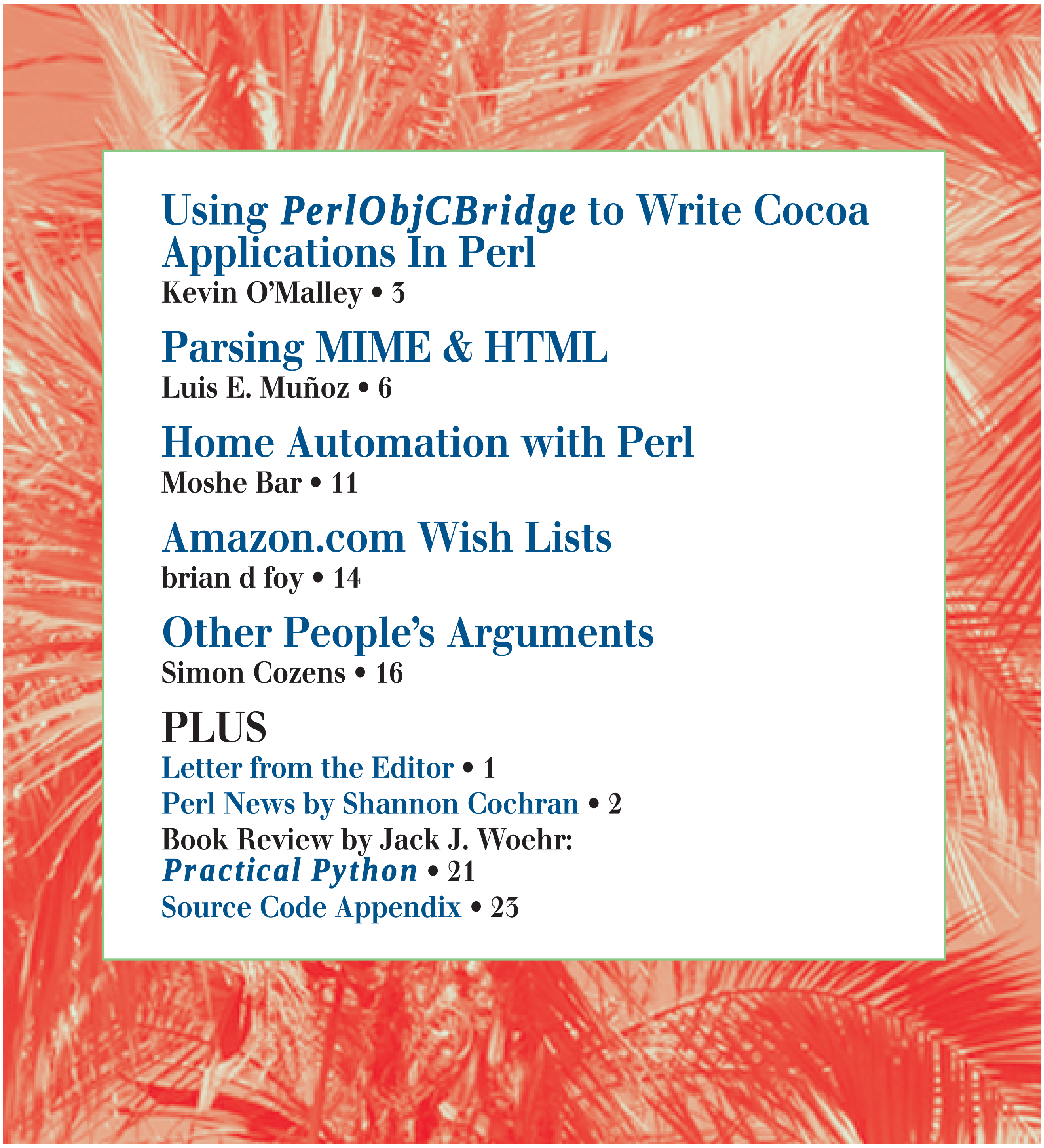
CMP
United Business Media

February 2003

The Perl Journal



Using *PerlObjCBridge* to Write Cocoa Applications In Perl
Kevin O'Malley • 3

Parsing MIME & HTML
Luis E. Muñoz • 6

Home Automation with Perl
Moshe Bar • 11

Amazon.com Wish Lists
brian d foy • 14

Other People's Arguments
Simon Cozens • 16

PLUS
Letter from the Editor • 1
Perl News by Shannon Cochran • 2
Book Review by Jack J. Woehr:
Practical Python • 21
Source Code Appendix • 23

LETTER FROM THE EDITOR

Technophiles, Tinkerers, and TiVo

The best part of programming—at least with scripting languages like Perl—is the thrill you get from watching something do the work for you. For some of us, it doesn't matter how long we've been writing code: We still get a little giddy with the thought of how much drudgery our code saves us.

Once you've caught this bug, possibilities for automation pop up everywhere and the world turns into one big nail for a programming hammer. Whether it's the joy of creativity, or just a healthy disdain of manual labor (or a little of both), it's central to what programmers do. It's all about looking for a better way to do things.

For example, take Moshe Bar's "Home Automation with Perl" article in this issue. You could argue that the technologies he employs (X10, Festival, Perl, and AppleScript, among others) represent a better way of interacting with your house and the things in it. Of course, we technophiles are optimistic about these things. It takes a lot of refinement before technologies like these are realized in a way that nontechies can get excited about. If I told my wife I wanted to automate our home, her first reaction would be: "Please tell me I'm not going to have to talk to a computer to open the living room curtains. Just promise me that. The pull-cord works just fine." And my wife is no Luddite—she just has a healthy respect for Keeping It Simple. She's often my barometer for whether or not a technology has reached the point of sensible simplicity that makes it attractive to average users.

Or take digital video recorders (DVRs). For years, people have been digitally recording television on PCs. With a big enough hard drive and a little automation, you could set up a fairly sophisticated system for recording shows you want, skipping the junk you don't. But most people see computers as objects of frustration, or at the very least confusion, and don't want to have to use them for a simple activity like watching TV. It took TiVo and ReplayTV to widen the appeal of this practice beyond the technophiles, culminating recently in FCC Chairman Michael Powell's declaration that TiVo was "God's machine." (If you haven't heard about this, just Google "God's machine." You can't miss it.)

Now, Powell's pitch wasn't the reason, but I purchased a TiVo and I have to say he's right. Here's the thing: I love this machine. I picked the shows I wanted to watch, and TiVo went and found them. No messing with time and channel. No punching in codes. Now, every time I turn on the television, I have several shows prerecorded that I already knew I wanted to watch. And I can fast-forward through the commercials. It's marvellous, and perilous. I think I'm watching way more TV than is good for me.

This device takes automation to a glorious extreme, but with such simplicity that you really don't even need to read the manual. Everything you need to know you get from the interface. There are no buttons or panels anywhere on the unit (a testament to its software design), and it seems to instinctively do the right thing. It's a triumph of interface design, and it's built on the shoulders of technophile tinkerers because it runs Linux under the hood. And yes, my wife loves it. Now if only the engineers at TiVo would build a home-automation interface…

Kevin Carlson

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.

Shannon Cochran

Perl News

Yet Another…

No less than four Yet Another Perl Conferences have been scheduled for 2003, including YAPC::Israel (announced last month). New this year is a specifically Canadian YAPC, scheduled to take place at Carlton University in Ottawa from May 15–16. Registration costs are $75 Canadian (about $50 in U.S. dollars) or $45 Canadian for students. There will be two speaker tracks with about 12 total presentions and a free tutorial, as well as professional training opportunities. Submissions for presentations will be accepted until March 21. Preregistration is available online at http://www.yapc.ca/.

The larger YAPC::America::North will be held June 16–18 in Boca Raton, Florida. Preregistration began February 1st, and the call for participation is online at http://www.yapc.org/America/cfp.shtml. Standard presentations will run 20 minutes, but lightning talks, long or extra-long talks, and half-day tutorials will also be considered. Conference fees are $85.

And yes, there's yet another Perl conference—YAPC::Europe, which will be held in Paris, from July 23–25, in the Conservatoire National des Arts et Métiers. The conference will cost 99 Euros, and preregistration will begin March 1. The call for participation is at http://www.yapc.org/Europe/2003/cfp.en.html.

PerlASPX Release Candidate Available

ActiveState's PerlASPX project allows ASP.NET-compliant web pages and web services to be written in Perl. ASP.NET apps can use early binding, just-in-time compilation, native optimization, and caching services to improve performance, but Microsoft provides support for only C#, Visual Basic, and JScript.

While ActiveState's web site, at http://aspn.activestate.com/ASPN/Downloads/PerlASPX, mentions only a beta version of the software (which expired in December), a more recent release candidate was actually announced in January—contact Support@ActiveState.com to request a copy. The company expects to release the final version later this month. PerlASPX requires ActiveState's Perl Dev Kit and ActivePerl, as well as either the Microsoft .NET Framework Redistributable or the Microsoft .NET Framework SDK.

TinyPerl 2.0 Released

TinyPerl, a bare-bones, 616 KB version of Perl 5.8, is now available as a SourceForge project (http://tinyperl.sourceforge.net/). The project is described as "Perl on a floppy," and is designed to aid developers seeking to publish their Perl applications. The 2.0 release is able to generate Windows executables from .pl scripts (there is currently no Linux or Mac version of TinyPerl). Author Graciliano Monteiro Passos posted on Perl Monks that "This will be the last release [for] a long, long time! It's very stable, and with everything that I want for now."

Mason E-Book Freely Available

The complete text of *Embedding Perl in HTML with Mason* by Dave Rolsky and Ken Williams (including corrections and example code, but so far excluding the index) has been posted to http://www.masonbook.com/. The authors say that the tool used to convert the book to HTML will also be published online. "Though," they add, "we of course encourage you to buy a copy."

Module Maintainers Sought

Following the passing of Israel.pm member Arial Brosh, new maintainers are sought for his modules. Brosh's author directory on CPAN—http://search.cpan.org/author/SCHOP/—list fourteen modules, including code for whois queries and a project to bring Hebrew character support to Perl. Volunteers should contact Gabor Szabo at gabor@perl.org.il.

A memorial web site for Brosh is also under consideration.

Browsing the Source

Zach Lipton has installed LXR—"Linux Cross Reference," a source-code indexing tool originally developed for use on the Linux kernel—at http://tinderbox.perl.org/lxr/. Now the Parrot and Perl5 source code is searchable by filename or contents, and hyperlinks within the source files allow browsers to find all references to a function or subroutine. LXR can also diff two separate versions of a file. The source tree is updated every night.

Bug reports or feature requests should be sent to Lipton at zach@zachlipton.com. He did warn on the perl6-internals list that "occasionally, LXR will return a blank or near-blank page. If you hit this bug, hitting reload a couple of times should make the correct file show up."

Archive Toolkit Mailing List Launched

A new mailing list has been set up on perl.org for users and developers of the Perl Archive Toolkit, and for discussion of general Perl packaging issues. App::Packer and Apache::PAR users are also encouraged to join. You can subscribe by sending mail to par-subscribe@perl.org, and the list archives are also posted at http://archive.develooper.com/par@perl.org/.

Setting Priorities

Damian Conway wrote on the perl6-language list that "It's my understanding that TPF is not intending to offer Larry [Wall] (or Dan [Sugalski]) another grant for 2003," on the grounds that Perl 6 and Parrot should not eclipse other Perl projects. Discussion on this issue continues; the Perl Foundation is reportedly in the process of setting up an online questionnaire to solicit feedback about where advocacy efforts and grant money should be best applied.

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

Kevin O'Malley

Using *PerlObjCBridge* to Write Cocoa Applications In Perl

On March 24, 2001, Apple released its new operating system, called "Mac OS X," to the public. Mac OS X (pronounced "OS Ten") is built on an open-source, UNIX-based core operating system called "Darwin." Mac OS X includes a full set of UNIX commands and tools, and a host of new development frameworks and technologies. One of these is Cocoa, an object-oriented environment for developing native Mac OS X applications. Cocoa provides developers with a rich component framework that greatly simplifies and facilitates the development of Mac OS X applications and GUIs. In fact, Apple recommends that developers use Cocoa when writing new applications for Mac OS X. For more information on Cocoa, see the Cocoa Resources section at the end of the article.

The latest release of Mac OS X (10.2), called "Jaguar," includes a Perl module called *PerlObjCBridge*. *PerlObjCBridge* enables Perl programmers to access Cocoa objects and services from their Perl scripts. This is very exciting news for Perl programmers working on the Mac OS X platform.

