

The Perl Journal

Audio on Demand with Mr. Voice

H. Wade Minter • 3

Reformatting Text Using Pattern Matching

Julius C. Duque • 7

On Perl as a Natural Language

Russell J.T. Dyer • 12

Cooking with Maypole, Part I

Simon Cozens • 15

Graphical Interfacing with POE and Tk

Randal L. Schwartz • 20

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Jack J. Woehr:

***Python Programming:
An Introduction to Computer Science*** • 24

Source Code Appendix • 25

LETTER FROM THE EDITOR

Cleaning Up the Markup Mess

The dividing line between the printed word and the digital word is often a murky place to have to work. While it's true that almost all documents these days are digital, it doesn't necessarily follow that all those documents have all the potential advantages that digitized information can provide, such as easy categorization and retrieval and easy transformation to various other formats. Our modern world creates a chaotic storm of these documents every day, and it's often the job of Perl to pull order out of that chaos.

All of this became obvious in a recent conversation with a *TPJ* reader whose job involves the unenviable task of converting PDF documents to HTML. As he puts it: "PDF has NO structure. It's just digital paper. What's in the file is: 'Put this text at this location.'"

The documents he has to parse contain no information about the logical characteristics of their own parts. Basically, this means you don't really know what's what—how do you tell a magazine article's title from its subtitle from the caption of a figure or diagram? Having written some Perl here at *TPJ* to convert our documents from our page-layout program's proprietary format to HTML (and XML), I sympathize.

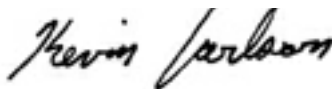
Why is this such a nasty issue? Shouldn't SGML and XML solve this problem? Well, yes. If your documents are created from the beginning in these information-rich markup languages, and your word processor or page-layout program saves your documents in these formats, you're golden. But good luck finding a WYSIWYG word processor or page-layout program that does a halfway-decent job of this. The current industry standards either have very naïve XML implementations, or require a more complicated configuration process than most users are willing to tolerate in order to enable such XML awareness.

And in order for a document format (like XML) to be information-rich, someone has to input that information. It's not enough to type in the title of an article—the author must then also label this title with the meta-information that in some way states that "this is a title." This is an extra step that most document creators won't take unless they are forced to. When faced with the choice of meeting a printer's deadline or labeling a document for easy processing later on, you can guess where the document author's priority would (and should) fall.

The real problem here is that document authors are concerned mostly with what the document *looks* like, not how it's logically structured. After all, if some text is at the top of a document, and it's large and bold, we naturally see it as a title. If it's text aligned underneath a picture, we see it as a caption describing that picture. This is a visual process that involves only our eyes and our brains. But there's a solution here, too: In many cases, these visual characteristics are the only meta-information you need to parse the document's structure.

At *TPJ*, we use a combination of these text characteristics and an object's position on the page to make decisions about tagging our content. We do it all in Perl, naturally. It allows us to entirely automate the process of conversion to HTML—but only because our articles are laid out in a very consistent way. If a document's text styles and the positions of various objects within the document change unpredictably from one document to the next, there are no patterns to work with and you're back to manually labeling all the parts of your document.

As great as Perl is, it can't help us to glean logical structure from our documents when there's nothing to be gleaned.



Kevin Carlson
Executive Editor
kcarlson@tpj.com

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

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



The Perl Journal

EXECUTIVE EDITOR

Kevin Carlson

MANAGING EDITOR

Della Song

ART DIRECTOR

Margaret A. Anderson

NEWS EDITOR

Shannon Cochran

EDITORIAL DIRECTOR

Jonathan Erickson

COLUMNISTS

Simon Cozens, Brian D'Joy, Moshe Bar, Randal Schwartz, Andy Lester

CONTRIBUTING EDITORS

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

INTERNET OPERATIONS

DIRECTOR

Michael Calderon

SENIOR WEB DEVELOPER

Steve Goyette

WEBMASTERS

Sean Coady, Joe Lucca

MARKETING / ADVERTISING

PUBLISHER

Timothy Trickett

MARKETING DIRECTOR

Jessica Hamilton

GRAPHIC DESIGNER

Carey Perez

THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403

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

CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND

MARKETING Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

PRESIDENT, TECHNOLOGY SOLUTIONS Robert Faletta

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED

TECHNOLOGIES Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER ENTERPRISE

ARCHITECTURE GROUP Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE

DEVELOPMENT MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING

SOLUTIONS Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES

Marie Myers

Perl News

Apocalypse 12 Unleashed

Apocalypse 12, a 20-page essay addressing the Perl 6 object model, is now upon us. This is the latest Apocalypse since Apocalypse 7, which consisted of just two sentences. As Larry explained on the perl6-language list, the Apocalypses are numbered in order of their importance to Perl 5, but they're being written in order of their importance to Perl 6. Therefore, Apocalypses 8 through 11—covering References, Data Structures, Packages, and Modules—have been skipped for the time being.

“What we’re proposing,” Larry writes, “is to develop a set of conventions for how object orientation ought to work in Perl 6—by default. But there should also be enough hooks to customize things to your heart’s content, hopefully without undue impact on the sensibilities of others. And in particular, there’s enough flexibility in the new approach that, if you want to, you can still program in a way much like the old Perl 5 approach.”

The full text of Apocalypse 12 is at <http://www.perl.com/pub/a/2004/04/16/a12.html>.

Perl 5.8.4 Nears Release

As of press time, Perl 5.8.4 has made it to Release Candidate 2, as RC1 had unexpected problems installing `suidperl`. Barring further surprises, the final 5.8.4 release should be out within a few days.

`Suidperl` had to be changed to fix a security vulnerability. Although backwards compatibility with scripts that invoke `#!/usr/bin/suidperl` has been preserved, the `perldata` warns: “For new projects, the core perl team would strongly recommend that you use dedicated, single purpose security tools such as `sudo` in preference to `suidperl`.” Perl 5.8.4 also fixes another security hole involving attempts to assign unreasonably large amounts of memory, which would previously crash perl, potentially opening up a “stack smashing” vulnerability.

The Perl Foundation Awards New Grant

The Perl Foundation has announced a new development grant: Simon Cozens has been awarded \$1000 to work on Maypole, a Model-View-Controller (MVC) framework for web applications. Maypole is designed as a Perl equivalent to Jakarta’s Struts framework. (See Simon’s article on Maypole in this issue.) As Simon writes on <http://maypole.simon-cozens.org/>, “Many web applications follow the same kind of flow of operation: In response to a request from a user, they mess about with a database, and present the results of that messing about back to the user through some templating system...Maypole provides a generic way of handling that compartmentalisation.” Bottom line: “It means you can write fully featured interfaces to a database in less than 20 lines of code.”

Maypole is currently in Version 1.4 (available from CPAN); the Perl Foundation grant will go towards developing view and template classes for `HTML::Mason` and `DBIx::SearchBuilder`, as well as finishing the Maypole documentation and producing example sites. Existing Maypole users are encouraged to get involved; there’s a mailing list accessible at <http://lists.netthink.co.uk/listinfo/maypole/>.

MySQL Issues License Exception

MySQL AB has solved a potential problem for `DBD::mysql` users by issuing a specific exemption to the license under which MySQL client libraries are distributed. As of MySQL 4.0, these client libraries were distributed under the GPL rather than the LGPL, which raised problems for developers who work with Perl, PHP, or other languages distributed under nonGPL licenses. The issue came to a head when the PHP packagers removed the MySQL client libraries from PHP 5.

MySQL AB’s exemption, published at <http://www.mysql.com/products/licensing/foss-exception.html>, specifies that applications distributed under any one of 18 open-source licenses—including Perl’s Artistic license—are free to incorporate MySQL client libraries without changing their own license terms. It’s formally termed the MySQL FOSS (Free and Open Source Software) License Exception.

Correction

Andy Adler has discovered a problem in the code he presented in his article on `Inline::Octave` in last month’s *TPJ*. According to Andy:

Mea culpa. I made a mistake in the code presented in ‘Perl and Inline Octave Code,’ which meant that the ‘temp-analyse.pl’ code would hang in some situations. The short solution is that I’ve released a new version (0.21) of `Inline::Octave`, which fixes the bug.

For those interested in more detail: Since `Inline::Octave` starts a child (Octave) process, it would like to know if it dies for any reason. Therefore, it sets `$_SIG{CHLD}` and `$_SIG{PIPE}` to handle this event. Unfortunately, the module user may well be starting child processes (such as ‘unzip,’ in the example given) or setting signal handlers. In the example, the user’s child process termination got caught by the wrong handler. In order to solve this problem, `Inline::Octave` now only sets signal handlers (using `local`) immediately before any interaction with ‘Octave’ and allows them to be reset afterwards.

Sorry for any inconvenience. If nothing else, I’ve managed to point out that this kind of IPC can be tricky.

—Andy Adler

Audio-on-Demand With Mr. Voice

We all know how well Perl fits in to solve problems with system administration, text processing, and web applications. There are, however, many less obvious ways that Perl can make a difference. For instance, helping to improve the quality of an improv comedy show.

Let me give you a little background. I joined the ComedySportz improv comedy troupe (now known as ComedyWorx—<http://www.comedyworx.com/>) in 1999. ComedyWorx puts on a competitive, team-on-team improv show, similar to *Whose Line Is It Anyway*, but with a sports motif. I started out as a player in the show on-stage, but as a geek, I was quickly drawn to the sound booth. The player who provides music, sound effects, and general commentary during the show is known as “Mr. Voice.”

When I started working as Mr. Voice, the sound process was very manual. Sound clips were supplied via tapes, CDs, and a MIDI device. Even as well organized as the other Voices had the work area, it still took ages (in improv time) to find the right clip, get it into the player, and start it playing. As things are unfolding on-stage, you only have a few seconds to find the right sound clip. Too often, by the time we locate and start playing the music, the moment has been lost. To further complicate matters, consider the delay in moving to the proper CD track or the danger of finding that the last Voice who used the tape you need was inconsiderate and didn’t rewind it to the proper point.

I wanted to bring digital audio into the picture for two reasons—ease of use (no need to reposition media, quick access times) and disaster recovery. It was impractical to make lossy dubs of all the tapes in the collection, but the danger of having the only tape holding a particular sound clip break was very real. However, it is trivial to back up perfect copies of audio files onto removable media or a network. I started off by putting some songs into an XMMS playlist and using them that way. It was certainly faster for small numbers of songs, but quickly became unwieldy as the number of songs grew and the need for organization and categorization arose. Scrolling through the playlist started to become as cumbersome as rifling through stacks of CDs.

H. Wade Minter is a UNIX system administrator and an improv comedian. He lives in Raleigh, NC and can be reached at minter@lunenburg.org.

A search through Google and Freshmeat didn’t yield any apps that I felt would solve the problem. I knew that for a situation this specific, I needed to write something myself. I named my project “Mr. Voice” after the job title.

I’ve never considered myself much of a coder. I did earn a Bachelor of Science in Computer Science at the College of William & Mary, where the curriculum focused primarily on C programming, but I’ve always been more of a systems guy than a coder. I was pretty comfortable in Perl, though, as it had been my longtime choice for CGI and systems programming work. But I knew that the other Voices would not adopt a command-line solution—I needed to pick a GUI toolkit to make this project work in a way other users would accept.

So I picked up *Learning Perl/Tk* by Nancy Walsh (O’Reilly & Associates, 2001). I can’t recommend this book enough. Within a few weeks, I was able to build a very usable system. I think Perl/Tk was a good choice—even three years later, when I look around at other Perl GUI bindings like wxPerl and Gtk-Perl, the quality of the Perl/Tk documentation stands out. I also recommend *Mastering Perl/Tk* by Steve Lidie and Nancy Walsh (O’Reilly & Associates, 2001).

With a GUI toolkit in hand, it was time to figure out how things would fit together. I decided to use a simple MySQL database for the metadata back end, storing things like title, artist, category, and filename. While I had some concerns that MySQL might be a little too heavy for the data that I was storing, I concluded that using it would allow for long-term growth. After all, I needed to build a system that was sustainable and could be enhanced as our needs continue to evolve. To actually play the audio files, I went with the venerable XMMS. That meant that my code just had to provide the glue between the two.

An easy-to-use interface was key. I started off by laying out how I wanted the main screen to look. I decided on a fairly vertical layout including (from top to bottom) a menu bar, a search area, a list-box to display the results, and a status bar (see Figure 1). The *Pack* layout manager seemed to be the right choice to achieve that. Once I wrapped my head around the allocation rectangle system, it turned out that *Pack* did everything I wanted it to do. And, honestly, the books did a great job of making *Pack* easy to use.

The justification on the labels and entry fields was achieved by packing each row into its own frame, then packing the label

with a set width to the left, followed by the Entry widget to the left.

```
$title_frame = $mw->Frame()->pack(
    -side => 'top',
    -fill => 'x'
);
$title_frame->Label(
    -text => "Title contains",
    -width => 25,
    -anchor => 'w'
)->pack( -side => 'left' );
$title_frame->Entry( -textvariable => \$title )->pack( -side => 'left' );
```

That way, everything lines up nicely.

Mr. Voice allows users to assign songs to F1 through F12 as hotkeys, so you can preload songs and quickly start music playing at the press of a single button. In my first few releases, the way you assigned a hotkey was pretty manual—you selected a song, hit an “assign hotkey” button, and selected a function key from this list. This process was separate from the *Toplevel* window that actually listed which songs were assigned. I knew there had to be a better way. The logical answer seemed to be drag and drop.

