

The Perl Journal

Web Localization & Perl

Autrijus Tang • 3

Data Manipulation & Perl Command-Line Options

Andy Lester • 8

Google and Perl

brian d foy • 12

Tracking Finances with *WWW::Mechanize* and *HTML::Parser*

Simon Cozens • 14

HTML Filtering in Perl, Part 1

Randal Schwartz • 18

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Piers Cawley:

***Programming Web Services with Perl* • 21**

Source Code Appendix • 23

LETTER FROM THE EDITOR

Apple's Musical Gambit

Apple Computer has sent word that it would like to be music to your ears. The company is talking, of course, about the launch of Music Store, its online music-purchase service initially for Mac users via Apple's free iTunes jukebox software. Apple isn't the first outfit to offer such a service, yet the music industry must be watching this endeavor very closely since the service represents a mass-market test balloon for the sort of Digital Rights Management (DRM) the music industry wants.

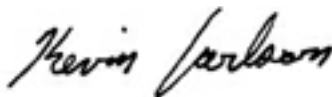
Here's how it works: Music you download through Apple's service comes in the form of a 128-kbps Advanced Audio Codec (AAC) file. (More on AAC at <http://www.vialicensing.com/products/mpeg4aac/standard.html>.) You are authorized to play this file on three Macs simultaneously. You can deauthorize the files on one machine, and authorize them on a new one when you upgrade your computer. You can download these songs to an unlimited number of iPods (Apple's popular digital media player). You can burn these files to standard audio CDs (up to 10 times for a single playlist). You can even stream them to other computers.

Given the impasse that has developed over DRM between the music industry (who wants control over copying) and the users (who have gotten used to being able to do pretty much whatever they want with MP3s), it seems that Apple has struck a compromise that, on the surface, seems reasonable for both parties. And perhaps if nothing else, Apple is to be applauded for trying. But there's an aspect of this DRM scheme that is sticking in the craw of more than a few of Apple's potential customers—the authorization process.

Each authorization on a new Mac requires communication with Apple's servers. For their DRM to work, authorization records need to be kept in a central location. In the short term, that might not be a problem. But fast-forward five years. Customers who have made a sizable investment in music are now dependent on the continued existence (and efficient functioning) of Apple's authorization service for the future viability of their music files. The music is held hostage by the whims of a single third party. If Apple goes under, or even bails out of the music business, the purchaser's music can't be moved to new machines. What about those CDs you can burn? Sure, you can reencode from those, producing files unencumbered by DRM, but those files will be lower quality than the originals you purchased.

That's not to say we don't all face risks when we purchase content on any media. Vinyl and cassette buyers had to repurchase their music on CD when the format changed. We're not guaranteed future compatibility. But there are key differences between these scenarios: You can still buy turntables and cassette decks. And those format shifts were industry-wide, and weren't controlled by any one party.

The DRM scheme Apple is using changes the agreement between music seller and music buyer in a fundamental way. Instead of purchasing a product, music buyers are paying for something that is somewhere between a product and a service—part of the payment is for the music itself, and part is for the continued authorization service. Whether Apple succeeds in its music endeavor, and whether the music industry gets the kind of DRM it wants, depends on whether buyers are willing to accept this change.



Kevin Carlson
Executive Editor
kcarlson@tpj.com

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

THE PERL JOURNAL is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: \$12.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2003 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

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

WEB DEVELOPER

Bryan McCormick

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 COO

Steve Weitzner

EXECUTIVE VICE PRESIDENT AND CFO

John Day

CHIEF INFORMATION OFFICER

Mike Mikos

PRESIDENT, TECHNOLOGY SOLUTIONS GROUP

Robert Faletra

PRESIDENT, HEALTHCARE GROUP

Vicki Masseria

PRESIDENT, ELECTRONICS GROUP

Jeff Patterson

SENIOR VICE PRESIDENT, GLOBAL SALES AND MARKETING

Bill Howard

SENIOR VICE PRESIDENT, HUMAN RESOURCES AND COMMUNICATIONS

Leah Landro

PRESIDENT AND GENERAL COUNSEL

Sandra Grayson

Perl News

The Perl Foundation Wants Your Two Cents

Following discussion about whether Perl 6 and Parrot have been eclipsing other Perl projects, The Perl Foundation has set up an online survey to determine where advocacy efforts and grant money would be best applied. The survey asks respondents to rate the relative importance of funding Perl 6 and Perl 5 projects, and also asks whether The Perl Foundation should continue to award two large, full-year grants or instead bestow several shorter grants.

For 2003, The Perl Foundation's course is already set: The organization will award several "small targeted grants to individuals." The Foundation has set up a grant proposal form on its web site for individuals wishing to apply for funding. ("Please don't submit a project for other people," the administrators note, "It's not a given that everyone can take time off their career to work for the Foundation.") Projects will be judged on their potential benefit to the Perl community and the clarity of their objectives.

Both the survey and the grant proposal form can be found on The Perl Foundation's web site (<http://www.perlfoundation.org/>).

Perl Plug-ins for Eclipse Under Development

EPIC is the Eclipse Perl Integration Collection, consisting of a set of plug-ins for the Eclipse tool platform. Three plug-ins are currently under development: the Perl Editor Plug-in, the Perl Debugger Plug-in, and the RegExp Plug-in. The EPIC web site is at <http://e-p-i-c.sourceforge.net/>.

The Eclipse project (<http://www.eclipse.org/>) was initiated by IBM; the idea is to create an infinitely extensible IDE in which any specific editor, debugger, or compiler can be swapped in as the developer chooses. The project started with Java tools and has added C/C++ support; the Perl plug-ins are still in an alpha stage of development.

ActiveState Bundles Visual Studio Tools

ActiveState has upgraded Perl ASPX, the Perl Dev Kit, and Visual Perl (along with Visual Python and Visual XSLT) to be compatible with the newly launched Visual Studio .NET 2003. The most significant upgrade is to Visual Python, which now features an interactive testing window and adds enhanced statement completions, language reference help, and support for remote debugging. Visual Perl, Visual Python, and Visual XSLT are also now available as a bundle; the "ActiveState Open Source Language Suite for Visual Studio .NET" sells for \$495. Go to http://www.activestate.com/Products/dot_net/index.plex for documentation.

Perl Beginners' Site Launched

Every wizened Perl guru was once a fresh-faced beginner. Shlomi Fish's new web site (<http://perl-begin.berlios.de/>) is a collection of resources for those just getting started with Perl. The site links to online tutorials, useful books, article collections, mailing lists, web forums, and IRC channels. Fish also maintains a "perl-begin-help" mailing list.

Parrot Win32 Binary Released

Clinton A. Pierce has compiled a binary distribution of Parrot for Win32 from the 0.10 sources. You can download it at http://www.geeksalad.org/parrot/Parrot_Dist_20030410.zip. "It's still *big*," he writes. "I dropped all of the .obj and .lib files after compilation and it's still a hefty 9 MB. I removed nothing else because in this early stage of development I figure: The people downloading this...know they're looking at a work-in-progress and the source (even the intermediate files) are probably instructive. CVS could be used to update non-object files (like documentation) if someone were so inclined." Users who run into missing DLLs should try installing Microsoft's .NET run time.

Cramming It All In At YAPC

The schedule has been set for the North American Yet Another Perl Conference, to be held June 16–18 in Boca Raton. You can see the list of talks at <http://yapc.org/America/talks.shtml>. A preliminary list of presentations has also been drawn up for the YAPC::Israel::2003 conference (coming up on May 11th in Haifa, Israel)—those 21 talks are detailed at <http://www.perl.org.il/YAPC/2003/presentations.html>.

The Whys and Hows of Parrot

Dan Sugalski posted an entry to his blog this month outlining the history and purpose of Parrot, starting with the bit where Jon Orwant flung coffee mugs at a wall during a Perl 5 porters meeting while insisting "Perl is dead" (or "Perl is f*cked," depending on which version of the story you prefer). The story proceeds from those shards of broken ceramics, continues through an April Fool's joke or two, and winds up explaining how Parrot's mandate has expanded, and why Perl may not always hold Most Favored Language status in the Parrot development process. The full history is at <http://www.sidhe.org/~dan/blog/archives/000168.html>.

We want your news! Send tips to editors@tpj.com.

Web Localization & Perl

Designing software for a worldwide audience involves two processes—internationalization (often abbreviated “i18n,” because of the 18 letters between “i” and “n”) and localization (abbreviated “l10n” for the same reason). Internationalization is an engineering process: It means building your application so that it can support multiple languages, date/currency formats, and local customs without deep structural changes in the code. Localization is the process of implementing your internationalized application for various locales. It is during the localization process, for example, that text translation takes place. Today, web-application developers have perhaps the greatest need to localize their software. Often, web-application user interfaces are text-based, increasing the need for translation and other localization efforts.

For proprietary applications, localization typically has been done as a prerequisite for competing in a foreign market. That implies that if the localization cost exceeded estimated profit in a given locale, the company would not localize its application at all; and it would be difficult (and maybe illegal) for users to do it themselves without the source code. If a vendor did not design its software with a good i18n framework in mind, well, international users were just out of luck.

Fortunately, the case is much simpler and more rewarding with open-source applications. As with proprietary applications, the first few versions are often designed with only one locale in mind, but open-source apps can be internationalized at any time by anyone.

In this article, I’ll describe techniques to make l10n straightforward. While I focus on web-based applications written in Perl, the principle should also apply to other languages and application types.

Localizing Static Web Sites

Web pages come in two different flavors: static pages that don’t change until they are manually updated, and dynamic pages that can change for each viewer based on various factors. These two types of pages are often referred to as “web documents” and “web applications,” respectively.

However, even static pages may have multiple representations—different people may prefer different languages, styles, or media (for example, an auditory representation instead of a visual one). Part of the Web’s strength is its ability to let clients negotiate with the server and determine the most preferred representation.

For example, consider my hypothetical homepage <http://www.autrijus.org/index.html>, written in Chinese (Figure 1). Assume that one day, I decide to translate it for my English-speaking friends (Figure 2).

Autrijus is a software developer who specializes in localization and internationalization. He can be contacted at autrijus@autrijus.org.

At this point, many web sites would decide to offer a language-selection page to let visitors pick their favorite language, as in Figure 3. For both nontechnical users and automated programs, this page is confusing, redundant, and irritating. Besides demanding an extra search-and-click for each visit, it creates a considerable amount of difficulty for web-agent programmers, as they now have to parse the page and follow the correct link, which is a highly error-prone thing to do.

MultiViews: The Easiest L10n Framework

Of course, it is better if everybody can see their preferred language automatically. Thankfully, the content negotiation feature in HTTP/1.1 addresses this problem quite neatly. Content negotiation is defined as “the process of selecting the best representation for a given response when there are multiple representations available” (<http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html>).

Under this scheme, browsers always send an Accept-Language request-header field, which specifies one or more preferred language codes. For example, “zh-tw, en-us, en” would mean “Traditional Chinese, American English, or English, in this order.”

Upon receiving this information, the web server is responsible for presenting the request content in the most preferred language. Different web servers may implement this process differently;

```
<html><head><title>唐宗漢 - 家</title></head>
<body>施工中, 請見諒</body></html>
```

Figure 1: A simple Chinese page.

```
<html><head><title>Autrijus.Home</title></head>
<body>Sorry, this page is under construction.</body></html>
```

Figure 2: Page translated to English.

Please choose your language:			
Čeština	Deutsch	English	Español
Français	Hrvatski	Italiano	日本語
한국어	Nederlands	Polski	Русский язык
Slovensky	Slovensci	Svenska	中文 (GB) 中文 (Big5)

Figure 3: A typical language-selection page.

under Apache (the most popular web server), a technique called “MultiViews” is widely used.

