hybrid data structure available in, Toss-up Questions modules bundled for first time with, Bonus Questions regular expressions, scariest addition to, Toss-up Questions

Perl Adept, Toss-up Questions

Perl compiler, evaluating code using Fathom, The Perl Compiler to the Rescue, Acknowledgments, Future Directions, Acknowledgments

Perl Data Language (PDL), PDL: The Perl Data Language, Where Are We Now?, The perldl Shell, The perldl Shell, Listing of a Few PDL Functions, Listing of a Few PDL Functions, Where Are We Now?

Perl Guru, Toss-up Questions

Perl Mongers chapters, Bonus Questions

Perl Novice, Toss-up Questions

Perl poems from Perl One-Liners, Perl Poem: down.pl

Perl Poetry Contest, The Perl Poetry Contest, Best of Show, The Results, Best Poem Port, Obfuscated Poetry?, Best of Show

Perl programs, Perl-fect Sundials, Building Your Own, Perl-fect Sundials, Building Your Own, How Perl Saved the Human Genome Project, Problems with Perl, How Perl Saved the Human Genome Project, How Perl Saved the Human Genome Project, Bioinformatics and Perl, Problems with Perl, Problems with Perl, Buzzword: Inheritance, Proof in the Pudding: Localizing Web Sites, Red Alert! creating sundials with, Perl-fect Sundials, Building Your Own, Perl-fect Sundials, Building Your Own localizing, Buzzword: Inheritance, Proof in the Pudding: Localizing Web Sites nuclear weapons and, Red Alert! problems with, Problems with Perl saving the human genome project, How Perl Saved the Human Genome Project, Problems with Perl, How Perl Saved the Human Genome Project, How Perl Saved the Human Genome Project, Bioinformatics and Perl, Problems with Perl

Perl warnings, turning on, Toss-up Questions, Lines 0–4: Load Modules

Perl Whirl Quiz Show, The Perl Whirl Quiz Show, Bonus Answers, Toss-up Questions, Toss-up Questions, Toss-up Questions, Bonus Questions, Bonus Questions, Bonus Questions, Bonus Questions, Bonus Questions, Bonus Answers, Bonus Answers

Perl Wizard, Toss-up Questions

Perl Wizard's Quiz, The Perl Wizard's Quiz, The Perl Wizard's Quiz, The Perl Wizard's Quiz, Answers, Answers, Answers

Perl, Martin, Nobel prize in Physics winner, Toss-up Answers

Perl/Internet Quiz Show, Toss-up Questions, Bonus Questions

First, Toss-up Questions, Bonus Questions

Perl/Tk solitaire game, Tktk: A Perl/Tk Solitaire Game, Finishing Touches, Finishing Touches perl2exe program, MisterHouse Perl2Exe utility, Games in Perl peridi shell, The peridi Shell, The peridi Shell, The peridi Shell, The peridi Shell

Perle Systems, Toss-up Answers

perllocal.pod, Toss-up Answers

perllocale documentation, Locale-Based Sorting, Toss-up Answers

PerlMX mail filtering, Bonus Questions, Bonus Questions PerlScript package, Toss-up Answers

PERL_EMERGENCY_SBRK, Toss-up Questions

pertainyms, Other Word Relationships

Peters, Les, The First Circle: Print "The Perl Journal", The Old Standby

PGPLOT graphics module functions, Listing of a Few PDL Functions

phase meronyms, Other Word Relationships

phlegmatic fluid, **Bonus Questions**

Phoenix grammars, What Are You Thinking?

Phoenix, Tom, The Regulars, Bonus Answers

phone calls, monitoring with caller ID, Whole House CallerID

phonebox (synthesizer), Text-to-Phoneme Conversion

phonemes, The Out-of-Vocabulary Problem: Synthesis in One s///e, Text-to-Phoneme Conversion, More Context: Two Substitutions, More Context: Two Substitutions, More Context: Two Substitutions

context-dependent sounds, More Context: Two Substitutions

text-to-phoneme conversion, Text-to-Phoneme Conversion, More Context: Two Substitutions, More Context: Two Substitutions

phonetic rules for natural language writing systems, Generativity Systems photographs, TPJ cover, TPJ #1: The Camel, TPJ #2: The Pearl, TPJ #4: Etch-a-Sketch., TPJ #5: Commodities, TPJ #6: Scrabble, TPJ #7: Spiderball, TPJ #8: The Coffee Cup Fiasco, TPJ #8: The Coffee Cup Fiasco, TPJ #9: Drummer/Coder Wanted, TPJ #10: The Underwood Typewriter, TPJ #11: The Conspiracy, TPJ #13: Dance Remixes, TPJ #14: Outlook Not So Good, TPJ #15: Braille blocks, TPJ #16: e. e. cummings' Gravestone, TPJ #17: Napster, TPJ #18: Spam, TPJ #19: Monopoly Money

Phusion genome assembly software, How Perl Saved the Human Genome Project

Physics Nobel prize winner, Martin Perl, Toss-up Answers

PICS (Platform for Internet Content Selectivity), Toss-up Questions

Pierce, Clinton, Haiku

Pike, Lori, Bonus Questions

Pike, Rob, Would You Like to Play a Game?, Toss-up Answers

pilcrow symbol, Bonus Questions

pilfer command (rezrov), Pilfer

Pinyan, Jeff, Best "The Perl Journal"

pipe-based I/O streams and DNA sequencing, Bioinformatics and Perl

pitch (note property), Easy Things Easy, Percussion, Uniformity, and noop

percussion and MIDI channel 9, Percussion, Uniformity, and noop

plaintext, reformatting with autoformat, Quoting, Lists, Widows

Plan 9 operating system, Bonus Questions questions about, Bonus Questions planarity and Link Grammar parser, Link Grammar Planetfall game, Pilfer, Toss-up Questions plausibility of simulated typos, How to Identify Words, How to Identify Words Pliny the Elder, Toss-up Questions plurality voting, Pairwise Voting, The Algorithms pluralizing nouns, rules for, Morphology: Word Form and Structure poems, The Perl Poetry Contest, Best of Show, The Categories, Best Poem Port, A Perl Program That Generates Poetry, Best of Show, Best of Show from Perl Poetry Contest, The Perl Poetry Contest, Best of Show, The Categories, Best Poem Port, A Perl Program That Generates Poetry, Best of Show, Best of Show