The drag-and-drop tutorial from perl.tk.org (<http://www.perl.tk.org/articles/dnd/dnd.html>) got me started. I implemented a system where you can click and drag a song from the main listbox, then drop an icon on one of the hotkey areas in the hotkey window. The selected item will then get assigned to the proper hotkey. I cheated somewhat on the back end, though, because the actual dragged token is meaningless—it’s just eye candy. The important thing is which item in the listbox is selected at the time of the drop, as the code below shows:

```
sub Hotkey_Drop
{
    my ($fkey_var) = @_;
    my $widget = $current_token->parent;
    my (@selection) = $widget->curselection();
    my $id = get_song_id($widget, $selection[0] );
    my $filename = get_info_from_id($id)->(filename);
    my $title = get_info_from_id($id)->(fulltitle);
    $fkeys{$fkey_var}->{id} = $id;
    $fkeys{$fkey_var}->{filename} = $filename;
    $fkeys{$fkey_var}->{title} = $title;
}
```

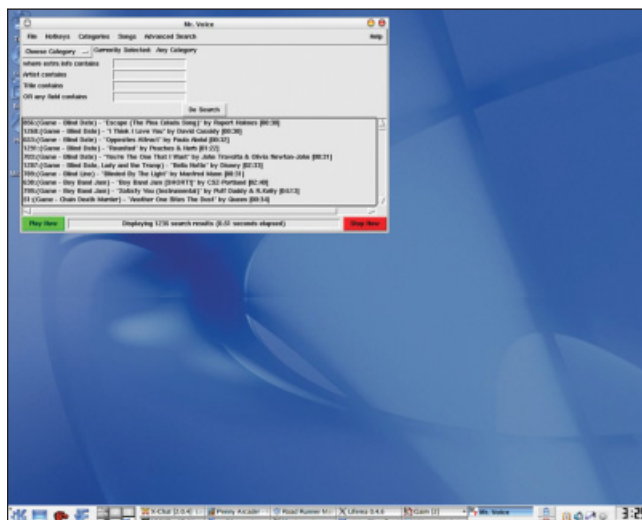


Figure 1: Mr. Voice's interface.

The actual drag-and-drop token (*\$current_token*) is only referenced to get the widget ID of the parent, which is one of two listboxes. The listbox is then queried directly to see which item is highlighted. Even if the code isn't the most elegant, it looks good and is easy to use, which also makes the users happy.

ComedyWorx puts on a competitive, team-on-team improv show, similar to Whose Line Is It Anyway, but with a sports motif

This little cheat actually came in quite handy when I added the Holding Tank (an extra *Listbox*-based *Toplevel* that is part of the application). Users can drag and drop multiple items from the main listbox into the Holding Tank by way of control-clicking to select multiple items. I'm not sure if I could get the standard drag-and-drop token system to handle multiple items in a single drag. To work around this, when a multiple-item selection is dropped into the Holding Tank, it queries the main listbox, receives an array of indexes, then iterates over them to populate the Holding Tank (see Figure 2).

```
sub Tank_Drop
{
    my ($dnd_source) = @_;
    my $parent = $dnd_source->parent;
    my (@indices) = $parent->curselection();
    foreach $index (@indices)
    {
        my $entry = $parent->get($index);
        $tankbox->insert('end', $entry );
    }
    if ( $#indices > 1 )
```

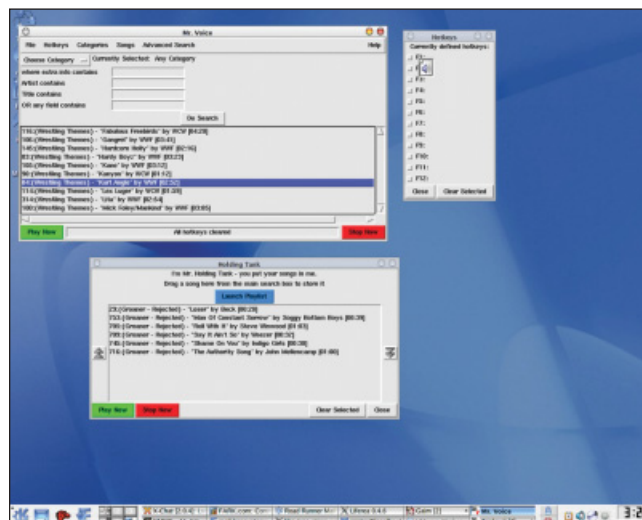


Figure 2: The Holding Tank.

```

{
    $parent->selectionClear( 0, 'end' );
}
}

```

While we're on the subject of drag and drop, I ran into a functional problem using it within my main listbox. The listbox select mode was set to "extended," which enabled the familiar "Control-click to select multiple items" selections. However, the extended mode also has a feature where you can click-drag to select multiple items. Unfortunately, it led to a race condition where people attempting to drag and drop items ended up with multiple items selected, and they'd drop the wrong thing.

Looking at the *Tk::Listbox* source, I found that there was a *Motion* method. To solve my problem, I redefined the method within my code to return immediately without actually doing anything. Of course, it also broke the click-drag selects native to the *Listbox* widget, but my app didn't really take advantage of that anyway, so I didn't lose anything there.

```

# Try to override the motion part of Tk::Listbox extended mode.
sub Tk::Listbox::Motion
{
    return;
}

```

With those three lines, starting a drag no longer selected multiple items, and one of my biggest complaints disappeared.

I discovered an added bonus in my choice of GUI toolkits: Perl/Tk makes developing a cross-platform application extremely easy. I designed Mr. Voice under Linux, which is how I set up our troupe's computer in Raleigh. However, when other improv clubs started using it, they all ran on Win32. Fortunately, all it took were a handful of checks of *\$^O* and I was able to run the same code on UNIX, Mac OS X (X11 mode), and Win32. An example of this is seen early in the program where I check for the OS and pull in OS-specific modules as needed (see Listing 1).

(The time zone setting is for code where I allow people to query the database for songs that have been added or updated in the last *x* period of time using methods in *Date::Manip*, which is discussed later in this article.)

Another example of OS-specific behavior occurs before I query an MP3 or OGG file for metadata—I get the short pathname of the actual audio file if we're on Win32:

```

if ( $filename =~ /\.mp3$/i )
{

```

```

$filename = Win32::GetShortPathName($filename) if ( $^O eq "MSWin32" )
my $tag = get_mp3tag($filename)
$title = $tag->{TITLE};
$artist = $tag->{ARTIST};
}

```

Figure 3 shows Mr. Voice running on Windows. When it's that easy to make your application cross platform, it's a shame not to

All it took were a handful of checks of \$^O and I was able to run the same code on UNIX, Mac OS X (X11 mode), and Win32

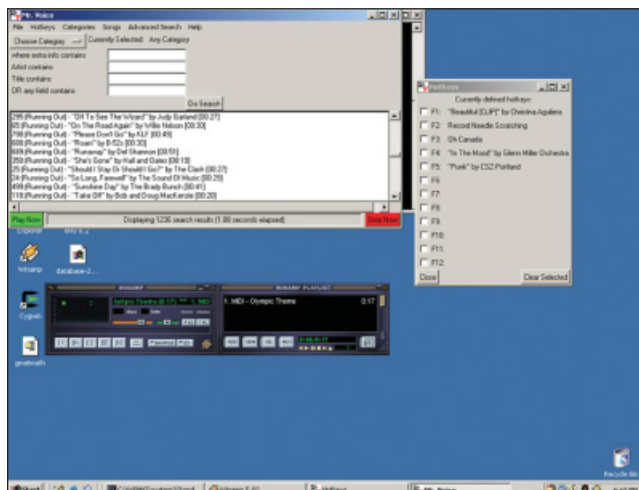


Figure 3: Mr. Voice on Windows.

do it. With Tk-804 now done, Steve Lidie has indicated that it should now be possible to build-in support for native Aqua widgets under Mac OS X, instead of having to use X11.app. I'll be watching progress on that front with great interest.

In improv clubs like ours, you normally have a large handful of people who are both qualified to run the sound system and interested in doing so, and people generally take turns. You might be Mr. Voice one weekend, then not be back behind the mic for a few weeks. Meanwhile, the other people all have the ability to add new sound files and modify existing entries. We needed a way to identify changes within the database.

I solved that problem with the help of the *Date::Manip* module. In Mr. Voice, there is an Advanced Search menu that has several options. Among those are "Show me what has changed in the last 0, 7, 14, or 30 days." The Mr. Voice MySQL database schema is set up with a *TIMESTAMP* column, which is set to the current time when a row is added or updated. When you choose one of those canned searches, it triggers the following code in the main search function:

```

if ( ( $_[0] ) && ( $_[0] eq "timespan" ) )
{
    $date = DateCalc( "today", "- $_[1]" );
    $date =~ /\d{4}(\d{2})(\d{2})\./;
    $year    = $1;
    $month   = $2;
    $date    = $3;
    $datestring = "$year-$month-$date";
}

```

\$_[1] is set to "0 days," "7 days," "14 days," or "30 days." *DateCalc* does the heavy lifting to return the proper date from today, then format it in a MySQL-friendly way. Then, when constructing the MySQL query, we do the following:

```

$query = $query . "AND modtime >= '$datestring' "
if ( ( $_[0] ) && ( $_[0] eq "timespan" ) );

```

This narrows the standard search query to only return rows that have changed in the specified timeframe. So the first thing I do when I get to the computer after time away is to run one of those queries to see what my fellow Voices have been working on. There is also an option to specify absolute start and end dates, which works in a similar fashion.

These are just a few examples of how I simplified my improv life using Perl. I don't consider any of the code I've written for Mr. Voice to be revolutionary. Instead, the magic of Mr. Voice comes from the ease with which Perl lets you put the building blocks together to create your own work of art. There is a well-documented GUI toolkit with an outstandingly helpful user community (<http://comp.lang.perl.tk/>). CPAN provides its collection of modules that allowed me to quickly do everything from grabbing title and artist information from OGG files to creating platform-neutral file paths with *File::Spec*. Of course, Perl itself, with the TMTOWTDI philosophy, lets you use the language rather than having the language use you. All of those pieces let me, a guy with no particular skill in coding or GUI design, put together a very useful application that not only makes my job easier, but allows other people with the same problem to do the same. If you have occasion to use Mr. Voice, I'm always grateful for suggestions as to ways I could improve the application or make my code better. The parts for Mr. Voice, including a web-based CVS repository, are available on my web site for any interested parties.

Finally, a quick note on packaging. As I said before, everyone (that I know of) except me runs Mr. Voice on Win32 systems. Mr. Voice has a fairly large list of module dependencies. I quickly found out that it was unreasonable to expect a group of fairly non-technical people scattered from Spokane to Los Angeles to Richmond to keep up with installing Perl modules on systems that are not, as a general rule, connected to a network. I originally used Perl2EXE to package my script and its modules into a single .exe (with an icon—Win32 folks love their icons) that I could distribute, which worked well enough. However, I've recently switched to PAR (<http://par.perl.org/>). Autrijus Tang has done an incredible

job with PAR—it can do everything that Perl2EXE can do, and more, plus it's both free and Free. If you're distributing a Perl application of any significant size or complexity, I highly encourage you to check out PAR.

ComedyWorx now runs its shows almost completely on digital audio. It works so well that people come up to us after shows and say, "I can't believe how quickly you were able to start playing music up there!" And since our music is all digital, we're able to take backups of sound files and the database offsite to protect them from accidents. We're able to do more with our shows, now that we have a way to store more audio and get to it quicker—and Perl makes it all happen. If you're ever in Raleigh, stop by and see a show. You'll be able to appreciate it on both an artistic and technical level.

Mr. Voice can be found at <http://www.lunenburg.org/mrvoice/>.

TPJ



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

Listing 1

```
if ( "$0" eq "MSWin32" )
{
    our $srcfile = "C:\\mrvoice.cfg";

    BEGIN
    {
        if ( "$0" eq "MSWin32" )
        {
            require LWP::UserAgent;
            LWP::UserAgent->import();
            require HTTP::Request;
            HTTP::Request->import();
            require Win32::Process;
            Win32::Process->import();
            require Tk::Radiobutton;
            Tk::Radiobutton->import();
            require Win32::FileOp;
            Win32::FileOp->import();
        }
    }

    $agent = LWP::UserAgent->new;
    $agent->agent("Mr. Voice Audio Software/$0 ");

    # You have to manually set the time zone for Windows.
    my ( $l_min, $l_hour, $l_year, $l_yday ) = ( localtime $^T )[ 1, 2, 5,
7 ];
```

```
my ( $g_min, $g_hour, $g_year, $g_yday ) = ( gmtime $^T )[ 1, 2, 5, 7
];
my $tzval =
    ( $l_min - $g_min ) / 60 + $l_hour - $g_hour + 24 *
    $tzval = sprintf( "%2.2d00", $tzval );
    Date_Init("TZ=$tzval");
}
else
{
    our $homedir = "~";
    $homedir =~ s{ ^ ~ ( [^/]* ) }
        { $1
            ? (getpwnam($1))[7]
            : ( $ENV{HOME} || $ENV{LOGDIR}
                || (getpwuid($>))[7]
            )
        }ex;
    our $srcfile = "$homedir/.mrvoicerc";
}
```

TPJ

Reformatting Text Using Pattern Matching

Whenever I write documents, I spend a considerable amount of time just reformatting lines of text so that they fit within a 70-character-wide column. I do this because I don't like reading long lines that wrap across the screen.

Because adjusting lines of text by hand is time consuming, I decided to put my Perl skills to good use by automating this cumbersome task. Being very good at text manipulation, Perl is the ideal tool for this job. Sure, there's already the *fmt* command, but it's available only on UNIX/Linux systems. Besides, it can only output left-justified lines.

The full source code of my Perl script, called *pretty*, can be found in Listing 1. Examples 1 through 6 show us all possible output formats when the input file, *gettysburg.txt*, which contains Lincoln's Gettysburg Address, is reformatted. Both the source for *pretty* and *gettysburg.txt* are available for download at <http://www.tpj.com/source/>.

Note: When using *pretty*, it expects unformatted paragraphs to be separated by at least two consecutive newlines.

Anatomy of *pretty*

Lines 9–12 of *pretty* declare the switches available to the user. The *--width* switch, which takes a mandatory integer value (*=i*), is used to control the line width of output lines. The decision to make *--width* take a mandatory value is just a matter of preference.

The options *--help*, *--left*, *--right*, *--centered*, *--both*, and *--newline* don't take any value; they're activated if explicitly specified on the command line.

--left and *--indent* are the only options that have default values. Lines are printed left-justified, regardless of whether *--left* (or its short form *-l*) is specified, unless overridden by *--right*, *--centered*, or *--both*. The switch *--indent* (or *-i*), whose integer argument specifies the amount of indentation at the start of a paragraph, defaults to a value of 0 (no indentation) if it is not explicitly specified (line 15).

--centered (or *-c*) places a line of text of equal spacing from the left and right margin. *--right* (or its short form *-r*) outputs lines that are right-justified, while *--both* (or *-b*) produces both left-justified and right-justified lines. To put empty lines between paragraphs, specify *--newlines* (or *-n*).

To aid the user, the function *syntax()* is called if *--help* is specified or if *--width* is omitted (line 14).

Julius is a freelance network consultant in the Philippines. He can be contacted at jcdueque@lycos.com.