Using MultiViews, I save the English version as `index.html.en` (note the extra file extension), then put the following line into Apache’s configuration file (`httpd.conf` or `.htaccess`):

```
Options +MultiViews
```

After that, Apache examines all requests to `http://www.autrijus.org/index.html`, to see if the client prefers “en” in its Accept-Language request-header field. People who prefer English see the English page; others see the original `index.html` page.

This technique allows gradual introduction of new localized versions of the same documents, so my international friends can contribute more languages over time—`index.html.fr` for French, `index.html.he` for Hebrew, and so on.

Since much of the international online population speaks only English and one other (native) language, most of the contributed versions would be translated from English, not Chinese. But because both versions represent the same content, that is not a problem.

Or is it? What if I go back to update the original, Chinese page?

The Difficulty of Maintaining Translations

It’s impossible to get my French and Hebrew friends to translate from Chinese. Clearly, I must use English as the base version. The same reasoning also applies to most free software projects, even if the principal developers do not speak English natively.

Moreover, even if it is merely a change to the background color (for example, `<body bgcolor=gold>`), I still need to modify all translated pages, to keep the layout consistent.

Now, if both the layout and contents are changed, things quickly become very complicated. Since the old HTML tags are gone, my translator friends must work from scratch every time. Unless all of them are HTML wizards, errors and conflicts will surely arise. If there are 20 regularly updated pages in my personal site, then pretty soon, I will run out of translators—or even out of friends. Clearly, what’s needed is a way to automate the process of generating localized pages.

Separate Data and Code with CGI.pm

To prepare web applications for localization, you must find a way to separate data from code as much as possible.

As the long-established web-development language of choice, Perl offers a wide variety of modules and toolkits for web site construction. The most popular one is probably *CGI.pm*, which has been merged into the core Perl release since 1997. Example 1 is a code snippet that uses it to automatically generate translated pages.

Unlike the HTML pages, this program enforces data/code separation via *CGI.pm*’s HTML-related routines. Tags (`<html>`, for instance) now become functions calls (`start_html()`), and text is turned into Perl strings. Therefore, when the localized version is written out to the corresponding static page (`index.html.zh_tw`, `index.html.en`, and so on), the HTML layout is always identical for each of the four languages listed.

```
use CGI ':standard'; # our templating system
foreach my $language (qw(zh_tw en de fr)) {
    open OUT, ">index.html.$language" or die $!;
    print OUT start_html({ title => _("Autrijus.Home") },
        _("Sorry, this page is under construction."),
        end_html);
    sub _ { some_function($language, @_); } # XXX: put L10n framework
                                           # here
}
```

Example 1: Using *CGI.pm* to automatically generate translated pages.

English	Haitian
This costs ____ dollars.	Bagay la kute ____ dola yo.

Figure 4: An English => Haitian lexicon.

set_id	1	2	3	4	5	6	7	8	9
msg_id	1	Autrijus'.Haus
2	Wir bitten um Entschuldigung...
3

Figure 5: The content of `nls/de.cat`.

The `sub _` function is responsible for localizing any text into the current `$language`, by passing the language and text strings to a hypothetical `some_function()`. `some_function()` is known as the “localization framework.”

After writing the code in Example 1, it is a simple matter to grep for all strings inside `_()` within the code, extract them into a lexicon, and ask translators to complete this lexicon in other languages. Here, lexicon means a set of things that you know how to say in another language—sometimes single words like “Cancel,” but usually whole phrases such as “Do you want to overwrite?” or “5 files found.” Strings in a lexicon are like entries in a traveler’s pocket phrasebook, sometimes with blanks to fill in, as in Figure 4.

Ideally, the translator should focus solely on this lexicon, instead of peeking at HTML files or the source code. But here’s the rub: Different localization frameworks use different lexicon formats, so you have to choose the framework that best suits the project.

Localization Frameworks

To implement the `some_function()` in Example 1, you need a library to manipulate lexicon files, look up the corresponding strings in it, and maybe incrementally extract new strings for insertion into the lexicon. These abilities are collectively provided by a localization framework.

From my observation, frameworks mostly differ in their idea about how lexicons should be structured. Here, I discuss the Perl interfaces for three such frameworks, starting with *Msgcat*.

Msgcat: Lexicons are Arrays

As one of the earliest I10n frameworks and as part of XPG3/XPG4 standards, *Msgcat* enjoys ubiquity on all UNIX platforms. It represents the first-generation paradigm of lexicons: Treat entries as numbered strings in an array (aka., a message catalog). This approach is straightforward to implement, needs little memory, and is fast to look up. The resource files used in Windows programming and other platforms use basically the same idea.

For each page or source file, *Msgcat* requires us to make a lexicon file for each language, as in Example 2.

Example 2 contains the German translation for the text strings within `index.html`, which is represented by a unique “set number” of 7. Once you finish building the lexicons for all pages, the `gencat` utility is used to generate the binary lexicon:

```
% gencat nls/de.cat nls/de/*.m
```

It is best to imagine the internals of the binary lexicon as a two-dimensional array, as in Figure 5.

```
$set 7 # $Id: nls/de/index.pl.m
1 Autrijus'.Haus
2 Wir bitten um Entschuldigung. Diese Seite ist im Aufbau.
```

Example 2: A *Msgcat* lexicon.

```
msgid ""
"This is a multiline string"
"with %1$s and %2$s as arguments"
msgstr ""
"これは多線ひも変数として"
"%2$s と %1$s のである"
```

Figure 6: A multiline entry with numbered arguments.

To read from the lexicon file, you use the Perl module *Locale::Msgcat* (available from CPAN) and implement the *sub _()* function (Example 3). Only the *msg_id* matters here; the string *"Autrijus.House"* is only used as an optional fallback when the lookup fails, as well as to improve the program's readability.

Because *set_id* and *msg_id* must both be unique and immutable, future revisions may only delete entries, and never reassign the number to represent other strings. This characteristic makes revisions very costly. For this reason, one should consider using *Msgcat* only if the lexicon is very stable.

Another shortcoming of *Msgcat* is the plurality problem. Consider the code snippet *printf(_(8, "%d files were deleted."), \$files);*. This is obviously incorrect when *\$files == 1*, and *"%d file(s) were deleted"* is grammatically invalid as well. Hence, you are often forced to use two entries:

```
printf(($files == 1) ? _(8, "%d file was deleted.")
      : _(9, "%d files were deleted."), $files);
```

This is still not satisfactory, however, because it is English specific. French, for example, uses singular with *\$files == 0*, and Slavic languages have three or four plural forms. Trying to retrofit those languages to the *Msgcat* infrastructure is often a futile exercise.

Gettext: Lexicons are Hashes

Due to the various problems of *Msgcat*, the GNU Project developed its own implementation of the UniForum Gettext interface in 1995. This implementation, written by Ulrich Drepper, has since become the de facto i18n framework for C-based free software projects, and has been widely adopted by C++, Tcl, and Python programmers.

Instead of requiring one lexicon for each source file, Gettext maintains a single lexicon (called a PO file) for each language of the entire project. For example, the German lexicon *de.po* for the homepage would look like Example 4.

The *#:* lines are automatically generated from the source file by the program *xgettext*, which can extract strings inside invocations of *gettext()*, and sort them out into a lexicon. Now, we may run *msgfmt* to compile the binary lexicon *locale/de/LC_MESSAGES/web.mo* from *po/de.po*:

```
% msgfmt locale/de/LC_MESSAGES/web.mo po/de.po
```

You can then access the binary lexicon using *Locale::gettext* from CPAN, as in Example 5. Recent versions (glibc 2.2+) of Gettext also introduced the *ngettext* ("*%d file*", "*%d files*", *\$files*) syntax. Unfortunately, *Locale::gettext* does not support that interface yet.

```
use POSIX;
use Locale::gettext;
POSIX::setlocale(LC_MESSAGES, $language); # Set target language
textdomain("web"); # Usually the same as the application's name
sub _ { gettext(@_) } # it's just a shorthand for gettext()
print _("Sorry, this site is under construction.");
```

Example 5: Sample usage of *Locale::gettext*.

```
use Locale::Msgcat;
my $cat = Locale::Msgcat->new;
$cat->catopen("nls/$language.cat", 1); # it's like a 2D array
sub _ { $cat->catgets(7, @_) } # 7 is the set_id for index.html
print _(1, "Autrijus.House"); # 1 is the msg_id for this text
```

Example 3: Sample usage of *Locale::Msgcat*.

```
#: index.pl:4
msgid "Autrijus.Home"
msgstr "Autrijus'.Haus"

#: index.pl:5
msgid "Sorry, this site is under construction."
msgstr "Wir bitten um Entschuldigung. Diese Seite ist im Aufbau."
```

Example 4: A *Gettext* lexicon.

Also, Gettext lexicons support multiline strings, as well as re-ordering via *printf* and *sprintf* (Figure 6). Finally, GNU Gettext comes with a very complete tool chain (*msgattrib*, *msgcmp*, *msgconv*, *msgexec*, *msgfmt*, *msgcat*, *msgcomm*...), which greatly simplifies the process of merging, updating, and managing lexicon files.

Maketext: Lexicons are Dispatch Tables

First written in 1998 by Sean Burke, the *Locale::Maketext* module was revamped in May 2001 and is included in the Perl 5.8 core. Unlike the function-based interface of *Msgcat* and *Gettext*, its basic design is object oriented, with *Locale::Maketext* as an abstract base class from which a project class is derived. The project class (with a name like *MyApp::L10N*) is, in turn, the base class for all the language classes in the project (which may have names like *MyApp::L10N::it*, *MyApp::L10N::fr*, and the like).

A language class is really a Perl module containing a *%Lexicon* hash as class data, which contains strings in the native language (usually English) as keys, and localized strings as values. The language class may also contain some methods that are useful for interpreting phrases in the lexicon, or otherwise dealing with text in that language. Example 6 illustrates *Locale::Maketext*'s use.

Under its square bracket notation, translators can make use of various language-specific functions inside their translated strings. Example 6 demonstrates the built-in plural and quantifier support—for languages with other kinds of plural-form characteristics, it is a simple matter of implementing a corresponding *quant()* function. Ordinal and time formats are easy to add, too.

Each language class may also implement an *->encoding()* method to describe the encoding of its lexicons, which may be linked with *Encode* for transcoding purposes. Language families are also inheritable and subclassable: missing entries in *fr_ca.pm* (Canadian French) would fall back to *fr.pm* (Generic French).

The handy built-in method *->get_handle()*, used with no arguments, magically detects HTTP, POSIX, and Win32 locale

```
package MyApp::L10N;
use base 'Locale::Maketext';

package MyApp::L10N::de;
use base 'MyApp::L10N';
our %Lexicon = (
    "[quant,_1,camel was,camels were] released." =>
    "[quant,_1,Kamel wurde,Kamele wurden] freigegeben.",
);

package main;
my $lh = MyApp::L10N->get_handle('de');
print $lh->maketext("[quant,_1,camel was,camels were] released.", 5);
```

Example 6: A *Locale::Maketext* lexicon and its usage.

settings in CGI, mod_perl, or from the command line; it spares you from parsing those settings manually.

However, *Locale::Maketext* is not without problems. The most serious issue is its lack of a toolchain such as GNU Gettext's. *Locale::Maketext* classes are full-fledged Perl modules and, as such, can have arbitrarily obscure syntactic structure. This makes writing a toolchain targeting *Locale::Maketext* classes all but impossible. For the same reason, there are also few text editors that can support it as well as Emacs PO Mode for Gettext.

Finally, since different projects may use different styles to write the language class, the translator must know some basic Perl syntax.

Locale::Maketext::Lexicon: The Best of Both Worlds

Irritated by the irregularity of *Locale::Maketext* lexicons, I implemented my own lexicon format for my company's internal use in May 2002, and asked the perl-i18n mailing list for ideas and feedback. Jesse Vincent suggested: "Why not simply standardize on Gettext's PO File format?" So I implemented it to accept lexicons in various formats, handled by different lexicon back-end modules. Thus, *Locale::Maketext::Lexicon* was born.