Pointer, Robey, Bots and Infobots Pollock, Bill, Best "The Perl Journal" polysemy, concept of, Movin' It Up a Level Pontifex cryptosystem, The Pontifex Cryptosystem portion meronyms, Other Word Relationships POSIX module, Toss-up Answers pound perl (#perl), Bonus Questions Powell, David, Most Creative power consumption of laptops, Toss-up Answers power line modulation (X10), Home Automation: The X10 Nitty-Gritty, Bidirectional I/O with the CM11 bidirectional I/O with CM11, Bidirectional I/O with the CM11

Prakash, Vipul Ved, Most Powerful Pratchett, Terry, Toss-up Questions pre-recorded speech, Pre-Recorded Sentences precedence of Perl operators, Bonus Questions Precision Entry and Reentry Launchings (PERL), Perl and Nuclear Weapons Don't Mix preference ballots, The Hare Method Hare method, The Hare Method

pregnant chads, Secure Internet Voting

Prisoner's Dilemma, The Prisoner's Dilemma, The Prisoner's Dilemma, The Iterated Prisoner's Dilemma, The Prisoner's Dilemma Programming Contest, The Prisoner's Dilemma Programming Contest, Results of the Contest, Results of the Contest, Results of the Contest, Results of the Contest

programming contest, The Prisoner's Dilemma Programming Contest, Results of the Contest, Results of the Contest, Results of the Contest

Process_Item (mh object), Objects in the Home Project Gutenberg, How to Identify Words, How to Identify Words prototyping, using Perl for, Other Uses for Perl pseudohash data structure, Toss-up Answers Psycho Killer, qu'est que c'est, Using synch, and Some Actual Music punk rocker smiley, Toss-up Questions Purity method, Toss-up Questions purl (infobot), Are You Spongeworthy?, Are You Spongeworthy?, You Can't Do That in Public!, You Can't Do That in Public!, What Are You Thinking?, A Sense of Play Purple Moon, now-defunct company, Bonus Answers

Puzzle Bobble game, Twitch Games

Pyramid solitaire game, Tktk: A Perl/Tk Solitaire Game

Pytho, temple at Delphi, Toss-up Questions

Q

quadrathorpe symbol, Bonus Questions Quadrivium (group of four liberal arts), Bonus Questions Quake logfile, accessing, Twitch Games quantum computing, Bonus Questions Queen Elizabeth II, Toss-up Answers quicksort sorting algorithm, Toss-up Answers quizzes, Toss-up Questions, Toss-up Questions, Sample Questions

rating your knowledge, Toss-up Questions sample questions, Sample Questions unnaturally high scores point to cheating, Toss-up Questions

quotations and autoformat, Quotations quoted text and autoformat, Quoting

QWERTY keymap vs. Dvorak keymap, Simulating Typos with Perl, References, Simulating Typos with Perl, Simulating the Typos, Simulating the Typos, How to Identify Words, How to Identify Words, Typos in Other Languages, Typos in Other Languages, References QWERTZ keyboards, Toss-up Questions R

radio buttons, Lines 72–104: The voting_matrix Subroutine, Lines 124–165: The validate Subroutine

creating for candidates, Lines 72–104: The voting matrix Subroutine

validating selection of, Lines 124–165: The validate Subroutine

Radio Shack Armatron on TPJ cover, TPJ #4: Etch-a-Sketch.

Railroad Tycoon game, Toss-up Questions

Ramos, Art, The Third Circle: Be Creative

random organisms, Genetic Algorithms, The Genetic Code, Assembling an Organism, Assembling an Organism, Assembling an Organism, Survival of the Fittest, Sex and Mutation, Survival of the Fittest, Survival of the Fittest, Sex and Mutation, Sex and Mutation

assembling, Assembling an Organism creating genetic code for, The Genetic Code, Assembling an Organism, Assembling an Organism culling from list, Survival of the Fittest ranking by fitness, Survival of the Fittest, Sex and Mutation, Survival of the Fittest, Sex and Mutation, Sex and Mutation

Raymond, Eric S., References, Bonus Questions read assemblers, Other Uses for Perl readability of Perl programs, grading, Existing Measures, Acknowledgments, Existing Measures, Why This Should Be Hard To Do, The Perl Compiler to the Rescue, The Perl Compiler to the Rescue, Acknowledgments ReadMode function, Bonus Questions reads, sequencing DNA as, How Perl Saved the Human Genome Project read all method. A Smart TV Guide read score function (MIDI::Simple), **Behold MIDI::Simple!** real-time service, **Toss-up Questions** reassemble rules in chatterbot configuration files. Chatbot::Eliza records, multiple sequence, **Bioinformatics and Perl** recursively building organisms, Assembling an Organism, Sex and Mutation. Survival of the Fittest. Sex and **Mutation** recycling memory, Toss-up Questions red beans and rice on TPJ cover, TPJ #5: Commodities Redford, John, Perl-fect Sundials ref function, Bonus Questions, Bonus Answers references, Future Features, References **MIDI and computer music, Future Features** voting systems/elections, References

reflowing text sent to autoformat, Text::Autoformat, Lists, Paragraphs, Lists, Lists

reformatting text, Lazy Text Formatting, Future Features, Text::Autoformat, Paragraphs, Justification and Sentencing, Future Features, Future Features, Future Features

refrigerator and Coke can temperature, Toss-up Answers registered voters, About Secure Elections