Parsing a Paragraph

Normally, Perl reads a chunk of data one line at a time—a “line” being a string of characters terminated by a newline (*\n*). Since we want to reformat paragraphs that span multiple lines, we need to change the meaning of “line.” We tell Perl to parse paragraphs instead of single lines. Now, the meaning of a “line” becomes “a string of characters delimited by two or more consecutive newlines.” This is, essentially, the meaning of a “paragraph.”

Fortunately, Perl offers a special variable that can be set to change the meaning of a “line.” That variable is *\$/*, the input record separator. You may set it to a multicharacter string to match a multicharacter delimiter. In our problem of reformatting paragraphs, be warned, though, that the choice of a new value for *\$/* can be tricky. For instance, the obvious delimiter, *\n\n*, is wrong. If Perl sees three consecutive newlines, for example, Perl will assume that the third newline belongs to the next paragraph. Meanwhile, Perl will swallow the whole input, from the first up to the last character, if *\$/* is set to *undef*. The correct approach is to set *\$/* to an empty string, *""*. This tells Perl to treat two or more consecutive newlines as a single newline; see line 17.

The Basic Idea

Lines 19–23 illustrate the core idea of *pretty*. On line 19, the use of the loop *while (<>)* enables *pretty* to act as a filter. With this loop, you can use the script like this:

```
cat file1.txt file2.txt file3.txt | ./pretty --width=64 --right
```

as well as like this:

```
./pretty --width=64 --right file1.txt file2.txt file3.txt
```

where *file1.txt*, *file2.txt*, and *file3.txt* are input files.

The paragraph read by Perl on line 19 is implicitly loaded into another special variable, *\$_*. On line 20, the split function implicitly acts on *\$_* and strips off all whitespaces (spaces, tabs, and newlines), storing only chunks of nonwhitespaces into an array, *@linein*. On Line 21, function *printpar()* takes *@linein* as argument and prints out the formatted paragraph. Last, on line 22, a newline is printed to separate two consecutive paragraphs if *--newline* is specified.

Printing a Line

The function `printpar()` performs the actual work of reformatting paragraphs (starting on line 25). When this function is called on line 21, the argument, `@linein`, is passed on to another variable, `@par` (line 27). From now on, the paragraph read by Perl will be manipulated through this new variable. The variable `$firstline` (on lines 28, 31, and 36–39) is relevant only if the option `--newline` is specified. See the section “Indentation” for more details.

The logic of `printpar()` is as follows:

1. We make use of a temporary line buffer that is, at most, as long as the line width (`$width`) specified by the user. We call this buffer `$buffer`. The unit of length is 1 character.
2. We maintain a running total of characters read so far. Store this running total to variable `$charcount`. Initially, this is set to 0. `$charcount` must not exceed `$width`.
3. Extract an element (a nonwhitespace chunk) from `@par` one at a time and take note of the element’s length. Insert the element into `$buffer`. Increment `$charcount` by an amount equal to the extracted element’s length. Then, insert a single space into `$buffer` to serve as a word separator. Increment `$charcount` by 1 to account for this single space. In doing so, realize that the last inserted element in `$buffer` is always a single space.
4. Repeat Step 3 until `$charcount` either exceeds or equals `$width`. Note that `$charcount` may exceed or equal `$width` by the insertion of an extracted element into `$buffer`, even before the mandatory word separator is added to `$buffer`.
5. If Step 3 terminates because `$charcount` either exceeds or equals `$width`, discard the last single space inserted in `$buffer`. Decrement `$charcount` by 1. Jump to Step 7 if the new value of `$charcount` becomes less than or equal to `$width`; otherwise, proceed to Step 6.
6. If, after decrementing by 1 (in Step 5), `$charcount` still exceeds the `$width`, requeue the last, extracted nonwhitespace element back into `@par` and update `$charcount` by subtracting from it the length of the excess element. Note that there is a word separator inserted into `$buffer` just before the returned element was inserted into `$buffer`. Delete this extra space also and decrement `$charcount` by 1.
7. Transfer the contents of `$buffer` to another variable, `$lineout`, and print it.
8. Repeat Steps 1–7 until there are no more elements in `@par`.

Lines 30–67, except those lines that contain the variable `$firstline`, show us how to implement the 8 steps above. Lines containing `$firstline` are used only when `--newline` is switched on. Let’s ignore these lines for the moment; we’ll get to that when we discuss indentation later.

On line 32, we use two temporary variables: `$buffer` (to hold the line to be printed) and `$word` (to hold the extracted element from `@par`). On line 33, `$wordlen` holds the length of the element just extracted, and `$charcount` is the number of characters in `$buffer` so far. On line 34, we use a temporary variable, `$linewidth`, to hold the value of `$width`, the line width specified by the user on the command line. We’re doing this because we don’t want to alter `$width` directly.

On lines 30 and 41, there is a constant reference to scalar `@par`. This is necessary because the function scalar returns the number of elements in an array. Recall that we always remove elements from the array `@par`. We must make sure, then, that it still has elements to be extracted; we stop extracting if it is already empty, even if `$charcount` has not yet exceeded the line width.

When `$buffer` is ready to be printed out, we transfer its contents to another variable, `$lineout`, for final printing (lines 60 and 102).

In case the line width is too short to accommodate even a single word, I have provided an easy way out (lines 46–50). The solution is to use a wider line width.

Indentation

`pretty` does not produce indented paragraphs by default. But you can alter this behavior by specifying the `--indent` (or `-i`) switch, with an integer as argument. This integer tells `pretty` to pad that many spaces at the start of a line. Note, however, that indentation takes effect only on the first line of each paragraph.

Line 12 declares `--indent` as a switch that takes an optional integer value (`:i`). Line 15 initializes `--indent` with a value of 0, in case it is not specified.

To help `pretty` distinguish the first line of a paragraph from the rest of the lines, I make use of a flag, the `$firstline` variable, on line 28. Before `pretty` starts scanning for input lines (on line 30), `$firstline` is initially set to 0, corresponding to a `false` value. When

```
1 ./pretty --width=64 --left --newline gettysburg.txt
2 ./pretty -w=64 -l -n gettysburg.txt
3 ./pretty -w 64 -l -n gettysburg.txt
4 ./pretty -w 64 -n gettysburg.txt
5 cat gettysburg.txt | ./pretty --width=64 --left --newline
```

```
1 Four score and seven years ago our fathers brought forth on this
2 continent, a new nation, conceived in Liberty, and dedicated to
3 the proposition that all men are created equal.
4
5 Now we are engaged in a great civil war, testing whether that
6 nation, or any nation so conceived and so dedicated, can long
7 endure. We are met on a great battle-field of that war. We have
8 come to dedicate a portion of that field, as a final resting
9 place for those who here gave their lives that that nation might
10 live. It is altogether fitting and proper that we should do
11 this.
```

Example 1: Left-justified output. Any of the five lines at the top will produce the output shown. You may omit `--left` (or `-l`) if you want your line to be left-justified, since it is switched on by default; see line 4 at the top.

```
1 ./pretty --width=64 --right --newline gettysburg.txt
2 ./pretty -w=64 -r -n gettysburg.txt
3 ./pretty -w 64 -r -n gettysburg.txt
4 cat gettysburg.txt | ./pretty --width=64 --right --newline
```

```
1 Four score and seven years ago our fathers brought forth on this
2 continent, a new nation, conceived in Liberty, and dedicated to
3 the proposition that all men are created equal.
4
5 Now we are engaged in a great civil war, testing whether that
6 nation, or any nation so conceived and so dedicated, can long
7 endure. We are met on a great battle-field of that war. We have
8 come to dedicate a portion of that field, as a final resting
9 place for those who here gave their lives that that nation might
10 live. It is altogether fitting and proper that we should do
11 this.
```

Example 2: Right-justified output.

```
1 ./pretty --width=64 --both --newline gettysburg.txt
2 ./pretty -w=64 -b -n gettysburg.txt
3 ./pretty -w 64 -b -n gettysburg.txt
4 cat gettysburg.txt | ./pretty --width=64 --both --newline
```

```
1 Four score and seven years ago our fathers brought forth on this
2 continent, a new nation, conceived in Liberty, and dedicated to
3 the proposition that all men are created equal.
4
5 Now we are engaged in a great civil war, testing whether that
6 nation, or any nation so conceived and so dedicated, can long
7 endure. We are met on a great battle-field of that war. We have
8 come to dedicate a portion of that field, as a final resting
9 place for those who here gave their lives that that nation might
10 live. It is altogether fitting and proper that we should do
11 this.
```

Example 3: Left- and right-justified output.

scanning begins, *\$firstline* is incremented by 1 (its value now becomes *true*), signifying that *pretty* has just found the first line of the paragraph (line 31).

On lines 36–39, we check the value of *\$firstline*. If it is equal to 1, then we know that we are currently dealing with the first line of the paragraph.

On line 37, *\$linewidth* is decreased by an amount equal to the integer specified as an argument to *--indent*. On line 38, we print out that many leading spaces to serve as indentation. We then fill up the remaining portion of the first line with texts, whose accumulated lengths must be less than or equal to the value of the updated *\$linewidth*.

On the next iteration, *\$firstline* is no longer equal to 1. Hence, the value of *\$linewidth* on line 34 remains unchanged. This also means that initial padding of spaces will no longer occur. Examples 5 and 6 present sample outputs using indentation.

Left-Justified Output

Undoubtedly, this is the simplest paragraph format to implement: just print *\$lineout* as is (line 102).

Right-Justified Output

For a right-justified format, we must know how many spaces to pad the left portion of *\$lineout*. This amount of space is simply the difference between the specified line width and the length of *\$lineout*. On line 69, we use the variable *\$spaces_to_fill* to hold this value. On line 75, we print this many leading spaces before printing *\$lineout*.

Centered Output

The centered format is very similar to the right-justified format, only this time, we divide the value of *\$spaces_to_fill* by 2. If the result has a fractional part, take only the integer part (line 72), and print this many leading spaces (line 73).

Left- and Right-Justified Output

This is the hardest part to implement. The trick is to take *\$lineout* and modify it by adding extra spaces between nonwhitespaces. But how do we distribute the spaces evenly within the line? I'll illustrate my own solution using examples.

Suppose that we want the line width to be 39 characters long, and the string to be printed is:

```
fathers•brought•forth•on•this
```

where • represents a single space.

The line above is 29 characters long (including 4 embedded spaces), 10 spaces short of being both left- and right-justified. My solution consists of two steps:

1. Scanning from the left, look for the first occurrence of a single space, and replace it with a double space.

```
fathers••brought•forth•on•this
```

2. Reverse the string.

```
siht•no•htrof•thguorb••srehtaf
```

Consider this as one round.

Since the new string is not yet 39 characters long, do another round:

1. Scanning from the left, look for the first occurrence of a single space, and replace it with a double space.

```
siht••no•htrof•thguorb••srehtaf
```

```
1 ./pretty --width=64 --centered --newline gettysburg.txt
2 ./pretty -w=64 -c -n gettysburg.txt
3 ./pretty -w 64 -c -n gettysburg.txt
4 cat gettysburg.txt | ./pretty --width=64 --centered --newline
```

```
1 Four score and seven years ago our fathers brought forth on this
2 continent, a new nation, conceived in Liberty, and dedicated to
3 the proposition that all men are created equal.
4
5 Now we are engaged in a great civil war, testing whether that
6 nation, or any nation so conceived and so dedicated, can long
7 endure. We are met on a great battle-field of that war. We have
8 come to dedicate a portion of that field, as a final resting
9 place for those who here gave their lives that that nation might
10 live. It is altogether fitting and proper that we should do
11 this.
```

Example 4: Centered output.

```
1 ./pretty --width=64 --both --newline --indent=4 gettysburg.txt
2 ./pretty -w=64 -b -n -i=4 gettysburg.txt
3 ./pretty -w 64 -b -n -i 4 gettysburg.txt
4 cat gettysburg.txt | ./pretty --width=64 --both --newline -i=4
```

```
1 Four score and seven years ago our fathers brought forth on
2 this continent, a new nation, conceived in Liberty, and
3 dedicated to the proposition that all men are created equal.
4
5 Now we are engaged in a great civil war, testing whether
6 that nation, or any nation so conceived and so dedicated, can
7 long endure. We are met on a great battle-field of that war. We
8 have come to dedicate a portion of that field, as a final
9 resting place for those who here gave their lives that that
10 nation might live. It is altogether fitting and proper that we
11 should do this.
```

Example 5: Indented, left- and right-justified output.

```
1 ./pretty --width=64 --both --indent=4 gettysburg.txt
2 ./pretty -w=64 -b -i=4 gettysburg.txt
3 ./pretty -w 64 -b -i 4 gettysburg.txt
4 cat gettysburg.txt | ./pretty --width=64 --both --indent=4
```

```
1 Four score and seven years ago our fathers brought forth on
2 this continent, a new nation, conceived in Liberty, and
3 dedicated to the proposition that all men are created equal.
4 Now we are engaged in a great civil war, testing whether
5 that nation, or any nation so conceived and so dedicated, can
6 long endure. We are met on a great battle-field of that war. We
7 have come to dedicate a portion of that field, as a final
8 resting place for those who here gave their lives that that
9 nation might live. It is altogether fitting and proper that we
10 should do this.
```

Example 6: Indented and left- and right-justified output with no newlines.

2. Reverse the string.

```
fathers••brought•forth•on•this
```

Notice that we have now evenly distributed two “filler” single spaces, one on the left and the other on the right. We repeat the two steps over and over until the string becomes 39 characters long.

We are now ready to formulate our strategy:

1. Scanning from the left, look for the first occurrence of a single space and replace it with a double space. Reverse the string.
2. Repeat Step 1 until the required line width is reached.

If we ran out of single spaces to replace, rephrase Step 1 as “Scanning from the left, look for the first occurrence of a double

space, and replace it with a triple space. Reverse the string.” Repeat the two steps.

If we ran out of double spaces to replace, rephrase Step 1 as “Scanning from the left, look for the first occurrence of a triple space, and replace it with a quadruple space. Reverse the string.” Repeat Steps 1 and 2, and so on. Get the picture?

Lines 83–92 show us how to implement the strategy. The actual work of replacing spaces is found on lines 85–86. On line 81, we make use of a counter, *\$reps*, that tracks the kind of spaces

*If the pattern is enclosed
in parentheses, Perl will
remember the substring that
watches this pattern*

(single, double, triple, and so on) to look for. Initially, *\$reps* is set to 1, meaning we begin our search for single spaces.