This article presents an introduction to *PerlObjCBridge* and shows how to use it in a real Perl program.

PerlObjCBridge Fundamentals

PerlObjCBridge provides the following functions:

- Enables access to many Objective-C objects from Perl.
- Enables Perl scripts to be Cocoa delegates or targets of notifications.
- Enables Perl scripts to access Cocoa's Distributed Objects (DO) mechanism, letting Perl objects send and receive messages to and from Objective-C or Perl objects running on different machines.

One limitation of *PerlObjCBridge* is its lack of support for accessing Cocoa GUI objects. This means you cannot use it to construct user interfaces for your Perl scripts. (See http://camelbones.sourceforge.net/ for information about a GUI framework for constructing Cocoa interfaces in Perl.)

Kevin is a long-time Macintosh and UNIX developer. He is the author of the upcoming book Mac OS X Programming: A Guide for UNIX Developers *(Manning Publications, 2003), on which this article is based. He can be contacted at omalley@umich.edu.*

In terms of syntax, Objective-C uses ":" to delimit arguments, which is not legal in Perl. Therefore, the "_" character is used in its place. To access Objective-C objects and methods from Perl, you use the following forms:

- Static method (through the class)—ClassName->method(…args…)
- Instance method (through the instance)—$object->method(…args…)

The following code shows a few examples of how to use these constructs:

```
# Accessing a method through its class (static method).
$pref = NSMutableDictionary->dictionary();
# Accessing a method through an instance (instance method).
$pref->objectForKey_($key);
```

One of the more powerful features of *PerlObjCBridge* is its ability to register Perl objects as recipients of notifications from Cocoa frameworks. For example, *PerlObjCBridge* automatically provides the stubs, or Objective-C objects, that act as proxies for Perl objects. If you have a Perl object like this:

```
package Foo;
sub new { ... }
sub aCallBack { ... }
```

you register *Foo* objects to receive *NSNotification* messages as follows:

```
$foo = new Foo();
NSNotificationCenter->defaultCenter()
->addObserver_selector_name_object_($foo,
"aCallBack", "theNotificationName", undef);
```

When the event *theNotificationName* occurs, the Cocoa Foundation sends the *aCallBack* message to *$foo*. Behind the scenes, *PerlObjCBridge* automatically creates a *PerlProxy* object to stand in for *$foo* wherever an Objective-C object is expected, as in the observer argument to the *addObserver* method. Cocoa's DO mechanism enables Cocoa programs to access objects from different programs, possibly running on different machines. You can access DO from *PerlObjCBridge*, enabling interprocess messaging between Perl objects. Basically, you write Perl scripts that run on different machines—or in different address spaces on the same machine—and that send messages to one another. Doing so enables your scripts to communicate with other scripts by directly calling their methods as if they were part of the same program.

Let's look at how to apply this knowledge in a Perl script.

A Perl-Based PIM Using *PerlObjCBridge*

These days, Palm devices are everywhere. They are used to track contacts and schedules, enter information into databases, access e-mail and the Web, and play games. In this example, we'll use software that comes with most UNIX systems to replicate some of this functionally at little or no cost.

The script, called "pim.pl," uses standard UNIX tools to track contacts, take notes, generate and view calendars, and even keep a list of quotes. The main UNIX programs used are *cal* and *remind*. (The *remind* program is freely available from http://www.roaringpenguin.com/remind/. It does not come with the system, so you will need to download it, compile it, and install it for Mac OS X.) The *cal* program displays a text-based calendar. If you have never used *remind*, you are in for a treat. Basically, *remind* is a calendar generator and reminder program with lots of options and uses. For the complete source code, see http://www.tpj.com/source/.

The first step in using *remind* is to create and edit your reminders file, called ".reminders", which is located in your home directory.

```
# The .reminders file.
REM 15 November +1 2002 MSG 4-5 Development meeting
REM 05 November 2002 *7 MSG 4:00-5:00 AI seminar
REM 04 November 2002 *1 until November 06 2002 MSG
    Out of office
```

Entries in this file represent calendar events or reminders. Once you add entries to the file, you run the remind command to process the file. Depending on the options, *remind* will output everything from a reminder list to a text- or Postscript-formatted calendar. Figure 1 shows a text-based calendar generated from the reminders file.

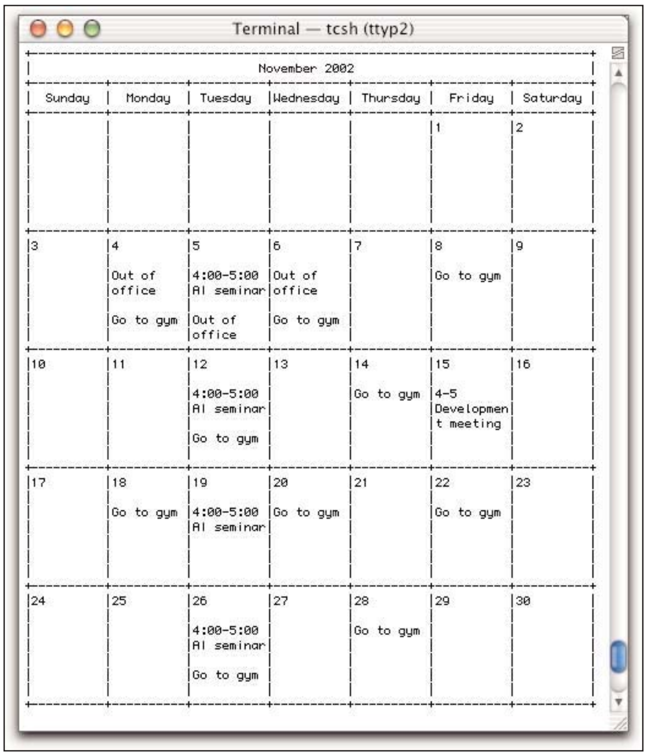
To view or edit your contact list, the Perl-based pim script opens the file in an editor; to edit tasks, it opens the task file in an editor; and to view tasks, it processes the file and prints a formatted version of the task list.

The Cocoa classes include a particularly useful set of methods: the *writeToFile* and *stringWithContentsOfFile* family of methods. Collectively, these methods let you to take an object, serialize its data to disk, and read the stored data from disk into an object at run time. When used in conjunction with the *NSMutableDictionary*, a hash data structure, you do not need to deal with formatting or parsing data; it's all done for you. This feature is particularly attractive and is a strong reason to use Cocoa objects in your Perl scripts. This program uses these features to store and access preference settings.

Application preferences are stored in a preference file, which is a text file formatted as XML. Each key/value pair in the file describes a particular program option. For example, the editor keyword is used to look up the editor that the script uses to open files (contacts-file for the name of the contacts file). See Listing 1 for an example of the preferences file.

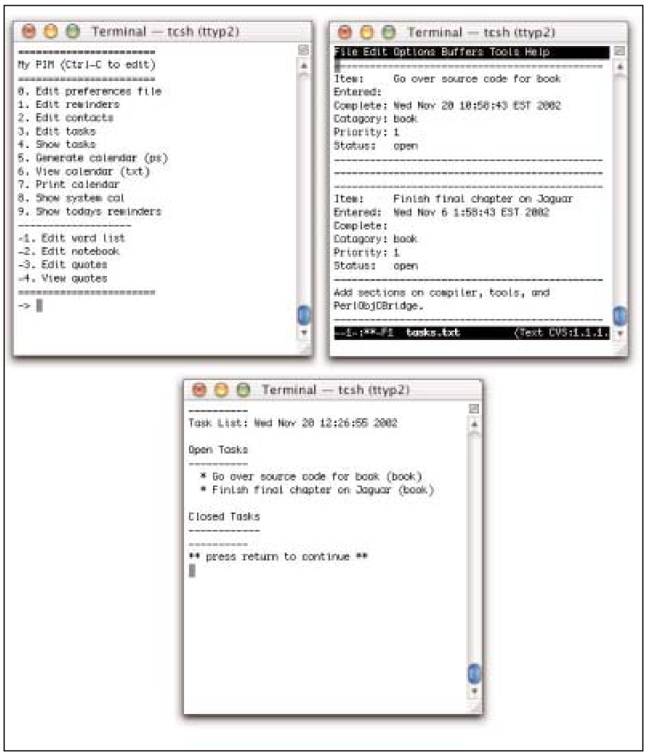
You can initially create this file either programmatically or by hand. To create it programmatically, you can use elements of the following code fragment:

```
my $pref = NSMutableDictionary->dictionary();
setPrefVal("editor", "/usr/bin/emacs");
setPrefVal("contacts-file", "contacts.txt");
writePrefs("pim.prefs");

sub setPrefVal {
  my ($key, $val) = @_;
  $pref->setObject_forKey_($val, $key);
}
sub writePrefs {
```



```
+----------------------------------------------------------------------------+
|                               November 2002                                |
+----------+----------+----------+----------+----------+----------+----------+
| Sunday   | Monday   | Tuesday  |Wednesday | Thursday | Friday   | Saturday |
+----------+----------+----------+----------+----------+----------+----------+
|          |          |          |          |          |1         |2         |
|          |          |          |          |          |          |          |
|          |          |          |          |          |          |          |
|          |          |          |          |          |          |          |
|          |          |          |          |          |          |          |
|          |          |          |          |          |          |          |
+----------+----------+----------+----------+----------+----------+----------+
|3         |4         |5         |6         |7         |8         |9         |
|          |          |          |          |          |          |          |
|          |Out of    |4:00-5:00 |Out of    |          |Go to gym |          |
|          |office    |AI seminar|office    |          |          |          |
|          |          |          |          |          |          |          |
|          |Go to gym |Out of    |Go to gym |          |          |          |
|          |          |office    |          |          |          |          |
+----------+----------+----------+----------+----------+----------+----------+
|10        |11        |12        |13        |14        |15        |16        |
|          |          |          |          |          |          |          |
|          |          |4:00-5:00 |          |Go to gym |4-5       |          |
|          |          |AI seminar|          |          |Developmen|          |
|          |          |          |          |          |t meeting |          |
|          |          |Go to gym |          |          |          |          |
|          |          |          |          |          |          |          |
+----------+----------+----------+----------+----------+----------+----------+
|17        |18        |19        |20        |21        |22        |23        |
|          |          |          |          |          |          |          |
|          |Go to gym |4:00-5:00 |Go to gym |          |Go to gym |          |
|          |          |AI seminar|          |          |          |          |
|          |          |          |          |          |          |          |
|          |          |          |          |          |          |          |
|          |          |          |          |          |          |          |
+----------+----------+----------+----------+----------+----------+----------+
|24        |25        |26        |27        |28        |29        |30        |
|          |          |          |          |          |          |          |
|          |          |4:00-5:00 |          |Go to gym |          |          |
|          |          |AI seminar|          |          |          |          |
|          |          |          |          |          |          |          |
|          |          |Go to gym |          |          |          |          |
|          |          |          |          |          |          |          |
+----------+----------+----------+----------+----------+----------+----------+
```

Figure 1: A text-based calendar generated from the reminders file.



```
========================
My PIM (Ctrl-C to edit)
========================
0. Edit preferences file
1. Edit reminders
2. Edit contacts
3. Edit tasks
4. Show tasks
5. Generate calendar (ps)
6. View calendar (txt)
7. Print calendar
8. Show system cal
9. Show todays reminders
--------------------
-1. Edit word list
-2. Edit notebook
-3. Edit quotes
-4. View quotes
========================
->
```

```
----------------------------------------------------
Item:     Go over source code for book
Entered:
Complete: Wed Nov 20 10:58:43 EST 2002
Catagory: book
Priority: 1
Status:   open
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
Item:     Finish final chapter on Jaguar
Entered:  Wed Nov 6 1:58:43 EST 2002
Complete:
Catagory: book
Priority: 1
Status:   open
----------------------------------------------------
Add sections on compiler, tools, and
PerlObjCBridge.
----------------------------------------------------
--1-:**-F1  tasks.txt             (Text CVS:1.1.1.
```

```
------------
Task List: Wed Nov 20 12:26:55 2002

Open Tasks
------------
  * Go over source code for book (book)
  * Finish final chapter on Jaguar (book)

Closed Tasks
--------------
------------
** press return to continue **
```

Figure 2: The program responding to the Task feature.

```
    my ($fName) = @_;
    $pref->writeToFile_atomically_($fName, 1);
  }
```

When the script starts, it creates a new, empty *dictionary* object by calling the static method dictionary from Foundation's *NSMutableDictionary* class. Next, it populates the dictionary (*readPrefs*) with key/value pairs from the preference file. To do so, it uses the *NSDictionary* static method *dictionaryWithContentsOfFile*. In a single call, it reads the preference file and stores each key/value pair into the *dictionary* object. This saves you the trouble of creating a new file format and writing code to parse and store the preference values:

```
  my $prefs = readPrefs($PREFS_FILE);
  sub readPrefs {
    my ($fName) = @_;
    my($dict) = NSDictionary->dictionaryWithContentsOf
                                                   File_($fName);
    if (!defined($dict)) {
      logExit("preferences file not read: $PREFS_FILE");
      exit;
    }
    return $dict;
  }
```

The rest of the script is quite simple. It goes into an infinite loop in which it displays a menu and handles user selections (see Listing 2). To exit the program, press Control-c. The handling code is also straightforward, as demonstrated by the *if/elsif* block. In both cases, it gets the appropriate value from the dictionary (based on a key) and handles the operation. Figure 2 shows the program responding to the Task feature.