registration table in database schema, The Database Schema, Generating VRNs

adding VRNs to, Generating VRNs

registration_number subroutine (e-ballot), Lines 105–116: The registration_number Subroutine regular expressions, Lexical Synthesis in One s///, Rules as Exceptions, Rules as Exceptions, It's Alive!, I See a Pattern Developing, A Full Lingua::LinkParser Example, Link Grammar, Link Grammar, Lingua::LinkParser, Lingua::LinkParser, Irregular Regular Expressions, Overloaded, Irregular Regular Expressions, Overloaded, What Sucks? What Rocks?, What Sucks, Regex Style, What Sucks, Regex Style, There's Lots More Here, A Full Lingua::LinkParser Example, Matching Nicknames, The CGI Script, Toss-up Questions, Toss-up Questions, Bonus Questions

converting English into kanji, The CGI Script

creating grouping without creating backreference, Toss-up Questions

Link Grammar parser and, I See a Pattern Developing, A Full Lingua::LinkParser Example, Link Grammar, Link Grammar, Lingua::LinkParser, Lingua::LinkParser, Irregular Regular Expressions, Overloaded, What Sucks? What Rocks?, What Sucks, Regex Style, What Sucks, Regex Style, There's Lots More Here, A Full Lingua::LinkParser Example

matching human names and, Matching Nicknames metacharacter matching beginning of string, Bonus Questions

overloading, Irregular Regular Expressions, Overloaded

rules as exceptions, Rules as Exceptions scariest addition to, Toss-up Questions speech synthesis and, Lexical Synthesis in One s/// used for Braille encoding, Rules as Exceptions, It's Alive!

Rehfeld, Georg, Results of the Contest

relative notes, specifying, "Relative" Notes relay-controlled PA system, Whole House CallerID renumbering paragraphs with autoformat, Lists rep pattern function, Simulating the Typos **Resnick, Mitch, Results of the Contest Description** Framework (RDF). Resource Toss-up **Ouestions** resume vs. résumé, using correct sort order for, English: **Résumé and Resume** Return of the King, Bonus Answers rezrov interpreter, Enter the Camel, References **RFC Internet standards**, **Toss-up Ouestions** rfits function (PDL), The perild Shell rhymes, searching for with Perl, Searching for Rhymes with Perl, Accommodating Another Notation, Preparing the Data, Searching the Prepared Data, Coping with (Syllabic) Stress, Coping with (Syllabic) Stress, A Simple mpron Searcher, Accommodating Another Notation Rhythm Lab program, Toss-up Questions Rich, Ken, Best "The Perl Journal" right ascension, Building Your Own **Riker, William H., References Rinehart, Daniel, Most Powerful, Best "The Perl Journal" Rivest, Ron, Toss-up Questions RJ** jacks used for Ethernet connections, **Bonus Questions** robbot and #riskybus channel, Bots and Infobots **Rocca, Mo, Toss-up Questions** rocks, what, What Sucks? What Rocks? Rokicki, Tomas, Most Powerful romaji, The Web Roman numerals, renumbering lists with, Lists **Romantic period, Music to My Ears Rossler, Larry, The Regulars**

rotation of the earth, Building Your Own, Building Your **Own, Building Your Own, Building Your Own, Building** Your Own, Building Your Own, Building Your Own rounds and measure counters, Mod, Canons, and Rounds Row, Row, Row Your Boat, Mod, Canons, and Rounds **RS232** serial port. **Toss-up Ouestions RSA, Bonus Ouestions** RSA-in-three-lines-of-Perl, TPJ #3: RSA on Greenbar **RSVP protocol, Toss-up Answers RSYNC utility, Toss-up Answers** run-time vs. compile-time, Bonus Questions Russian, translating English into, A Localization Horror Story: It Could Happen to You, Buzzword: Inheritance, **Buzzword: Concision** rvals function (PDL), The peridi Shell **Ryan, Kim, Installing the Modules Rvan, Mark, World Wide Wasteland**

S

s/// expressions, Morphology: Word Form and Structure, Morphological Inflections and Derivations, Implementing It, Representing a Single Rule, Representing Many Rules, Telling Good from Bad, Implementing It

derivation and inflection rules, Morphological Inflections and Derivations, Implementing It, Representing a Single Rule, Representing Many Rules, Telling Good from Bad, Implementing It

inflecting nouns/verbs, Morphology: Word Form and Structure

Salzenberg, Chip, Bonus Answers Samba, developer of, Toss-up Questions Sanderson, Hugh, Best "The Perl Journal" sanguine fluid, Bonus Questions Sarathy, Gurusamy, Toss-up Questions SAX (Simple API for XML), Toss-up Answers scalar variables, The Joy of vec, Bonus Questions, Toss-up Questions, Toss-up Questions

holding the time when program begins, Toss-up Questions memory consumption and, The Joy of vec serpentine, Toss-up Questions verbose alternatives to, Bonus Questions

schema, database, for tallying votes, The Database Schema, The Database Schema, The Database Schema, Generating VRNs

Schneier, Bruce, Toss-up Questions, Toss-up Answers, About Secure Elections

Schwartz, Randal, The Regulars

Schwartzian transform, Contexts in Regular Expressions

Schwern, Michael, It Seemed Like a Good Idea at the Time

Scooter web crawler, Toss-up Answers score objects, Behold MIDI::Simple!

scores, Behold MIDI::Simple!, Percussion, Uniformity, and noop, The Object-Oriented Interface, Using synch, and Some Actual Music, Mod, Canons, and Rounds, Mod, Canons, and Rounds, Mod, Canons, and Rounds

creating music with synch function, Using synch, and Some Actual Music, Mod, Canons, and Rounds, Mod, Canons, and Rounds, Mod, Canons, and Rounds manipulating several, using OOP interface, The Object-Oriented Interface

noop function and, Percussion, Uniformity, and noop state variables used in, Behold MIDI::Simple!