If the line width is not yet reached (line 83), replace the first single space found with a double space (lines 85–86). If you fail to see the single space in there, here’s a closer view of lines 85–86 (a • stands for a single space):

```
if ($tempbuf =~ /\S+•{$reps}\S+/) {  
    $tempbuf =~ s/\S+•{$reps}\S+/$1•$2/;
```

When *\$reps* is 1, lines 85–86 become:

```
if ($tempbuf =~ /\S+•{1}\S+/) {  
    $tempbuf =~ s/\S+•{1}\S+/$1•$2/;
```

When searching for double spaces (*\$reps* = 2), the two lines are equivalent to:

```
if ($tempbuf =~ /\S+•{2}\S+/) {  
    $tempbuf =~ s/\S+•{2}\S+/$1•$2/;
```

or, to put it in another way:

```
if ($tempbuf =~ /\S+••\S+/) {  
    $tempbuf =~ s/\S+••\S+/$1•$2/;
```

When given the pattern *\S+•{*n*}*, Perl will look for the presence of exactly *n* consecutive spaces preceded by one or more nonwhitespaces. And if the pattern is enclosed in parentheses, Perl will remember the substring that matches this pattern. The remembered substrings can be accessed via the special variables, *\$1*, *\$2*, *\$3*, etc.

As an example, testing the string *fathers•brought•forth* for the pattern *(\S+•{1}) (\S+)*, we find that the substring *fathers•* matches the first subpattern, *(\S+•{1})*; thus, *fathers•* gets assigned to *\$1*. Likewise, the substring *brought* matches the second subpattern, *(\S+)*, and gets assigned to *\$2*. And so, the original string, *fathers•brought•forth*, now becomes *fathers••brought•forth*.

What makes this implementation so challenging is that it is not readily apparent that we need to include the nonwhitespaces in the search. Naively searching for whitespaces only, Perl will only replace the first whitespace it sees and not the spaces in the middle of the line. The result is that only the whitespaces at the beginning and end of the line get replaced. It’s important to realize that the nonwhitespaces serve as reference points in the substitution.

Let’s now turn our attention to line 84. I’ve only included this line to make *pretty* as idiot-proof as possible. Line 84 handles a special case. Here’s the scenario: Suppose we want a line to be 12 characters wide, and we have the string, *democratic•institutions*. We can see immediately that only the word, *democratic*, can fit on the line buffer. With only one word, it’s impossible to make the line both left- and right-justified.

Line 85 will repeatedly search for at least one whitespace between two nonwhitespaces. Finding only one nonwhitespace on the line, line 85 will fail, and two things will happen:

1. *\$tempbuf* will never be updated (line 86). As a consequence, there will be an infinite loop on line 83.
2. Since the *if* condition on line 85 fails, the alternative *else* condition increments *\$reps* continuously (line 90).

Soon enough, Perl will complain and will generate error messages similar to the following:

```
Quantifier in {,} bigger than 32766 in regex; marked by <-- HERE  
in m/(\S+ { <-- HERE 32767}) (\S+)/ (#1)  
(F) There is currently a limit to the size of the min and max  
values of the {min,max} construct. The <-- HERE shows in the  
regular expression about where the problem was discovered. See  
perlre.
```

The infinite loop on line 83 was averted because *\$reps* exceeded the maximum value of the *{min,max}* regular expression construct.

The fix, therefore, is to check if *\$tempbuf* has any embedded spaces. If there is a space in it, then we are sure that there are at least two words in *\$tempbuf*. Break out of the *while* loop as soon as possible if no space is found—this means that *\$tempbuf* contains only one word. In doing so, we print the one-word line as is, making that line left-justified.

Finally, recall our previous example:

```
fathers•brought•forth•on•this
```

After the first substitution, the string is *reversed*, and becomes

```
siht•no•htrof•thguorb••srehtaf
```

We see here that, for an odd number of substitutions, the last call to reverse is unnecessary. So, we employ another variable, *\$replacements_made*, to keep track of how many spaces have been inserted. *\$replacements_made*, initially set to 0 (line 78), is incremented every time a substitution is made (line 87). So, on lines 95–98, we check *\$replacements_made* to know whether to undo the last reverse. If *\$replacements_made* is odd, we reverse the string one more time (line 98); otherwise, we leave the string alone (line 96).

Sample Outputs

Examples 1–6 show some example output using *pretty*. Any of the lines at the top of these examples will produce the lines below them. Line width is set to 64 characters. Indentations are set to four single spaces. The input file is *gettysburg.txt* (available with the source code for this article at <http://www.tpj.com/source/>).

TPJ

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

Listing 1

```
1  #!/usr/local/bin/perl
2  use diagnostics;
3  use strict;
4  use warnings;
5  use Getopt::Long;
6
7  my ($width, $help, $left, $centered, $right, $both);
8  my ($indent, $newline);
9  GetOptions("width=i" => \$width, "help" => \$help,
10 "left" => \$left, "centered" => \$centered,
11 "right" => \$right, "both" => \$both,
12 "indent:i" => \$indent, "newline" => \$newline);
13
14 syntax() if ($help or !$width);
15 $indent = 0 if (!$indent);
16
17 local $/ = "";
18
19 while (<>) {
20     my @linein = split;
21     printpar(@linein);
22     print "\n" if ($newline);
23 }
24
25 sub printpar
26 {
27     my (@par) = @_;
28     my $firstline = 0;
29
30     while (scalar @par) {
31         $firstline++;
32         my ($buffer, $word);
33         my ($charcount, $wordlen) = (0, 0);
34         my $linewidth = $width;
35
36         if ($firstline == 1) {
37             $linewidth -= $indent;
38             print " " x $indent;
39         }
40
41         while (($charcount < $linewidth) and (scalar @par)) {
42             $word = shift @par;
43             $buffer .= $word;
44             $wordlen = length($word);
45
46             if ($wordlen > $linewidth) {
47                 print "\nERROR: The word \"$word\"";
48                 print " ($wordlen chars) cannot be accommodated\n";
49                 exit 1;
50             }
51
52             $charcount += $wordlen;
53             $buffer .= " ";
54             $charcount++;
55         }
56
57         chop $buffer;
58         $charcount--;
59
60         my $lineout = $buffer;
61
62         if ($charcount > $linewidth) {
63             unshift(@par, $word);
64             $charcount -= $wordlen;
65             $charcount--;
66             $lineout = substr $buffer, 0, $charcount;
67         }
68
69         my $spaces_to_fill = $linewidth - $charcount;
70
71         if ($centered) {
```

```
72     my $leftfill = int($spaces_to_fill/2);
73     print " " x $leftfill;
74 } elsif ($right) {
75     print " " x $spaces_to_fill;
76 } elsif ($both) {
77     my $tempbuf = $lineout;
78     my $replacements_made = 0;
79
80     if (scalar @par) {
81         my $reps = 1;
82
83         while (length($tempbuf) < $linewidth) {
84             last if ($tempbuf !~ /\s/);
85             if ($tempbuf =~ /\S+ {$reps}(\S+)/) {
86                 $tempbuf =~ s/\S+ {$reps}(\S+)/$1 $2/;
87                 $replacements_made++;
88                 $tempbuf = reverse $tempbuf;
89             } else {
90                 $reps++;
91             }
92         } # while
93     }
94
95     if ($replacements_made % 2 == 0) {
96         $lineout = $tempbuf;
97     } else {
98         $lineout = reverse $tempbuf;
99     }
100 }
101
102 print "$lineout\n";
103 }
104 }
105
106 sub syntax
107 {
108     print "Options:\n";
109     print "--width=n (or -w=n or -w n)   Line width is n chars ";
110     print "long\n";
111     print "--left (or -l)                 Left-justified";
112     print " (default)\n";
113     print "--right (or -r)                 Right-justified\n";
114     print "--centered (or -c)             Centered\n";
115     print "--both (or -b)                 Both left- and\n";
116     print "                                right-justified\n";
117     print "--indent=n (or -i=n or -i n)   Leave n spaces for ";
118     print "initial\n";
119     print "                                indentation (defaults\n";
120     print "to 0)\n";
121     print "--newline (or -n)                 Output an empty line\n";
122     print "                                between ";
123     print "paragraphs\n";
124     exit 0;
125 }
```

TPJ

On Perl as a Natural Language

With a formal education in linguistics, Larry Wall created the computer programming language Perl in 1987. Given his liberal attitudes, Wall has minimally guided the Perl community's growth for more than 15 years now. He has, however, retained control over the committee responsible for Perl's linguistic development so that he may ensure that Perl adheres to his linguistic vision. Part of Wall's linguistic philosophy appears in his famous essay entitled "Natural Language Principles in Perl." In his essay, Wall argues that Perl is superior and very different from all other computer languages because it is a natural language, much like English. In this article, I will explore Wall's natural language principles as they relate to Perl, expand on them, and give code examples, as well as some analogies to English where appropriate.

Learning Perl

The first principle that Wall cites as proof of Perl as a natural language is the fact that with Perl, one can be very expressive. "You learn a natural language once and use it many times" (Wall). Unlike the designers of the programming language BASIC, which was designed to make it easy for beginners to learn the language fully, Perl designers haven't shied away from complexities in order to make the language easier to learn. This would take away from the possibilities of Perl. For instance, if you only know the language English and you're trying to learn Italian, you might find it difficult when faced with the various inflections of the definite article (that is to say, *the*). In Italian, you use *lo*, *l'*, *il*, *la*, *gli*, *i*, or *le*, depending on the gender of the associated noun and whether it's singular or plural, as well as the first one or two letters of the noun.

In English, it's *the* in all cases, regardless of quantity or gender or the starting letters of the noun that follows. Eliminating inflections from Italian would make it easier for Americans to learn the language, but it would also eliminate some of its richness and functionality.

In Perl, although a flow-control statement such as *while* may be a little difficult for a newcomer to programming to comprehend, once you learn it, you are able to use it many times and never outgrow it. You can feel some comfort as you begin each new script knowing that you have already acquired some skills.

The English language is immense. The second edition of the Oxford English Dictionary contains about 290,000 entries, with about 616,500 word forms. This is an enormous number of words for

anyone to learn. However, "an educated person has a vocabulary of about 20,000 words and uses about 2,000 in a week's conversation" (Wilton). These kind of numbers are common to most users of natural languages. As a result, Wall says, "Nobody has ever learned any natural language completely." In Perl, there are hundreds of built-in functions and hundreds of commands and operators. Also, there are tens of thousands of extension objects and modules. CPAN, the depository of public Perl modules, boasts over 5500 modules. Each module has its own commands and methods. This adds up to well over 10,000 "words," half of what an average educated person knows in a native spoken language. With these kind of numbers, no one can be expected to learn every command and nuance of Perl. Not even Larry Wall knows it all—it's not practical. Therefore, you're not expected to learn all of Perl. To start with, you just need to learn what you need to accomplish your tasks. With Perl, if your vocabulary is minimal and your methods are simple, although your code might not be very tight, it can still be useful in communicating what you want. And that makes it valid.

The third principle of a natural language that Larry Wall suggests about Perl is related to the previous one. It has to do with the acceptance of many levels of competence in Perl programmers by the Perl community. Wall says that "if a language is designed so that you can 'learn as you go,' then the expectation is that everyone is learning, and that's okay." It is this mature attitude that accounts for the success of Perl community web sites like The Perl Monastery (<http://perlmonks.org/>). Perl monks of all levels (from novice to monk to saint) can post questions on the site about Perl and not feel ignorant, nor worry about being ridiculed for asking for help. Again, it's understood that we're all learning Perl and that we're all at different levels of competence, improving at our own pace.

The same attitude about many levels of competence exists to some extent in spoken human languages. People don't tend to poke fun at a child for having a small vocabulary and for writing simple sentences. Nor do people of polite company ridicule people of lesser formal education for their limited vocabulary and weak grammar skills. We are aware of these shortcomings and we react to them at times, but we don't necessarily find fault in the speakers because of them. We accept them as they are and interface with them on their terms as best we can. What's important is each individual's need to express himself.

Influences and Borrowings

A phrase that is often repeated in the Perl community is, "there's more than one way to do things in Perl." This is another aspect of the expressiveness of Perl. There's no one way or "right" way to write a Perl script. Each programmer has an individual style and way of communicating through code. Some use *strict* and declare

Russell is a Perl programmer, MySQL developer, and web designer living and working on a consulting basis in New Orleans. He is also an adjunct instructor at a local college where he teaches Linux and other open-source software. He can be reached at russell@dyerhouse.com.

all variables. Some programmers are sloppier and don't allow for error checking. Some take a couple hundred lines of code to say what could be said more efficiently in fewer than 50 lines. And some use one- and two-letter names for variables while others use lengthier descriptive names. These deviations are because of the flexibility inherent in Perl and because Perl and the members of the Perl community have grown out of several different programming languages and backgrounds. A programmer who learned the structured language of C before learning Perl will write a different program to accomplish the same task as someone who first learned the object-oriented language of Java. Neither method is the right way; which method is better is debatable. You have to do what works for you, what conforms to your background and your skills.

One of the reasons for the large number of words and seemingly inconsistent pronunciations in English is due to the fact that many words have been borrowed from other languages and eventually became part of English. For instance, many English words come directly from French (e.g., entrepreneur) and German (e.g., kindergarten). In Perl, it's very much the same. Perl has borrowed commands and methods from C, sed, awk, Lisp, Python, shell, and a few others, in addition to English. In fact, Perl continues to get ideas from other languages. For instance, while there is the command *case* in C, until the Perl module *Switch* (by Jarkko Hietaniemi) added *case* to Perl by extension, one had to use a series of *if* and *elsif* statements to accomplish the same effects. With Version 6 of Perl, though, *case* will become part of the vocabulary of Perl without the use of a module. Whereas it was felt unnecessary over the years by some of the designers to include a command for *case* in Perl, an individual on his own decided to create a module to make it available for all and, thereby, has proved the need for it to be part of standard Perl. Otherwise, "efforts to maintain the 'purity' of a language only succeed in establishing an elite class of people" (Wall) and do nothing for the development of the language and thereby the service of the community that uses the language. Without changes coming from the community through modules and usage, Perl would be extremely limited and probably would not survive.