Overall, this script does the job, but it could be improved. The program could be extended to use Cocoa's Distributed Objects mechanism. For example, the script could act as a server running on one machine, and you could access it from another machine to read and update information. Additionally, you could write a Cocoa GUI client that displays the information in an interface while the main handling and storage code runs on the server.

Another addition would be to add items to your calendar or task list from your e-mail inbox. This functionality would let you send yourself e-mail that goes directly into your PIM. For example, to add a task to your PIM, you could send yourself an e-mail with a special tag (TASK, for example) in the subject. Then, your PIM could read your inbox looking for message subjects with this tag and add the appropriate e-mails to your task list.

Summary

This article has given you a brief look at *PerlObjCBridge*, a Perl module that comes installed under Mac OS X 10.2, Jaguar. Use the *PerlObjCBridge* man page to get more information on its features. Using *PerlObjCBridge*, Perl programmers can now take advantage of Mac OS X's Cocoa Foundation directly from their Perl scripts. *PerlObjCBridge* is a good piece of software engineering, providing many more functions than I've discussed here. I hope it will continue to grow and include more features, including the ability to construct Cocoa GUIs from Perl scripts. If you are a Perl programmer working under Mac OS X, take a serious look at *PerlObjCBridge*.

Cocoa Resources

- Apple Computer Inc. *Learning Cocoa*. Ed. Troy Mott. Sebastopol, CA: O'Reilly, 2001.
- Buck, Eric, Donald Yacktman, and Scott Anguish. *Cocoa Programming: Programming for the MAC OS X*. Indianapolis: Sams, 2001.
- Garfinkel, Simson, and Michael K. Mahoney. *Building Cocoa Applications*. Sebastopol, CA: O'Reilly, 2002.
- Hillegass, Aaron. *Cocoa Programming for Mac OS X*. Boston: Addison-Wesley, 2002.
- Cocoa Developer Documentation: "New to Cocoa Programming." http://developer.apple.com/techpubs/macosx/Cocoa/SiteInfo/NewToCocoa.html.

Acknowledgement

Thanks to Doug Wiebe of Apple Computer, the author of the *PerlObjCBridge*, for his information and insights on this topic, and for some of the code examples that appear in this article.

TPJ

Listing 1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//
DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>prefs-path</key>
  <string>./</string>
  <key>editor</key>
  <string>/usr/bin/emacs</string>
  <key>viewer</key>
  <string>more</string>
  <key>file-path</key>
  <string>./</string>
  <key>contacts-file</key>
  <string>contacts.txt</string>
  <key>tasks-file</key>
  <string>tasks.txt</string>
  <key>words-file</key>
  <string>word_list.txt</string>
  <key>notebook-file</key>
  <string>notebook.txt</string>
  <key>quotes-file</key>
  <string>quotes.txt</string>
</dict>
</plist>
```

Listing 2

```
for(;;) {
  system("clear");
  print "======================\n";
  print "My PIM\n";
  print "======================\n";
  print "0. Edit preferences file\n";
  print "1. Edit reminders\n";
  print "2. Edit contacts\n";
  print "3. Edit tasks\n";
  print "4. Show tasks\n";
  print "5. Generate calendar (ps)\n";
  print "6. View calendar (txt)\n";
  print "7. Print calendar's";
  print "8. Show system cal\n";
  print "9. Show today's reminders\n";
  print "------------------\n";
  print "-1. Edit word list\n";
  print "-2. Edit notebook\n";
  print "-3. Edit quotes\n";
  print "-4. View quotes\n";
  print "======================\n";
  print "-> ";
  my $s = <STDIN>;
  chop($s);
  if ($s == 0) {
    system(getPrefVal("editor") . " " . $PREFS_FILE);
  }
  elsif ($s == 1) {
  system(getPrefVal("editor") . " ~/.reminders");
}
```

TPJ

Luis E. Muñoz

Parsing MIME & HTML

Understanding an e-mail message encoded with MIME can be very difficult. There are many options and many different ways to do the actual encoding. Add to that the sometimes too-liberal interpretations of the relevant RFCs by the e-mail client designers, and you will begin to get the idea. In this article, I will show you how this task can be laughably simple, thanks to Perl's extensive bag of tricks, CPAN.

I started out with a simple and straightforward mission: Fetch an e-mail from a POP mailbox and display it in a 7-bit, text-only capable device. This article describes the different stages for a simple tool that accomplishes this task, written in Perl with a lot of help from CPAN modules. I hope this is useful to other Perl folks who might have a similar mission.

Let's discuss each part of this task, in turn, as we read through mfetch, the script I prepared as an example. The script appears in its entirety in Listing 1.

Setting Up the Script

The first thing is to load all the necessary modules. You should be familiar with *strict* and *warnings*. We'll see how to use the rest of the modules a bit later.

```
 1   #!/usr/bin/perl
 2
 3   # This script is (c) 2002 Luis E. Muñoz, All
     # Rights Reserved
 4   # This code can be used under the same terms as
     # Perl itself. It comes with absolutely
 5   # NO WARRANTY. Use at your own risk.
 6
 7   use strict;
 8   use warnings;
 9   use IO::File;
10   use Net::POP3;
11   use NetAddr::IP;
12   use Getopt::Std;
13   use MIME::Parser;
14   use HTML::Parser;
15   use Unicode::Map8;
16   use MIME::WordDecoder;
17
18  use vars qw($opt_s $opt_u $opt_p $opt_m $wd $e $map);
19
20   getopts('s:u:p:m:');
21
22   usage_die("-s server is required\n") unless $opt_s;
23   usage_die("-u username is required\n") unless $opt_u;
24   usage_die("-p password is required\n") unless $opt_p;
25   usage_die("-m message is required\n") unless $opt_m;
26
27   $opt_s = NetAddr::IP->new($opt_s)
28   or die "Cannot make sense of given server\n";
```

Note lines 27 and 28, where I use *NetAddr::IP* to convert whatever the user gave us through the *-s* option into an IP address. This is a very common use of this module, as its *new()* method will convert many common IP notations into an object from which an IP address can later be extracted. It will even perform a name resolution if required. So far, everything should look familiar.

It is worth noting that the error handling in lines 22–25 is not a brilliant example of good coding or documentation. It is much better to write your script's documentation in POD, and to use a module such as *Pod::Usage* to provide useful error messages to the user. At the very least, try to provide an informative usage message. You can see the *usage_die()* function at the end of Listing 1.

Fetching a Message Via POP3

On to deeper waters. The first step in parsing a message is getting at the message itself. For this, I'll use *Net::POP3*, which implements the POP3 protocol described in RFC-1939. This is all done in Example 1.

At line 30, a connection to the POP server is attempted. This is a TCP connection, in this case to port 110. If this connection succeeds, the USER and PASS commands are issued at line 33, which are the simplest form of authentication supported by the POP protocol. Your username and password are being sent here through the network without the protection of cryptography, so a bit of caution is in order.

Net::POP3 supports many operations defined in the POP protocol that allow for more complex operations, such as fetching a list of messages, unseen messages, and the like. It can also fetch messages for us in a variety of ways. Since I want this script to be as lightweight as possible (i.e., to burn as little memory as possible), I want to fetch the message to a temporary on-disk file. The temporary file is nicely provided by the *new_tmpfile* method of *IO::File* in line 36, which returns a file handle to a deleted file.

Luis is an Open Source and Perl advocate at a nationwide ISP in Venezuela. He can be contacted at luismunoz@cpan.org.

I can work on this file, which will magically disappear when the script is finished.

Later, I instruct the *Net::POP3* object to fetch the required message from the mail server and write it to the supplied file handle using the *get* method on line 39. After this, the connection is terminated gracefully by invoking *quit* and destroying the object. Destroying the object ensures that the TCP connection with the server is terminated. This ensures that the resources being held in the POP server are freed as soon as possible, which is a good programming practice for network clients. Note that in line 45, I rewind the file so that the fetched message can be read back by subsequent code.

The interaction of mfetch with the POP server is very simple. *Net::POP3* provides a very complete implementation of the protocol, and allows for much more sophisticated applications—I'm only using a small fraction of its potential here.

For this particular example, we could also have used *Net::POP3Client*, which provides a somewhat similar interface. The code would have looked more or less like the following fragment:

```
my $pops = new Net::POP3Client(
           USER => $opt_u,
           PASSWORD => $opt_p,
           HOST => $opt_s->addr)
    or die "Error connecting or logging in: $!\n";

my $fh = new_tmpfile IO::File
    or die "Cannot create temporary file: $!\n";

$pops->HeadAndBodyToFile($fh, $opt_m)
    or die "Cannot fetch message: $!\n";

$pops->Close();
```

Parsing the MIME Structure

Just as e-mail travels inside a sort of envelope (the headers), complex messages that include attachments (and generally, HTML messages) travel within a collection of MIME entities. You can think of these entities as containers that can transfer any kind of binary information through the e-mail infrastructure, which in general does not know how to deal with 8-bit data. The code in Example 2 takes care of parsing this MIME structure.

Perl has a wonderful class that provides the ability to understand this MIME encapsulation, returning a nice hierarchy of objects that represent the message. You access this facility through the *MIME::Parser* class, which is part of the *MIME-Tools* bundle. *MIME::Parser* returns a hierarchy of *MIME::Entity* representing your message. The parser is smart; if you pass it a non-MIME e-mail, it will be returned to you as a text/plain entity.

MIME::Parser can be tweaked in many ways, as its documentation will tell you. One thing that can be tweaked is the decoding process. For my purposes, I needed to be as light on memory usage as possible. The default behavior of *MIME::Parser* involves the use of temporary files for decoding of the message. These temporary files can be spared and core memory used instead by invoking *output_to_core()*. Before doing this, note all the caveats cited in the module's documentation. The most important one is that if a 100-MB file ends up in your inbox, this whole thing needs to be slurped into RAM.

In line 47 (Example 2), I create the parser object. The call to *ignore_errors()* in line 48 is an attempt to make this parser as tolerant as possible. *extract_uuencode()* on line 49 takes care of pieces of the e-mail that are uu-encoded automatically, translating them back into a more readable form.

The actual request to parse the message, available through reading the *$fh* filehandle, is in line 51. Note that it is enclosed in an *eval* block. I have to do this as the parser might throw an exception if certain errors are encountered. The *eval* allows me to catch

```
30  my $pops = Net::POP3->new($opt_s->addr)
31      or die "Failed to connect to POP3 server: $!\n";
32
33  $pops->login($opt_u, $opt_p)
34      or die "Authentication failed\n";
35
36  my $fh = new_tmpfile IO::File
37      or die "Cannot create temporary file: $!\n";
38
39  $pops->get($opt_m, $fh)
40      or die "No such message $opt_m\n";
41
42  $pops->quit();
43  $pops = undef;
44
45  $fh->seek(0, SEEK_SET);
```

Example 1: Using Net::POP3 ***to read the message.***

```
47  my $mp = new MIME::Parser;
48  $mp->ignore_errors(1);
49  $mp->extract_uuencode(1);
50
51  eval { $e = $mp->parse($fh); };
52  my $error = ($@ || $mp->last_error);
53
54  if ($error)
55  {
56      $mp->filer->purge;      # Get rid of the temp files
57      die "Error parsing the message: $error\n";
58  }
```

Example 2: Parsing the MIME structure.

this exception and react in a way that is sensible. In this case, I want to be sure that any temporary file created by the parsing process is cleared by a call to *purge()*, as seen in lines 56 and 57.

Setting Up the HTML Parser

Parsing HTML can be a tricky and tedious task. Thankfully, Perl has a number of nice ways to help you do this job. Several excellent books, such as *The Perl Cookbook* (O'Reilly, 1998), had a couple of examples that came very close to what I needed, especially recipe 20.5, "Converting HTML to ASCII," which I reproduce below.