Scour, dotcom that indexes your hard drive for public access, Toss-up Questions

Scrabble game on TPJ cover, TPJ #6: Scrabble search term, most popular, Toss-up Questions searching for rhymes with Perl, Searching for Rhymes with Perl, Accommodating Another Notation, Searching the Prepared Data, Accommodating Another Notation seasons, haiku should pertain to, Haiku

Second Perl/Internet Quiz Show, The Second Perl/ Internet Quiz Show, Toss-up Answers, Toss-up Questions, Toss-up Questions, Toss-up Questions, Bonus Questions, Bonus Questions, Toss-up Answers, Toss-up Answers

secret messages in plaintext, hiding, TPJ #3: RSA on Greenbar

secure Internet voting, Secure Internet Voting, Tallying the Votes, About Secure Elections, The E-Ballot, Tallying the Votes seek function, Toss-up Questions

Seibel, Peter, Results of the Contest

Sejnowski, T.J., References

semantic phrase parsers, What Are You Thinking?

send cm11 method, Bidirectional I/O with the CM11

send_cm17 method, The ControlX10::CM17 and ControlX10::CM11 Modules

Seneff, Stephanie, References

sense numbers for synsets, Concepts in Wordnet, Basic Usage, Other Word Relationships

looking up correct numbers, Other Word Relationships

sequencing technology, DNA, How Perl Saved the Human Genome Project

serial ports, Home Automation: The X10 Nitty-Gritty, What's in a Command?, The ControlX10::CM17 and ControlX10::CM11 Modules, The ControlX10::CM17 and ControlX10::CM11 Modules, What's in a Command?, What's in a Command?, Bidirectional I/O with the CM11

CM11 controller and, Bidirectional I/O with the CM11 CM17 controller and, Home Automation: The X10 Nitty-Gritty, What's in a Command?, The ControlX10::CM17 and ControlX10::CM11 Modules, The ControlX10::CM17 and ControlX10::CM11 Modules, What's in a Command?, What's in a Command?

SerialPort modules, What's in a Command?, What's in a Command?, What's in a Command?, What's in a Command?, What's in a Command?

SerialStub.pl program, The ControlX10::CM17 and ControlX10::CM11 Modules Serial_Item (mh object), Objects in the Home Sethuraman, Krishna, Best Four-Line Signature set_byte_at method (rezrov), The Joy of vec set_watch method, A Smart TV Guide sex, Genetic Algorithms

hot algorithmic, Genetic Algorithms

Shamir, Adi, Toss-up Ouestions shark, jumping the, Toss-up Answers Shatner, William, Bonus Answers Sheiness, Frank, Best "The Perl Journal" shell interface for PDL, The perild Shell, Listing of a Few PDL Functions. The perild Shell, Listing of a Few PDL Functions numbers, Library of Congress, Toss-up shelving **Ouestions** Shergold, Craig, Toss-up Answers Sherman, Aaron, Most Powerful, Best "The Perl Journal" Shift-JIS encoding, Taking It a Step Further **"OWERTY"** keymap vs. Dvorak Sholes keymap. **Simulating Typos with Perl, References, References** shuffling cards using Fisher-Yates algorithm, Shuffling Shute, James, Most Creative Sidebotham, Bob, Most Creative Sierpinski fractals in 504 bytes, World Wide Wasteland signal handlers for trapping Ctrl-Z or Ctrl-C, Toss-up Questions Sigue Sigue Sputnik, TPJ #9: Drummer/Coder Wanted Simple Mail Transfer Protocol (SMTP), Bonus Answers

simulating typos with Perl, Simulating Typos with Perl, References, Simulating the Typos, Simulating the Typos,
How to Identify Words. How to Identify Words. Typos in **Other Languages, Typos in Other Languages, References** single-character Perl variable, Toss-up Questions Sklyarov, Dmitry, Toss-up Answers slash/underscore notation in metrical patterns, Coping with (Syllabic) Stress, Accommodating Another Notation Slashdot, software underlying, Bonus Questions Sleator, Daniel, References slow speed.pl program, What's in a Command? **Smart Go Format files, Strategy Games** smart houses, Home Automation with MisterHouse, HA! Perl?, Talking and Listening, A Smart TV Guide smilev face :-), Toss-up Ouestions **Smileys, Bonus Questions** Smith, Matthew, The Old Standby Smith's method for pairwise voting, Smith's Method smurfing, ways of defeating, Toss-up Answers SNMP (Simple Network Management Protocol), Toss-up **Ouestions** -snoop-obj command-line switch, Tinkering with the **Z-Machine** (Simple Object Access Protocol), SOAP Toss-up **Ouestions** Socket Item (mh object), Objects in the Home solidus symbol, Bonus Questions solitaire game (Tktk), The Rules, Finishing Touches, A First Cut, Layout, Would You Like to Play a Game?, Laying the Base, Pickup Lines, Oops!, Finishing Touches, **Finishing Touches** sonnets, generated by Perl program, A Perl Program That **Generates Poetry** Sony Discman on TPJ cover, TPJ #13: Dance Remixes

Sørensen, Poul, Best Four-Line Signature, Best "The Perl Journal", Best "The Perl Journal"

sort criterion, Default sort Versus "Normal" English Sorting, Default sort Versus "Normal" English Sorting, Locale-Based Sorting

bi-level sorting and, Locale-Based Sorting making it predictable, Default sort Versus "Normal" English Sorting

sort function, performing crime of omission, Toss-up Answers

sort order for Spanish, defining, Sorting it All Out Sort::ArbBiLex module, Sorting it All Out

sorting, Locale-Based Sorting, Spanish: Cana y Caña according to Spanish conventions, Spanish: Cana y Caña

locale-based, Locale-Based Sorting

sorting algorithm used by Perl, Toss-up Questions soundex matching, Matching Nicknames, How Does It Work?

for misspelled words, How Does It Work? for names, Matching Nicknames

source filtering technique, Bonus Answers SourceForge, Toss-up Answers, Toss-up Questions SHARC project, Toss-up Questions

SpaceWar game, Twitch Games

SPAM (Standby Potential Armageddon Meeting), Red Alert!