The design goal was to combine the flexibility of *Locale::Maketext*'s lexicon expression with standard formats supported by utilities designed for Gettext or Msgcat. It also supports the Tie interface, which comes in handy for accessing lexicons stored in relational databases or DBM files.

Figure 7 demonstrates a typical application using *Locale::Maketext::Lexicon* and the extended PO File syntax supported by the Gettext back end. Line 2 tells the current package *main* to inherit from *Locale::Maketext*, so it can acquire the *get_handle* method. Lines 5–8 build four language classes using a variety of lexicon formats and sources:

- The Auto back end tells *Locale::Maketext* that no localizing is needed for the English language—just use the lookup key as the returned string. It is especially useful if you are just starting to prototype a program and do not want deal with the localization files yet.
- The Tie back end links the French %Lexicon hash to a Berkeley DB file; entries will then be fetched whenever it is used, so it will not waste any memory on unused lexicon entries.
- The Gettext back end reads a compiled MO file from disk for Chinese, and reads the German lexicon from the DATA file-handle in PO file format.

Lines 11–13 implement the *ord* method for each language subclass of the package *main*, which converts its argument to ordinate numbers (1st, 2nd, 3rd...) in that language. Two CPAN modules are used to handle English and French, while German and Chinese need only straightforward string interpolation.

```
1 use CGI 'standard';
2 use base 'Locale::Maketext'; # inherits get_handle()
3
4 # Various lexicon formats and sources
5 use Locale::Maketext::Lexicon {
6   en => ['Auto'],          fr => ['Tie' => 'DB_File', 'fr.db'],
7   de => ['Gettext' => \%DATA], zh_tw => ['Gettext' => 'zh_tw.mo'],
8 };
9
10 # Ordinate functions for each subclasses of 'main'
11 use Lingua::EN::Numbers::Ordinate; use Lingua::FR::Numbers::Ordinate;
12 sub en::ord { ordinate($_[1]) } sub fr::ord { ordinate_fr($_[1]) }
13 sub de::ord { "$_[1]." } sub zh_tw::ord { "第 $_[1] 個" }
14
15 my $lh = __PACKAGE__->get_handle; # magically gets the current locale
16 sub _ { $lh->maketext($_) } # may also convert encodings if needed
17
18 print header, start_html, # [*,...] is a shorthand for [quant,...]
19 _("You are my [ord,1] guest in [*,_2,day].", $hits, $days), end_html;
20
21 _DATA
22 # The German Lexicon, in extended PO File format
23 magid "You are my [ord($1)] guest in [*($2,day)]."
24 magstr "Innerhalb [*($2,Tages,Tagen), sie sind mein [ord($1)] Gast."
```

Figure 7: A sample application using *Locale::Maketext::Lexicon*.

Line 15 gets a language handle object for the current package. Because it did not specify the language argument, it automatically guesses the current locale by probing the HTTP_ACCEPT_LANGUAGE environment variable, POSIX *setlocale()* settings, or *Win32::Locale* on Windows. Line 16 sets up a simple wrapper function that passes all arguments to the handle's *maketext* method.

Finally, lines 18–19 print a message containing one string to be localized. The first argument, *\$hits*, is passed to the *ord* method, and the second argument, *\$days*, calls the built-in *quant* method—the *[*...]* notation is shorthand for the previously discussed *[quant,...]*.

Lines 22–24 are a sample lexicon, in extended PO file format. In addition to ordered arguments via %1 and %2, it also supports *%function(args...)* in entries, which will be transformed to *[function,args...]*. Any %1, %2... sequences inside the *args* has their percent signs (%) replaced by underscores (_).

Summary

The localization process consists of these steps:

1. Assess the web site's templating system.
2. Choose a localization framework and hook it up.
3. Write a program to locate text strings in templates, and put filters around them.
4. Extract a test lexicon; fix obvious problems manually.
5. Locate text strings in the source code by hand; replace them with *_(...)* calls.
6. Extract another test lexicon and machine-translate it.
7. Try the localized version out; fix any remaining problems.
8. Extract the beta lexicon; mail it to your translator teams for review. Fix problems reported by translators; extract the official lexicon and mail it out.

www.amzi.com

Move your Rules to Logic Base

90% Less Code
99% Less Maintenance

Create logic bases for pricing, shipping, taxes, insurance, regulations, workflow, claims, and more with the Amzi® Logic Server™ Query your logic base like a database from Java, C++, VB, Delphi, ASP, .NET Web Servers and other languages/ tools under Windows and Unix.

FREE Download!
info@amzi.com www.amzi.com

Put Your Expertise on the Web and in Your Product!

KnowledgeWright® is the next generation of tools for building expert systems. Use a graphical interface to develop and debug your knowledge-bases. Run them on the Web or under Java, C++, Visual Basic, Delphi, etc. Let us customize the reasoning engine (in a flash) to meet your specific needs.

FREE Download!
info@amzi.com www.amzi.com

www.amzi.com

- Periodically notify translators of new lexicon entries before each release.

Following these steps, you could manage a i18n project fairly easily, keep the translations up-to-date, and minimize errors.

Localization Tips

Here are some tips for localizing web applications, and other software in general:

- Separate data and code, both in design and in practice.
- Don't work on i18n/i10n before the web site or application takes shape.
- Avoid graphic files with text in them.
- Leave enough spaces around labels and buttons—do not overcrowd the UI.
- Use complete sentences, instead of concatenated fragments:

```
_("Found ") . $files . _(" file(s)."); # Fragmented sentence - wrong!  
sprintf(_("Found %s file(s)."), $files); # Complete (with sprintf)  
_("Found [*,_l,file].", $files); # Complete (Locale::Maketext)
```

- Distinguish the same string in different contexts (for example, "Home" in the context of "Homepage" and "Home Phone Number").
- Work with your translators as equals; do not apply lexicon patches by yourself without their consent.
- One person doing draft translations works best.
- In lexicons, provide as many comments and as much metadata as possible:

```
#: lib/RT/Transaction_Overlay.pm:579  
#. ($field, $self->OldValue, $self->NewValue)  
# Note that 'changed to' here means 'has been modified to...'.  
msgid "%1 %2 changed to %3"  
msgstr "%1 %2 cambiado a %3"
```

Using the `xgettext.pl` utility provided in the `Locale::Maketext::Lexicon` package, the source file, line number (marked by `#:`), and variables (marked by `#.`) can be deduced automatically and incrementally. It is also helpful to clarify the meaning of short or ambiguous phrases with normal comments (marked by `#`).

Conclusion

In nonEnglish speaking countries, localization efforts are often a prerequisite for participating in free software projects. These localization projects are principal places for community contributions, but such efforts are also historically time consuming and error prone, partly because of English-specific frameworks and rigid coding practices used by existing applications. The entry barrier for translators has been unnecessarily high.

On the other hand, the increasing internationalization of the Web makes it increasingly likely that the interfaces to web-based dynamic content services will be localized to two or more languages. For example, Sean Burke led enthusiastic users to localize the popular `Apache::MP3` module, which powers homegrown Internet jukeboxes everywhere, to dozens of languages in 2002. Lincoln Stein, the module's author, was not involved with the project at all—all he needed to do was integrate the i18n patches and lexicons into the next release.

Free software projects are not abstractions filled with code, but rather depend on people caring enough to share code and give useful feedback to improve each other's code. Hence, it is my hope that techniques presented in this article will encourage programmers and eager users to actively internationalize existing applications, instead of passively translating for the relatively few applications with established i18n frameworks.

TPJ

Sign Up For BYTE.com!

DON'T GET LEFT BEHIND!
Register for BYTE.com today
and have access to...

- Jerry Pournelle's "Chaos Manor"
- Moshe Bar's "Serving with Linux"
- Martin Heller's "Mr. Computer Language Person"
- David Em's "Media Lab"
- and much more!...

BYTE.com will keep you up-to-date on emerging trends and technologies with even more rich technical articles and opinions than ever before! Expert opinions, in-depth analysis, trusted information...find all this and more on BYTE.com!



**ONLY \$19.95
FOR ANNUAL
ACCESS!**



As a special thank you for signing up, receive the BYTE CD-ROM and a year of access for the low price of \$26.95!

Registering is easy...go to www.byte.com and sign up today! Don't delay!



BYTE

Data Manipulation & Perl Command-Line Options

In his article “Something For Nothing” (*TPJ*, March 2003), Simon Cozens talked about using the tools on CPAN to avoid reinventing the wheel. Even without CPAN, Perl itself provides a number of command-line options that do the heavy lifting for many data- and file-manipulation tasks. In this article, I’ll provide an overview of Perl’s most useful and commonly used data-manipulation options.

-e Command

The most useful way to use the command-line options is by writing Perl one-liners right in the shell. The `-e` option is the basis for most command-line programs. It accepts the value of the parameter as the source text for a program:

```
$ perl -e'print "Hello, World!\n"'
Hello, World!
```

Since this is a single statement in a block, you can omit the semicolon. Also, when the `-e` option is used, Perl no longer looks for a program name on the command line. This means you can’t mix code with `-e` and a program file.

The `-e` option is repeatable, which lets you create entire scripts on the command line:

```
$ perl -e'print "Hello, ";' -e'print "World!\n"'
Hello, World!
```

When chaining together multiple `-e` options, make sure you keep your semicolons in the right place. I reflexively put semicolons in my `-e` lines just for safety’s sake, even if it’s not strictly necessary because there’s only one `-e` option.

With the `-e` option, any shell window becomes a Perl IDE. Use it as your calculator to figure out how many 80-line records are in a megabyte:

```
$ perl -e'print 1024*1024/80, "\n"'
13107.2
```

Escaping Shell Characters

When you’re creating command-line programs, it’s important to pay attention to quoting issues. In all my examples, I’ve quoted with single quotes—not double quotes—for two reasons. First, I want to be able to use double quotes inside my programs for lit-

erals, and double quotes don’t nest in the shell. Second, I have to prevent shell interpolation, and single quotes make it easy. For example, if I use double quotes, then

```
$ perl -MCGI -e'print $CGI::VERSION'
```

gets the `$CGI` interpolated as a shell variable. Consequently, unless you have a shell variable called `$CGI`, Perl sees

```
print ::VERSION
```

You can escape the shell variables with a backslash:

```
$ perl -MCGI -e'print \$CGI::VERSION'
```

but that gets to be tough to maintain. That’s why I stick with single quotes:

```
$ perl -MCGI -e'print $CGI::VERSION'
```

Windows has slightly different quoting issues. Windows doesn’t have shell variable interpolation, so there’s no need for escaping variables with dollar signs in them. On the other hand, you can use only double quotes under Windows, which can be a challenge if you want to use double quotes in your program. Under Windows, your “Hello, World” would look like this:

```
C:\> perl -e"print \"Hello, World!\n\""
```

he inner double quotes are escaped with backslashes.

The Diamond Operator

Perl’s diamond operator, `<>`, has a great deal of magic built into it, making operations on multiple files easy.

Have you ever written something like this:

```
for my $file ( @ARGV ) {
    open( my $fh, $file ) or die "Can't open $file: $!\n";
    while ( my $line = <$fh> ) {
        # do something with $line
    }
    close $fh;
}
```

```
$ perl myprog.pl file1.txt file2.txt file3.txt
```

Andy manages programmers for Follett Library Resources. He can be contacted at andy@petdance.com.

so that your program can operate on three files at once? Use the diamond operator instead. Perl keeps track of which file you're on, and opens and closes the filehandle as appropriate. With the diamond operator, it's as simple as:

```
while ( my $line = <> ) {
    # do something
}
```

Perl keeps the name of the currently open file in `$ARGV`. The `$.` line counter does not reset at the beginning of each file.

The diamond operator figures prominently in much Perl command-line magic, so it behooves you to get comfortable with it.