```
use HTML::TreeBuilder;
use HTML::FormatText;

$html = HTML::TreeBuilder->new();
$html->parse($document);

$formatter = HTML::FormatText->new(
               leftmargin => 0,
               rightmargin => 50);

$ascii = $formatter->format($html);
```

I did not want to use this recipe for two reasons: I needed fine-grained control in the HTML to ASCII conversion, and I wanted to have as little impact as possible on resources. I did a small benchmark (available electronically from www.tpj.com/) from that, which shows the performance difference between the two options while parsing a copy of one of my web articles. The following result shows that the custom parser runs faster than the *Cookbook*'s recipe. This does not mean that the recipe or the modules it uses are bad. This result simply means that the recipe is actually doing a lot of additional work, which just happens to not be all that useful for this particular task.

```
bash-2.05a$ ./mbench
Benchmark: timing 100 iterations of Cookbook's,
      Custom...
```

```
Cookbook's: 73 wallclock secs (52.82 usr +  0.00 sys
                 = 52.82 CPU) @  1.89/s (n=100)
     Custom:  1 wallclock secs ( 1.17 usr +  0.00 sys
                 =  1.17 CPU) @ 85.47/s (n=100)
                Rate Cookbook's     Custom
Cookbook's 1.89/s          --        -98%
Custom     85.5/s       4415%          --
```

HTML::FormatText does a great job of converting the HTML to plain text. Unfortunately, I have a set of guidelines that I need to follow in the conversion that are not compatible with the output of this module. Additionally, *HTML::TreeBuilder* does an excellent job of parsing an HTML document, but produces an intermediate structure—the parse tree—which, in my case, wastes resources.

However, Perl has an excellent HTML parser in the *HTML::Parser* module. In this case, I chose to use this class to implement an event-driven parser, where tokens (syntactic elements) in the source document cause the parser to call functions I provide. This allowed me complete control over the translation while sparing the intermediate data structure.

Converting HTML to text is a lossy transformation. This means that what comes out of this transformation is not exactly equivalent to what went in. Pictures, text layout, style, and a few other information elements are lost. My needs required that I noted the existence of images as well as a reasonably accurate rendition of the page's text, but nothing else. Remember that the target device can only display 7-bit text, and this is within a very small and limited display. This piece of code sets up the parser to do what I need:

```
62  my $parser = HTML::Parser->new
63       (
64        api_version => 3,
65        default_h => [ "" ],
66        start_h   => [ sub { print "[IMG ",
67              d($_[1]->{alt}) ||  $_[1]->{src},"]\n"
68                if $_[0] eq 'img';
69                             }, "tagname, attr" ],
70       text_h    => [ sub { print d(shift); }, "dtext" ],
71        ) or die "Cannot create HTML parser\n";
72
73  $parser->ignore_elements(qw(script style));
74 $parser->strict_comment(1);
```

Starting on line 71, I set up the *HTML::Parser* object that will help me do this. First, I tell it I want to use the latest (as of this writing) interface style, which provides more flexibility than earlier interfaces. On line 65, I tell the object that by default, no parse events should do anything. There are other ways to say this, but the one shown is the most efficient.

Lines 66 through 69 define a handler for the start events. This handler will be called each time an opening tag such as <A> or <IMG> is recognized in the source being parsed. Handlers are specified as a reference to an array whose first element tells the parser what to do and its second element tells the parser what information to pass to the code. In this example, I supply a function that for any IMG tag will output descriptive text composed with either the ALT or the SRC attributes. I request this handler to be called with the name of the tag as the first argument, and the list of attributes as further arguments, through the string "tagname, attr" found in line 69. The *d()* function will be explained a bit later—it has to do with decoding its argument.

The *text* event will be triggered by anything inside tags in the input text. I've set up a simpler handler for this event that merely prints out whatever is recognized. I also request that HTML entities such as *&euro;* or *&ntilde;* be decoded for me through the string "dtext" on line 70. HTML entities are used to represent special characters outside the traditional ASCII range. In the interest of document accuracy, you should always use entities instead of directly placing 8-bit characters in the text.

Some syntactic elements are used to enclose information that is not important for this application, such as <style>...</style> and <script>...</script>. I ask the parser to ignore those elements with the call to *ignore_elements()* at line 73. I also request the parser to follow strict comment syntax through the call to *strict_comment()* on line 74.

Converting HTML to text is a lossy transformation. This means that what comes out of this transformation is not exactly equivalent to what went in

Setting Up the Unicode Mappings

MIME defines various ways to encode binary data depending on the frequency of octets greater than 127. With relatively few high-bit octets, Quoted-Printable encoding is used. When many high-bit octets are present, Base-64 encoding is used instead. The reason is that Quoted-Printable is slightly more readable but very inefficient in space, while Base-64 is completely unreadable by standard humans and adds much less overhead in the size of encoded files. Often, message headers such as the sender's name are encoded using Quoted-Printable when they contain characters such as a "ñ". These headers look like "From: =?ISO-8859-1?Q?Luis_Mu=F1oz?= <some@body.org>" and should be converted to "From: Luis Muñoz <some@body.org>." In plain English, Quoted-Printable encoding is being used to make the extended ISO-8859-1 characters acceptable for any 7-bit transport, such as e-mail. Many contemporary mail transport agents can properly handle message bodies that contain high-bit octets but will choke on headers with binary data, in case you were wondering about all this fuss.

Lines 92 through 102 define *setup_decoder()*, which can use the headers contained in a *MIME::Head* object to set up a suitable decoder based on the *MIME::WordDecoder* class. This will translate instances of Quoted-Printable text to their high-bit equivalents. Note that I selected ISO-8859-1 as the default when no proper character set can be identified. This was a sensible choice for me, as ISO-8859-1 encloses Spanish characters, and Spanish happens to be my native language.

```
92    sub setup_decoder
93    {
94        my $head = shift;
95        if ($head->get('Content-Type')
96            and $head->get('Content-Type') =
                 ~ m!charset="([^\"]+)"!)
97        {
98            $wd = supported MIME::WordDecoder uc $1;
```

```
103 sub decode_entities
104 {
105     my $ent = shift;
106
107     if (my @parts = $ent->parts)
108     {
109         decode_entities($_) for @parts;
110     }
111     elsif (my $body = $ent->bodyhandle)
112     {
113         my $type = $ent->head->mime_type;
114
115         setup_decoder($ent->head);
116
117         if ($type eq 'text/plain')
118         { print d($body->as_string); }
119         elsif ($type eq 'text/html')
120         { $parser->parse($body->as_string); }
121         else
122         { print "[Unhandled part of type $type]"; }
123     }
124 }
```

Example 3: The decode_entities ***function.***

```
 99         }
100         $wd = supported MIME::WordDecoder
                  "ISO-8859-1" unless $wd;
101    }
```

But this clever decoding is not enough. Getting at the original high-bit characters is not enough. I must recode these high characters into something usable by the 7-bit display device. So in line 76, I set up a mapping based on *Unicode::Map8*. This module can convert 8-bit characters such as ISO-8859-1 or ASCII into wider characters (Unicode) and then back into our chosen representation, ASCII, which only defines 7-bit characters. This means that any character that cannot be properly represented will be lost, which is acceptable for our application.

```
76    $map = Unicode::Map8->new('ASCII')
77        or die "Cannot create character map\n";
```

The decoding and character mapping is then brought together at line 90, where I define the *d()* function, which simply invokes the appropriate MIME decoding method, transforms the resulting string into Unicode via the *to16()* method, and then transforms it back into ASCII using *to8()* to ensure printable results on our device. Since I am allergic to warnings related to undef values, I make sure that *decode()* always gets a defined string to work with.

```
90    sub d { $map->to8($map->to16($wd->decode
             (shift||''))); }
```

As you might notice if you try this code, the conversion is again lossy because there are characters that do not exist in ASCII. You can experiment with the *addpair()* method to *Unicode::Map8* in order to add custom character transformations (for instance, 'ñ' might be 'n'). Another way to achieve this is by deriving a class from *Unicode::Map8* and implementing the *unmapped_to8* method to supply your own interpretation of the missing characters. Take a look at the module's documentation for more information.

Starting the Decode Process

With all the pieces in place, all that's left is to traverse the hierarchy of entities that *MIME::Parser* provides after parsing a message. I implemented a very simple recursive function, *decode_entities*, shown in Example 3.

```
bash-2.05a$ ./mfetch -s pop.foo.bar -u myself \
            -p very_secure_password -m 2 | head -20
Date: Sun, 22 Dec 2002 23:22:25 -0400
From: Luis Muñoz <lem@foo.bar>
To: Myself <myself@foo.bar>
Subject: Fwd: Get $860 Free - Come, Play, Have Fun!



Begin forwarded message:

> From: Cosmic Offers <spam@spammer.com>
> Date: Sun Dec 22, 2002  20:59:43 America/Caracas
> To: spam@victim.net
> Subject: Get $860 Free - Come, Play, Have Fun!
>

>
[IMG http://www.spammer.com/email/Flc_600_550_liberty_mailer_.gif]
[IMG http://www.spammer.com/email/Flc_600_550_liberty_mail-02.gif]
[IMG http://www.spammer.com/email/Flc_600_550_liberty_mail-03.gif]
[IMG http://www.spammer.com/email/Flc_600_550_liberty_mail-04.gif]
```

Example 4: Decoding a more complex MIME message.

The condition at line 107 asks if this part or entity contains other parts. If it does, it extracts them and invokes itself recursively to process each subpart at line 109.

If this part is a leaf, its body is processed. Line 111 gets it as a *MIME::Body* object. On line 155, I set up a decoder for this part's encoding and based on the type of this part, which is determined at line 113; the code on lines 117 to 122 calls the proper handlers.

In order to fire off the decoding process, I call *decode_entities()* with the result of the MIME decoding of the message on line 86. This will invoke the HTML parser when needed and, in general, produce the output I look for in this example. After this processing is done, I make sure to wipe temporary files created by *MIME::Parser* on line 88. Note that if the message is not actually encoded with MIME, *MIME::Parser* will arrange for you to receive a single part of type text/plain that contains the whole message text, which is perfect for our application.