Roughness and Ambiguity

Coming from the previous French colony of New Orleans, Louisiana, I can appreciate the roughness allowed in Perl. In the old Northern European colonies, many believe that planning leads to success. Here in New Orleans, there is a cultural attitude that out of chaos comes creativity and then order and thereby greater success than one could have planned. With Perl, I can choose to *use strict* or not. I can ignore errors or I can stomp them out. More importantly, I can write a program in Perl that will work under basic and expected conditions to get started and I can refine the details later. Besides functionality, because there is more than one way to do things in Perl, I can write a program in a simple manner to solve an immediate problem and then go back later to tighten the code to allow for more possibilities and to improve performance. "In terms of [a written] language [like English], you say something that gets close to what you want to say, and then you start refining it around the edges" (Wall). This feature of flexibility and expandability (or rather contractability) makes Perl easier to learn and use, and it encourages true creativity.

Since a variety of styles are possible, each programmer can take pride in his programming style or he can strive strictly for functionality. "Natural languages are used by people who for the most part don't give a rip how elegant the design of their language is. Ordinary folks scatter all sorts of redundancy throughout their communication to make sure of being understood" (Wall). I know that there are times when I could use the default variable of `$_`, but choose to name a variable for clarity. There are times when a public module or one of my own private libraries could be clean-

er and take up less code. However, I will choose to hack through my own code within the script I'm working on just to keep things more straightforward and obvious for myself and anyone who comes behind me. "Stylistic limits should be self-imposed, or at most be policed by consensus among your buddies" (Wall).

One flexible component of Perl that many outsiders seem to dislike is the allowing of local ambiguity. "Generally, within a natural language, ambiguity is resolved rapidly using recently spoken words and topics" (Wall). In English, if one is careful of syntactical elements (i.e., the word order and proximity), pronouns can be

It's understood that we're all learning Perl and that we're all at different levels of competence, improving at our own pace

used and reused without risk of confusing the listener. For instance, consider this sentence: "My brother said that his boss told him that he needed him to work late even though he wanted to leave early." Without asking me any questions or reading any other related sentences, you probably know who I'm referring to each time I say "him" and "he." The same situation exists in Perl. "There are a number of pronouns in Perl: `$_` means 'it', and `@_` tends to mean 'them'" (Wall). Therefore, we can write a script like this:

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

use strict;

@_ = qw(one two three);
$_ = 'test';

print $_;

foreach $_ (@_){
    print $_;
}

print $_;

exit;
```

After the obligatory initial lines, we set our initial variables. We load the default array (`@_`) with the words, one, two, and three. We then set the default scalar variable (`$_`) to a value of test. Before starting the flow control statement, the script prints the value of `$_` to the screen. The *foreach* statement reads through each element of the array, places each in `$_`, and then prints it out before going onto the next element. When all of the elements of the array have been processed, the script leaves the scope of the *foreach* and then prints the value of `$_`, which has now reverted back to test due to its context and without being instructed. The results from running this script are as follows:

Perl understands, in both instances, that `$_` outside of the *foreach* statement means test and that within it means the value of each element of `@_`. That's just like a pronoun in English and it's something that other programming languages don't do. It's this flexibility that makes Perl better and makes it difficult for programmers coming from a more rigid programming language to learn Perl. By the way, with the exception of the fourth line, the `$_` could be eliminated everywhere else and the script will still work:

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

use strict;

@_ = qw(one two three);
$_ = 'test';

print;

foreach (@_){
    print;
}

print;

exit;
```

In this tighter script, we nod to what we want and Perl knows that we mean the pronoun and knows what its value should be in each context.

Clarifications

"Part of the reason a language can get away with certain local ambiguities is that other ambiguities are suppressed by various mechanisms. English uses number and word order..." (Wall). English also uses syntax to give the listener signals as to what is meant by each use of a pronoun. "Similarly Perl has number markers on its nouns" (Wall). So `$color` contains one element (one color) and is singular, while the array `@color` potentially contains more than one element and is plural. "So `$` and `@` are a little like 'this' and 'these' in English" (Wall)—`$color` basically means this color and `@color` means these colors. While case is not a factor with Perl, it does make distinctions sometimes by word order. The example Wall gives in his essay is that *sub use* starts a subroutine named "use" and *use sub* calls a module named "sub."

Another signal to the listener for clarifying and reducing ambiguity is the use of topicalizers. "A topicalizer simply introduces the subject you're intending to talk about" (Wall). In English, this can be done with just a few words at the beginning of a sentence: "As for myself" and "With regard to the students" are both topicalizers to orient the listener to the context by which the words that follow will be made. In Perl, *for* and *foreach* statements are topicalizers. For instance, `foreach(@color) { print; }` will print the pronoun `$_` based on the context of the `@color` elements, not what `$_` meant before and after the `@color` topic is introduced and discussed.

Bigger Pictures and Conclusion

With artificial languages, rules about discourse structures can be rigid. The order in which code is laid out is sometimes irrelevant. For instance, you could group related lines of code and consider them to be paragraphs. These paragraphs can be dropped into different subroutines or functions. By using functions, one could greatly mix up the order of a script. "Perl tends to be pretty free about what order you put your statements, except that it's rather Aris-

totelian in requiring you to provide an explicit beginning and end for larger structures, using curlies" (Wall).

"Because a language is designed by many people, any language inevitably diverges into dialects" (Wall). In a sense, some of the Perl modules can represent specialized vocabularies and grammars; they can be said to be dialectal differences in Perl. For instance, Perl web developers often speak in *CGI.pm* terms. MySQL developers are typically well versed in *DBI.pm*. In some ways, this is analogous to comparing speakers of various New Orleans dialects and speakers of the dialects of New York City. Speakers of

*It's this flexibility that makes
Perl better and makes it
difficult for programmers coming
from a more rigid programming
language to learn Perl*

both sets of regional dialects have a commonality in English—they both know standard English. The same is true for CGI and DBI Perl programmers: While they have their specialized nouns and verbs associated with each module, they also know the basic commands of Perl (i.e., *if* and *print*), which could be considered the koine dialect. On the other hand, referring to the use of a module as a dialect may not be appropriate: "Differences in language that depend on who we are constitute dialect. Differences in language that depend on where, why, or how we are using language are matters of register" (Pyles). One could argue that a web developer uses the CGI module because she's a web developer and that would make CGI.pm a dialect. One could also argue that she uses CGI.pm because she's trying to be more efficient in web development and, therefore, the module is a register.

Ultimately, what makes a language natural is its growth that comes out of the community, out of its users diverging and creating new words and new grammar rules. This kind of growth cannot be controlled by anyone effectively. Creativity comes naturally from the chaos and anyone and everyone can have an effect. "We all contribute to the design of our language by our borrowings and coinages, by copying what we think is cool and eschewing what we think is obfuscatory" (Wall). Perl is not Larry Wall's language, it's our language. Unlike a Microsoft language, its source code is open to all of us and anyone can be part of the design process for Perl. Just by using it, we are helping it to grow and are adding to its status as a natural language.

References

Pyles, Thomas and John Algeo. *The Origins and Development of the English Language*, 4th ed. 1993. Harcourt Brace & Company. Fort Worth, Texas.

Wall, Larry. "Natural Language Principles in Perl." <http://www.wall.org/~larry/>.

Wilton, David. "How many words are there in the English language?" May 2003. <http://www.wordorigins.org/>.



Cooking with Maypole, Part I

Simon Cozens

We've all done it. We need to whip up a web application based on a database or some other structured data source; we drag out our favorite templating library, our favorite database abstraction layer, and write a bunch of code to glue the two together, maybe using something like *CGI::Application* to handle some of the front-end work. We extend the application in particular ways as the specifications change, but at least we have something working.

Then, the next time we need to create a database-backed web application, we drag out the same tools and do it again. And again. Wouldn't it be nice if there were a decent, extensible web application framework like Java's Struts? Well, there are one or two: OpenInteract and OpenFrame, to start with. But in this article, and its follow-up next month, I'd like to talk about a new one: Maypole.

Maypole started when I couldn't find any tools to "magically" put a web front end on a database without the need for lots of code or customization. Since then, it's become generalized to support extensions in all directions: adding functionality to the application itself using different templating modules and database abstractions. However, it's retained its focus of allowing you to get as much done as possible without writing very much code.

To introduce Maypole and some other friends, we'll look at an application I described a few months ago—the house-sharing database. I mentioned then that I planned to extend the application to track what food was in the cupboards. We're going to do that now, and our eventual application will also collect a database of recipes and have the system suggest the best recipe to use up the food before it goes off.

Tracking Provisions

We'll start, as usual, by developing our database schema, beginning with the recipes. The recipes will come in as XML, as we'll see in a few moments, so the table looks like this:

```
CREATE TABLE recipe (
  id int not null auto_increment primary key,
  name varchar(255),
  xml text
);
```

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

Now, each recipe will have multiple ingredients, so we need a many-to-many table linking ingredients to recipes:

```
CREATE TABLE ingredient (
  id int not null auto_increment primary key,
  recipe int,
  food int,
  quantity varchar(255)
);
```

Then, we're going to need a table for the names of food. One of the things you notice about recipes is that they're not always consistent about their use of the names of ingredients. "Lamb mince" and "minced lamb" are the same thing, so we'll use a column in the table to identify the "canonical" name of a food to help us when we're importing recipes. We'll call this the normalized name, and so our schema looks like this:

```
CREATE TABLE food (
  id int not null auto_increment primary key,
  name varchar(255),
  normalized varchar(255)
);
```

The recipes have categories as well as ingredients, so here's another many-to-many linking table:

```
CREATE TABLE category (
  id int not null auto_increment primary key,
  name varchar(255)
);

CREATE TABLE categorization (
  id int not null auto_increment primary key,
  recipe int,
  category int
);
```

And finally, there's a table for the food in the cupboards. I called my database "larder," so "larder.contents" is a good name for this table:

```
CREATE TABLE contents (
  id int not null auto_increment primary key,
  food int,
```



```

    quantity varchar(255),
    use_by date
);

```

Now I could use `Class::DBI` to set up the database, create classes for each of the tables, load them up with the column names, declare the relationships between them, and start writing my templating controller code. Or I could just say (see Listing 1):

```

package Larder;
use base 'Apache::MVC';
Larder->setup("dbd:mysql:larder");

```

`Apache::MVC` is a subclass of Maypole that is specifically suited to creating Apache `mod_perl` handlers, and it takes care of pretty much everything.

We still need to declare our relationships between the tables, but we'll do this in several parts: First, we'll get a system going where we can add and edit the contents of our larder, then we'll look at displaying recipes as well; finally, in next month's article, we'll look at how to link the recipes with their ingredients and also the ingredients with what's in the larder.

To relate the food types in the contents table to the food table, we use a `has_a` relationship:

```

Larder::Contents->has_a( food => "Larder::Food" );

```

Deploying the Web Site

Believe it or not, we're almost done with setting up a simple interface to the database. All we need to do is tell the system where it's going to live, so that it can construct URLs to itself, and we're done:

```

package Larder;
use base 'Apache::MVC';
Larder->setup("dbi:mysql:larder");
Larder->config->{uri_base} = "http://localhost/larder/";
Larder::Contents->has_a( food => "Larder::Food" );
1;

```

Well, almost done. That's as much Perl as we need to write for the moment. The rest are HTML templates but, thankfully, Maypole includes a useful bunch of those, too. We copy the factory templates directory inside our web root:

```

% cp -r templates/factory /var/www/larder/

```

We probably want a CSS file in there also; the sample one provided with Maypole works well. And then set up the Apache configuration to use the `Larder` Perl module:

```

<Location /larder>
    SetHandler perl-script
    PerlHandler Larder
</Location>

```

That's all.

Restart Apache, go to the site, and you should be able to view, search, and browse the database. This works equally well for any database, and it's a great way to look around an established database that has a lot of relationships defined. But, of course, our larder database doesn't have any data in it at the moment.

Editing and Untainting

To allow us to add and edit records, we need to tell Maypole more about the kinds of data we're expecting. To do this, May-

pole makes use of `Class::DBI::Untaint`, an application of `CGI::Untaint`.

`CGI::Untaint` is a mechanism for testing to make sure that incoming form data conforms to various properties. For instance, given a `CGI::Untaint` object that encapsulates some POST parameters, we can extract an integer like so:

```

my $i = CGI::Untaint->new(CGI->Vars);
$i->extract(-as_integer => "food");

```

Maypole started when I couldn't find any tools to "magically" put a web front end on a database without the need for lots of code or customization

This checks that the `food` parameter is an integer and returns it if it is; if not, `$i->error` will be set to an appropriate error message. Other tests by which you can extract your data are `as_hex` and `as_printable`, which tests for a valid hex number and an ordinary printable string, respectively; there are other handlers available on CPAN, and you can make your own, as documented in `CGI::Untaint`.

To tell the Maypole handler what handler to use for each of your columns, you need to use the `untaint_columns` methods in the classes representing your tables. We have a class representing food in the cupboard, `Larder::Contents`, which has a string column `quantity` and a date column `use_by`, so we declare these:

```

Larder::Contents->untaint_columns(
    printable => [ "quantity" ],
    date      => [ "use_by" ]
);

```

This will allow us to edit these columns, but there's one column we've forgotten—the `food` column will have to contain an integer that refers to a primary key in the food table. Even though this will be displayed as a drop-down list, we need to ensure that the number passed in to the edit process is an integer:

```

Larder::Contents->untaint_columns(
    printable => [ "quantity" ],
    date      => [ "use_by" ],
    integer   => [ "food" ]
);

```

Now we can add and edit rows in the contents table.

Tomatoes and Triggers

The food table itself is slightly different. We have three columns, `id`, `name`, and `normalized`. The only one we need to edit is `name`, as the others should be hidden. Maypole automatically hides the `id` column, but we can specify the columns

to be displayed by overriding the *display_columns* method in the *Larder::Food* class:

```
package Larder::Food;
sub display_columns { "name" }
```

We also need to make sure that the *normalized* column gets set every time there's an update to *name*. We'll use a very simple canonicalization subroutine for demonstration purposes, but a proper foodstuff canonicalization routine needs to know about specialist cooking terms; for example, that chopped tomatoes and diced tomatoes are both tomatoes. Our routine is a common one for doing fuzzy searches: It ditches vowels and punctuation characters, and compresses whitespace and repeated letters.

```
sub normalize {
    my $word = lc shift;
    $word =~ s/[aeiou]//g;
    $word =~ s/[^a-zA-Z]//g;
    $word =~ tr/a-z /a-z /s;
    return $word;
}
```

To arrange this subroutine to set the value of *normalized*, we use a feature of *Class::DBI* called “triggers.” These are subroutine references fired off by particular events; the *Class::Trigger* class helps you to define these. For instance, every time a record is created: *Class::DBI*-based classes will call their *after_create* trigger. We can use this trigger to make sure that the *normalized* field is set to the right thing:

```
Larder::Food->add_trigger( after_create => sub {
    my $self = shift;
    $self->normalized(normalize($self->name));
} );
```

Similarly, when we update the name from the web interface, we want to ensure that the new name is correctly normalized, if the normalized version is now different:

```
Larder::Food->add_trigger( after_update => sub {
    my $self = shift;
    my $old = $self->normalized;
    my $new = normalize($self->name);
    if ($old ne $new) {
        $self->normalized($new);
    }
} );
```

Finally, we set our *untaint_columns* so we can update the name of a foodstuff:

```
Larder::Food->untaint_columns( printable => [ "name" ] );
```

The screenshot shows a web application titled "Listing of all content". It features a table with columns "Food", "Quantity", and "Use By". The table contains two rows: "Baked beans" (1 can, 2009-12-31) and "Rice" (450g). Each row has "edit" and "delete" buttons. To the right of the table is a "Search" section with input fields for "Food", "Quantity", and "Use By", and a "search" button. Below the table is an "Add a new contents" section with similar input fields and a "create" button.

