

# *The Perl Journal*

## **A mod\_perl 2 Primer**

Peter Sergeant • 3

## **Metamodeling with Perl and AMPL**

Christian Hicks and Dessislava Pachamanoval • 6

## **Perl, VMWare, and Virtual Solutions**

Sam Tregar • 13

## **Sorting Out the Linguistic Mess**

Simon Cozens • 16

## **Making New Distributions**

brian d foy • 19

## **PLUS**

**Letter from the Editor • 1**

**Perl News by Shannon Cochran • 2**

**Book Review by Jack J. Woehr:**

***Perl 6 and Parrot Essentials* • 21**

**Source Code Appendix • 22**

## LETTER FROM THE EDITOR

### Emulation Evolution

Programming often boils down to modeling something. User interfaces that we build often look like real-world physical objects. Our applications sometimes model physical tasks we perform, such as punching buttons on a keypad or wiring objects together with cables. The analog world is so often virtualized that we don't even bat an eye at it anymore. In fact, we expect it, and take it for granted.

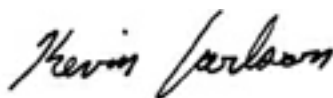
But we don't just model the analog world. We run entire operating systems in emulation. We now digitally model the digital. This isn't new—I distinctly remember running DOS in emulation on a Macintosh back in '92. But the power and usability of such emulation has increased dramatically in recent years. Take VMWare, for instance. The fact that Sam Tregar (see his article on page 13) can run not just one, but six virtual operating systems on one machine attests to just how smart and useful our emulation has become.

The rise in emulation and the concept of the virtual machine at one time promised an end to platform dependency. The Java VM was supposed to allow us to stop worrying about what hardware and underlying OS we were running. But the devil was in the details, of course. The Java VM has provided some wonderful benefits, to be sure, but it's hardly the liberation we were promised. The phrase "platform independent" still comes with many qualifications and exceptions, whether you program in Perl, C++, Java, or any other language you care to name.

As long as multiple operating systems and multiple types of microprocessors exist, the virtual machine designer's job will remain one of lowest common denominators—figuring out what subset of functionality exists on all host platforms, and limiting the VM to that subset. There are ways to fudge that, of course, but you'll never be able to, for instance, provide accelerated 3D rendering on a system with no accelerated 3D video hardware. Highly cross-compatible emulation is directly opposed to platform-specific optimization. That's a law that can't be gotten around, and it's what has kept us from that hardware- and OS-agnostic promised land. We want optimization and performance, so we have to care about what's under the hood.

Within those limitations, however, emulation can do wonders. It's very useful when you aren't particularly concerned about performance, but you just have to run that Mac program on your PC, or vice versa. And when emulation is freed from the yoke of cross-platform operation, it can really shine. Apple's "Classic" environment is a good example. It allows the execution of OS 9 apps in OS X, and because it doesn't need to run on multiple platforms, it can be highly optimized. Classic apps on the Mac run impressively fast. With some apps, performance is indistinguishable from native operation.

While we haven't entered the emulation promised land, there's still a lot to like about the current state of emulation. We get access to applications that would otherwise require a prohibitively expensive hardware purchase. We get some apps that, if carefully designed, can be mostly cross platform. And we get a backwards-compatibility insurance policy for cherished or necessary older programs. All in all, I'd call that real progress.



Kevin Carlson  
Executive Editor  
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to [editors@tpj.com](mailto: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: \$18.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2004 by CMP Media LLC, unless otherwise noted. All rights reserved.



### The Perl Journal

#### EXECUTIVE EDITOR

Kevin Carlson

#### MANAGING EDITOR

Della Wyser

#### ART DIRECTOR

Margaret A. Anderson

#### NEWS EDITOR

Shannon Cochran

#### EDITORIAL DIRECTOR

Jonathan Erickson

#### COLUMNISTS

Simon Cozens, Brian d'Joy, Moshe Bar, Andy Lester

#### CONTRIBUTING EDITORS

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

#### INTERNET OPERATIONS

##### DIRECTOR

Michael Calderon

##### SENIOR WEB DEVELOPER

Steve Goyette

##### WEBMASTERS

Sean Coady, Joe Lucca

#### MARKETING / ADVERTISING

##### PUBLISHER

Michael Goodman

##### MARKETING DIRECTOR

Jessica Hamilton

##### GRAPHIC DESIGNER

Carey Perez

#### THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403

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

#### CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND MARKETING

Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT/GROUP DIRECTOR INTERNET BUSINESS

Mike Azara

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

VICE PRESIDENT, COMMUNICATIONS Alexandra Raine

PRESIDENT, CHANNEL GROUP Robert Faletra

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED TECHNOLOGIES

Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER NETWORK COMPUTING

ENTERPRISE ARCHITECTURE GROUP Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE DEVELOPMENT

MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING SOLUTIONS

Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES Marie Myers

# Perl News

## The Once and Future Perl

The sixth maintenance release of Perl 5.8, mostly incorporating bug fixes and performance enhancements, is now available from CPAN. New functionality in the update includes better tolerance of UTF-16-encoded scripts; the ability to use nonIFS compatible LSPs on 32-bit Windows systems, allowing Perl to work in conjunction with some firewalls; and a new `-dt` command-line flag, which enables threads support in the debugger.