SPAM, can of, on TPJ cover, TPJ #18: Spam

Spanish, Default sort Versus "Normal" English Sorting, Spanish: Cana y Caña, Bi-Level Sorting to the Rescue, English: Résumé and Resume, Optimizing with Memoization, Sorting it All Out

defining sort order for, Spanish: Cana y Caña, Sorting it All Out

internationalized sorting and, Default sort Versus "Normal" English Sorting, Bi-Level Sorting to the Rescue, English: Résumé and Resume, Optimizing with Memoization

special variable holding name of Perl program, Bonus Questions

speech log file, example of, Say What?

speech svnthesis. Speech Synthesis. **References. Pre-Recorded Sentences, Pre-Recorded Sentences, Lexical** Synthesis in One s///, The Out-of-Vocabulary Problem: Synthesis in One s///e, Text-to-Phoneme Conversion, Conversion. More Text-to-Phoneme **Context:** Two Substitutions. Text-to-Phoneme Conversion. More Context: Two Substitutions. More **Context:** Two Substitutions, More Context: Two Substitutions, What **Else?**, **References**

pre-recorded speech and, Pre-Recorded Sentences

regular expressions and, The Out-of-Vocabulary Problem: Synthesis in One s///e, More Context: Two Substitutions

text-to-phoneme conversion, Text-to-Phoneme Conversion, More Context: Two Substitutions, More Context: Two Substitutions

tokenizing input, Lexical Synthesis in One s///

spell-check feature of Google, Afterword

Spencer, Henry, Bonus Questions spiderball on TPJ cover, TPJ #7: Spiderball spiral galaxy, Messier 51, The peridi Shell splice function, **Bonus Questions** split function, Bonus Questions Spragg, Adam, Create a Diversion spread spectrum wireless protocol, Toss-up Ouestions spreadtoemail (virus function), Toss-up Ouestions sprintf function, Toss-up Answers Sproat, Richard, References squeaky stairs, simulating with MisterHouse, Squeaky Stairs Squid web proxy cache, Toss-up Ouestions Stallman, Richard, Toss-up Answers standards, prices for, Bonus Questions Star Wars leitmotif, adding to .covrc file, Is There a Poet in the House? start multipart form function (CGI), Lines 61-71: The generate ballot Subroutine start port.pl program, What's in a Command? stat function, Toss-up Questions statements per subroutine, measuring, The Basic Units stats function (PDL), The perild Shell steganography, TPJ #3: RSA on Greenbar Stein, Lincoln D., The CGI Script Stephenson, Neal, Toss-up Questions, The Pontifex Cryptosystem storage problems for DNA sequences, How Perl Saved the **Human Genome Project STORE function**, **Bonus Questions** story files for computer games, What's It Do?, The Joy of vec

loading into Z-machine's memory, The Joy of vec

StoryFile.pm package, Teleport Stowe, Jonathan, The Regulars

strategy games, Strategy Games

stress patterns in words, Coping with (Syllabic) Stress, A Simple mpron Searcher, Coping with (Syllabic) Stress, Coping with (Syllabic) Stress, A Simple mpron Searcher, A Simple mpron Searcher

String::Approx module, Installing the Modules, Matching Nicknames, The Sub::Approx Module, A Little Rocket Science

stroke_groups function, Simulating the Typos

stub interface, creating, Toss-up Questions

study function, Toss-up Answers

Sturdy, John C. G., References

style, Perl, Perl Style, Acknowledgments, The Tool, Usage, Why This Should Be Hard To Do, Future Directions, Acknowledgments

Sub::Approx module, The Sub::Approx Module

subroutines, calling with misspelled names, Correcting Typos with Perl, Afterword, How Does It Work?, How Does It Work?, How Does It Work?, The Sub::Approx Module, Afterword

substance meronyms, Other Word Relationships substr function, Bonus Ouestions

sucks, what, What Sucks? What Rocks?, What Sucks, Regex Style, A Full Lingua::LinkParser Example

Lingua::LinkParser example, A Full Lingua::LinkParser Example

using regular expressions, What Sucks, Regex Style

summer solstice, Perl-fect Sundials, Building Your Own, Building Your Own expressing the sun vector, Building Your Own tracking gnomon shadow on, Building Your Own

sun vector, Building Your Own, Building Your Own sundials, Perl-fect Sundials, Building Your Own, Building Your Own, Building Your Own

background information on, Perl-fect Sundials building your own, Building Your Own, Building Your Own, Building Your Own

Suomi, land of most Internet sites per capita, Toss-up Answers

supercomputing Linux clusters, Toss-up Questions

Sussman, Gerald Jay, References

swapping chunks of data to minimize memory consumption, What's It Do?

syllables, Coping with (Syllabic) Stress, A Simple mpron Searcher, Coping with (Syllabic) Stress, Coping with (Syllabic) Stress, A Simple mpron Searcher

stressed vs. unstressed, Coping with (Syllabic) Stress, A Simple mpron Searcher, Coping with (Syllabic) Stress, Coping with (Syllabic) Stress, A Simple mpron Searcher

symbol table hashes (stashes), How Does It Work?

Symbol::Approx::Scalarmodule,TheSymbol::Approx::Sub Module

Symbol::Approx::Sub module, Correcting Typos with Perl, Afterword, How Does It Work?, How Does It Work?, How Does It Work?, The Symbol::Approx::Sub Module, Afterword Symbol::Approx::Sub::Google module, Afterword symbolic debugger in Perl, Bonus Questions synch function (MIDI::Simple), Behold MIDI::Simple! synsets (sets of synonyms), Concepts in Wordnet, What's Next?, Converting the Data, Basic Usage, Basic Usage, Basic Usage, Basic Usage, Movin' It Up a Level, Movin' It Up a Level, Other Word Relationships, Other Useful Functions, What's Next?, What's Next?

adding meronyms to, Other Word Relationships sense numbers for, Other Word Relationships word relationships and, Other Word Relationships, Other Word Relationships, Other Word Relationships, Other Word Relationships, Other Word Relationships writing a password checker, Basic Usage

syntax trees, The Genetic Code, Assembling an Organism, Survival of the Fittest, Sex and Mutation, Sex and Mutation, Sex and Mutation

assembling organisms, Assembling an Organism calculating output of, Survival of the Fittest, Sex and Mutation, Sex and Mutation mutating organisms, Sex and Mutation