-n and -p: Automatic Looping Powerhouses

The `-n` and `-p` options are the real workhorse options. They derive from the Awk metaphor of “Do something to every line in the file,” and work closely with the diamond operator.

The following program prepends each line with its line number:

```
while (<>) {
    $_ = sprintf( "%05d: %s", $., $_ );
    print; # implicitly print $_
}
```

The construct of “Walk through a file, and *print* `$_` after you do some magic to it” is so common that Perl gives us the `-p` option to implement it for us. The previous example can be written as:

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

$_ = sprintf( "%05d: %s", $., $_ );
```

or even shorter as:

```
$ perl -p -e'$_ = sprintf( "%05d: %s", $., $_ )'
```

The `-n` option is just like `-p`, except that there's no print at the bottom of the implicit loop. This is useful for grep-like programs when you're only interested in selected information. You might use it to print only commented-out lines from your input, defined as beginning with optional whitespace and a pound sign:

```
$ perl -n -e'print if /\s*#/'
```

The next program prints every numeric value that looks like it's part of a dollar value, as in “\$43.50.”

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

while ( /\$(\d+\.\d\d)/g ) {
    print $1, "\n";
}
```

This *while* loop is inside the implicit *while(<>)* loop. If you want to do something before or after your main *while()* loop, use BEGIN and END blocks. For example, to total up all those dollar values:

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

BEGIN { $total=0 }
END { printf( "%.2f\n", $total ) }

while ( /\$(\d+\.\d\d)/g ) {
    $total += $1;
}
```

The order of the BEGIN and END blocks doesn't matter, so don't worry about having them in the right order. You can specify them with the `-e` option, too. Here's a quick one that I used while writing this article to strip verbatim paragraphs from POD:

```
$ perl -n -e'BEGIN {$/=""}; print unless /\s+/;' article.pod
```

If you want an empty *while* loop with `-n` or `-p`, you still must specify the `-e` option, or else Perl waits for the body of the loop to be entered on standard input. Using `-el` gives Perl a dummy loop body that does nothing.

-l: Line-Ending Handling

When you're working with lines in a file, you'll find you're doing lots of chomping and *print \$something, "\n"*. Perl has the `-l` (dash el) option to take care of this for you.

In the simplest sense, adding `-l` when you're using `-n` or `-p` automatically does a chomp on the input record, and adds a `"\n"` after everything you print. It makes command-line one-liners much easier, as in:

```
perl -l -e'print substr($_,0,40)'
```

This example only shows the first 40 characters of each line in the input, whether or not the line is longer than 40 characters, not counting the line-ending `"\n"`. For command-line programmers, `-l` is a godsend because it means you can use *print \$foo* instead of *print \$foo, "\n"* to get the results you want.

The mechanics of how the *print ... "\n"* happens are a little more complex. The `-l` actually sets `$\`, the output record separator, to `$/`, the input record separator. You can override this by specifying the octal value for `$/` on `-l`. For instance, if you wanted to have all output lines have a Ctrl-M as the record terminator, specify `-l015` (that's “dash el zero one five”).

-i: Edit in Place

All the options you've learned so far are great for writing filters, where a number of files or standard input get fed out to standard output. Unfortunately, you're left to do the foo-move dance, as in:

```
perl -p -e"slick code" input.txt > foo
mv foo input.txt
```

Perl comes to the rescue again with the `-i` option. Adding `-i` tells Perl to edit your files in place, so you can replace the previous example with

```
perl -p -i -e"slick code" input.txt
```

You can tell Perl to keep a copy of the original file(s) by specifying a string to tack on to the end of the file. Common examples are `-i~` or `-i.bak`. Perl doesn't treat “.bak” as an “extension” in the sense of replacing one extension with another:

```
perl -i.bak input.txt
```

leaves the original file called “input.txt.bak.” Of course, the `-i` option does the right thing if you specify multiple files by creating a backup file for each of the files processed.

-0(octal): Specify Input Record Separator

Often when working on the command line, you'll want to specify your input record separator. Although this is possible with *e'BEGIN {\$/=...}*, it's easier with the `-0` option. (That's dash-zero, not dash-oh.) To specify an input record separator of *chr(13)*, use `-015`. Two special values for the `-0` option are `-00` for paragraph mode, equivalent to `$/=""`, and `-0777` to slurp entire files, equivalent to `$/=undef`.

101 Perl Articles!



From the pages of *The Perl Journal*, *Dr. Dobbs's Journal*, *Web Techniques*, *Webreview.com*, and *Byte.com*, we've brought together 101 articles written by the world's leading experts on Perl programming. Including everything from programming tricks and techniques, to utilities ranging from web site searching and embedding dynamic images, this unique collection of *101 Perl Articles* has something for every Perl programmer.

Plus, this collection of articles is fully searchable, and includes a cross-platform search engine so you can immediately find answers you're looking for. Delivered as HTML files in a ZIP archive or CD-ROM image, download *101 Perl Articles* and burn your own CD-ROM or store it on hard disk.

\$9.95 For subscribers to
The Perl Journal

\$12.95 For nonsubscribers to
The Perl Journal

\$22.90 To subscribe to
The Perl Journal and
receive *101 Perl Articles*

Go to
<http://www.tpj.com/>
now!

The earlier example for filtering POD paragraphs:

```
$ perl -n -e'BEGIN {$/=""}; print unless /\s+/' article.pod
```

can now be shortened to:

```
$ perl -n -00 -e'print unless /\s+/' article.pod
```

-a and -F: Autosplit Input Records

The *-a* and *-F* options only work with *-n* and *-p*. Specifying *-a* tells Perl to run *@F=split* on your input line. Without the *-F* option, this means breaking up the input line on whitespace, which is most handy for log files. If you don't specify the *-l* option, your final element in *@F* has a "\n" at the end of it, which is probably not what you want.

Here's a quick way to count the bytes that your Apache server has sent out:

```
$ perl -l -a -n -e'$n+=$F[9];END{print $n}' access_log
```

Each line of the Apache log file is broken up on whitespace, and the number of bytes is the 10th field in the line.

If you don't want to split on whitespace, specify the regex to use with the *-F* option. This example walks through the */etc/passwd* file, printing all usernames that have a login shell. The fields in */etc/passwd* are separated by colons, with the user's name as the first field and login shell as the last.

```
perl -l -n -a -F: \
-e'print $F[0] unless $F[-1] eq "/bin/false"' /etc/passwd
```

Even though there are no slashes here, *-F:* still means that the regex is */:*.

Option Stacking

If you want to make the most of your keystrokes on the command line, you may want to stack your options. Single-character options may be combined with the following option. For example, our */etc/passwd*-processing examples that start:

```
perl -l -n -a -F: -e'....'
```

can be written as:

```
perl -lnaF: -e'....'
```

I don't recommend combining options because it adds a layer of complexity for the small benefit of saving a few keystrokes. There are also pitfalls when combining options, especially with the *-i* option. For example, say you have a program where you're editing a file in place to truncate each line to 40 characters:

```
$ perl -p -i -l -e'$_=substr($_,0,40)' myfile.txt
```

This works just fine. Now, combine those options overoptimistically into:

```
$ perl -pil -e'$_=substr($_,0,40)' myfile.txt
```

The *-p* option is just fine, but now you've told the *-i* options to append the letter "I" at the end of the backup file's name, and lost the *-l* functionality of handling line endings. The results aren't pretty.

-(mM)(-) Module

-m and *-M* are the module-loading options. They obviate the need to have a *-e'use ModuleName;*'.

The `-mModuleName` performs a `use ModuleName ();` before your program executes. `-MModuleName` is the same, but without the parentheses. The difference can be subtle, depending on the import semantics of the module you're importing, as we'll see below. You can also do a "no ModuleName" with `-M-ModuleName`.

The `-M` option is also a handy way to find out if a module is installed, and what version. Want to see which version of the CGI module you have installed?

```
$ perl -MCGI -le'print $CGI::VERSION'
2.89
```

Of course, if you don't have the module installed, Perl will give an error.

Many modules have specific tricks built in for use on the command line. Probably the most common example is the CPAN module, where you can do:

```
$ perl -MCPAN -e'install "Module::Name"'
```

`Text::Autoformat` exports the `autoformat` function by default, making it easy to write a one-liner to format a block of text from standard input:

```
$ perl -MText::Autoformat -e'autoformat'
```

Applying What You've Learned

With all these marvelous file-mangling command-line options at your disposal, you have a great deal of power at hand. For instance, a program to convert standard line endings to the Mac's Ctrl-M takes only 24 characters:

```
$ perl -i.bak -l015 -pe' *.*.txt
```

A global search and replace of all occurrences of "FOO" to "BAR" in .html files in a directory is as easy as:

```
$ perl -i -pe's/FOO/BAR/g' *.html
```

Wrapping Up

A few final notes about command-line options: Perl respects command-line options on the `#!/perl` line of your script, so a script that you write as:

```
$ perl -i -pe's/FOO/BAR/g'
```

could also be written as:

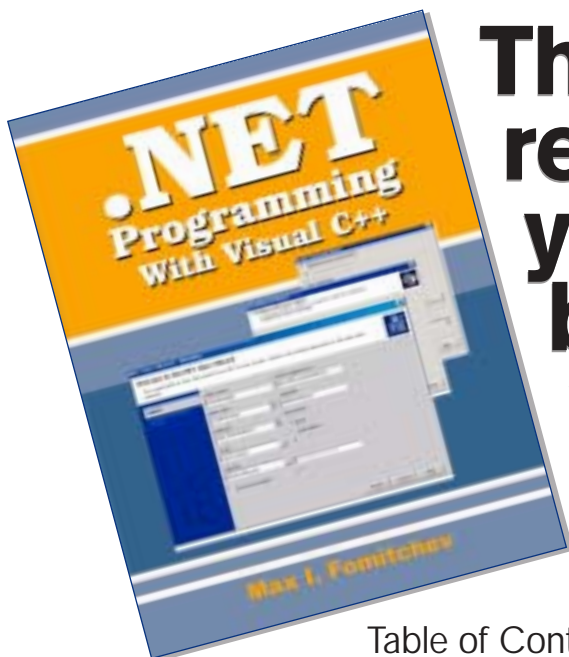
```
#!/usr/bin/perl -i -p
s/FOO/BAR/g;
```

Even in operating systems that don't use the `#!/perl` line (like Windows), Perl still checks for it and will respect the options.

For future reference, if you don't have this article handy, you can run `perl -h` to get an alphabetical list of options, or run `perl-doc perlrn` for the manpage.

Now go forth with your newfound power and keep getting something for nothing.

TPJ



The .NET resource you've been waiting for!



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

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

Get your copy now!

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

Google and Perl

Google has become the most popular search engine for techies, even to the extent that a lot of people have been treating Google as just another part of their application. In response, Google set up a SOAP service to allow people to access their services and get a response in XML. At the moment, the program is an experimental beta service and Google limits the number of SOAP queries to 1000 per developer, which is about one query every minute and a half.

To use the Google Web API, I need to register as a developer and get a developer's token at <http://www.google.com/apis/>. This leads me through the sign up process and provides links to documentation and FAQs. The developer's kit comes with a Web Services Description Language (WSDL) file that I can use with `SOAP::Lite` to set up the service and insulate me from the specifics of the SOAP stuff going on. I just want to get my search results.

The process is simple: I load the `SOAP::Lite` module and use its `service` method to create my search object, `$search`. The argument to `service` is the location of the WSDL file, which I have in the same directory as the script. I store my Google Developer's Key in my shell's environment and access it as `$ENV{GOOGLE_KEY}`. I print the results with `Data::Dumper` because it is quick and easy. Later, I will use the Google API to access the information.

Making a Query

The `doGoogleSearch()` method comes from the WSDL file and takes several arguments. If I leave off any of the arguments, the search fails.