```
86  decode_entities($e);
87
88 $mp->filer->purge;
```

Conclusion

After these less than 130 lines of code, I can easily fetch and decode a message, such as in the following example:

```
bash-2.05a$ ./mfetch -s pop.foo.bar -u myself \
             -p very_secure_password -m 5
Date: Sat, 28 Dec 2002 20:14:37 -0400
From: root <root@foo.bar>
To: myself@foo.bar
Subject: This is the plain subject

This is a boring and plain message.
```

More complex MIME messages can also be decoded. Look at Example 4, where I dissect a dreaded piece of junk mail. I edited it to spare you pages and pages of worthless image links.

That about covers the operation of the mfetch script. I hope you find it useful if you have a similar MIME or HTML decoding task to accomplish.

TPJ

Listing 1

```
#!/usr/bin/perl

# This script is (c) 2002 Luis E. MuÒoz, All Rights Reserved
# This code can be used under the same terms as Perl itself. It comes
# with absolutely NO WARRANTY. Use at your own risk.

use strict;
use warnings;
use IO::File;
use Net::POP3;
use NetAddr::IP;
use Getopt::Std;
use MIME::Parser;
use HTML::Parser;
use Unicode::Map8;
use MIME::WordDecoder;

use vars qw($opt_s $opt_u $opt_p $opt_m $wd $e $map);

getopts('s:u:p:m:');

usage_die("-s server is required\n") unless $opt_s;
usage_die("-u username is required\n") unless $opt_u;
usage_die("-p password is required\n") unless $opt_p;
usage_die("-m message is required\n") unless $opt_m;

$opt_s = NetAddr::IP->new($opt_s)
    or die "Cannot make sense of given server\n";

my $pops = Net::POP3->new($opt_s->addr)
    or die "Failed to connect to POP3 server: $!\n";

$pops->login($opt_u, $opt_p)
    or die "Authentication failed\n";

my $fh = new_tmpfile IO::File
    or die "Cannot create temporary file: $!\n";

$pops->get($opt_m, $fh)
    or die "No such message $opt_m\n";

$pops->quit();
$pops = undef;

$fh->seek(0, SEEK_SET);

my $mp = new MIME::Parser;
$mp->ignore_errors(1);
$mp->extract_uuencode(1);

eval { $e = $mp->parse($fh); };
my $error = ($@ || $mp->last_error);

if ($error)
{
    $mp->filer->purge;      # Get rid of the temp files
    die "Error parsing the message: $error\n";
}

                # Setup the HTML parser

my $parser = HTML::Parser->new
    (
     api_version => 3,
     default_h      => [ "" ],
     start_h => [ sub { print "[IMG ",
            d($_[1]->{alt}) || $_[1]->{src},"]\n"
                if $_[0] eq 'img';
            }, "tagname, attr" ],
     text_h => [ sub { print d(shift); }, "dtext" ],
     ) or die "Cannot create HTML parser\n";

$parser->ignore_elements(qw(script style));
$parser->strict_comment(1);

$map = Unicode::Map8->new('ASCII')
    or die "Cannot create character map\n";

setup_decoder($e->head);

print "Date: ", $e->head->get('date');
print "From: ", d($e->head->get('from'));
print "To: ", d($e->head->get('to'));
print "Subject: ", d($e->head->get('subject')), "\n";

decode_entities($e);

$mp->filer->purge;

sub d { $map->to8($map->to16($wd->decode(shift||''))); }

sub setup_decoder
{
    my $head = shift;
    if ($head->get('Content-Type')
    and $head->get('Content-Type') =~ m!charset="([^\"]+)"!)
    {
    $wd = supported MIME::WordDecoder uc $1;
    }
    $wd = supported MIME::WordDecoder "ISO-8859-1" unless $wd;
}

sub decode_entities
{
    my $ent = shift;

    if (my @parts = $ent->parts)
    {
    decode_entities($_) for @parts;
    }
    elsif (my $body = $ent->bodyhandle)
    {
    my $type = $ent->head->mime_type;

    setup_decoder($ent->head);

    if ($type eq 'text/plain')
    { print d($body->as_string); }
    elsif ($type eq 'text/html')
    { $parser->parse($body->as_string); }
    else
    { print "[Unhandled part of type $type]"; }
    }
}

sub usage_die
{
    my $msg = <<EOF;
Usage: mfetch -s pop-server -u pop-user -p pop-password -m msgnum
EOF
    ;
    $msg .= shift;
    die $msg, "\n";
}
```

TPJ

Moshe Bar

Home Automation with Perl

This is the story of how I automated my new home with Perl. You see, I am a bit of a geek and I love for computers to do things for me and to control tedious tasks. I recently moved into a new home, and instead of spending money on specialized appliances for security and control, I decided to go all wireless and use X10 (see http://www.x10.com/products/x10_ck11a.htm) for controlling analog devices like lamps, the fridge, the garage door, the entrance, shutters, phone messages, and the thermostat. Being of the Perl persuasion when it comes to automation and scripting, I naturally looked for ways to do it in Perl. Fortunately, there is a way to control the CM17 (a small RF remote that plugs into the serial port of your computer) through the SerialPort module and the X10 transceiver (a thing that plugs into the wall and receives RF signals from the CM17) through the X10 Perl module. Both modules are available from CPAN. More information about the X10 module can be found at http://members.aol.com/Bbirthisel/x10.d.

With the SerialPort module, you can turn a lamp on and off like this:

```
#!/usr/bin/perl -w
use strict;
use Device::SerialPort;
use ControlX10::CM17;
my $sp = Device::SerialPort->new('/dev/ttyS0');
if ($ARGV[0] =~ /^on$/i) {
    ControlX10::CM17::send($sp,"G1J");
}
elsif ($ARGV[0] =~ /^off$/i) {
    ControlX10::CM17::send($sp,"G1K");
}
else {
    print "You can only turn the lamp 'on' or 'off'\n";
}
```

The above code works properly on a Linux box, as the /dev/ttyS0 type naming of the serial port suggests.

One could write a series of scripts for each and every device in the house and control them through cron jobs. That, however, would be tedious and would create a management nightmare once you hooked up a significant number of devices. Instead, a general scheduler for home automation coupled with a generalized frontend for input and output using the X10 remote and the Festival text-to-speech system for speech seems a much better strategy. (For more information on Festival, see http://www.cstr.ed.ac.uk/projects/festival/.)

Before sitting down to invent the wheel, it is always a good idea to make sure that it hasn't already been invented by somebody else. A quick search of the Web revealed that somebody had actually done exactly what I needed to do for his own home and had put the source to his Perl software in the public domain. It's called "MisterHouse," and it is hosted on SourceForge at http://misterhouse.sourceforge.net/. MisterHouse knows to fire events based on time, web access, socket, voice, and serial data.

Perl subroutines and objects are used to give a powerful programming interface. Here is some example code:

```
$fountain = new X10_Item 'B1';
set $fountain ON if time_now '6:00 PM';

$movement_sensor = new Serial_Item 'XA2', 'stair';
play(file => 'stairs_creek*.wav') if state_now
     $movement_sensor eq 'stair';

$v_bedroom_curtain = new Voice_Cmd '[open,close] the
     bedroom curtains';
curtain('bedroom', $state) if $state = said v_bed
     room_curtain;
```

In this example, we turn on the water fountain at 6:00PM and play a creaky sound if there is movement detected at a particular location. Finally, the bedroom curtains can be closed or opened depending on spoken commands via a voice-recognition system such as IBM's ViaVoice for Linux. MisterHouse can use Festival to talk back to the user.

MisterHouse was exactly what I was looking for as a foundation for my own home-automation system. I installed a house sound system with loudspeakers in most rooms of the house and had the wiring for it done by a professional installation company. I also had them install little X10 remote-control receivers in strategic locations around the house and, finally, put the whole environment on a sturdy IBM NetFinity server in the basement.

Moshe is a systems administrator and operating-system researcher and has a M.Sc and a Ph.D. in computer science. He can be contacted at moshe@moelabs.com.

Through Perl scripts responding to input from the remote control, I hooked up all the doors and the main lights both inside and outside the house; I then hooked up flat-panel touch screens to some cheap Apple iMacs I bought on eBay. The Apple iMacs make no noise at all, don't look like computers, and have more than ample resources to run the web-based home control interface in the living room and kitchen.

The web interface is necessary for operations that are difficult to do with a remote control, such as choosing an MP3 track to play or entering a text string.

It is extremely easy to integrate Perl with Festival. A simple print to the standard input of Festival with the –tts switch will make Festival read the payload of stdin over the soundcard. Next to Festival's ability to act as a speech server of standard socket connections, there is also a Festival module for Perl available at CPAN. The module can be used for blocking and nonblocking speech mode and also knows to return the waveform of the text to be spoken (see http://www.cpan.org/modules/by-module/Festival/Festival-Client-Async-0.0301.readme).

We use our MisterHouse extensions at home to have certain incoming e-mails read to us aloud over the house sound system if we are at home. Festival, however, doesn't know how to read certain characters; for instance, the @ in an e-mail address. For this, a little trickery is needed; see Listing 1.

Up to this stage, our home automation system could select and listen to MP3s, turn on and off most home appliances, listen to incoming e-mails, alert us when Portsentry (a port-scanning alerter for UNIX systems, including Linux) noticed suspicious activity on our DSL line, and control access to and from our home to the outside world.

The next stage involved making the system automate the house according to real-time events such as weather, time of day, and visitors. One example of such an event is when we drive away and forget to close the garage door. With MisterHouse, you can close the garage door automatically with a script like this:

```
#————————————————————————————
# Example of a door monitor
#————————————————————————————
$timer_garage_door = new  Timer();
$garage_door         = new  Serial_Item('DCCH',
                                          'opened');
$garage_door        -> add              ('DCCL',
                                          'closed'');
if($state = state_now $garage_door) {
    set $timer_garage_door 120;
    play('rooms' => 'all', 'file' => "garage_door_" .
         $state . "*.wav");
}

if ((time_cron('0,5,10,15,30,45 22,23 * * *') and
    ('opened' eq ($garage_door->{state})) and
    inactive $timer_garage_door)) {
    speak("The garage door has been left opened.  I
          am now closing it.");
    set $garage_door_button ON;
    set $garage_door_button OFF;
}
```

Quite obviously, this script could also be used to automate the opening and closing of curtains, to turn auxiliary computers on and off, and to have coffee ready for you after you get out of the shower in the morning by monitoring the shower's heater for activity (indicating someone is having a shower).

I don't usually need an alarm clock to wake me up in the morning, but when I have to catch an early flight, I like to be sure to be up on time. A few lines of Perl turn the television on to a music channel and switch the bedroom lights on:

```
   set $left_bedroom_light ON;
        set $right_bedroom_light ON;

        runit("min", "ir_cmd TV,POWER,5,1");
```

Just as I was finishing this column, I noticed a very nice article by brian d foy at The O'Reilly Network (see http://www.oreillynet.com/). The article outlines how to remotely control iTunes, the excellent Mac OS X music player, through a mixture of Perl and OS X AppleScript.

brian d foy is the creator of the Mac::iTunes Perl module, which allows iTunes to be controlled from any computer in the network. Since iTunes itself, like many OS X applications, is an AppleScript-aware application, you can have Perl programs launch AppleScripts.

AppleScript can be controlled either interactively or in batch mode through the CLI tool osascript with the Mac::iTunes::AppleScript module, which wraps common AppleScripts in Perl functions. At the core of that module lies the *_osascript* routine, which creates an AppleScript string and then calls the OS X CLI tool osascript. It is very easy with the Mac::iTunes module to play, for example, a random MP3 track off the database; see Listing 2.

Apple actually provides a whole tarball of AppleScripts for iTunes, available for download from its web site. Some examples of the scripts provided are:

- Rewind Tracks
- Replace Text in Track Names
- Enable/Disable Selected Tracks
- eMusic Search by Song Title
- eMusic Search by Artist
- eMusic Search by Album
- Make Playlist by Artist
- Create Library Summary
- Remove Playlists from Source
- Remove Missing Tracks
- Mute On/Off

There are many more ready-made scripts from the Apple web site and almost unlimited possibilities exist to create others using the simple tools I describe here. In my home, I made use of these scripts to automate my MP3 jukebox, which is based on a Mac Cube connected to the stereo for good sound quality.

TPJ

Listing 1

```
# Modify this rule for use with get_email
# Rename to get_email_rule.pl to enable
#  - $from_full has the full email address, not just the name portion.

sub get_email_rule {
    my ($from, $to, $subject, $from_full) = @_;
    $from = 'The S F gals'            if $to =~ /FEM-SF/;
    $from = 'The E C S guys'          if $to =~ /ecs/;
    $from = 'The Mister House guys'   if $to =~ /misterhouse/;
    $from = 'The perl guys'           if $to =~ /Perl-Win32-Users/;
   $from = 'The phone guys'           if $to =~ /ktx/ or $subject =~ /kx-t/i;
    $from = 'junk mail'               if $from =~ /\S+[0-9]{3,}/;
# If we get a joe#### type address, assume it is junk mail.
    return                           if $from =~ /X10 Newsletter/;