Figure 1: Larder contents.

```
<recipe version="0.5">
  <recipe>
    <head>
      <title>Aioli</title>
      <categories>
        <cat>Salads</cat> <cat>Condiment</cat> <cat>Classic</cat>
      </categories>
    </head>
    <ingredients>
      <ing>
        <amt> <qty>2</qty> <unit></unit> <item>Cloves garlic</item>
      </ing>
      <ing>
        <amt> <qty>1/2</qty> <unit>cups</unit></amt>
        <item>Mayonnaise</item>
      </ing>
    </ingredients>
    <directions>
      <step> Mash garlic to a paste with salt and stir into
      mayonnaise.
    </step>
    </directions>
  </recipe>
</recipeML>
```

Example 1: A RecipeML recipe.

and now we have two tables we can display, edit, and add to. With a few bits of data in it, the site looks like Figure 1.

RecipeML

The next stage of this is to get the recipes into the system. The state of machine-readable recipes is somewhat lamentable at the moment, but I did find a large archive of recipes in RecipeML, an application of XML. A RecipeML recipe looks like Example 1.

We want to extract the categories and the ingredients, then dump the data into the database. First, we fire up our old friend *XML::Simple* to parse the XML into a Perl data structure:

```
use XML::Simple;
use File::Slurp;
for my $recipe (<xml/*>) {
    my $xml = read_file($recipe);
    my $structure = XMLin($xml)->(recipe);
}
```

XML::Simple has some quirky features, and one of these is that if there's an *<ingredients>* tag with several *<ing>* tags inside it, it will present these as an array, as one might expect; unfortunately, if there's only one ingredient, it presents it as a hash. Naturally, this causes problems when we come to dereference it. To get around this, we force *XML::Simple* to present everything as arrays, regardless of the number of subelements. This has the unfortunate side effect of making the rest of the code ugly, but at least it's consistently ugly:

```
my $structure = XMLin($xml, ForceArray => 1)->(recipe)->[0];
my $name = $structure->(head)->[0]->(title)->[0];
my @ingredients = @{$structure->(ingredients)[0](ing)};
my @cats = @{$structure->(head)[0](categories)[0](cat)};
```

We have all the data we need.

For the time being, we'll only worry about the name and the XML; the ingredients and categories require slightly trickier many-to-many relationships, so we'll deal with those in next month's column.

To create the database rows, we will load up the same *Larder* module that we've been using as an Apache handler, since this does the hard work of setting up the *Class::DBI* classes for us.

```
use Larder;
use XML::Simple;
use File::Slurp;
for my $recipe (<xml/*>) {
    my $xml = read_file($recipe);
```

```

my $structure = XMLin($xml, ForceArray => 1)->{recipe}->[0];
my $name = $structure->{head}->[0]->{title}->[0];
my @ingredients = @{$structure->{ingredients}[0]{ing}};
my @cats = @{$structure->{head}[0]{categories}[0]{cat}};
Larder::Recipe->find_or_create({
    name => $name,
    xml => $xml
});
}

```

XSL is a language, itself expressed in terms of XML, for turning one XML document into another

Now we have a load of recipes in our system and we can list them by visiting the URL `/larder/recipe/list`. However, we have one problem. Since Maypole tries to display all columns in a table by default, it shows the recipe XML alongside the name. We don't want it to display the XML by default, and we need to work out a way to turn that XML into HTML.

The first stage in this is to use the `display_columns` method as before to have it only display the name. We also need to put the recipe back into the list of allowable tables:

```

package Larder::Recipe;
sub display_columns { "name" }

```

Our final version of *Recipe.pm* will be shown next month. The next stage, actually displaying the XML as HTML, requires a bit of trickery.

Cooking the Source

We only want the recipe to be displayed in full when we go to the view page for an individual recipe; we can do this by writing our own custom view template. Maypole searches for templates in three directories to allow flexibility in overloading the default templates. When you go to a URL such as `/recipe/view/10`, it looks for the view template in the recipe directory; if there isn't one specific to the recipe table, it looks in the custom directory to allow you to specify site-wide defaults; finally, it checks the factory directory, which contains the generic templates that come with Maypole.

So we want to write our own custom template specifically for recipes, which should live in `recipe/view`. However, we still need to get the RecipeXML into HTML somehow. What we'll use to do that is called XSL.

XSL is a language, itself expressed in terms of XML, for turning one XML document into another. For instance, here's the fragment of an XSL stylesheet for transforming a recipe:

```

<xsl:template match="recipe">
<html>

```

```

<head><title><xsl:value-of select="head/title"/></title>
</head>
<body>
    <xsl:apply-templates />
</body>
</html>
</xsl:template>

```

Most of the tags here are supposed to be output verbatim, apart from the tags with the namespace *xsl:*, which are magic and refer to the transformation process itself. In these cases, the first line says, "if you see a recipe tag, spit out this chunk of XML." The third line outputs ordinary HTML head and title tags, then looks for the first title element inside a head tag in the source recipe XML, and outputs the value inside those tags. The *apply-templates* in the middle means "keep working through the source document and apply any other fragments for tags that you find."

To apply the XSL stylesheet to a recipe, we need to use an XSL processor. One of the best in the business (or at least, in the Perl business) comes from the GNOME project and is wrapped in the *XML::LibXSLT* module. We can take an *XML::LibXML* parser and parse a document:

```

use XML::LibXML;
my $parser = XML::LibXML->new();
my $source = $parser->parse_file("recipe.xml");

```

And then parse a stylesheet because that's just plain XML, as well:

```

my $xsl = $parser->parse_file("recipe.xsl");

```

And now we use *XML::LibXSLT* to turn that into a stylesheet object:

```

use XML::LibXSLT;
my $stylesheet_parser = XML::LibXSLT->new();
my $stylesheet = $stylesheet_parser->parse_stylesheet($xsl);

```

Now that stylesheet object can be used to transform the original recipe:

```

my $results = $stylesheet->transform($source);
print $stylesheet->output_string($results);

```

This should output some nice HTML for our recipe. Now we need to know how to get that nice HTML into the context of our web application.

Simmer and Serve

Since we'll be processing our Maypole templates using the Template Toolkit, the most natural way to do this is as a Template plugin. We're going to write our own plugin using the *XML::LibXSLT* module to transform the RecipeXML into HTML. We'll start by envisaging the syntax we want out of it, which will look something like this:

```

[% USE transform = XSLT("recipe.xsl") %]

[% recipe.xml | $transform %]

```

We can inherit from the basic filter class *Template::Plugin::Filter* and plan to override the two methods *init* and *filter* with methods that set up our XSLT parser and apply the stylesheet to the filtered text, respectively.

The *init* method needs to load the stylesheet, and may as well parse it and turn it into an *XML::LibXSLT::Stylesheet* object there and then. First, we get the name of the stylesheet we want to use; this will be provided in the `_ARGS` slot of the filter object:

```

sub init {
    my $self = shift;
    my $file = $self->{ _ARGS }->[0]
    or return $self->error('No filename specified!')
}

```

Next, we load up the parsers and try parsing the stylesheet's XML:

```

$self->{ parser } = XML::LibXML->new();
$self->{ XSLT } = XML::LibXSLT->new();
my $xml;
eval {
    $xml = $self->{ parser }->parse_file($file)
};
return $self->error("Stylesheet parsing error: $@" ) if $@;
return $self->error("Stylesheet parsing failed") unless $xml;

```

If that works, then we can try feeding the stylesheet to *XML::LibXSLT*:

```

eval {
    $self->{ stylesheet } =
        $self->{ XSLT }->parse_stylesheet( $xml );
};
return $self->error("Stylesheet not valid XSL: $@" ) if $@;
return $self->error("Stylesheet parsing failed")
    unless $self->{stylesheet};
return $self;

```

This handles what happens when the *USE* call is made; the *filter* method is called when the returned object is used as a filter.

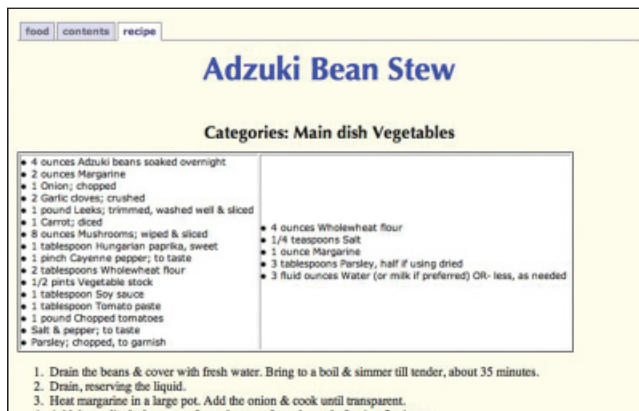


Figure 2: A recipe viewed using an XSL template.

This gets handed some text and needs to parse it:

```

my ($self, $text) = @_;
my $xml;
eval { $xml = $self->{ parser }->parse_string($text); };
return $self->error("XML parsing error: $@" ) if $@;
return $self->error("XML parsing failed") unless $xml;

```

And then it needs to apply the stylesheet to it:

```

return $self->{ stylesheet }->output_string(
    $self->{ stylesheet }->transform( $xml )
);

```

That's essentially the core of the *Template::Plugin::XSLT* module, which I wrote precisely in order to display these recipes. Now we can write our recipe/view template, based on the generic one in *factory/view*:

```

[% INCLUDE header %]
<h2> [% recipe.title %] </h2>
[% INCLUDE navbar;
USE transform = XSLT("recipe.xml");
recipe.xml | $transform %]

```

We can now view a recipe using an XSL template. Figure 2 is one I prepared earlier.

Serving Suggestion

We've looked at a wide variety of things in this article: using XSLT to transform XML into HTML; writing Template Toolkit filter plugins; using *Class::DBI::FromCGI* to restrict the possible input for form fields; and, of course, using Maypole, a new web application framework.

While we've not gone into much depth about how Maypole does its stuff—we'll look at that next month—I hope I've given you the flavor (ho, ho) of how easy it is to construct web applications in Maypole. We've put together an interface to a larder inventory system, together with an XSL-based recipe display in around 40 lines of Perl code.

Next month, we'll look at linking recipes with their ingredients and searching for optimal recipes to use up fading food! Until then, happy cooking!

TPJ

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

Listing 1

```

package Larder;
use strict;
use base 'Apache::MVC';
Larder->setup("dbi:mysql:larder");
Larder->config->{display_tables} = [qw[food contents recipe]];
Larder->config->{uri_base} = "http://localhost/larder/";
Larder::Contents->has_a( food => "Larder::Food" );
Larder::Contents->untaint_columns(
    printable => [ "quantity" ],
    date      => [ "use_by" ],
    integer   => [ "food" ]
);

package Larder::Food;
sub display_columns { "name" }
Larder::Food->add_trigger( after_update => sub {
    my $self = shift;
    my $old = $self->normalized;
    my $new = normalize($self->name);
    if ($old ne $new) {

```

```

        $self->normalized($new);
    }
} );
Larder::Food->add_trigger( after_create => sub {
    my $self = shift;
    $self->normalized(normalize($self->name));
} );

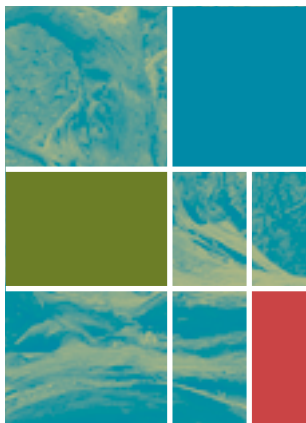
Larder::Food->untaint_columns( printable => [ "name" ] );
sub normalize {
    my $word = lc shift;
    $word =~ s/[aeiou]/g;
    $word =~ s/[^s\w]/g;
    $word =~ tr/a-z /a-z /s;
    return $word;
}

package Larder::Recipe;
sub display_columns { "name" }

1;

```

TPJ



Graphical Interaction with POE and Tk

Randal L. Schwartz

In an article I wrote for *The Perl Journal* a year ago (“Tailing Web Logs, April 2003), I introduced the Perl Object Environment, better known as “POE,” as a means of executing many tasks at once. Recently, I’ve been playing with the Tk toolkit through the Perl-Tk interface. With Perl-Tk, I can write Perl programs that use standardized graphical widgets that respond to various interactions.

Perl-Tk is an event-driven framework. A large portion of writing a graphical application consists of responses to various events, such as pressing a button or typing into a text field. Perl-Tk also provides for a few, simple noninterface events as well: the passage of time, and a filehandle becoming ready to read or write. In theory, these two additional event categories are sufficient to write nearly everything I was doing with POE. But in practice, it’s silly to reinvent the myriad of gadgets that the POE community has already created.

But why choose one or the other? The POE event loop can safely coexist with the Perl-Tk event loop! In fact, it’s as simple as keeping in mind a few important rules:

- Start your POE/Tk program with `use Tk` before `use POE`. This informs the POE framework that it will have to play nice with Perl-Tk.
- Use `$poe_main_window` (exported automatically) as Perl-Tk’s `MainWindow`, instead of instantiating one of your own. POE’s events have to be bound to Perl-Tk’s active main window, and this puts the hooks in place.
- It’s easiest to set up your graphic interface in the `_start` state in your primary POE session. This ensures that the graphic window is initialized and that POE is running the event loop.