On the perl5-porters list, Rafael Garcia-Suarez sketched out a roadmap for Perl 5.10, which could arrive as early as next summer. Rafael identified seven central tasks left to complete. First is to finish assertions: “The current way to combine assertions is, in my opinion, ugly.” Second is to integrate the lexical pragma patch submitted by Mark Jason Dominus and Autrijus Tang. The `encoding::warnings` module from CPAN will also be integrated; “make it lexical,” Rafael noted. A replacement needs to be implemented for the deprecated `my $x if 0`; Rafael noted that “in perl 6 style, that would be `state $x = "initial value"`. (Ooh, a new keyword.) Speaking of new keywords, `state` and `err` should be second-class keywords. At least. I bet `state()` is a common function name...” Lastly, Perl 5.10 will also serve as the Perl 5 core to ponie. For future-proofing purposes, Rafael suggested adding “a `no ponie` pragma, for hairy code that won’t run under ponie.”

## Parrot Grammar Engine Is Born

Patrick R. Michaud has completed the first incarnation of the Parrot Grammar Engine. According to the README, “Basically, PGE consists of a parser and a PIR code generator written in C. The parser takes a string representing a [Perl 6] rule and builds a parse tree for the rule. The generator then produces a PIR subroutine (matching engine) that can match strings according to the components of the rule.”

Other contributors quickly began to build on Patrick’s work. Andy Dougherty contributed a patch to help PGE build more cleanly on OS X, and Jared Rhine helped to construct an initial testing framework. “I know there are lots of existing regular expression test suites out there,” Patrick noted on the perl6-compiler list, “so if others could convert the relevant portions to work with the above framework, that’d be great.” He also outlined a direction of future development: “My current plan is to implement cut and backreferences next, then some level of character class support (even if it’s just `\d`, `\D`, `\s`, `\S`, `\w`, `\W` while we wait for Parrot strings to stabilize), and then subrules and lookarounds.”

## The Secret Synopses

The latest versions of the Synopses and Apocalypses can now be found at <http://dev.perl.org/perl6/synopsis/> and <http://dev.perl.org/>

[perl6/apocalypse/](http://perl6/apocalypse/), respectively; Larry says the older [www.wall.org](http://www.wall.org) links are to be considered deprecated. But use this knowledge with care: “If everyone’d think \*real\* hard before actually submitting to [Slashdot] (and, hopefully, deciding not to) I’d appreciate it,” Dan Sugalski requested on the perl6-compiler list. “I feel obligated to actually \*read\* the comments on parrot and perl 6 stories on Slashdot, at 0, so if I don’t actually have to do so, well...so much the better usually.”

## Perls of Wisdom

Mark Fowler is once again marking the gift-giving season with a Perl Advent Calendar (<http://perladvent.org/2004/>). This holiday calendar, featuring an image of the legendary Three Wise Camels, bears 25 numbered boxes that each become clickable on their corresponding date in December. “Behind” each day is a window presenting a different CPAN module, with documentation and a tutorial. This is the fifth year Mark has continued the advent calendar tradition: “Not much has been added this year,” he notes on the site. “The calendar’s maturing, and all the features that I have time to add have been added. This year, I’ve tidied up the HTML, and got rid of most of the tables (it now uses `divs` for nearly all its layout—suggestions on how to remove the last few tables would be great). Mainly this year, I’ve worked out that trying to do the advent calendar, while being really busy at work, while trying to help help organise a Perl conference, while trying to conduct the no small matter of planning a wedding, is very exhausting. More features next year maybe...”

## Perl DBI is 10

It’s been 10 years since the first public release of the Perl DBI, and Tim Bunce is looking toward the future. In a roadmap posted to CPAN (<http://search.cpan.org/~timb/DBI/Roadmap.pod>), he describes plans for new enhancements in “testing, performance, high availability and load balancing, batch statements, Unicode, database portability, and more. Addressing these issues together, in a coordinated way, will help ensure maximum future functionality with minimal disruptive (incompatible) upgrades.”

Toward this end, The Perl Foundation has set up a DBI Development Fund at <http://dbi.perl.org/donate/>. A TPF Grant Manager will reward Tim from the fund as milestones are reached. Donations to the fund are tax deductible. “If your company has benefited from the DBI then this is an opportunity to both give back, by way of thanks, and help ensure greater benefits in the future,” Tim explains. Alternatively, companies can choose to sponsor the development of specific new functionality. See <http://groups-beta.google.com/group/perl.dbi.users/msg/caf189d7b404a003> for Tim’s birthday announcement and call for support.