   $from =~ s/\./ Dot /g ;     # ...change "." to the word "Dot"
   $from =~ s/\@/ At /g ;      # ...change \@  to the word "At"

    return $from;
}

return 1;
```

Listing 2

```
sub _osascript
        {
        my $script = "tell source 'Library'
                     tell playlist 'Library'
                                set this_track to some track
                                set this_name to the name of this_track
                                set this_artist to the artist of this_track
                                set this_album to the album of this_track
                                play this_track
                     end tell
           end tell";

        require IPC::Open2;

        my( $read, $write );
        my $pid = IPC::Open2::open2( $read, $write, 'osascript' );

        print $write qq(tell application "iTunes"\n), $script,
                        qq(\nend tell\n);
        close $write;

        my $data = do { local $/; <$read> };

        return $data;
        }
```

TPJ

Amazon.com Wish Lists

brian d foy

Over the holiday, I wanted to check my Amazon.com wish list, which I use to keep track of the books that I would like to read, even if I do not intend to buy them. With Amazon.com web services, I can easily download my wish list in XML format, and with *XML::Simple*, I can easily parse and access the information. The *Text::Template* module gives me a flexible way to display the information once I get it.

Recently, Amazon.com opened its data, which it calls "Properties," to the public through their "Amazon.com Web Services" (http://soap.amazon.com/). Anyone can use these services by signing up for the program and getting a "Developer's Token" that allows them to access the web service. Once you have a token, you can access book, DVD, and author information as well as the results of the many sorts of searches available on the Amazon.com web site. I started using the web service for some publishing clients who wanted to check sales rank and price data for their books, and now I am finding it useful for my personal information as well.

A Web Service, despite its general name, typically applies to something on a web server that returns XML data. Sometimes this XML result is a Simple Object Access Protocol (SOAP) message, but it can also be an XML/HTTP response. Amazon.com web services allows me to use either. For my wish list, I use the XML/HTTP method, which only requires a URL with the right parameters. This is much easier to use than SOAP for simple tasks.

Before I can start, I need to find my wish list ID string, which Amazon.com hides in a lot of information in their URLs, including session and user identifiers. If I look at my wish list page, I see a link to "Share Your Wishlist," which has as part of its link "/wishlist/1VGWQEYUDRN9V/." The string after "wishlist" (1VGWQEYUDRN9V) is my wish list ID.

Once I have my Developer's Token and wish list ID, I can download my wish list information. If I use XML/HTTP, I need to create the URL. The base of the request URL is <URL:http://xml.amazon.com/onca/xml2>. After the base, I form a query string with the information that I have collected; see Table 1.

I create the URL with the URL module, using my Amazon.com Associates ID (theperlreview-20); my Developer's Token, which I have in my shell's environment variable AMAZON_DEV_T; the "lite" version of the output; and my wish list ID (1VGWQEYUDRN9V). Amazon.com returns up to ten items in the wish list for each request. The first ten are page one, the next ten are page two, and so on. In this request, I set page to one to get the first ten results. To get all the results, I need to make multiple requests, increasing the page number by one each time, until Amazon.com returns no more results.

```
my $url = do {
    require URI;
    my %values = (
        t               => 'theperlreview-20',
        'dev-t'         => $ENV{AMAZON_DEV_T},
        type            => 'lite',
        f               => 'xml',
        page            => 1,
        WishlistSearch  => '1VGWQEYUDRN9V'
        );

    my $u = URI->new( 'http://xml.amazon.com/onca/xml2' );
    $u->query_form( %values );
    $u->as_string;
    };
```

I fetch this URL with *LWP::Simple*, and store the result in the scalar *$xml*.

```
use LWP::Simple qw(get);
my $xml = get( $url );
```

The result in *$xml*, for the "lite" output, is a *ProductInfo* node containing several *Details* nodes; see Example 1. In this example output, I have replaced long URLs with "…" to show the structure more clearly. The "heavy" output includes much more detailed information on the product, including category names, similar items, and sales rank.

I can parse this any way that I like, including using *XML::Parser* or simply using built-in pattern matching and text-manipulation functions. I use *XML::Simple* to get a hash, which makes the next step easier.

```
use XML::Simple;
my $hash = XMLin( $xml );
```

t	My Amazon.com Associates ID
dev-t	My developer's token
type	The verbosity of output (heavy or lite)
f	"xml" for XML output
WishlistSearch	The wish list ID

Table 1: Wish list data.

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.

The anonymous hash in *$hash* has a *Details* key, which is an array of hashes because I have more than one item in my wish list. (If I have only one item in my wish list, the value of the *Details* key will be just the hash for the product information.)

```
{
'Details' => [
     {
     'OurPrice' => '$10.47',
     'ImageUrlLarge' => '.../2290319740.01.LZZZZZZZ.jpg',
     'ReleaseDate' => 'January, 2003',
     'ImageUrlMedium' => '.../2290319740.01.MZZZZZZZ.jpg',
     'Catalog'' => 'Book',
     'Asin' => '2290319740',
     'url' => '...',
     'Manufacturer' => 'J Ai Lu Editions',
     'ListPrice' => '$14.95',
     'ProductName' => 'Je Parler Francais',
     'ImageUrlSmall' => '.../2290319740.01.THUMBZZZ.jpg',
     'Authors' => {
          'Author' => 'David Sedaris'
          }
     },
     ]
}
```

The *Text::Template* module gives me a basic framework to hand off data to a presentation layer so I can change the way that I show the information without changing the logic code. I give *Text::Template*'s *fill_in_file()* function the filename of the template that I want to use, and pass the data as the value for the HASH key.

```
use Text::Template qw(fill_in_file);

my $Template = 'wish_list-template.txt';

print fill_in_file( $Template, HASH => $wish_list );
```

Inside the template, *Text::Template* turns *$wish_list* into variables. The *Details* key turns into the array *@Details* because it has an anonymous array for a value. Each element of *@Details* is an anonymous hash with keys that are the names of the XML tags. I use a very simple template to print the titles of everything in my wish list. *Text::Template* evaluates the parts of the file between the braces {} as Perl code. Everything outside of the braces is literal text.

```
My Amazon.com wish list:

{
my $string = '';
foreach my $product ( @Details )
     {
     $string .= "Title: $product->{ProductName}\n";
     }

$string;
}
```

The template reformats the wish list information, which I print to the screen. In this example, I use plain text, but I can output any format, including HTML.

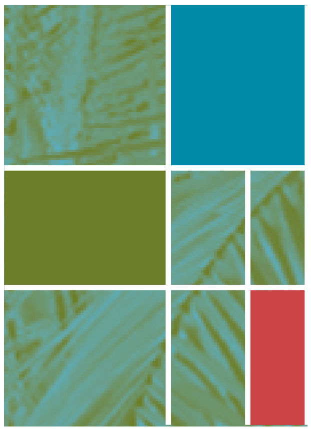
```
My Amazon.com wish list:

Je Parler Francais
The Rise of the Meritocracy
1421: The Year China Discovered America
The Declaration of Independence and Other Great
      Documents of American History, 1775-1865 (Dover
      Thrift Editions)
Common Sense (Dover Thrift Editions)
Democracy in America
The Path to Victory: America's Army and the
      Revolution in Human Affairs
Boyd: The Fighter Pilot Who Changed the Art of War
I May Be Wrong but I Doubt It
My Losing Season
```

That's it—a couple of Perl modules and a little help from Amazon.com, and I can download my wish list data. With a little work, so can you.

TPJ

```
<ProductInfo xmlns:xsi="...">
   <Details url="...">
      <Asin>2290319740</Asin>
      <ProductName>Je Parler Francais</ProductName>
      <Catalog>Book</Catalog>
      <Authors>
         <Author>David Sedaris</Author>
      </Authors>
      <ReleaseDate>January, 2003</ReleaseDate>
      <Manufacturer>J Ai Lu Editions</Manufacturer>
      <ImageUrlSmall>.../2290319740.01.THUMBZZZ.jpg</ImageUrlSmall>
      <ImageUrlMedium>.../2290319740.01.MZZZZZZZ.jpg</ImageUrlMedium>
      <ImageUrlLarge>.../2290319740.01.LZZZZZZZ.jpg</ImageUrlLarge>
      <ListPrice>$14.95</ListPrice>
      <OurPrice>$10.47</OurPrice>
   </Details>
</ProductInfo>
```

***Example 1:** XML result from fetching an Amazon.com wish list URL.*



Other People's Arguments

Simon Cozens

I'm going to let you into a secret about writing technical articles. The trick that I often use to plan an article is to think of a particular technique I want to illustrate, then find a practical use for it and, finally, find a problem that the practical application solves. Here comes the trick: You then present it all back-to-front. That way, you've got an article that looks like it's showing you how to solve a particular problem, and the technique you want to talk about pops up magically as the answer to all your problems in the end.

In this article, for instance, the technique that I want to talk about is the little-known *@DB::args* magic variable; the application is my recent *rubyisms* Perl module; the problem, if you want to call it that, is my recent dabbling with Ruby.

As you can probably tell from last month's column, I've grown fond of some of the features I've been using in Ruby. This is a common problem—or so I'm told—with love affairs: If and when you come back to your first love, you can't help but want some of the things you've left behind. Thankfully, however, programming languages are a good deal easier to change than people. So the more frustrated I got with the things from Ruby that I thought Perl lacked, the more I wanted to sit down and fix them up.

The first thing I found myself missing was the *super* keyword. It came up in Perl as I was specializing a *Class::DBI*-based module. I had a class representing a database table, which I could search by its two columns, *real_name* and *displayed_name*. But I also wanted a "magical" search term name that searched through both columns. So, in my subclass, I would say:

```
sub search {
    my ($self, $terms) = @_;
    if (exists $terms->{name}) {
        # Do our special search
    } else {
        # Call the superclass's search.
    }
}
```

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

Now, calling the *superclass* method is easy. We all know how to do that. Here's how you'd naturally write it in Perl:

```
$self->SUPER::search($terms);
```

However, this is what it looks like in Ruby:

```
super
```

You can probably understand why I felt spoiled by Ruby. No problem, I thought, I can find a way to do this in Perl. So I thought about what the *super* subroutine had to do:

- It should have a prototype of *()* so it could be called like a built-in.
- It should use *caller* to get at the method that called it.
- It should have to somehow get at the object and the arguments to the method, so it could work out the *superclass* method and call it with the same arguments.

Well, the first two were pretty easy:

```
sub super () {
    my $caller= (caller(1))[3];
    $caller =~ s/.*:://;

}
```

But the third had me completely confused. How on earth could I retrieve my caller's subroutine arguments? Well, the obvious place to start looking was the documentation for *caller*; and there I found something I had never noticed before:

> "Furthermore, when called from within the DB package, *caller* returns more detailed information: it sets the list variable *@DB::args* to be the arguments with which the subroutine was invoked."

Wow, just perfect. So I wrote up a little subroutine to call *caller* from package *DB*:

```
package DB;
sub uplevel_args { my @x = caller(2); return @DB::args }
```

This looks two frames up the stack (*DB::uplevel_args* is the zeroth frame, *SUPER::super* is the first, and the method that called *super* is the second), and returns the arguments from the method. The array is needed to stop Perl from optimizing the call to *caller*.

So now we know how the method was called, which tells us the object.

```
sub super () {
    my $caller= (caller(1))[3];
    $caller =~ s/.*:://;

    @_ = DB::uplevel_args();
    my $self = $_[0];
}
```

Unfortunately, it gets tricky again here: We'd like to say *$self->SUPER::$caller*, but that gives us a "*Bad name after ::*" error. And we want to avoid using *eval*, if possible. What we need is to somehow get hold of a reference to the appropriate *superclass* method. Since we already know the class and the method name, we are half way there. Let's assume a putative *UNIVERSAL::super* subroutine that works like *UNIVERSAL::can* and returns a reference to the method if one is available. Then we can say:

```
sub super () {
    my $caller= (caller(1))[3];
    $caller =~ s/.*:://;

    my @their_args = DB::uplevel_args();
    my $self = $their_args[0];
    $self->UNIVERSAL::super($caller)->(@their_args);
}
```

Now this is pretty clever, but it has a slight untidyness problem. Suppose we have a class *Wibble::Simple* that inherits from class *Wibble*. With our current use of *super*, we'd see a call stack like this:

```
Wibble::Simple::do_it
   SUPER::super
      Wibble::do_it
```

Whereas we'd prefer to see this:

```
Wibble::Simple::do_it
   Wibble::do_it
```

Now, there is a way to make *SUPER::super* morph itself into the appropriate method: the *goto &subroutine* technique. Since *UNIVERSAL::super*—when it's written—returns a subroutine reference, we merely need to set the @_ to be what we want the superclass to see, and then *goto* the reference:

```
sub super () {
    my $caller= (caller(1))[3];
    $caller =~ s/.*:://;

    @_ = DB::uplevel_args();
    my $self = $_[0];
    my $supermethod = $self->UNIVERSAL::super($caller);
    goto &$supermethod;
}
```

Right, we're done! Well, apart from the little matter of that *UNIVERSAL::super* method, that is. But this isn't too much of a problem—all we need to do to work out an object's *super* method is to think of what Perl would do. And what Perl does is ask each member of that object's class's *@ISA* array if it can perform the method. This can be done with the *can* method, which returns a code reference if the class can perform the given method—precisely what we want!