```
Developer's key
Query terms
Starting search result
Maximum search results (10 the highest)
...and some others the API describes and I do not use.
```

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

brian is the founder of the first Perl Users Group, NY.pm, and Perl Mongers, the Perl advocacy organization. He has been teaching Perl through Stonehenge Consulting for the past five years, and can be contacted at comdog@panix.com.

```
use strict;

use Data::Dumper;
use SOAP::Lite;

my $search = SOAP::Lite->service("File:search.wsdl");

my $results = $search->doGoogleSearch(
    $ENV{GOOGLE_KEY}, 'TPJ', 0, 10,
    "false", "", "false", "", "latin1", "latin1");

print Data::Dumper::Dumper( $results );
```

When I first ran this program, the initial search result I got had the title “TPJ Sues State Lawmaker Over Open Records,” and “The Perl Journal” was the second search result. The other TPJ is Texans for Public Justice. The third argument to `doGoogleSearch()`, the results index, is zero based just like Perl, and I had used “1.” Further down in the list, I found another *TPJ* magazine, the *Tube & Pipe Journal* (<http://www.fmametalfab.org/croydon2k/tubepipe.htm>), in case you can't get enough *TPJ*.

Manipulating the Results

The return value is a Perl data structure, and since I already dumped it with `Data::Dumper`, I know what it looks like. It is an anonymous hash that has keys that describe the search, including the arguments to `doGoogleSearch()`, and how long it took to do the search. Some modules on CPAN, such as `Net::Google`, provide object access to this structure, but I find it just as easy to play with the data structure.

The search results are in the `resultsElements` key, and each result is another anonymous hash.

```
$VAR1 = bless( {
    'endIndex' => '1',
    'searchTime' => '0.11542',
    'searchComments' => '',
    'documentFiltering' => 0,
    'searchQuery' => 'TPJ',
    'estimatedTotalResultsCount' => '48400',
    'searchTips' => '',
    'startIndex' => '1',
```

```

'resultElements' => [
    bless( {
        'URL' => 'http://www.tpj.com/',
        'snippet' => ' <b>...</b> <b>TPJ</b> was published as a standalone quarterly magazine until 2001 when<br> it became a quarterly supplement to Sys Admin magazine. Beginning <b>...</b> ',
        'directoryTitle' => 'The Perl Journal',
        'hostName' => '',
        'relatedInformationPresent' => 1,
        'directoryCategory' => bless( {
            'fullViewableName' => 'Top/Computers/Programming/Languages/Perl/Magazines_and_E-zines',
            'specialEncoding' => ''
        }, 'DirectoryCategory' ),
        'summary' => 'The first and only periodical devoted to Perl.',
        'cachedSize' => '12k',
        'title' => 'The Perl Journal'
    }, 'ResultElement' )
],
'directoryCategories' => [],
'estimateIsExact' => 0
}, 'searchResult' );

```

I want to print the title and the link of the results. I modify my earlier program to extract the *resultElements* portion of the anonymous hash and go through each element of it. In my *foreach* loop, I store each element in *\$link*, and then access the *directoryTitle* and URL values with a hash slice.

```

#!/usr/bin/perl
use strict;

use SOAP::Lite;

my $search = SOAP::Lite->service("File:search.wsdl");

my $results = $search->doGoogleSearch(
    $ENV{GOOGLE_KEY}, 'TPJ', 0, 10,
    "false", "", "false", "", "latin1", "latin1");

my $links = $results->{resultElements};

foreach my $link ( @$links )
{
    print join "\n", @{ $link }{ qw(directoryTitle URL) }, "\n";
}

```

If I want to reinvent Google's "I feel lucky" technology, I fetch only the first result and immediately access the URL with *LWP::Simple*.

```

#!/usr/bin/perl
use strict;

use LWP::Simple;
use SOAP::Lite;

my $search = SOAP::Lite->service("File:search.wsdl");

my $results = $search->doGoogleSearch(
    $ENV{GOOGLE_KEY}, 'TPJ', 0, 1,
    "false", "", "false", "", "latin1", "latin1");

my $link = $results->{resultElements}[0]{URL};

getprint( $link );

```

Google limits the number of search results I can get back in one query to 10, although the *\$results* anonymous hash does have an estimated count of results in the *estimatedTotalResultsCount* key. This estimate can be much greater than the total number of results since Google can remove results that are the same as previous results, even though they may have different URLs. If I want to get past the first 10 results, I use a different starting number each time I call *doGoogleSearch()*.

```

#!/usr/bin/perl
use strict;

use SOAP::Lite;

my $search = SOAP::Lite->service("File:search.wsdl");

for( my ( $start, $total ) = ( 0, 1 );
    $start < 50 and $start < $total;
    $start += 10
    )
{

    my $results = $search->doGoogleSearch(
        $ENV{GOOGLE_KEY}, 'TPJ', $start, 10,
        "false", "", "false", "", "latin1", "latin1");

    $total = $results->{estimatedTotalResultsCount};

    my $links = $results->{resultElements};

    foreach my $link ( @$links )
    {
        print join "\n", @{ $link }{ qw(directoryTitle URL) }, "\n";
    }
}

```

The Google Web API allows me to easily access the most popular Google features. I can search, then access the results and information about the search without parsing HTML or writing a web client. I did not cover everything that I can do, but the Developer's Kit provides greater detail. Now that you know how simple it is, you have no excuse not to do it yourself.

TPJ





Tracking Finances with *WWW::Mechanize* and *HTML::Parser*

Simon Cozens

One of the things I like about my bank is that it has a very comprehensive online banking service. I can fire up a web browser, log in, and transfer money, pay bills, check balances, and so on.

One of the things I really hate about it, though, is that this requires me to fire up a web browser, log in, and so on. So one of the first things I did when dealing with the bank was to write an *LWP::UserAgent*-based module that handled all the quirks of logging in and doing trivial tasks like checking balances and getting statements. I had *Finance::Bank::LloydsTSB*, and I was happy.

Unfortunately, the bank recently made matters worse by insisting on another layer of security. Now, don't get me wrong, I'm not averse to more security on my bank account per se, but I am averse to having my nice labor-saving module break.

However, I realized that this would give me the opportunity to rewrite *F::B::LloydsTSB* in terms of the *WWW::Mechanize* module, and also give me an opportunity to tell you how I did it.

WWW::Mechanize

But first, what is *WWW::Mechanize*, and why is it better than *LWP::UserAgent*? *WWW::Mechanize*, written by Andy Lester, is a fork of Kirrily Robert (Skud)'s *WWW::Automate*. Skud wrote *WWW::Automate* in order to help test web-based applications. The idea was to have a subclass of *LWP::UserAgent* that did more work behind the scenes, which would enable it to feel like it was emulating a web browser, rather than just a dumb web client.

For instance, it knows about the forms on a page, and can help you fill them in; it knows about following links, hitting a back button, reading the title of a page, and so on.

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

Let's take a simple example. We'll visit the Perl home page, <http://www.perl.com/>, follow the first link we find, and see where we end up:

```
use WWW::Mechanize;
my $agent = WWW::Mechanize->new();
$agent->get("http://www.perl.com/");
$agent->follow(0);
print $agent->title, " : ", $agent->uri, "\n";
```

(As it happens, it's a link back to www.perl.com, but hey.)

As *WWW::Mechanize* is a subclass of *LWP::UserAgent*, all the familiar methods are there—*new* creates a new user agent and *get* downloads a page.

However, there's also the new *follow* method, which takes the number of a link on the page, starting from zero; 0 is the first link. *Mechanize* also stores the URL of the page it's currently visiting, and we can retrieve that with the *uri* method. It also knows how to parse the HTML for the page and extract the HTML title, which we retrieve with *title*.

Examining the Lloyds Site

To work out how we'd make *Mechanize* automate access to our web site, we first need to look at how we'd do so in a browser. As it turns out, we'll find some unpleasant secrets along the way.

We start by going to the site's entry URL, <https://online.lloydstsb.co.uk/customer.ibt/>. This gives us the main login box, prompting us for a username and password. We fill this in, and post the form back to the site (see Figure 1).

Once we've done that, we come up against the new "memorable information" page, presumably created in order to stop people like us writing programmatic interfaces to log into the system. As well as a password, we've supplied the bank with a nine-character piece of information, and the new page presents us with a form with three drop-down menus, and asks us to select certain characters from the memorable information phrase (see Figure 2).

Once we post that back, we're finally logged in. From the post-login page, the account balances are stored in a table that we'll parse



Figure 1: Main login form.

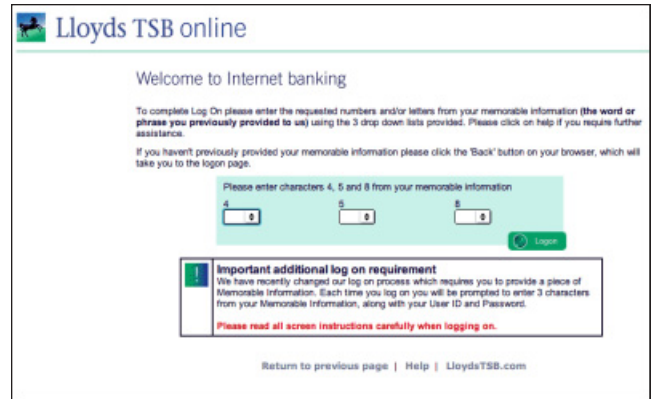


Figure 2: Memorable information form.

out using *HTML::Parser*—more on that later. First, we’ll translate this complicated login sequence into *WWW::Mechanize* code.

Mechanizing the Login

The first phase is obvious—we want to get the front page, fill in the form, and click the “log in” button. Here’s what this code looked like originally:

```
my $orig_r = $ua->get
    ("https://online.lloydstsb.co.uk/customer.abc");
croak $orig_r->error_as_HTML unless $orig_r->is_success;

my $orig = $orig_r->content;
my $key;

$orig =~ /name="Key" type="HIDDEN" value="(\\d+)/
    or croak "Couldn't parse key!";
$key = $1;
my $check = $ua->post
    ("https://online.lloydstsb.co.uk/customer.abc", {
        Key => $key,
        LOGINPAGE => "LOGINPAGE",
        UserId => $opts{username},
        Password => $opts{password},
    });
```

As you can see, a lot of this is handling the hidden form fields that the front page provides. With *Mechanize*, all of that is taken away as the forms are parsed and the form values persist nicely:

```
$ua->get("https://online.lloydstsb.co.uk/customer.abc");
croak $ua->res->error_as_HTML unless $ua->res->is_success;
$ua->field(UserId => $opts{username});
$ua->field(Password => $opts{password});
$ua->click;
```

Much more Perl-ish! This is what I like about *WWW::Mechanize*; it allows us to code at the appropriate level—we’re simply saying what we want done, instead of how we want to do it. We don’t want to get bogged down in the mechanics, we just want to fill in two fields and click the button.

Another useful feature is that we don’t need to explicitly store the *HTTP::Response* object returned by *get*; *Mechanize* automatically stashes that away for us, as well as the current *HTTP::Request*.

Unfortunately, this doesn’t work. It would, were it not for this little snippet of HTML:

```
<input type="UserId" name="UserId" ...
```

Mechanize uses *HTML::Form*, which quite rightly gets very distraught with the idea of a *UserId*-type input. Most browsers render this as a text input, and we should do the same. Hence we have to get a little dirty with the *HTML::Form* object. First, we find the object that represents the user ID form input:

```
my $input = $ua->current_form->find_input("UserId");
```

This is currently an ignored input, an instance of *HTML::Form::IgnoreInput*. We want to change it into a text input:

```
$input->{type} = "text";
bless $input, "WWW::Form::TextInput";
```

And now we should find ourselves at a page with the three drop-down menus on it. If not, we’ve probably failed to login. The three characters it wants are stored in *ResponseKey0* through *ResponseKey2*, so we extract those from our memorable information phrase and put them back in the form:

```
for (0..2) {
    my $key;
    eval { $key = $ua->current_form->find_input
        ("ResponseKey$_")->value; };
    croak "Couldn't log in; check your password and username"
        if $@;
    my $value = substr($opts{memorable}, $key-1, 1);
    $ua->field("ResponseValue$_" => $value);
}
```

Now this happens to return a redirect (it’s funny how bank sites tend to bring out all the interesting edge cases in screen scraping...), so we go to the location it specifies:

```
$response = $ua->click;
$ua->get($response->{headers}->{location});
```

And now, finally, we’re logged in! Now what?

Extracting the Account Balances

The account balances are stored in a table on the front page. We need to parse the HTML for the table and extract the account name, number, and the balance. Thankfully, Johnathan Stowe has written a very neat little *HTML::Parser*-based table parser in his collection of *HTML::Parser* examples (<http://www.gellyfish.com/htexamples/>), and so we steal his code pretty much wholesale.

HTML::Parser works by calling various methods every time it sees certain elements of the HTML document—a method for an opening tag, one for a closing tag, and one for any text in the middle.

We want to collect the data out of a table, so we want to be concerned with the *table*, *tr*, and *td* tags. In particular, when we see a *tr* tag, we want to start a new row; when we see a *td* tag, we start a new entry:

```
sub start {
    my ($self,$tag,$attr,$attrseq,$orig) = @_;
    if ($tag eq 'table') { $intable++; $self->{Table} = []; }
    if ($tag eq 'tr') { $inrecord++; $self->{Row} = []; }
    if ($tag eq 'td') { $infield++; $self->{Field} = ''; }
}
```

*Now, don't get me wrong, I'm not
averse to more security on my
bank account per se, but I am
averse to having my nice
labor-saving module break*

HTML::Parser actually passes in a lot of things we don't really care about, such as the attributes; we're only interested in storing data structures for the table, row, and cell.

Now, if we're inside a table and a row and a cell, we want to collect any text we see into the current field:

```
sub text {
    my ($self,$text) = @_;
    if ($intable && $inrecord && $infield) { $self->{Field} .= $text; }
}
```

We have to concatenate onto *\$self->{Field}*, as the cell may contain other tags. If we have:

```
<TD> Hello, <B>esteemed</B> visitor</TD>
```

then *text* will actually be called three times, and *start* and *end* will be called for the *B* tag as well. Because we want all three pieces of text instead of just the last one, we concatenate them all together.

The real hard work is done by the end tag processor. We've been gathering text into *\$self->{Field}*, and when we come to the end of our cell, the *</td>* tag, we push the contents we've accumulated into the current row; similarly, when we come to the end of a row, we push that row onto the table:

```
sub end {
    my ($self,$tag) = @_;
    if ($tag eq 'table') { $intable--; }
    if ($tag eq 'td') { $infield--;
        push @{$self->{Row}}, $self->{Field}; }
    if ($tag eq 'tr') { $inrecord--;
        push @{$self->{Table}}, $self->{Row}; }
}
```

And, well, that's it; once this package inherits from *HTML::Parser* like so:

```
package TableThing;
use strict;
use vars qw(@ISA $infield $inrecord $intable);
use base 'HTML::Parser';
```

we will be able to use it to parse our tables. Once we've logged in, we can say:

```
my $table_parser = TableThing->new;
$table_parser->parse($ua->content);
```

and *\$table_parser* will now contain a data structure representing the table of accounts. Extracting the balances is now a matter of ordinary Perl data-structure munging.

Putting It Together

Let's now put this lot together into a module, *Finance::Bank::LloydsTSB*. We'll start with the ordinary module preamble, and set up our browser object:

```
package Finance::Bank::LloydsTSB;
use strict;
use Carp;
our $VERSION = '1.2';
use WWW::Mechanize;
our $ua = WWW::Mechanize->new(
    env_proxy => 1,
    keep_alive => 1,
    timeout => 30,
);
```

Our constructor will be called *check_balance* since it will return a bunch of account objects. We make sure that it has the requisite parameters, and bless that into an object:

```
sub check_balance {
    my ($class, %opts) = @_;
    croak "Must provide a password"
        unless exists $opts{password};
    croak "Must provide a username"
        unless exists $opts{username};
    croak "Must provide memorable information"
        unless exists $opts{memorable};

    my $self = bless { %opts }, $class;
```

And now comes the *Mechanize* code we established before:

```
$ua->get("https://online.lloydstsb.co.uk/customer.ibc");
my $field = $ua->current_form->find_input("UserId1");
$field->{type}="input";
bless $field, "HTML::Form::TextInput";
$ua->field(UserId1 => $opts{username});
$ua->field>Password => $opts{password});
$ua->click;

for (0..2) {
    my $key;
    eval { $key = $ua->current_form->find_input
        ("ResponseKey$_")->value; };
    croak "Couldn't log in; check your password and username" if $@;
    my $value = substr(lc $opts{memorable}, $key-1, 1);
    $ua->field("ResponseValue$_" => $value);
}

my $response = $ua->click;
$ua->get($response->{_headers}->{location});
```


Getting the data out of the table turns out to be slightly tricky; first, we extract those table rows that contain nonwhitespace:

```
my @table = @{$foo->{Table}};  
@table = grep { grep { s/\s//g; s/\s{2,}//g; /\S/ } @$_ } @table;
```

The top row is a header, so we get rid of that:

```
shift @table;
```

And now we look for cells that contain nonwhitespace:

```
for (@table) {  
    my @line = grep /\S/, @$_;
```

The balance is the last cell in the line and is specified as a number, followed by either *CR* for credit or *DR* for overdrawn, so we fix that up to be a real number:

```
my $balance = pop @line;  
$balance =~ s/ CR//;  
$balance = -$balance if $balance =~ s/ DR//;
```

We can extract the other components of the table directly and bless them into the *Finance::Bank::LloydsTSB::Account* class. We'll also throw in a link to the current *\$self* because, although it's not used at the moment, we can later use this for reconfirming the password when we make transfers or payments:

```
push @accounts, (bless {  
    balance => $balance,  
    name => $line[0],  
    sort_code => $line[1],
```

```
    account_no => $line[2],  
    parent => $self  
}, "Finance::Bank::LloydsTSB::Account");  
}  
return @accounts;
```

And that's basically our module. All that remains is to provide accessors for the *name*, *sort_code*, *account_no*, and *balance*, and we do this extremely lazily:

```
package Finance::Bank::LloydsTSB::Account;  
sub AUTOLOAD { my $self=shift; $AUTOLOAD =~ s/.*:://;  
    $self->{$AUTOLOAD} }
```

So before we know it, we've written an interface to our online banking system. Interestingly, even though we added another screen to go through in the form of our memorable information page, the module ended up being four lines shorter than the previous incarnation—this was directly due to changing from *LWP::UserAgent* to *WWW::Mechanize* and programming at a more appropriate level.

I've found *WWW::Mechanize* useful for hacking up all kinds of screen-scraping code, from simple tests of web-based services right up to full-featured CPAN modules as we've seen in this article. *Finance::Bank::LloydsTSB* is available from CPAN, and has spawned several other online banking access modules, many of which switched to *Mechanize* much earlier than I did. I hope from this article you've gained some impression of how to go about writing something to interface to your own banking service, and an idea of how to use *WWW::Mechanize* in order to automate web access from Perl.

TPJ

Subscribe now to

Dr. Dobb's E-mail Newsletters

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

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

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



HTML Filtering in Perl, Part I

Randal Schwartz

The simplicity of HTML is sometimes deceiving. Sure, it's pretty easy for your average Perl hacker to set up a web-based bulletin board system, allowing people to come along and write comments. It's even tempting to allow those comments to contain HTML rather than being escaped into monospaced `<pre>` purgatory. But "there be dragons there," as the old maps used to say.

The problem is that arbitrary HTML permits arbitrary activities to be triggered by merely visiting the site, thanks to these fancy scriptable browsers. As reported in the security journals, these attacks are generally known as "cross-site scripting." They usually come in the form of a JavaScript chunk embedded in a web page where at least part of the content can be controlled by arbitrary visitors, such as a guestbook or a web-based message system. Left unchecked, such attacks can unknowingly leak a person's credentials (such as cookies) to the bad guys, and that can lead to some pretty bad stuff.

Even without the issue of cross-site scripting, we still have to watch out for arbitrary HTML and JavaScript that can trigger browser bugs, which can again lead to denial-of-service attacks or usurped credentials. While keeping up with the latest browser release usually prevents this, most people I know don't upgrade at the first notice, leading to a vulnerability window.

And then there are the just plain annoyances. People who put HTML "start bold" tags in without the end bold. Or worse yet, including a start comment marker without the matching end comment. This isn't always a malicious act: It could happen just as easily by accident.

Because there are so many ways to go wrong, people tend to forbid all HTML, escape everything through an entity escaper, and leave it at that. But how do you permit some "safe" HTML while being very careful not to let "dangerous" HTML or comments into your code? For example, what if inline images were deemed to be annoying? How do you ensure that you are stripping all `img` elements?

This month, I'll present, in detail, an HTML stripper program to tackle this problem. The program depends on an HTML filter module that I've written. Next month, I'll give a line-by-line breakdown of the filter module. (Both listings are shown at the end of this column, and will be repeated next month.)

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

I've seen a few solutions to tidy up HTML, usually based on a series of regular-expression replacements (such as `HTML::Sanitizer` in the CPAN). But these often fail to consider the matching-tag or the implicit close-tag problems of HTML. For example, consider the valid HTML of:

```
<table><tr><td><b>foo<td>bar</table>
```

In this case, the bolding really does end at the end of *foo*, so *bar* should be rendered as unbolded. But to know that, you have to know that the `td` element closes off the previous `td` element, and therefore also the `b` element as well. That's a bit hard to get into the regular expressions.

One all-encompassing solution is `HTML::TreeBuilder` from the CPAN. This code understands the nesting and optional closing tags of HTML, and wraps itself around `HTML::Parser` to find the tags and other syntax of an HTML document. Once we have a nice clean tree of properly parsed and nested HTML elements, we merely need to walk through the tree, throwing away the dangerous elements. As long as we don't mangle the tree, we should get properly nested tags out of the mix as well.

The problem with a solution based around `HTML::TreeBuilder` is that it is too expensive to use repeatedly (such as every time a page is reloaded). While `HTML::Parser` is pretty fast, `HTML::TreeBuilder` has to build a lot of heavily connected heavy Perl objects, at least one for every element of the tree. This kind of tree is slow to create and slow to discard, so a heavily hit web site would be bogged down in short order.

But, from the XML realm (of all places) comes another interesting solution, in the form of `XML::LibXML`, which is a wrapper around the GNOME `libxml2` parser. Although it can be a bit finicky to install, many interesting things become possible once you've got it there.

The `XML::LibXML` library can parse things in HTML mode, not just XML mode. In HTML mode, missing close tags are automatically deduced, HTML entities are optional and error-corrected, and quotes around attribute values are optional. All of these would be fatal to a normal XML parser. The result of an HTML parsing is an in-memory Document Object Model (DOM) that can then be accessed with XPath or DOM APIs. The advantage is that the DOM stays in the library (C code) side of the picture until requested, rather than in a bag of Perl objects.

In my time trials, regardless of whether the HTML file was small or huge, an HTML parse with `XML::LibXML` was 10 to 20 times faster than the equivalent parse with `HTML::TreeBuilder`.

This is good news because most of the time is spent recognizing the data and building the tree, so reducing that gives us a big win.

So, once we build the DOM, it's a matter of walking the DOM, removing the forbidden elements and attributes, and then spitting the result out as HTML. And I've constructed a proof-of-concept module for that, which I'll describe next month.

To test my code, I needed a list representing a typical web-based community system's permitted HTML elements and attributes. Since I frequent the Perl Monastery at <http://www.perlmonks.org/>, I decided to grab their list of approved HTML for typical questions or answers. I extracted the list, and put it into the center of Listing 1.

Lines 1–3 of this program begin nearly every program I write, and turn on compiler restrictions and disable the normal *STDOUT* buffering.

Line 5 pulls in the *My_HTML_Filter* module, containing my HTML filtering code. This module is expected to be somewhere within my *@INC* path. Because I was always invoking this program from the current directory, I put the *.pm* file in the same directory for testing. In a production system, I might have had to alter *@INC* to access the locally installed module.

Line 7 gives the URL from which these elements and attributes are extracted. Lines 9–49 create the hash of permitted elements and attributes, as a nested hash. The first level of the hash has a key for every valid element. The corresponding value is a hashref, pointing to a second hash of where the keys represent every valid attribute for that element. The corresponding values for those keys are simply the number 1, permitting a truth test rather than an existence test for when we finally want to check for validity.

The code to create this hash from the “here document” is in lines 10 and 11. First, the data is split on newline, and then for each line, a further split on whitespace puts the first word of the line into *\$k*, and the remaining words into *@v*. Then, two elements are generated for each input element: the *\$k* value, and a hashref of a hash where the keys are all the *@v* elements and the values are all 1.

The list of elements and attributes given here is by no means promised to be safe. It just happens to be what is in use at the moment at the Perl Monastery, and has evolved over time.

Line 51 and beyond create a *Test::More* document, usually used in testing a module within a distribution, but handy here while I was developing and understanding the module. The *no_plan* in line 51 indicates that *Test::More* will count the number of tests and put the “plan” for the tests at the end of the output rather than the beginning.

Line 53 creates a filter object *\$f*, passing it the permitted elements and attributes hash. Line 54 tests *\$f* to ensure that it's actually an object of the intended type.

Lines 56–88 illustrate some of the transformations of this HTML stripper. Each is in the form of:

```
is($trial_text, $reference_text, $explanation)
```

The *\$trial_text* comes from running the filter on the given string, resulting in some HTML output. This is compared to the *\$reference_text*, which is what we are hoping the output resembles. The *\$explanation* describes the particular test. A sample run of this part of the code looks like:

```
ok 1 - The object isa My_HTML_Filter
ok 2 - basic text gets paragraphed
ok 3 - bogons gets stripped
ok 4 - links are permitted
ok 5 - attributes get quoted
ok 6 - bad attributes get stripped
ok 7 - comments get stripped
ok 8 - tags get balanced
ok 9 - b/i tags get balanced
ok 10 - b/i tags get nested properly
ok 11 - tags get lowercased
ok 12 - br comes out as HTML not XHTML
```

This test list is by no means a full suite of tests that I would use for a production module, but it shows the basics. Bad attributes and comments are removed, bad elements are stripped (and their contents pulled up inline), close tags are automatically added according to HTML rules, and generally, life is good. The resulting HTML could be inserted into an output page safely.

And then the fun part—lines 90–97 show me just how fast or slow this code actually can be. I placed the home page for <http://www.stonehenge.com/> into a local file, then brought the contents into *\$homepage* in line 91 (using the *autovivified* filehandle mechanism new to Perl 5.8). I then ran the stripper on the text (about 8K as I'm testing this) until a CPU second passed, and reported the number of passes per second that can be achieved. On an 8K chunk of HTML (much larger than a typical question or answer at the Monastery), I see about 40 to 50 results per second on my 1-GHz laptop. This is well within reasonable bounds, assuming we cache the result in some nice place on a high-performance website. Thus, the code is useful.

So that's the HTML stripper. You can examine the code for the *My_HTML_Filter* module, which is shown in Listing 2; and next month, I'll walk through that module in detail.

TPJ

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

Listing 1

```
=0= ##### LISTING ONE (main program) #####
=1= #!/usr/bin/perl
=2= use strict;
=3= $|++;
=4=
=5= use My_HTML_Filter;
=6=
=7= ## from http://www.perlmonks.org/index.pl?node_id=29281
=8=
=9= my %PERMITTED =
=10= map { my($k, @v) = split; ($k, {map {$_, 1} @v}) }
=11= split /\n/, <<'END';
=12= a href name target class title
=13= b
=14= big
=15= blockquote class
=16= br
=17= center
=18= dd
=19= div class
=20= dl
```

```
=21= dt
=22= em
=23= font size color class
=24= h1
=25= h2
=26= h3
=27= h4
=28= h5
=29= h6
=30= hr
=31= i
=32= li
=33= ol type start
=34= p align class
=35= pre class
=36= small
=37= span class title
=38= strike
=39= strong
=40= sub
=41= sup
=42= table width cellpadding cellspacing border bgcolor class
=43= td width align valign colspan rowspan bgcolor height class
```



```

=44=   th colspan width align bgcolor height class
=45=   tr width align valign class
=46=   tt class
=47=   u
=48=   ul
=49=   END
=50=
=51=   use Test::More qw(no_plan);
=52=
=53=   my $f = My_HTML_Filter->new(\\%PERMITTED) or die;
=54=   isa_ok($f, "My_HTML_Filter");
=55=
=56=   is($f->strip(qq{Hello}),
=57=       qq{<p>Hello</p>\n},
=58=       "basic text gets paragraphed");
=59=   is($f->strip(qq{<p><bogus>Thing}),
=60=       qq{<p>Thing</p>\n},
=61=       "bogons gets stripped");
=62=   is($f->strip(qq{<a href="foo">bar</a>}),
=63=       qq{<a href="foo">bar</a>\n},
=64=       "links are permitted");
=65=   is($f->strip(qq{<a href=foo>bar</a>}),
=66=       qq{<a href="foo">bar</a>\n},
=67=       "attributes get quoted");
=68=   is($f->strip(qq{<a href=foo bogus=place>bar</a>}),
=69=       qq{<a href="foo">bar</a>\n},
=70=       "bad attributes get stripped");
=71=   is($f->strip(qq{<p>What do <!-- comment -->you say?}),
=72=       qq{<p>What do you say?</p>\n},
=73=       "comments get stripped");
=74=   is($f->strip(qq{<table><tr><td>Hi</td></tr></table>\n}),
=75=       qq{<table><tr><td>Hi</td></tr></table>\n},
=76=       "tags get balanced");
=77=   is($f->strip(qq{<b><i>bold italic!</b></i>}),
=78=       qq{<b><i>bold italic!</i></b>\n},
=79=       "b/i tags get balanced");
=80=   is($f->strip(qq{<b><i>bold italic!</b></i>}),
=81=       qq{<b><i>bold italic!</i></b>\n},
=82=       "b/i tags get nested properly");
=83=   is($f->strip(qq{<B><I>bold italic!</I></B>}),
=84=       qq{<b><i>bold italic!</i></b>\n},
=85=       "tags get lowercased");
=86=   is($f->strip(qq{<hl>hey</hl>one<br>two}),
=87=       qq{<hl>hey</hl>\n<p>one<br>two</p>\n},
=88=       "br comes out as HTML not XHTML");
=89=
=90=   use Benchmark;
=91=   my $homepage = do { open my $f, "homepage.html"; join "", <$f> };
=92=
=93=   timethese
=94=       (-1,
=95=       {
=96=         strip_homepage => sub { $f->strip($homepage) }
=97=       });

```

Listing 2

```

=0=   ##### LISTING TWO (My_HTML_Filter.pm) #####
=1=   package My_HTML_Filter;
=2=   use strict;
=3=   require XML::LibXML;
=4=   my $PARSER = XML::LibXML->new;
=5=
=6=   sub new {
=7=       my $class = shift;
=8=       my $permitted = shift;
=9=       return bless { permitted => $permitted }, $class;
=10=   }
=11=
=12=   sub strip {
=13=       my $self = shift;
=14=       my $html = shift;
=15=
=16=       my $dom = $PARSER->parse_html_string($html) or die "Cannot
=17=       parse";
=18=       my $permitted = $self->{permitted};
=19=
=20=       my $cur = $dom->firstChild;
=21=       while ($cur) {
=22=           my $delete = 0;           # default to safe
=23=
=24=           ## I really really hate switching on class names
=25=           ## but this is a bad interface design {sigh}
=26=           if (ref $cur eq "XML::LibXML::Element") {
=27=               ## "that which is not explicitly permitted is forbidden!"
=28=               if (my $ok_attr = $permitted->{$cur->nodeName}) {
=29=                   ## so this element is permitted, but what about its
=30=                   attributes?
=31=                   for my $att ($cur->attributes) {
=32=                       my $name = $att->nodeName;

```