# A mod\_perl 2 Primer

In the beginning, there were web servers. You had a nice transactionless way to allow remote users to browse areas of your computer's filesystem. But this wasn't enough—what if you wanted to allow remote users to execute local applications? Thus, CGI was born as a protocol for allowing remote users to pass arguments to applications on your machine. Suddenly, you could write guestbook applications and all sorts of other monstrosities, and make your web site “dynamic.” Great.

If your application was written in Perl, however, your web server had to fire up Perl, run your script, and tear Perl back down again each time someone wanted to run your code. This was somewhat processor and memory intensive, especially if your web server was trying to run your application for more than one user at once.

But, mod\_perl embeds a persistent Perl interpreter into Apache, allowing you to intercept requests and handle them with a Perl module. Thus, you can rewrite your application to be wrapped up in a Perl module, and execute it over and over again without having to initialize a new Perl instance and load up all the other libraries your application may use each time you want to run some code.

mod\_perl allows you to use Perl to control other parts of the Apache request process, too, such as HTTP authentication—these other uses, however, are outside of the scope of this article. This article is intended to give reasonably competent Perl programmers a quick-and-dirty introduction to mod\_perl 2, and to give those already familiar with mod\_perl 1 an idea of how mod\_perl 2 is different.

## What Is mod\_perl 2?

mod\_perl has been around for a while, and many people are familiar with it. However, Apache 2 includes many features not available in Apache 1, and presents an opportunity for mod\_perl developers to fix concepts that were “broken” in the original mod\_perl.

However, mod\_perl 2 isn't finished yet. Its stability and fitness for use in a production environment are discussed briefly later, but the fact remains that the latest release (at the time of writing) is on Version 1.99\_17—a dead giveaway that it's still beta. Therefore, when we refer to mod\_perl 2, this is what we're talking about—hopefully, in the near future, we'll see a mod\_perl with a version number of 2 or above, but that's not the case today.

*Pete works for Community Internet Ltd. (<http://www.community.co.uk/>) and can be contacted at [pete@clueball.com](mailto:pete@clueball.com).*

## Is mod\_perl 2 Ready to Use?

The low version number is an oft-cited reason people aren't comfortable with using mod\_perl 2 in a production environment yet. Geoffrey Young, coauthor of *mod\_perl Developer's Cookbook* (Pearson Education, 2002) and a mod\_perl developer, paints a different picture:

“mod\_perl 2.0 is as stable as a piece of software can be without large-scale deployment fleshing out the issues that only large-scale deployment can expose...in our minds, 2.0 will be released when we consider the API to be frozen and immutable. Not releasing an official 2.0 is just us committing to the user-base that we promise to not mess with how the official API looks [once released]. But really, we're at the immutable stage with 99% of the stuff at this point, so unless you're doing really funky stuff (specifically stuff you couldn't do in [mod\_perl 1]) you probably wouldn't notice.”

We're not going to be doing anything funky here, so we should be safe.

## Setting Up Apache

Fully configuring Apache is outside of the scope of this article. You can find complete details at: <http://perl.apache.org/docs/2.0/user/config/config.html>.

However, it's worth pointing out that all the following examples are run on my server using the following lines in the Apache configuration file, and if you're reasonably comfortable with the Apache configuration, you can probably steal and adapt these:

```
# Tell Apache where to find our Perl modules...

PerlSwitches -I/Users/sheriff/modules/

# Specify the location the mod_perl handler applies to

<Location /DisplayPage/>
    SetHandler perl-script

    # Specify the module to use to handle this 'location'
    PerlResponseHandler Module::Name

</Location>
```

When Apache invokes a handler from a request, it passes that handler a bundle of information about the request in the form of a C *struct* (a bit like a hash) called the “request record.” This is passed to your handler in the form of an *Apache::RequestRec* object; you are not, of course, expected to interact directly with the request record—*Apache::RequestRec*’s API takes care of all that for you. The function called *handler* in your Perl library will be invoked with this object as the first argument.

### *Apache::RequestIO* and *Apache::Const*

So far we’ve touched briefly on *Apache::RequestRec* for interacting with Apache. There are two other modules that are worth knowing about before we attempt a *Hello World* module.

*Apache::RequestRec* allows us to retrieve data about the incoming request. *Apache::RequestIO* provides us with IO methods we can use on the request object—for example, it allows us to print data to the user. Finally, *Apache::Const* sets up some constants for us to use for returning HTTP status codes from our *handler* routine.