```
package UNIVERSAL;

sub super {
    my ($class, $method) = @_;

    if (ref $class) { $class = ref $class; }
    my $x;
    for (@{$class."::ISA"}, "UNIVERSAL") {
        return $x if $x = $_->can($method);
    }
}
```

And with this—and a little testing and documentation—the *SUPER* module was born and released onto CPAN. Now I could write things like the following:

```
sub search {
    my ($self, $terms) = @_;
    if (exists $terms->{name}) {
        # Do our special search
    } else {
        super;
    }
}
```

This made me happy.

But then, a few more lines of code later, there was another problem. One of the nice things about Ruby's OO model as opposed to Python's and Perl's is that the recipient of a method is implicit.

It's not necessary to say:

```
my $self = shift;
```

to get it from the argument list. It's just there, and it's possible to get at it with the *self* keyword.

Furthermore, you can call one method from another just by naming it, and the *self* is again passed around implicitly.

Here's a bit of Ruby to demonstrate this:

```
class Thing

  def look
      _print
  end

  def _print
      puts self
  end

end

foo = Thing.new
foo.look
```

We create a new object and call its *look* method. This then calls another method, *_print*, implicitly passing the object around. *_print* receives the object, once again implicitly, before finally referencing it as *self*.

Of course, this can't be done in Perl—we can't change the fact that method calls do pass around the receiver and that we need to pass the receiver to a submethod. And we can't rewrite @_ (once we know the receiver) to pretend it was never there in the first place. But we can fake it.

Using the same *@DB::args* trick, we can create a subroutine that returns the first argument of its caller:

```
sub self () {
    return (DB::uplevel_args())[0];
}
```

This means we can say things like:

```
sub look   { self->_print }
sub _print { print self, "\n" }
```

Not a bad start. But we'd really like to be able to say:

```
sub look   { _print }
sub _print { print self, "\n" }
```

That's right: Even though we call *print* with no arguments, it should still know what the current receiver is.

Once again, being able to mess about with other subroutines' arguments comes to our aid. The key to this is realizing that we don't have to look just two levels up the stack—we can look farther if we want to. And as we look up the call stack, we'll eventually come to a frame that is called "properly," with the appropriate type of object as its first argument.

So, we first modify *DB::uplevel_args* so that we can look up a variable number of frames:

```
sub uplevel_args { my @x = caller($_[0]+1); return @DB::args };
```

We now look up the stack until we find a subroutine whose first parameter *is-a* whatever class we were called by:

```
sub self () {
    my $call_pack = (caller())[0];

    my $level =1;
    while (caller($level)) {
        my @their_args = DB::uplevel_args($level);
        if (ref $their_args[0]
           and eval { $their_args[0]->isa($call_pack) }) {
            return $their_args[0];
        }
        $level++;
    }

    return $call_pack;
}
```

We're only interested in objects that are inherited from the caller, because if we have

```
package Thing;
sub look_to_file { my $output = new IO::File (...);
                   _print($output)
                 }
```

we want the recipient of *_print* to be the *Thing*, not the *IO::File*. So in this case, we want to ignore the argument to *_print* but look back at the arguments of *look_to_file*.

Notice also that if we don't find any object of the relevant class at any time in the recent past, we assume that we're dealing with a class method, and that the *self* is the name of the calling package; this is a reasonable approach and is pretty much what Ruby does:

```
% ruby -e 'print self;'
main
```

```
% perl -Mrubyisms -e 'print self;'
main
```

So now we can use an implicit *self*, and pass it around between methods of the same class. Very neat, no?

But I've glossed over another little detail of my Ruby example: Our class, *Thing*, had a constructor, but we didn't define a *new* method. This is because all classes in Ruby inherit from class *Class*, which provides a generally-good-enough default constructor and then calls the *initialize* method to allow us to specify the object. This is a neat idea, so I wanted to steal that too. First, we need to make everything that imports the *rubyism* method inherit from class *Class*:

```
sub import {
    no strict 'refs';
    push @{(caller())[0]."::ISA"}, "Class";
    rubyisms->export_to_level(1, @_);
}
```

We find the calling package's package name, and slap *Class* onto the end of its *@ISA* array. Then we can use a little *Exporter* trick that deserves to be better known: Call *Exporter*'s *import* to make *super* and *self* available to calling packages in addition to doing our own *import*ish things. You might think that after all we have seen in this article, we could so far just say:

```
sub import {
    no strict 'refs';
    push @{(caller())[0]."::ISA"}, "Class";
    super;
}
```

to jump to our superclass. Unfortunately, that doesn't quite work. This is because *Exporter*'s *import* method moves symbols from *Exporter* to the class calling the method—in this case, *rubyisms*. This isn't what we want—we want to move symbols from *rubyisms* to whatever *use*d it. So we call the *import_to_level* method, which moves symbols around at a different calling level. This does the right thing.

Now we can populate the *Class* class with the methods we want. A generally-good-enough constructor in Perl blesses an empty hash and calls a specializer before returning the new object:

```
package Class;
sub new {
    my $class = shift;
    my $self = bless {}, $class;
    $self->initialize(@_);
    return $self;
}
```

(Note that we can't use *self* here to get the recipient because in the constructor, there isn't a recipient yet!)

We provide a dummy specializer for completeness:

```
sub initialize {}
```

We can now rewrite our Ruby example in Rubyish Perl:

```
package Thing;
use rubyisms;

sub _print { print self }
sub look   { _print }
my $foo = Thing->new;
$foo->look;
```

And I was happy again—until I found another feature from Ruby I wanted to steal…

Now that we have this technique of inspecting the caller's arguments, it is very simple to write our own keywords such as *super* and *self* that depend on the properties of a subroutine. Another subroutine-specific keyword in Ruby is *yield*.

We can bend Perl in some interesting and extraordinary directions without mucking about with XS, the Perl internals, or any other difficulty

As we saw last month, every method in Ruby can take an optional block, and the *yield* keyword calls back that block. In Perl, we don't have the same optional block syntax, but we do have something similar: If subroutines are given a prototype starting with the *&* character, they will behave somewhat like *map* or *grep*.

For instance, here's a truly simple array iterator. You might want to call it a "Visitor design pattern" if you're a Gang-of-Four devotee; if you're a Perl devotee, you might call it a highly redundant *for* loop. It simply visits each element of the array, calling a codeblock on the element:

```
sub each_arr (&@) {
    my ($code, @array) = @_;
    for (@array) {
         $code->($_);
    }
}
```

With the syntax-modifying *&* prototype, this can be called as follows:

```
each_arr { print $_[0], "\n" } (10, 20, 30);
```

But we'd prefer to write this in a more Rubyish way, like so:

```
sub each_arr {
    for (@_) { yield }
}
```

The Perl way is slightly different—instead of a block at the end, we expect a block at the beginning. So, once again, we look at our caller's arguments and ensure that the first argument is a code reference. If it isnt, we give a nice Ruby-friendly error:

```
sub yield {
    my @their_args = DB::uplevel_args(1);
    if ((!@their_args) or ref $their_args[0] ne "CODE") {
        croak "no block given (LocalJumpError)";
    }
```

And now we have the code reference; we can call it on *$_*:

```
    $their_args[0]->($_);
}
```

This is pretty good, but Ruby's *yield* doesn't just yield the default value. In fact, Ruby doesn't really have a "default value" equivalent to *$_*. *yield* can take arguments, and those arguments should be passed to the code reference. But this being Perl, we want to support both styles: implicit *$_* and explicit arguments. So our *yield* subroutine ends up looking like this:

```
sub yield (@) {
    my @their_args = DB::uplevel_args(1);
    if ((!@their_args) or ref $their_args[0] ne "CODE") {
        croak "no block given (LocalJumpError)";
    }
    my @stuff = (@_||$_);
    $their_args[0]->(@stuff);
}
```

But there is a slight problem. If we try out our shiny new *yield* with the *each_args* example, we might see something like this:

```
CODE(0x10774)
10
20
30
```

I must stress again that we're only faking it. We can't rewrite *each_arr*'s @_ array so that the codeblock is squirrelled away for *yield* and doesn't appear when we call *for*. The code reference is going to stay part of @_ whether you like it or not. So *yield* needs to be a bit tricky.

The obvious way to solve this problem in the majority of cases is to simply refuse to call the code reference on itself:

```
$their_args[0]->(@stuff)
    unless $stuff[0] == $their_args[0];
```

And that is By And Large Good Enough. The iterators now work the way we expect them to.

So that's all I've wanted from Ruby so far, and the whole module, *rubyisms.pm*, is available from CPAN. I'm sure that there'll be more features added as I keep dragging things across from Ruby.

But we've seen that with the knowledge gained from a relatively simple but relatively obscure technique—the interaction between *caller* and *@we* can bend Perl in some interesting and extraordinary directions without mucking about with XS, the Perl internals, or any other difficulty.

Sometimes, it seems, getting involved in other people's arguments isn't such a bad thing after all.

TPJ

Practical Python

Jack J. Woehr

Practical Python
Magnus Lie Hetland
Apress, 2002
619 pp., $49.95
ISBN 1-59059-006-6

Practical Python is a complete course in Python programming, from the elements of the language to some rather advanced example projects. It's a fine work, from the writing to the pacing to the layout and other production values, including technical review. It's undoubtedly one of the fastest and most painless ways to get Python under your belt.

You're probably familiar with Python, but in case you're not: Python is an objectified scripting language similar to Perl and occupying a similar niche, but possessing a more rigid and streamlined structure than Perl. Python's popularity is growing among those who find dealing with the immensity of Perl and its praxis a bit like trying to snow-swim your way out of an avalanche.

Author Magnus Hetland is the founder of the AnyGui project, which allows Pythoners to slap a GUI onto their code. He is a powerful Python programmer and a good teacher. His example projects are more interesting and useful than such projects usually are in books of this sort.

Hetland is big on Python, but he's not a language warrior. Aside from a few sly digs at C++, which frankly was never Python's main competition anyway, Hetland sticks to business. Practical Python is pretty much all pedagogy with little panegyric.

A neat web blurb on the book, along with the Table of Contents and a sample chapter, is at http://www.apress.com/book/bookDisplay.html?bID=93.

The copyright page of *Practical Python* notes a site for code download, but has the interesting caveat that the downloader "will have to answer a few questions about the book" in order to download, which I've never seen Apress do before. Perhaps they read their web logs and discovered they were giving away a few thousand copies of source code for each book they sell?

I recently chatted with Magnus via ICQ from his home in Norway.

TPJ: Please tell me about work—what you are doing these days, what projects are you working on?
MH: I'm a Ph.D. student at the Norwegian University of Science and Technology. Hopefully, I'll finish my thesis this spring. My main area of research at the moment is knowledge discovery in time series and sequences in general.

TPJ: What does that mean?
MH: It basically means finding interesting patterns and predicting trends in sequences of various kinds; for example, predicting stock prices and the like—although I haven't had too much luck in that particular area.

TPJ: Does Python help you in this? Or is Python something separate?
MH: Well—I use C++ for the really heavy stuff, such as genetic programming with costly computations and the like, but I use Python for almost everything else. I find it particularly useful for preparing data that I've gathered from various sources. File processing, and especially text-file processing, is much easier in Python than in C++ in my opinion. I guess, basically, I use Python where I find it feasible. I do use Python for some number crunching, too, with extensions such as *numarray* or *Pyrex*.

TPJ: Parsing and string handling in general was never easy in C/C++. Most people use Perl. What's the case for the Python?
MH: I used Perl quite a bit, too, earlier on. And I liked it quite a bit more than C at the time. But when I discovered Python, I found that it had the same advantages when compared to C, but the syntax and the elegance of the object model appealed to me.

TPJ: I'm sure our readers understand the syntax advantage. What's the "elegance" of the object model of Python?
MH: I guess in the days when I switched from Perl to Python, the difference for me was mainly that I couldn't completely grok the Perl model at all, while the Python model seemed quite obvious. What the status of Perl in this area is at the moment, I don't

Jack is an independent consultant specializing in mentoring programming teams and is also a contributing editor to Dr. Dobb's Journal. *His web site is http://www.softwoehr.com/.*

really know, so I'm not in a position to compare the two. But since version 2.2, Python has simplified its type and class systems, and I think the whole thing is now quite sleek—although much of the sleekness is "under the hood."

TPJ: The IBM language ObjectRexx is very aesthetically satisfying from a pure object point of view. You use the term "Mixin class" in the book. MIXIN is an ObjectRexx keyword. Tell me about this.
MH: Right. I've seen it used in C++ too, I think. It's usable in any language with multiple inheritance. Basically, you write simple classes that implement only a limited piece of functionality—a single method, for example. Then you can use that as one of several parent classes, and inherit the limited functionality in addition to that of the "main" parent. Not really a formal concept—more of a conventional way of structuring code. For example, in the Python Standard Library, you have a socket server class. In addition, you have other classes such as a threading mix-in that can be used to get a threaded socket server.

The similarity of [Python to] Rexx has not escaped Python programmers. There is a comparison of the two languages at http://www.scoug.com/os24u/2001/pyREXXex.html and another at http://www.python.org/tim_one/000330.html. Tim Peters, who makes the latter, is a top Python guru.

TPJ: Do Python users experience all the problems that Perl users do when the language undergoes a revision?
MH: Well—I don't really know what problems Perl users face, but of course there are problems. Any changes that are not backwards compatible are problematic. Thankfully, the policy of Guido van Rossum and the Python team on this point is to try to avoid backwards incompatibilities as much as possible. Also, there is a system of gradual change, with warnings being issued by the interpreter when you use features that are obsolete.

TPJ: How did you get started writing the book?
MH: Actually, I was contacted by W. Jason Gilmore from Apress, who had read my online tutorials "Instant Hacking" and "Instant Python." He liked them a lot, and wanted me to write a book in somewhat the same spirit. Even though the book is, of course, a lot longer than the tutorials, I have tried to keep it just as light and humorous.

TPJ: How did your English get so good? While I read the book, I thought "either his English is very good or he has a very good editor."
MH: Actually, I went to an English preschool, so I knew some English already at the age of six. I suppose that has given me a head start. I also had very good editors at Apress, of course.

TPJ: English preschool? Is that a common mode of education in Norway?
MH: No, not really. It was just one year—my parents thought it was a good idea. But we do have quite a bit of English in our school system, so I suppose Norwegians know more English than people in many other countries. Writing the book was a great learning experience—I learned about intricacies and subtleties both in the English language and in Python.

TPJ: What are some of the "intricacies and subtleties" you discovered in writing the book?
MH: Well, for one thing, I learned about standard traps like using "since" instead of "because," which may be misinterpreted. Similarly, I learned about such things as restrictive and nonrestrictive clauses and the uses of "which" and "that." As for Python, I learned so many things that I can hardly remember any of them. But most of what I learned is documented in the book. Much of what I learned can be attributed to my diligent and competent editors.

TPJ: I guess my perspective on Python is foreshortened because by reading your book, I more than doubled my knowledge of the language! How long have you been doing Python?
MH: From around 1997, I think. And, yes, there is quite a lot of knowledge packed into the book. Even though it starts out as a book for newbies, there is quite a lot to be learned. I've included lots of useful little things that many people might not know, and traps that I've seen many people fall into (through comp.lang.python).

TPJ: What are people doing with it? What is stimulating the growth? I mean, with Perl there's CPAN; why go play in the Python sandbox?
MH: Well, there are the typical case histories for a language at http://www.python-in-business.org/success/. But there is actually a repository somewhat like CPAN underway for Python, too (see http://www.python.org/peps/pep-0243.html and http://www.python.org/peps/pep-0301.html), that will be linked with the Python distribution system, Distutils. As for why one would want to use Python—it's hard to give any hard and fast reasons. I think that many features of the language (especially its syntax) make it easy to learn, and its modularity and focus on object-orientedness makes it easy to build large systems without the hassle of languages such as C and C++.

There is some empirical evidence for productivity increase with high-level languages such as LISP, Perl, Python, or Ruby. But choosing between them may be more of an irrational thing, I suppose. There are many arguments for most languages. I guess the best thing to do if one is curious about Python is to try it. There is virtually no cost in setting up an interpreter, and playing around with it can give you some notion of how the language works. My tutorial Instant Python (http://hetland.org/python/instant-python/) is meant as an introduction to people who already know other languages. It should give you an overview of the language in, say, half an hour.

TPJ: For some reason, your publisher, Apress, puts notes about URLs for source code on the copyright page where almost no one finds and reads them. And they're not very informative. This is just a small glitch from a publisher that is in other respects a very good publisher. But I don't see any URL in the book for corrections/errata (I've found minor typos here and there).
MH: There should be a link from the Apress page, and there is also a link from my page at ppython.com.

TPJ: Wasn't writing *Practical Python* quite a distraction as you work towards your Ph.D.?
MH: Yes, it was. My advisor wasn't all too happy with it. I hadn't planned on spending as much time on it as I ended up doing. But it seems that I'll be able to finish my thesis on time anyway (hopefully), and then the book is only a bonus. It certainly has given me a lot of training that may be useful when writing scientific articles and the like. I've tried my very best to convey the sense of fun I feel when programming in Python. I suppose that's been my main goal—to show that programming can be fun and that anyone can do it; to some degree, anyway.

If you're a programmer, you ought to love programming. I feel sorry for those entering the business only in the hope of earning loads of money. I noticed a sharp upturn in the number of students taking computer-related courses at our university at the end of the '90s. Many of them had little interest or aptitude for programming. Of course, you don't have to know programming very well to work with computers, but I think you should. It's basic craft.

TPJ

Source Code Appendix

Kevin O'Malley "Using *PerlObjCBridge* to Write Cocoa Applications in Perl"

Listing 1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//
DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>prefs-path</key>
  <string>./</string>
  <key>editor</key>
  <string>/usr/bin/emacs</string>
  <key>viewer</key>
  <string>more</string>
  <key>file-path</key>
  <string>./</string>
  <key>contacts-file</key>
  <string>contacts.txt</string>
  <key>tasks-file</key>
  <string>tasks.txt</string>
  <key>words-file</key>
  <string>word_list.txt</string>
  <key>notebook-file</key>
  <string>notebook.txt</string>
  <key>quotes-file</key>
  <string>quotes.txt</string>
</dict>
</plist>
```

Listing 2

```
for(;;) {
  system("clear");
  print "=======================\n";
  print "My PIM\n";
  print "=======================\n";
  print "0. Edit preferences file\n";
  print "1. Edit reminders\n";
  print "2. Edit contacts\n";
  print "3. Edit tasks\n";
  print "4. Show tasks\n";
  print "5. Generate calendar (ps)\n";
  print "6. View calendar (txt)\n";
  print "7. Print calendar's";
  print "8. Show system cal\n";
  print "9. Show today's reminders\n";
  print "-------------------\n";
  print "-1. Edit word list\n";
  print "-2. Edit notebook\n";
  print "-3. Edit quotes\n";
  print "-4. View quotes\n";
  print "=======================\n";
  print "-> ";
  my $s = <STDIN>;
  chop($s);
  if ($s == 0) {
    system(getPrefVal("editor") . " " . $PREFS_FILE);
  }
  elsif ($s == 1) {
  system(getPrefVal("editor") . " ~/.reminders");
}
```

Moshe Bar "Home Automation with Perl"

Listing 1

```
# Modify this rule for use with get_email
# Rename to get_email_rule.pl to enable
#  - $from_full has the full email address, not just the name portion.

sub get_email_rule {
    my ($from, $to, $subject, $from_full) = @_;
    $from = 'The S F gals'          if $to =~ /FEM-SF/;
    $from = 'The E C S guys'        if $to =~ /ecs/;
    $from = 'The Mister House guys' if $to =~ /misterhouse/;
    $from = 'The perl guys'         if $to =~ /Perl-Win32-Users/;
    $from = 'The phone guys'        if $to =~ /ktx/ or $subject =~ /kx-t/i;
    $from = 'junk mail'             if $from =~ /\S+[0-9]{3,}/; # If we get a joe#### type address, assume it is junk mail.
    return                           if $from =~ /X10 Newsletter/;

   $from =~ s/\./ Dot /g ;     # ...change "." to the word "Dot"
   $from =~ s/\@/ At /g ;      # ...change \@  to the word "At"

    return $from;
}

return 1;
```

Listing 2

```
sub _osascript
        {
        my $script = "tell source 'Library'
                     tell playlist 'Library'
                                set this_track to some track
                                set this_name to the name of this_track
                                set this_artist to the artist of this_track
                                set this_album to the album of this_track
                                play this_track
                     end tell
           end tell";

        require IPC::Open2;

        my( $read, $write );
        my $pid = IPC::Open2::open2( $read, $write, 'osascript' );

        print $write qq(tell application "iTunes"\n), $script,
                       qq(\nend tell\n);
        close $write;

        my $data = do { local $/; <$read> };

        return $data;
        }
```

Luis E. Muñoz "Parsing MIME & HTML"

Listing 1

```
#!/usr/bin/perl

# This script is (c) 2002 Luis E. MuÒoz, All Rights Reserved
# This code can be used under the same terms as Perl itself. It comes
# with absolutely NO WARRANTY. Use at your own risk.

use strict;
use warnings;
use IO::File;
use Net::POP3;
use NetAddr::IP;
use Getopt::Std;
use MIME::Parser;
use HTML::Parser;
use Unicode::Map8;
use MIME::WordDecoder;

use vars qw($opt_s $opt_u $opt_p $opt_m $wd $e $map);

getopts('s:u:p:m:');

usage_die("-s server is required\n") unless $opt_s;
usage_die("-u username is required\n") unless $opt_u;
usage_die("-p password is required\n") unless $opt_p;
usage_die("-m message is required\n") unless $opt_m;

$opt_s = NetAddr::IP->new($opt_s)
    or die "Cannot make sense of given server\n";

my $pops = Net::POP3->new($opt_s->addr)
    or die "Failed to connect to POP3 server: $!\n";

$pops->login($opt_u, $opt_p)
    or die "Authentication failed\n";

my $fh = new_tmpfile IO::File
    or die "Cannot create temporary file: $!\n";

$pops->get($opt_m, $fh)
    or die "No such message $opt_m\n";

$pops->quit();
$pops = undef;

$fh->seek(0, SEEK_SET);

my $mp = new MIME::Parser;
$mp->ignore_errors(1);
$mp->extract_uuencode(1);
```

```
eval { $e = $mp->parse($fh); };
my $error = ($@ || $mp->last_error);

if ($error)
{
    $mp->filer->purge;      # Get rid of the temp files
    die "Error parsing the message: $error\n";
}

                # Setup the HTML parser

my $parser = HTML::Parser->new
    (
     api_version => 3,
     default_h      => [ "" ],
     start_h => [ sub { print "[IMG ",
            d($_[1]->{alt}) || $_[1]->{src},"]\n"
                if $_[0] eq 'img';
            }, "tagname, attr" ],
     text_h => [ sub { print d(shift); }, "dtext" ],
     ) or die "Cannot create HTML parser\n";

$parser->ignore_elements(qw(script style));
$parser->strict_comment(1);

$map = Unicode::Map8->new('ASCII')
    or die "Cannot create character map\n";

setup_decoder($e->head);

print "Date: ", $e->head->get('date');
print "From: ", d($e->head->get('from'));
print "To: ", d($e->head->get('to'));
print "Subject: ", d($e->head->get('subject')), "\n";

decode_entities($e);

$mp->filer->purge;

sub d { $map->to8($map->to16($wd->decode(shift||''))); }

sub setup_decoder
{
    my $head = shift;
    if ($head->get('Content-Type')
    and $head->get('Content-Type') =~ m!charset="([^\"]+)"!)
    {
    $wd = supported MIME::WordDecoder uc $1;
    }
    $wd = supported MIME::WordDecoder "ISO-8859-1" unless $wd;
}

sub decode_entities
{
    my $ent = shift;

    if (my @parts = $ent->parts)
    {
    decode_entities($_) for @parts;
    }
    elsif (my $body = $ent->bodyhandle)
    {
    my $type = $ent->head->mime_type;

    setup_decoder($ent->head);

    if ($type eq 'text/plain')
    { print d($body->as_string); }
    elsif ($type eq 'text/html')
    { $parser->parse($body->as_string); }
    else
    { print "[Unhandled part of type $type]"; }
    }
}

sub usage_die
{
    my $msg = <<EOF;
Usage: mfetch -s pop-server -u pop-user -p pop-password -m msgnum
EOF
    ;
    $msg .= shift;
    die $msg, "\n";
}
```