```

=31=           $cur->removeAttribute($name) unless $ok_attr->{$name};
=32=       }
=33=       ## now descend if any kids
=34=       if (my $next = $cur->firstChild) {
=35=           $cur = $next;
=36=           next;           # don't execute code at bottom
=37=       }
=38=   } else {
=39=       ## bogus - delete!
=40=       ## we must hoist any kids to be after our current position
=41=       in
=42=       ## reverse order, since we always inserting right after old
=43=       my $parent = $cur->parentNode or die "Expecting parent of
=44=       $cur";
=45=       for (reverse $cur->childNodes) {
=46=           $parent->insertAfter($_, $cur);
=47=       }
=48=       ## and flag this one for deletion
=49=       $delete = 1;
=50=       ## fall out
=51=   } elsif (ref $cur eq "XML::LibXML::Text"
=52=           or ref $cur eq "XML::LibXML::CDATASection") {
=53=       ## fall out
=54=   } elsif (ref $cur eq "XML::LibXML::Dtd"
=55=           or ref $cur eq "XML::LibXML::Comment") {
=56=       ## delete these
=57=       $delete = 1;
=58=       ## fall out
=59=   } else {
=60=       warn "[what to do with a $cur?"; # I hope we don't hit this
=61=   }
=62=
=63=   ## determine next node ala XPath "following::node()[1]"
=64=   my $next = $cur;
=65=   {
=66=       if (my $sib = $next->nextSibling) {
=67=           $next = $sib;
=68=           last;
=69=       }
=70=       ## no sibling... must try parent node's sibling
=71=       $next = $next->parentNode;
=72=       redo if $next;
=73=   }
=74=   ## $next might be undef at this point, and we'll be done
=75=
=76=   ## delete the current node if needed
=77=   $cur->parentNode->removeChild($cur)
=78=       if $delete;
=79=
=80=   $cur = $next;
=81=
=82=   my $output_html = $dom->toStringHTML;
=83=   $output_html =~ s/.*\n//;           # strip the doctype
=84=
=85=   return $output_html;
=86=   }
=87=
=88=   1;

```

TPJ





Programming Web Services with Perl

Piers Cawley

I remember, lo these many years ago (okay, so it was only five years ago, but that's centuries in Internet time), reading Dave Winer's Davenet article about his new XML-RPC, which would use XML over HTTP to make remote procedure calls. "Huh?" I thought, "Isn't that what CGI does? All you have to do is return XML documents." So, I pretty much ignored it. Meanwhile, XML-RPC turned into SOAP, Pavel Kulchenko wrote *SOAP::Lite*, Randy J. Ray wrote *RPC::XML*, and the likes of Google and Amazon exposed SOAP interfaces to their services. So maybe it's time for me to take a closer look.

O'Reilly & Associates has been my touchstone over the years for technology-specific books, so their *Programming Web Services with Perl*, by Kulchenko and Ray, seemed like the obvious choice and, with the exception of a few quibbles about copyediting (not uncommon with first printings of O'Reilly titles, I'm afraid), it has proven to be a good choice. While I'm not yet convinced that I'm a fan of XML-RPC or SOAP, I will try in this review to keep my criticisms of web services in general separate from any criticisms of the book itself.

The book starts with an enjoyable, clear-headed overview of the state of web services today. The authors point out areas of hype and cynicism, and declare that the truth is somewhere in the middle between the points of view of web-services proponents and detractors. They then whet the reader's appetite with a couple of simple client applications for real-world web services.

The next chapter gives an overview of HTTP, XML, and XML Schema; the technologies on which the various web-service frameworks are built. The authors cover a lot of ground clearly and concisely.

In Chapter 3 we get an introduction to XML-RPC and its specifications, as well as barehanded implementations of a simple client and server. XML-RPC is a simple enough spec that it's possible to write applications using it without having to use a dedicated module (though I wouldn't recommend it). XML-RPC's chief virtue appears to be its simplicity, which is also its chief failing. Chapter 4 rewrites the barehanded examples using the three current XML-RPC libraries, *RPC::XMLSimple*, *XMLRPC::Lite*, and *RPC::XML*. From the point of view of the disinterested reader, there isn't really that much to choose from between the three toolkits shown; they all offer massive savings of programmer effort.

Piers is a freelance writer and programmer, and the writer of the Perl 6 Summary. He has been programming Perl since 1993, and is currently working on Pixie, an object persistence tool. He can be reached at pdcauley@bofh.org.uk.

Programming Web Services with Perl

Randy J. Ray and Pavel Kulchenko
O'Reilly & Associates, 2002
470 pp., \$39.95
ISBN 0-596-00206-8

The authors do describe the various trade-offs involved with each choice and show admirable restraint in their descriptions of the modules they themselves wrote. (It might please Randy J. Ray to know that, if I ever need to write an XML-RPC service, I'll use his *RPC::XML* module.)

Which brings us to Chapter 5. Described by the authors as an "XML bolus," Chapter 5 covers the low-level details of SOAP and I'm afraid it proved very hard to swallow. (Through no fault of the authors, I hasten to add.) As satirical songwriter Tom Lehrer once said, "I find that if you take the various popular song forms to their logical extremes, you can arrive at almost anything from the ridiculous to the obscene, or—as they say in New York—sophisticated." I'm still undecided as to whether SOAP is ridiculous or obscene, but it's certainly sophisticated. The authors do what they can to help the pill down though, and the reader is rewarded by arriving at the sunny uplands of Chapter 6, which deals with the basics of programming using SOAP, and more specifically, *SOAP::Lite*.

Interface is everything in *SOAP::Lite*. Kulchenko takes the crawling horror that is the SOAP spec and bundles it all up behind a really neat interface. Throughout the chapter, the authors use it to build increasingly complex applications, and the module just works, leaving the programmer free to concentrate on the business of solving the programming problem instead of the communication problem.

Chapter 7 continues to delight as we move on to the process of writing a SOAP server using the module. The authors concentrate on building a "wish list" application, similar to (but massively simplified, of course) Amazon's wishlist. Again, because the toolkit

Fame & Fortune Await You!

Become a **TPJ** author!

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

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

is so easy to use, the authors are free to concentrate on the process of designing an interface that's suitable for a web service, and on making sure that the choice of transport is decoupled from the application itself. This is all very neatly done; it wouldn't be hard to take the lessons learned in this chapter and apply them to developing your own web services.

Chapter 8 extends things still further by showing how SOAP services don't have to be accessed via HTTP. Examples are given using pure TCP sockets, Jabber, and even e-mail and FTP. Because of the care taken in designing the server application, the power of *SOAP::Lite*'s abstraction, and a cunning trick with *use* and *@ISA*, this turns out to be a doddle. The chapter finishes with an example of writing a custom encrypted transport.

There's a good deal worth paying attention to in here, especially if you're interested in using SOAP internally for enterprise applications

Chapters 9 and 10 cover WSDL and UDDI, which, from my point of view, is just more XML pain, I'm afraid. The exposition is again clear and helpful, though there did seem to be a higher incidence of typos in this chapter. I also noticed that some of the pseudo UUIDs used by the authors in the UDDI chapter weren't exactly standard: "FEEB-1E" and "EA8-69-COC5" don't look like any UUIDs I've ever seen.

Chapter 11 deals with the Representational State Transfer (REST) way of doing web services. At bottom, REST is a fancy name for my kneejerk response to XML-RPC from five years ago, but done right. Once again, the authors have done a great job of explaining the concepts involved with clarity, thoughtfulness, and some great example code. After reading their exposition, I have a much better understanding of what REST is, how to use it, why it's hard to get right, and why you should bother.

The last chapter deals with advanced topics, emerging standards for reliable messaging, distributed transactions, document-based processing, service discovery, the whole nine yards. The authors give an overview of the current state of the art. There's a good deal worth paying attention to in here, especially if you're interested in using SOAP internally for enterprise applications. There's also a useful section of internationalization issues and on improving application performance, which covers slightly more than the basic "use *mod_perl*;" rule.

The appendices include in-depth references for the XML-RPC and SOAP modules discussed in the main text, the full source code for all the examples discussed, and short but interesting Bibliography and Links sections.

After reading the book, I'm still not convinced by either XML-RPC or SOAP as technologies for reasons that I find hard to put my finger on, but convincing people like me wasn't the aim of the book. As both an overview of the various tools and techniques available and as a tutorial on the details of web-service implementation, this is an excellent book.

TPJ

Source Code Appendix

Randal Schwartz “HTML Filtering in Perl, Part I”

Listing 1

```
=0= ##### LISTING ONE (main program) #####
=1= #!/usr/bin/perl
=2= use strict;
=3= $|++;
=4=
=5= use My_HTML_Filter;
=6=
=7= ## from http://www.perlmonks.org/index.pl?node_id=29281
=8=
=9= my %PERMITTED =
=10=     map { my($k, @v) = split; ($k, {map {$_, 1} @v}) }
=11=     split /\n/, <<'END';
=12=     a href name target class title
=13=     b
=14=     big
=15=     blockquote class
=16=     br
=17=     center
=18=     dd
=19=     div class
=20=     dl
=21=     dt
=22=     em
=23=     font size color class
=24=     h1
=25=     h2
=26=     h3
=27=     h4
=28=     h5
=29=     h6
=30=     hr
=31=     i
=32=     li
=33=     ol type start
=34=     p align class
=35=     pre class
=36=     small
=37=     span class title
=38=     strike
=39=     strong
=40=     sub
=41=     sup
=42=     table width cellpadding cellspacing border bgcolor class
=43=     td width align valign colspan rowspan bgcolor height class
=44=     th colspan width align bgcolor height class
=45=     tr width align valign class
=46=     tt class
=47=     u
=48=     ul
=49=     END
=50=
=51= use Test::More qw(no_plan);
=52=
=53= my $f = My_HTML_Filter->new(\%PERMITTED) or die;
=54= isa_ok($f, "My_HTML_Filter");
=55=
=56= is($f->strip(qq{Hello}),
=57=     qq{<p>Hello</p>\n},
=58=     "basic text gets paragraphed");
=59= is($f->strip(qq{<p><bogus>Thing}),
=60=     qq{<p>Thing</p>\n},
=61=     "bogons gets stripped");
=62= is($f->strip(qq{<a href="foo">bar</a>}),
=63=     qq{<a href="foo">bar</a>\n},
=64=     "links are permitted");
=65= is($f->strip(qq{<a href=foo>bar</a>}),
=66=     qq{<a href="foo">bar</a>\n},
=67=     "attributes get quoted");
=68= is($f->strip(qq{<a href=foo bogus=place>bar</a>}),
=69=     qq{<a href="foo">bar</a>\n},
=70=     "bad attributes get stripped");
=71= is($f->strip(qq{<p>What do <!-- comment -->you say?}),
=72=     qq{<p>What do you say?</p>\n},
=73=     "comments get stripped");
=74= is($f->strip(qq{<table><tr><td>Hi!}),
=75=     qq{<table><tr><td>Hi!</td></tr></table>\n},
=76=     "tags get balanced");
=77= is($f->strip(qq{<b><i>bold italic!}),
=78=     qq{<b><i>bold italic!</i></b>\n},
=79=     "b/i tags get balanced");
=80= is($f->strip(qq{<b><i>bold italic!</b></i>}),
```



```

=81=         qq{<b><i>bold italic!</i></b>\n},
=82=         "b/i tags get nested properly");
=83=     is($f->strip(qq{<B><I>bold italic!</I></B>}),
=84=         qq{<b><i>bold italic!</i></b>\n},
=85=         "tags get lowercased");
=86=     is($f->strip(qq{<h1>hey</h1>one<br>two}),
=87=         qq{<h1>hey</h1>\n<p>one<br>two</p>\n},
=88=         "br comes out as HTML not XHTML");
=89=
=90=     use Benchmark;
=91=     my $homepage = do { open my $f, "homepage.html"; join "", <$f> };
=92=
=93=     timethese
=94=         (-1,
=95=         {
=96=             strip_homepage => sub { $f->strip($homepage) }
=97=         });

```

Listing 2

```

=0=     ##### LISTING TWO (My_HTML_Filter.pm) #####
=1=     package My_HTML_Filter;
=2=     use strict;
=3=     require XML::LibXML;
=4=     my $PARSER = XML::LibXML->new;
=5=
=6=     sub new {
=7=         my $class = shift;
=8=         my $permitted = shift;
=9=         return bless { permitted => $permitted }, $class;
=10=     }
=11=
=12=     sub strip {
=13=         my $self = shift;
=14=         my $html = shift;
=15=
=16=         my $dom = $PARSER->parse_html_string($html) or die "Cannot parse";
=17=         my $permitted = $self->{permitted};
=18=
=19=         my $cur = $dom->firstChild;
=20=         while ($cur) {
=21=             my $delete = 0;           # default to safe
=22=
=23=             ## I really really hate switching on class names
=24=             ## but this is a bad interface design {sigh}
=25=             if (ref $cur eq "XML::LibXML::Element") {
=26=                 ## "that which is not explicitly permitted is forbidden!"
=27=                 if (my $ok_attr = $permitted->{$cur->nodeName}) {
=28=                     ## so this element is permitted, but what about its attributes?
=29=                     for my $att ($cur->attributes) {
=30=                         my $name = $att->nodeName;
=31=                         $cur->removeAttribute($name) unless $ok_attr->{$name};
=32=                     }
=33=                     ## now descend if any kids
=34=                     if (my $next = $cur->firstChild) {
=35=                         $cur = $next;
=36=                         next;           # don't execute code at bottom
=37=                     }
=38=                 } else {
=39=                     ## bogon - delete!
=40=                     ## we must hoist any kids to be after our current position in
=41=                     ## reverse order, since we always inserting right after old node
=42=                     my $parent = $cur->parentNode or die "Expecting parent of $cur";
=43=                     for (reverse $cur->childNodes) {
=44=                         $parent->insertAfter($_, $cur);
=45=                     }
=46=                     ## and flag this one for deletion
=47=                     $delete = 1;
=48=                     ## fall out
=49=                 }
=50=             } elsif (ref $cur eq "XML::LibXML::Text"
=51=                 or ref $cur eq "XML::LibXML::CDATASection") {
=52=                 ## fall out
=53=             } elsif (ref $cur eq "XML::LibXML::Dtd"
=54=                 or ref $cur eq "XML::LibXML::Comment") {
=55=                 ## delete these
=56=                 $delete = 1;
=57=                 ## fall out
=58=             } else {
=59=                 warn "[what to do with a $cur?]"; # I hope we don't hit this
=60=             }
=61=
=62=             ## determine next node ala XPath "following::node()[1]"
=63=             my $next = $cur;
=64=             {
=65=                 if (my $sib = $next->nextSibling) {
=66=                     $next = $sib;
=67=                     last;
=68=                 }
=69=                 ## no sibling... must try parent node's sibling

```

```
=70=         $next = $next->parentNode;
=71=         redo if $next;
=72=     }
=73=     ## $next might be undef at this point, and we'll be done
=74=
=75=     ## delete the current node if needed
=76=     $cur->parentNode->removeChild($cur)
=77=         if $delete;
=78=
=79=     $cur = $next;
=80= }
=81=
=82= my $output_html = $dom->toStringHTML;
=83= $output_html =~ s/.*\n//;      # strip the doctype
=84=
=85= return $output_html;
=86= }
=87=
=88= 1;
```