System V IPC structures, Bonus Questions systems integration issues and Perl, Other Uses for Perl Т

-T command-line switch, Toss-up Questions -T flag, Bonus Questions tables in database schema, The Database Schema tag/value pairs and BoulderIO, Bioinformatics and Perl tagmemics, Toss-up Answers tags used in Tktk solitaire game, Making the Moves, Oops!, Pickup Lines, Oops! taint checking, turning on, Lines 0–4: Load Modules tally table in database schema, The Database Schema, Lines 166–205: The enter_ballot Subroutine tallying votes using database schema, The Database Schema, Tallying the Votes tanka, The Tao of Haiku tao of haiku, The Tao of Haiku

tattagad on forgerm which Parl program was

tattooed on forearm, which Perl program was, Toss-up Questions

tau lepton, discovery of, by Martin Perl, Toss-up Questions, Toss-up Answers

Taylor, Dennis, Acknowledgments

Taylor, Garry, Create a Diversion

Tcl program, most popular, Toss-up Questions

telephone calls, monitoring with caller ID, A Smart TV Guide

teleport command (rezrov), Tinkering with the Z-Machine, Bamf, Teleport, Teleport, Teleport, Bamf Temperley, Davy, References

Term::Readkey module, Bonus Questions

terminals used in genetic algorithms, The Genetic Code

Terse method, **Toss-up Questions**

Tesla, Nikola, Toss-up Questions

TeX text formatting package, Toss-up Answers, Toss-up Questions

inventor of, Toss-up Questions

text formatting and autoformat, Text::Autoformat, Future Features, Text::Autoformat, Paragraphs, Widows, Future Features, Future Features

Text To Speech (TTS) and home automation, MisterHouse, Talking and Listening

text, searching for with Perl, Searching for Rhymes with Perl, Accommodating Another Notation, Accommodating Another Notation

text-to-phoneme conversion, Text-to-Phoneme Conversion, More Context: Two Substitutions, More Context: Two Substitutions

Text::Metaphone module, Installing the Modules, Matching Nicknames, The Sub::Approx Module

Text::Soundex module, Installing the Modules, Matching Nicknames, How Does It Work?

Text::StripHigh module, Future Features

The Cathedral and the Bazaar, Future Directions, Bonus Answers

The Lurking Horror game, Toss-up Questions

The Onion, Toss-up Answers

The Psychology of Computer Programming, Percussion, Uniformity, and noop

The Semantic web project, Toss-up Answers

The Vaults of Parnassus, Bonus Answers

The Victorian Internet, Toss-up Questions

Third Obfuscated Perl Contest, Most Powerful, Best of Show, Most Powerful, Most Creative, Best "The Perl Journal", Best of Show

Third Perl/Internet Quiz Show, The Third Perl/Internet Quiz Show, Bonus Answers, Bonus Questions, Bonus Answers third-party candidates, difficulty in getting support for, Pairwise Voting

Three-Way Prisoner's Dilemma, The Three-Way Prisoner's Dilemma

thwarted-majorities paradox, Impossibility Theorems Tibetan, simulating typos in, Typos in Other Languages ticks (time measurement in music), Behold MIDI::Simple! tie function, Toss-up Questions, Bonus Questions Tie::BikePower module, Bonus Questions tiebreakers, Default sort Versus "Normal" English Sorting, Bi-Level Sorting to the Rescue in sort criteria, Default sort Versus "Normal" English Sorting, Di Level Sorting to the Rescue

Sorting, Bi-Level Sorting to the Rescue

tied function, Bonus Answers, Toss-up Questions tied hashes, Dictionary Database **TIESCALAR function, Bonus Questions** Timer (mh object), Objects in the Home time cron function, A Smart TV Guide tip vector, calculating, Building Your Own title casing text with autoformat, Justification and Sentencing **TiVo, Toss-up Answers** Tk interface for MisterHouse, MisterHouse **TK-707 program, Toss-up Questions Tk::bind object, Making the Moves Tk::**Canvas object, Lavout Tktk (Tk timekiller) solitaire game, Finishing Touches .to domain location, Toss-up Questions tokenizing speech input, Pre-Recorded Sentences tokens, The Basic Units, Constructing a Word Frequency List, Constructing a Word Frequency List, Morphological Inflections Implementing and **Derivations.** It. Representing a Single Rule, The Key Insight!, Implementing It

applying single/multiple rules, Morphological Inflections and Derivations, Implementing It, Representing a Single Rule, The Key Insight!, Implementing It

excluding low-frequency, Constructing a Word Frequency List

frequency lists and, Constructing a Word Frequency List

per expression, measuring, The Basic Units

Tolkein, J.R.R., Bonus Questions

Tonga, location of .to domain name, Toss-up Answers top-level domains ending in two-letter country codes, Bonus Questions

Torkington, Nathan, Toss-up Answers, The Fifth Obfuscated Perl Contest

Tower of the Winds (Greek sundial), Perl-fect Sundials trace editors, Other Uses for Perl

transducers, Text-to-Phoneme Conversion, More Context: Two Substitutions, Text-to-Phoneme Conversion, More Context: Two Substitutions

text-to-phoneme, Text-to-Phoneme Conversion, More Context: Two Substitutions, Text-to-Phoneme Conversion, More Context: Two Substitutions

transliteration and autoformat, Future Features Transmission Control Protocol (TCP), Bonus Answers Traveling Salesman Problem, solving, Toss-up Questions traversing hypernyms, Basic Usage, Other Word Relationships