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

As a simple application of a POE-Tk program, I chose to make use of the recently released “PoCo” (POE Component) called “*POE::Component::RSSAggregator*.” This PoCo automatically fetches and parses RSS feeds, providing a callback when the feed has new headlines. This PoCo was originally created to feed an IRC bot with news headlines, but thanks to the pluggability of POE code, I created a GUI-based newsreader instead.

I chose a tabbed-notebook-style interface, with each news feed being a tab along the top. Selecting a tab brings up the headlines, with new headlines in bold. Single-clicking on a headline marks it as read. Double-clicking on a headline not only marks it as read but also fires off a command to open the corresponding detail URL with my favorite browser. And all this is in the 145 lines of code presented in Listing 1. (Please note that I’m using the hot-off-the-presses Tk 804 release with this program. I’ve heard from one of my reviewers that the code may not work on older releases.)

Lines 1–3 start nearly every program I write, enabling warnings, compiler restrictions, and unbuffering STDOUT. For this program, a buffered STDOUT really didn’t hurt because there weren’t any print operations, but I do these lines out of habit now.

Lines 5–19 provide the user-serviceable parts of the program. The idea is that everything after line 19 can be left alone.

First, lines 7–11 give the `@FEEDS` that I’m monitoring. Each line has three whitespace-delimited fields. The first field is the unique short name for the feed. Because this name also goes on the tab (and the tabs don’t wrap), I’m interpreting a vertical bar in the feed name as a line break for the tab. The second field is the frequency in seconds with which to go back to check for updates. Here, 900 seconds is 15 minutes: a good compromise between having fresh news and not angering system administrators. The third field is the RSS-feed URL.

As sample feeds, I’ve included searches for new use Perl; journals, new Slashdot journals, and a wonderful little link exchange called del.icio.us to which I’ve only recently been introduced, but seems to be a continual source of cool corners of the Internet. (In

fact, you may want to crank up the frequency of *fetch* a bit higher on that one to avoid missing anything, and be prepared to lose a few hours attempting to follow every link that comes by.)

Line 13 defines a path to a DBM that'll hold our already seen links between invocations of this program. While you'll probably want to just leave this thing running all day, you don't want to lose your place when you have to go away. The DBM provides a simple memory between invocations.

Lines 15–17 define the *LAUNCH* subroutine, which is very platform specific. On my Mac OS X box, I call *open* with a URL, and it launches my preferred browser on the URL. If you're using this program, you'll need to figure out how to do that for yourself.

Note that I'm being a bit casual with the POE rules here: When this subroutine is invoked, it blocks for a bit while the command is executing, so no other events are being processed. If this wasn't a very fast launcher, I'd need to use something like *POE::Wheel::Run* for the child process. But in practice, I don't care because while I'm launching a URL to see, I'm really not expecting any of the rest of my application to work properly.

Lines 21–26 define my needed globals: the POE session alias for the PoCo and the DBM hash for my persistent memory.

Line 28 cleans up my memory storage by deleting any headline that was seen more than three days ago. This seemed like a fair compromise, and is merely an optimization: The program would run fine without it. Had I expected this program to run for weeks at a time, I'd have put this code into a state that got invoked once a day or so. Again, I'm building this program to my expected usage, and not necessarily a general-purpose application.

Lines 30 and 31 pull in Tk and POE, as stated earlier. Lines 33–143 create the main POE session, providing all the user-side interface for this program. Setting that aside at the moment, we see that line 145 starts the POE main event loop, which exits only when the Tk main window is closed.

When the POE event loop starts, the only session created thus far is sent a *start* event, and this begins the code starting in line 36. Lines 37 and 38 extract the parameters sent to this state handler. Line 39 pulls in the *POE::Component::RSSAggregator* module, found in CPAN.

Lines 42 to 45 start the RSS PoCo. A separate session is created, named as *\$READER*, with a callback to our current session's *handle_feed* state when one of the feeds has changed state.

Lines 47–52 set up the main Tk widget for the interface. A standard tabbed notebook widget called *NoteBook* is established as a child widget of the *\$poe_main_window*, packed so that it fills that window.

Lines 54 and 55 add the initial (and only) feeds to be watched. I chose to do this as a separate state to keep it modular, and also because I eventually wanted to add some GUI components to manage feeds, rather than simply displaying feeds that were hard coded into the program.

Speaking of which, the handler for adding feeds begins immediately thereafter, starting in line 57. Lines 58 and 59 extract the context variables for this handler, as well as getting the *\$name*, *\$delay*, and *\$url* values from the yield call in line 55.

Lines 61–68 add the page for the tabbed notebook, as a *RO-Text* widget. This is a text widget with the text entry bindings removed, so that I won't be tempted to type into the area where the URLs are listed. Line 63 patches up the vertical bars into newlines for the tab label. Line 64 adds the page, identified by the given name, and with a label that indicates that we haven't yet fetched the feed. Additionally, a *Scrolled* widget is created to provide a vertical scrollbar when necessary for the list of headlines. Some of this took a bit of fudging around, but this seems to have created the look that I needed.

Lines 69–77 add the bindings to the text area to respond to single and double clicks. First, two tags are created for new links and already seen links, differing only in their visual presentation (new

links are in bold text). Then, *tagBind* is used to associate clicks on those tags with callbacks to our *handle_click* state handler, including an additional 1 or 2 parameter to indicate a single or double click. The *Ev* call also passes the relative *x-y* coordinates of the click, which we can translate into a line number rather easily (shown later).

Finally, once the tab and text window is established, we're ready to get headlines, as requested in lines 80 and 81. By posting an *add_feed* event to the *\$READER* session, we'll start getting callbacks to our *handle_feed* handler as the data arrives and is fully parsed.

*Start your POE/Tk program with
use Tk before use POE. This
informs the POE framework that it
will have to play nice with Perl-Tk*

Lines 84–104 handle any clicks in the text area, as designated by the bindings in lines 73–76. Let's set that aside for a minute because we can't get any clicks until we actually have some text.

Lines 105–113 handle a notification from the RSS PoCo that we've got a new *XML::RSS::Feed* object, arriving in *\$feed* in line 108. In this handler, we merely cache the value into our POE heap, and then trigger a callback to our own *feed_changed* handler.

The *feed_changed* handler (starting in line 114) is triggered when any feed possibly has new headlines, and also when items are clicked (which changes which items of a feed we've already seen). This handler's job is to take the model data into the view.

Lines 118–121 locate the feed object, the notebook widget, the specific notebook page widget, and the scrolled text widget. Line 124 remembers the scroll percentage for the current text view. If the list of URLs is long enough so that a scrollbar appears, I don't want the scroll bar to jump annoyingly whenever the headlines are updated.

Line 125 clears out any previous URL list. For simplicity, when any part of the data model changes, I start over on the display. For efficiency, I could have tried to compute a small set of differences and redrawn only those, but this was much simpler and, as they say, "fast enough."

Lines 127–137 add the URLs to the display. First, a count of new items is initialized to 0 (line 127). Then, for each of the headline objects, we set *\$tag* to either link or seen, depending on whether the headline's ID property is present in the *%SEEN* data, counting the headline as new if needed (lines 129–134). Finally, lines 135 and 136 insert the headline text, tagged appropriately, followed by an untagged newline.

Line 138 restores the current scroll position as best as it can, as saved in line 124. Lines 140 and 141 update the tab label with the new number of items that are as-yet unread. That way, I can quickly glance across the tabs to see if there's anything new to read.

Now, back to handling clicks. Lines 88–92 get the feed name, the number of clicks, the text widget, and the *x-y* coordinates within that text widget where the click occurred. But we don't want

the pixel of the click: We want the line number. Luckily, line 94 shows that we can simplify the expression with a simple call to the *index* method, which returns a line number, dot, and character number within that line.

Lines 96–103 mark the headline corresponding to that line to having been seen (at the current time), and triggers an event to redraw the window (described earlier), since the model data has now effectively changed. (Strictly speaking, it's a view aspect of the model that has changed, but this is close enough for practical purposes.) In addition, if it was a double click instead of a

single click, we call *LAUNCH* on the URL, which launches my browser.

And that's all there is to it! I now have a nice desktop GUI application that watches RSS feeds in exactly the manner of my choosing, with lots of room to add further GUI improvements and functionality. It's also nicely resizable, and deals with large amounts of data as well. So, consider using POE and Tk together for your next GUI application. Until next time, enjoy!

TPJ

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

Listing 1

```
=1=      #!/usr/bin/perl -w
=2=      use strict;
=3=      $|++;
=4=
=5=      ## config
=6=
=7=      my @FEEDS = map [split], split /\n/, <<'THE_FEEDS';
=8=      useperl[journal 900 http://use.perl.org/search.pl?op=journals&content_type=rss
=9=      slashdot[journal 900 http://slashdot.org/search.pl?op=journals&content_type=rss
=10=     del.icio.us 900 http://del.icio.us/rss/
=11=     THE_FEEDS
=12=
=13=     my ($DB) = glob("~/newsee");    # save database here
=14=
=15=     sub LAUNCH {
=16=         system "open", shift;      # open $_[0] as a URL in favorite browser
=17=     }
=18=
=19=     ## end config
=20=
=21=     ## globals
=22=
=23=     my $READER = "reader";          # alias for reader session
=24=     dlopen(my %SEEN, $DB, 0600) or die;
=25=
=26=     ## end globals
=27=
=28=     delete @SEEN[grep $SEEN{$_} < time - 86400 * 3, keys %SEEN]; # quick cleanup
=29=
=30=     use Tk;
=31=     use POE;
=32=
=33=     POE::Session->create
=34=     (inline_states =>
=35=     {
=36=         _start => sub {
=37=             my ($kernel, $session, $heap) =
=38=             @_ [KERNEL, SESSION, HEAP];
=39=             require POE::Component::RSSAggregator;
=40=
=41=             ## start the reader
=42=             POE::Component::RSSAggregator->new
=43=             (alias => $READER,
=44=              callback => $session->postback('handle_feed'),
=45=             );
=46=
=47=             ## set up the NoteBook
=48=             require Tk::NoteBook;
=49=             $heap->(nb) = $poe_main_window
=50=             ->NoteBook
=51=             (-font => [-size => 10]
=52=              )->pack(-expand => 1, -fill => 'both');
=53=
=54=             ## add the initial subscriptions
=55=             $kernel->yield(add_feed => @FEEDS);
=56=         },
=57=         add_feed => sub {
=58=             my ($kernel, $session, $heap, $name, $delay, $url) =
=59=             @_ [KERNEL, SESSION, HEAP, ARG0, ARG1, ARG2];
=60=
=61=             ## add a notebook page
=62=             require Tk::ROText;
=63=             (my $label_name = $name) =~ tr/|/\n/;
=64=             my $scrolled = $heap->(nb)->add($name, -label => "$label_name: ?")
=65=             ->Scrolled
=66=             (ROText => -scrollbars => 'oe',
=67=              -spacing3 => '5',
=68=              )->pack(-expand => 1, -fill => 'both');
=69=             ## set up bindings on $scrolled here
=70=             $scrolled->tagConfigure('link', -font => [-weight => 'bold']);
```

```

=71=         $scrolled->tagConfigure('seen');
=72=         for my $tag (qw(link seen)) {
=73=             $scrolled->tagBind($tag, '<1>',
=74=                 [$session->postback(handle_click => $name, 1), Ev('@')]);
=75=             $scrolled->tagBind($tag, '<Double-1>',
=76=                 [$session->postback(handle_click => $name, 2), Ev('@')]);
=77=         }
=78=
=79=         ## start the feed, getting callbacks
=80=         $kernel->post($READER => add_feed =>
=81=             {url => $url, name => $name, delay => $delay});
=82=
=83=     },
=84=     handle_click => sub {
=85=         my ($kernel, $session, $heap, $postback_args, $callback_args) =
=86=             @_ [KERNEL, SESSION, HEAP, ARG0, ARG1];
=87=
=88=         my $name = $postback_args->[0];
=89=         my $count = $postback_args->[1]; # 1 = single click, 2 = double click
=90=
=91=         my $text = $callback_args->[0];
=92=         my $at = $callback_args->[1];
=93=
=94=         my ($line) = $text->index($at) =~ /\^(\\d+)\\.\\d+$/ or die;;
=95=
=96=         if (my $headline = $heap->{feed}{$name}->headlines->[$line - 1]) {
=97=             $SEEN{$headline->id} = time;
=98=             $kernel->yield(feed_changed => $name);
=99=
=100=             if ($count == 2) { # double click: open URL
=101=                 LAUNCH($headline->url);
=102=             }
=103=         }
=104=     },
=105=     handle_feed => sub {
=106=         my ($kernel, $session, $heap, $callback_args) =
=107=             @_ [KERNEL, SESSION, HEAP, ARG1];
=108=         my $feed = $callback_args->[0];
=109=
=110=         my $name = $feed->name;
=111=         $heap->{feed}{$name} = $feed;
=112=         $kernel->yield(feed_changed => $name);
=113=     },
=114=     feed_changed => sub {
=115=         my ($kernel, $session, $heap, $name) =
=116=             @_ [KERNEL, SESSION, HEAP, ARG0];
=117=
=118=         my $feed = $heap->{feed}{$name};
=119=         my $nb = $heap->{nb};
=120=         my $widget = $nb->page_widget($name);
=121=         my $scrolled = $widget->children->[0];
=122=
=123=         ## update the text
=124=         my ($pct) = $scrolled->yview;
=125=         $scrolled->delete("1.0", "end");
=126=
=127=         my $new_count = 0;
=128=         for my $headline (@{$feed->headlines}) {
=129=             my $tag = 'link';
=130=             if ($SEEN{$headline->id}) {
=131=                 $tag = 'seen';
=132=             } else {
=133=                 $new_count++;
=134=             }
=135=             $scrolled->insert('end', $headline->headline, $tag);
=136=             $scrolled->insert('end', "\n");
=137=         }
=138=         $scrolled->yviewMoveto($pct);
=139=
=140=         (my $label_name = $name) =~ tr/|/\n/;
=141=         $nb->pageconfigure($name, -label => "$label_name: $new_count");
=142=     },
=143= });
=144=
=145= $poe_kernel->run();

```

TPJ



Python Programming: An Introduction to Computer Science

Jack J. Woehr

Python Programming: An Introduction to Computer Science, by John Zelle, is a very good first programming book for beginners. On reading the title, my first (cynical) reaction was, “Python as an introduction to computer science? Why not RPG or Snobol?” The point was that it seems on the surface quite difficult to introduce computer science adequately merely by teaching a somewhat exotic, string-oriented, high-level functional interpreter. For that matter, what does computer science rigorously construed have to do with the languages used in modern software engineering? Computer programmers are evermore dealing less and less in computer science: We’re becoming tailors using a set of patterns preordained by the language designer, patterns to which we cut our fabric all day. Like fish, we hardly notice the water in which we swim, live, and breathe.