Treese, Win, Toss-up Questions, Toss-up Answers

Tridgell, Andrew, Toss-up Questions

trigraphs (three-letter clusters), How to Identify Words, Typos in Other Languages, Typos in Other Languages

triphones and speech synthesis, More Context: Two Substitutions

Trivium (group of three liberal arts), Bonus Questions

TTS (Text To Speech) and home automation, MisterHouse, Talking and Listening

Tux the penguin, Toss-up Answers

TV shows, announcing via MisterHouse, A Smart TV Guide

twitch games, Twitch Games

Two Towers, Bonus Answers

two-party duopoly, Pairwise Voting

type checking, lack of in Perl, Problems with Perl

typeglobs, Why Was It Written?, Afterword, How Does It Work?, How Does It Work?, Afterword

typewriter on TPJ cover, TPJ #10: The Underwood Typewriter

Typewriting Behavior, Simulating the Typos

typos (typographical errors), Typos in Other Languages, Typos in Other Languages, Correcting Typos with Perl, Afterword, Why Was It Written?, How Does It Work?, How Does It Work?, The Symbol::Approx::Sub Module, Afterword

correcting, with Perl, Correcting Typos with Perl, Afterword, Why Was It Written?, How Does It Work?, How Does It Work?, The Symbol::Approx::Sub Module, Afterword

simulating, with Perl, Typos in Other Languages, Typos in Other Languages typo_on_word function, Simulating the Typos, How to Identify Words, Simulating the Typos, Simulating the Typos, How to Identify Words U

U.S. Census accepting online forms, Bonus Questions, Toss-up Answers, Bonus Questions Ullman, Cayce, Most Creative Ultima Online game, Toss-up Questions Ultramaster RS101 program, Toss-up Questions undersea cable, miles of, Toss-up Questions Underwood typewriter on TPJ cover, TPJ #10: The Underwood Typewriter undo command, Universal Command Set

rezrov game interpreter, Universal Command Set

Unicode character set, Localizing Your Perl Programs, Toss-up Answers

Uniform Dispute Resolution Policy, Toss-up Questions Uniform Resource Locator (URL), Bonus Answers

uniformity, violation of, Percussion, Uniformity, and noop union method (Lingua::Wordnet:: Analysis), Movin' It Up a Level

United States Patent and Trademark Office, Bonus Questions

universal command set for Z-machines, Universal Command Set

Unix Database Package and Lingua::Wordnet, Converting the Data

Unix pipes and DNA sequencing, Bioinformatics and Perl unlink function, Toss-up Questions

Unsolicited Commercial Email (UCE), TPJ #18: Spam unstressed vs. stressed syllables in words, Coping with (Syllabic) Stress, A Simple mpron Searcher, Coping with (Syllabic) Stress, Coping with (Syllabic) Stress, A Simple mpron Searcher

url (infobot), What Are You Thinking? use constant pragma, A Little Rocket Science Usenet newsgroups, A Day in the Life of comp.lang.perl.misc, The Future of comp.lang.perl.misc, A Little History, The Day's Weather Report, The Future of comp.lang.perl.misc, The Future of comp.lang.perl.misc user interfaces, abstracting using Z-machine opcodes, Interface Abstraction

/usr/dict/words file, Telling Good from Bad uuencoding attachments, Uuencoding Attachments V

validate subroutine (e-ballot), Lines 23–26: Handle a Submitted Ballot, Lines 124–165: The validate Subroutine Valley of Gehennom, Toss-up Questions

van Dongen, Stijn, Most Creative

vandalism issues for bots, What? We Ordered No Pizzas! VAR (value-added reseller), Bonus Questions

vec function (rezrov), The Joy of vec, The Joy of vec, The Joy of vec, The Joy of vec

vector checks and DNA sequencing, Bioinformatics and Perl

vector for the sun, Building Your Own, Building Your Own

Veeraraghavan, Sriranga, Best Four-Line Signature vernal equinox, Building Your Own, Building Your Own, Building Your Own, Building Your Own, Building Your Own

version command (rezrov), Teleport

versioning system, concurrent, Toss-up Questions

vertical blanking interval, Bonus Questions

vi vs. Emacs, Bonus Questions, Bonus Answers

Village of Hommlet, Toss-up Questions

vinyl records on TPJ cover, TPJ #17: Napster

virtual computer (Z-machine), The Rezrov Infocom Game Interpreter

virus, computer, Toss-up Questions, Toss-up Answers ILOVEYOU, Toss-up Answers

VISCII character set, Localizing Your Perl Programs Visual Python plug-in, Bonus Questions Voice_Cmd (mh object), Objects in the Home voltage surges, protecting against, Toss-up Questions volume (note property), Easy Things Easy vote subroutine (e-ballot), Lines 117–123: The vote Subroutine

votes, About Secure Elections, Generating VRNs

tallying, using database schema, About Secure Elections, Generating VRNs

voting, Toss-up Questions, Pairwise Election Methods, Condorcet's Method, The Algorithms, Using CGI to Spit It All Out

not yet ready for Internet, Toss-up Questions pairwise, Pairwise Election Methods, Condorcet's Method, The Algorithms, Using CGI to Spit It All Out

voting_matrix subroutine (e-ballot), Lines 72–104: The voting_matrix Subroutine

VR (Voice Recognition), Say What? sample session, Say What?

VRNs (Voter Registration Numbers), About Secure Elections, Generating VRNs, The E-Ballot, Generating VRNs, The E-Ballot

entering into CEA database, Generating VRNs generating blocks of, Generating VRNs, The E-Ballot, The E-Ballot W

-w flag, Toss-up Questions

-W flag, Bonus Questions

Wall, Larry, Bonus Questions, Toss-up Questions, Bonus Questions

famous sayings of, Bonus Questions father of Perl, Toss-up Questions sanguine fluid and, Bonus Questions

Wallace, Sanford, Toss-up Answers Wallace, Shawn, Best "The Perl Journal" Walshaw, Chris, References wantarray function, Toss-up Questions Ward, Wayne, What Are You Thinking? warnings, Perl, turning on, Toss-up Questions, Lines 0–4: Load Modules web crawler used by AltaVista, Toss-up Questions web interface for MisterHouse, MisterHouse web proxy cache designed for Unix, Toss-up Questions web sites, Proof in the Pudding: Localizing Web Sites, Learning Japanese with an HTML Filter

learning/using Japanese, Learning Japanese with an HTML Filter

localizing, Proof in the Pudding: Localizing Web Sites

web-based voting, About Secure Elections, The vote Program

Webby awards, Toss-up Answers

webDAV (Web Distributed Authoring and Versioning), Toss-up Answers

WebLibs game, Word Games

weighing rules in optimality systems, Optimality Systems Weinberg, Gerald M., References

Weizenbaum, Joseph, Chatbot::Eliza

Wellcome Trust Sanger Institute, How Perl Saved the Human Genome Project

Wells-Jensen, Sheri, Braille and Contractions, Braille and Contractions, Linguistic Rule Systems, Linguistic Rule Systems

Wendling, Bill, Most Powerful, Best "The Perl Journal"

Westminster Chimes and measure counters, Mod, Canons, and Rounds, References, Mod, Canons, and Rounds, References

White Plume Mountain, Toss-up Questions

Whittier, John Greenleaf, Perl-fect Sundials

widows and autoformat, Widows

Win32::SerialPort module, What's in a Command?, What's in a Command?, What's in a Command?, What's in a Command?, What's in a Command?

winfrotz interpreter, Enter the Camel

Winstein, Keith, The First Circle: Print "The Perl Journal"

winter solstice, Perl-fect Sundials

Winter, Bruce, Whole House CallerID, Say What?

Winterbottom, Angie, Best of Show

wireless link module (X10), Simple Output Commands: The CM17

Wisehart, Dan, Results of the Contest

Wizard, Perl, Toss-up Questions

word boundaries, matching, Contexts in Regular Expressions

Word Finds, generating, Word Games

word frequency lists, Morphological Analysis and Perl, Morphological Inflections and Derivations, Constructing a Word Frequency List, Morphological Inflections and Derivations, Morphological Inflections and Derivations word games, Word Games, Twitch Games, Word Games, Twitch Games

word morphology, Word Morphology, Future Work, Morphological Inflections and Derivations, Implementing It, Implementing It, Future Work

word relationships, Other Word Relationships, Other Word Relationships, Other Word Relationships

hypernyms, Other Word Relationships meronyms, Other Word Relationships pertainyms, Other Word Relationships

World Wide Wasteland category, results of, World Wide Wasteland

WORM (write once read many) device, Bonus Questions worms, Toss-up Questions, Toss-up Questions

denial of service attack against www.whitehouse.com, Toss-up Questions

program created by R.T. Morris, Toss-up Questions

Worms game, Twitch Games

wrapman program, Toss-up Answers

wrapping text to fixed margins, Text::Autoformat, Future Features, Text::Autoformat, Quoting, Future Features write-in candidates, Lines 166–205: The enter_ballot Subroutine

entering ballot information into database, Lines 166–205: The enter ballot Subroutine

writein table in database schema, The Database Schema, Lines 166–205: The enter_ballot Subroutine

write_score function (MIDI::Simple), Behold MIDI::Simple!

wwwkan.pl script, The Web, wwwkan.pl

X

-x command-line switch, Toss-up Answers /x modifier, embedding comments in regular expressions, **Toss-up Answers** X10 ActiveHome (CM11) interface, MisterHouse, Give Your House a Brain X10 power line modulation, Simple Output Commands: The CM17 X10 wireless link module, Home Automation: The X10 **Nitty-Gritty** Xbox Microsoft game console, Toss-up Answers Xenakis, Iannis, References **XHTML, Toss-up Answers** xjdic program, Taking It a Step Further XSL, choosing numbering style with, Toss-up Questions **XUL files, Toss-up Questions** xzip interpreter, Enter the Camel

Y

Young, Benjamin, Inner Beauty

Ζ

Z-code (Zork Implementation Language), The Rezrov Infocom Game Interpreter, Enter the Camel, The Joy of vec

multiplatform interpreters available, Enter the Camel story files, The Joy of vec

loading into memory, The Joy of vec

Z-machines, Resurrecting the Z-Machine, What's It Do? opcodes, What's It Do? reverse-engineering, Resurrecting the Z-Machine

zeroes function (PDL), The peridi Shell

Zeroth Annual Obfuscated Perl Contest, The Zeroth Annual Obfuscated Perl Contest, Best "The Perl Journal", Best "The Perl Journal"

ZIO module (rezrov), Interface Abstraction

ZIP (Z-machine Interpreter Program), The Rezrov Infocom Game Interpreter, A Plea for Inlining

ZObject.pm package, Teleport

zombie process, Bonus Answers

Zork I, The Rezrov Infocom Game Interpreter, Tinkering with the Z-Machine, Quantity Is Job One, Toss-up Questions

Kitchen object description, Tinkering with the Z-Machine

opcodes required, Quantity Is Job One

Zork: The Undiscovered Underground, Resurrecting the Z-Machine

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *Games, Diversions, and Perl Culture: Best of the Perl Journal* is a flying dragon (genus *draco*). Found in the tropical rain forests of the East Indies and Southern Asia, this small lizard has five or six hind ribs on each side that are prolonged and covered with weblike skin, forming "wings." While jumping, the lizard spreads its wings and glides to the ground; it can generally glide almost nine yards. Gliding is used only as a means of locomotion and not for predator escape; to escape danger, the lizard always climbs. The lizard also never glides when it's raining or windy.

A flying dragon feeds mostly on small ants and termites and is described as a sit-and-wait feeder. It will sit next to a tree trunk waiting for insects to come to it.

A female flying dragon builds a nest for her eggs by forcing her head into the soil to create a small hole. She then lays five eggs into the hole and covers them with dirt, packing the soil on top with a patting motion of her head. The eggs take approximately 32 days to incubate.

Humans don't eat flying dragons, and they aren't currently listed as threatened.

Jane Ellin was the production editor and proofreader for *Games, Diversions, and Perl Culture: Best of the Perl Journal.* Colleen Gorman, Sarah Sherman, and Claire Cloutier provided quality control. Sue Willing, Linley Dolby, Jamie Peppard, and Sada Preisch provided production support. Judy Hoer wrote the index.

Hanna Dyer designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Archives. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

Bret Kerr designed the interior layout, based on a series design by David Futato. Erik Ray and Mike Sierra converted the files from pod to FrameMaker 5.5.6. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Mary Anne Weeks Mayo.