The mystery, however, is soon solved. The author explains: “Teaching Python is not the main point of this book...rather, Python is used to illustrate fundamental principles of design and programming that apply in any language or computer environment.” While *Python Programming* partly defines the term “computer science” by way of Dijkstra’s famous quote: “Computer science is no more about computers than astronomy is about telescopes,” the book uses the term “computer science” loosely. *Python Programming: An Introduction to Computer Science* introduces the absolute minimum of computer science corresponding to the needs of an introduction to computer programming. Chalk it up to hyperbolic inflation: The book should have been called something like “Learning Computer Programming Through Python”—as indeed it would have been in the 1980’s before our mundane-albeit-highly-engaging technical discipline had become so precocious and exalted that it is no longer enough merely to teach programming.

Be that as it may, computer programming needs to be taught, and teaching it is no disgrace. At that humble chore, *Python Programming* acquits itself honorably. The book is one-on-one with the reader in the engaging fashion that characterized the better beginner self-tutorials of previous decades. In a similar manner, it breezes around the compass of interesting simple tasks that can be done with a minimum of code. The author derives his examples from an aesthetically acceptable nerdy viewpoint: The first exercise in the book is a chaotic function that accepts its input and renders its output in floating point.

As Guido van Rossum, creator of Python, states in the foreword:

The author mentions in his preface that Python is near-ideal as a first programming language, without being a “toy lan-

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

*Python Programming: An
Introduction to Computer Science*
John M. Zelle
Franklin, Beedle & Associates, 2004
ISBN 1-887902-99-6

guage”...I don’t want to take full credit for this: Python was derived from ABC, a language designed by Lambert Meertens to teach programming.

Python Programming follows a pathway through the language well suited to student needs. As Zelle explains:

There are places in the book where the “Pythonic” way of doing things has been eschewed for pedagogical reasons...This is done purposefully in an effort to meet students “where they’re at.”

Basic statements and assignments are followed by simple math, input/output, strings, files, and graphical objects. Then come functions and flow control. The rest of the book is dedicated to design, object orientation, and problem solving. The level throughout is suitable for late high school or early college.

The appendices include a quick reference to Python, a guide to using the IDLE development environment with Python, and a glossary of terms. These appendices are well focused and admirably brief.

The CD-ROM content, while adequate, exhibits, in contrast to the book itself, a lack of attention to detail that has come to characterize the majority of CD-ROM offerings accompanying programming books lately. There is no unified HTML presentation/explanation of the content starting with an `/index.html`, which would make things so much simpler for today’s beginner who knows a lot more about reading web-based content than about any other computer task. Then there’s the `.tar` file on the accompanying disk, which turns out really to be a `.tar.gz` file. Maybe it’s just too much work nowadays to prepare CD-ROM content with the same care with which a book is designed. In that case, the day of CD-ROMs accompanying textbooks is over and all content should simply be distributed over the Web from the publisher’s or author’s web site, in which case the content would (one hopes) be well indexed by default.

The web site for the book, which includes a chapter listing and a form to request an instructor review copy, is <http://www.fbeedle.com/99-6.html>.

TPJ

Source Code Appendix

H. Wade Minter “Audio on Demand with Mr. Voice”

Listing 1

```
if ( "$^O" eq "MSWin32" )
{
    our $srcfile = "C:\\mrvoice.cfg";

    BEGIN
    {
        if ( $^O eq "MSWin32" )
        {
            require LWP::UserAgent;
            LWP::UserAgent->import();
            require HTTP::Request;
            HTTP::Request->import();
            require Win32::Process;
            Win32::Process->import();
            require Tk::Radiobutton;
            Tk::Radiobutton->import();
            require Win32::FileOp;
            Win32::FileOp->import();
        }
    }

    $agent = LWP::UserAgent->new;
    $agent->agent("Mr. Voice Audio Software/$0 ");

    # You have to manually set the time zone for Windows.
    my ( $l_min, $l_hour, $l_year, $l_yday ) = ( localtime $^T )[ 1, 2, 5, 7 ];
    my ( $g_min, $g_hour, $g_year, $g_yday ) = ( gmtime $^T )[ 1, 2, 5, 7 ];
    my $tzval =
        ( $l_min - $g_min ) / 60 + $l_hour - $g_hour + 24 *
    $tzval = sprintf( "%2.2d00", $tzval );
    Date_Init("TZ=$tzval");
}
else
{
    our $homedir = "~";
    $homedir =~ s{ ^ ~ ( [^/]* ) }
        { $1
            ? (getpwam($1))[7]
            : ( $ENV{HOME} || $ENV{LOGDIR}
                || (getpwuid($>))[7]
            )
        }ex;
    our $srcfile = "$homedir/.mrvoicerc";
}
```

Julius C. Duque “Reformatting Text Using Pattern Matching”

Listing 1

```
1  #!/usr/local/bin/perl
2  use diagnostics;
3  use strict;
4  use warnings;
5  use Getopt::Long;
6
7  my ($width, $help, $left, $centered, $right, $both);
8  my ($indent, $newline);
9  GetOptions("width=i" => \$width, "help" => \$help,
10 "left" => \$left, "centered" => \$centered,
11 "right" => \$right, "both" => \$both,
12 "indent:i" => \$indent, "newline" => \$newline);
13
14 syntax() if ($help or !$width);
15 $indent = 0 if (!$indent);
16
17 local $/ = "";
18
19 while (<>) {
20     my @linein = split;
21     printpar(@linein);
22     print "\n" if ($newline);
23 }
24
25 sub printpar
26 {
27     my (@par) = @_;
28     my $firstline = 0;
29
30     while (scalar @par) {
31         $firstline++;
```

```

32 my ($buffer, $word);
33 my ($charcount, $wordlen) = (0, 0);
34 my $linewidth = $width;
35
36 if ($firstline == 1) {
37     $linewidth -= $indent;
38     print " " x $indent;
39 }
40
41 while (($charcount < $linewidth) and (scalar @par)) {
42     $word = shift @par;
43     $buffer .= $word;
44     $wordlen = length($word);
45
46     if ($wordlen > $linewidth) {
47         print "\nERROR: The word \"$word\"";
48         print " ($wordlen chars) cannot be accommodated\n";
49         exit 1;
50     }
51
52     $charcount += $wordlen;
53     $buffer .= " ";
54     $charcount++;
55 }
56
57 chop $buffer;
58 $charcount--;
59
60 my $lineout = $buffer;
61
62 if ($charcount > $linewidth) {
63     unshift(@par, $word);
64     $charcount -= $wordlen;
65     $charcount--;
66     $lineout = substr $buffer, 0, $charcount;
67 }
68
69 my $spaces_to_fill = $linewidth - $charcount;
70
71 if ($centered) {
72     my $leftfill = int($spaces_to_fill/2);
73     print " " x $leftfill;
74 } elsif ($right) {
75     print " " x $spaces_to_fill;
76 } elsif ($both) {
77     my $tempbuf = $lineout;
78     my $replacements_made = 0;
79
80     if (scalar @par) {
81         my $reps = 1;
82
83         while (length($tempbuf) < $linewidth) {
84             last if ($tempbuf !~ /\s/);
85             if ($tempbuf =~ /\S+ {$reps}(\S+)/) {
86                 $tempbuf =~ s/(\S+ {$reps})(\S+)/$1 $2/;
87                 $replacements_made++;
88                 $tempbuf = reverse $tempbuf;
89             } else {
90                 $reps++;
91             }
92         } # while
93     }
94
95     if ($replacements_made % 2 == 0) {
96         $lineout = $tempbuf;
97     } else {
98         $lineout = reverse $tempbuf;
99     }
100 }
101
102 print "$lineout\n";
103 }
104 }
105
106 sub syntax
107 {
108     print "Options:\n";
109     print "--width=n (or -w=n or -w n)   Line width is n chars ";
110     print "long\n";
111     print "--left (or -l)                 Left-justified";
112     print " (default)\n";
113     print "--right (or -r)                 Right-justified\n";
114     print "--centered (or -c)              Centered\n";
115     print "--both (or -b)                  Both left- and\n";
116     print "                                     right-justified\n";
117     print "--indent=n (or -i=n or -i n)   Leave n spaces for ";
118     print "initial\n";
119     print "                                     indentation (defaults ";
120     print "to 0)\n";

```

```

121 print "--newline (or -n)          Output an empty line \n";
122 print "                          between ";
123 print "paragraphs\n";
124 exit 0;
125 }

```

Randal L. Schwartz “Graphical Interfacing with POE and Tk”

Listing 1

```

1=      #!/usr/bin/perl -w
2=      use strict;
3=      $|++;
4=
5=      ## config
6=
7=      my @FEEDS = map [split], split /\n/, <<'THE_FEEDS';
8=      useperl|journal 900 http://use.perl.org/search.pl?op=journals&content_type=rss
9=      slashdot|journal 900 http://slashdot.org/search.pl?op=journals&content_type=rss
10=     del.icio.us 900 http://del.icio.us/rss/
11=     THE_FEEDS
12=
13=     my ($DB) = glob("~/newsee"); # save database here
14=
15=     sub LAUNCH {
16=         system "open", shift;      # open $_[0] as a URL in favorite browser
17=     }
18=
19=     ## end config
20=
21=     ## globals
22=
23=     my $READER = "reader";          # alias for reader session
24=     dnmopen(my %SEEN, $DB, 0600) or die;
25=
26=     ## end globals
27=
28=     delete @SEEN(grep $SEEN{$_} < time - 86400 * 3, keys %SEEN); # quick cleanup
29=
30=     use Tk;
31=     use POE;
32=
33=     POE::Session->create
34=     (inline_states =>
35=      {
36=        _start => sub {
37=            my ($kernel, $session, $heap) =
38=              @_ [KERNEL, SESSION, HEAP];
39=            require POE::Component::RSSAggregator;
40=
41=            ## start the reader
42=            POE::Component::RSSAggregator->new
43=              (alias => $READER,
44=               callback => $session->postback('handle_feed'),
45=              );
46=
47=            ## set up the NoteBook
48=            require Tk::NoteBook;
49=            $heap->{nb} = $poe_main_window
50=              ->NoteBook
51=              (-font => [-size => 10]
52=               )->pack(-expand => 1, -fill => 'both');
53=
54=            ## add the initial subscriptions
55=            $kernel->yield(add_feed => @$_) for @FEEDS;
56=        },
57=      add_feed => sub {
58=          my ($kernel, $session, $heap, $name, $delay, $url) =
59=            @_ [KERNEL, SESSION, HEAP, ARG0, ARG1, ARG2];
60=
61=          ## add a notebook page
62=          require Tk::ROText;
63=          (my $label_name = $name) =~ tr//\n/;
64=          my $scrolled = $heap->{nb}->add($name, -label => "$label_name: ?")
65=            ->Scrolled
66=            (ROText => -scrollbars => 'oe',
67=             -spacing3 => '5',
68=             )->pack(-expand => 1, -fill => 'both');
69=          ## set up bindings on $scrolled here
70=          $scrolled->tagConfigure('link', -font => [-weight => 'bold']);
71=          $scrolled->tagConfigure('seen');
72=          for my $tag (qw(link seen)) {
73=              $scrolled->tagBind($tag, '<1>',
74=                [$session->postback(handle_click => $name, 1), Ev('@')]);
75=              $scrolled->tagBind($tag, '<Double-1>',
76=                [$session->postback(handle_click => $name, 2), Ev('@')]);
77=          }
78=      }

```



```

=79=      ## start the feed, getting callbacks
=80=      $kernel->post($READER => add_feed =>
=81=          {url => $url, name => $name, delay => $delay});
=82=
=83=      },
=84=      handle_click => sub {
=85=          my ($kernel, $session, $heap, $postback_args, $callback_args) =
=86=              @_ [KERNEL, SESSION, HEAP, ARG0, ARG1];
=87=
=88=          my $name = $postback_args->[0];
=89=          my $count = $postback_args->[1]; # 1 = single click, 2 = double click
=90=
=91=          my $text = $callback_args->[0];
=92=          my $at = $callback_args->[1];
=93=
=94=          my ($line) = $text->index($at) =~ /^(\d+)\.\d+$/ or die;
=95=
=96=          if (my $headline = $heap->{feed}{$name}->headlines->[$line - 1]) {
=97=              $SEEN{$headline->id} = time;
=98=              $kernel->yield(feed_changed => $name);
=99=
=100=             if ($count == 2) {          # double click: open URL
=101=                 LAUNCH($headline->url);
=102=             }
=103=         }
=104=     },
=105=     handle_feed => sub {
=106=         my ($kernel, $session, $heap, $callback_args) =
=107=             @_ [KERNEL, SESSION, HEAP, ARG1];
=108=         my $feed = $callback_args->[0];
=109=
=110=         my $name = $feed->name;
=111=         $heap->{feed}{$name} = $feed;
=112=         $kernel->yield(feed_changed => $name);
=113=     },
=114=     feed_changed => sub {
=115=         my ($kernel, $session, $heap, $name) =
=116=             @_ [KERNEL, SESSION, HEAP, ARG0];
=117=
=118=         my $feed = $heap->{feed}{$name};
=119=         my $nb = $heap->{nb};
=120=         my $widget = $nb->page_widget($name);
=121=         my $scrolled = $widget->children->[0];
=122=
=123=         ## update the text
=124=         my ($pct) = $scrolled->yview;
=125=         $scrolled->delete("1.0", "end");
=126=
=127=         my $new_count = 0;
=128=         for my $headline (@{$feed->headlines}) {
=129=             my $tag = 'link';
=130=             if ($SEEN{$headline->id}) {
=131=                 $tag = 'seen';
=132=             } else {
=133=                 $new_count++;
=134=             }
=135=             $scrolled->insert('end', $headline->headline, $tag);
=136=             $scrolled->insert('end', "\n");
=137=         }
=138=         $scrolled->yviewMoveto($pct);
=139=
=140=         (my $label_name = $name) =~ tr//\n/;
=141=         $nb->pageconfigure($name, -label => "$label_name: $new_count");
=142=     },
=143= });
=144=
=145= $poe_kernel->run();

```

TPJ