Here’s a very basic example:

```
01: package TestModules::Hello;
02: use strict;
03: use Apache::RequestRec ();
04: use Apache::RequestIO ();
05: use Apache::Const -compile => qw(:common);
06: sub handler {
07:     my $r = shift;
08:     # Grab the query string...
09:     my $query_string = $r->args;
10:     # Print out some info about this...
11:     $r->content_type( 'text/plain' );
12:     $r->print( "Hello world! Your query string
           was: $query_string\n" );
13:     return Apache::OK;
14: }
15: 1;
```

The first line declares the package name of our handler, and the second sets the *strict* pragma, forcing us to predeclare our variables and helping to make any mistakes that we make more visible.

The inclusion of the third line may strike some readers as odd, but we’ll come back to that in a moment—for the time being, let’s say (accurately) that it loads the *Apache::RequestRec* library. Line 4 loads the *Apache::RequestIO* library, and line 5 loads *Apache::Const* and asks it to load up the “common” set of constants.

*handler* is the name of the function that *mod\_perl* calls when it wants to pass a request to your library. It passes the request object to this handler as its first argument. Traditionally, people save the request object to the scalar *\$r*.

Line 9 retrieves the query string of the URL that led to our handler being called—the *args* method is provided by *Apache::RequestRec*. Line 11 sets the content-type of our response, and line 12 outputs some data to the user, including the query string they sent us. Finally, line 13 tells Apache that we finished what we wanted to do successfully and that an HTTP code of “200” (which indicates success) should be sent back to the user.

So if we’re being passed an *Apache::RequestRec* object, why do we need to explicitly load this library, too? This is a somewhat controversial design decision—you’re being passed an object that’s blessed into a class that may not exist in memory yet. Some people may consider this a crime against nature. On the other hand, it does help to keep your code footprint down if you’re not intending to use any of the methods provided by *Apache::RequestRec* itself—other modules, such as *Apache::RequestIO*, add methods to the *Apache::RequestRec* namespace.

You can find documentation for all these modules at <http://perl.apache.org/docs/2.0/api/index.html>.

### Redirect Script

So we’ve successfully written our first *mod\_perl* 2 handler. However, it doesn’t really do much that’s very useful. Next, I’ll describe a very simple redirect handler. For this, you’ll need to know something about *Apache::URI* and *APR::Table*.

### Headers and *APR::Table*

To do a simple redirect, we’re going to need to read in the URL, decide how to redirect based on that, construct a new URL, add a redirect header, and send it to the user. Let’s talk about setting a redirect header first.

---

*Apache 2...presents an opportunity for mod\_perl developers to fix concepts that were “broken” in the original mod\_perl*

---

HTTP allows you to set more than one header of the same name, which means *mod\_perl* needs to store the headers you want to send in a way that reflects this. A simple hash-based method for storage won’t work—you can’t easily assign more than one value to a hash without messing around with array references. Thus, headers are represented by *APR::Table* objects, which hide all this behind a nice, tidy API.

*Apache::RequestRec* gives us the method *headers\_out*, which returns an *APR::Table* object. We need to add a “Location” header, so we can use the *set* method on this object, which adds or overwrites a key’s value. For a more comprehensive discussion of *APR::Table*, see <http://perl.apache.org/docs/2.0/api/APR/Table.html>.

Essentially, what we need to do is:

```
# Retrieve the out-going headers APR::Table object
my $headers = $r->headers_out;
```

```
# Set/overwrite the 'Location' key's value
$headers->set( Location => 'http://whatever/' );
```

Or more concisely:

```
$r->headers_out->set( Location => 'http://whatever/' );
```

So we’re almost there. Now we just have to construct our URL to send out. We could just create a simple scalar and use that, but we might be hosting requests on a variety of hosts and ports, so let’s be a little more intelligent about it.

### *Apache::URI*

*Apache::URI* gives us a nice way to construct URLs using the requested URL as a base. *Apache::URI* brings to *\$r* the *construct\_url* method (among others—see <http://perl.apache.org/docs/2.0/api/APache/URI.html>). This allows us to create a fully qualified URL from a relative path. So, for example, we could say:

```
my $new_url = $r->construct_url( '/foo/' );
```

Which, assuming the request had been for “http://wherever:9000/asdf/,” would give us “http://wherever:9000/foo.” Excellent. Putting it all together, we get:

```
01: package TestModules::Which;
02: use strict;
03: use Apache::RequestRec ();
04: use APR::Table ();
05: # We only need to load the
   # REDIRECT status constant...
06: use Apache::Const -compile => qw( REDIRECT );
07: sub handler {
08:     my $r = shift;
09:     my $url = $r->construct_url('/new_location/');
10:     $r->headers_out->set( Location => $url );
11:     return Apache::REDIRECT;
12: }
13: 1;
```

## Using CGI.pm

CGI is a rather ambiguous term in this context. CGI stands for Common Gateway Interface, and describes how to pass data to code being executed by a web server. Due to the use of the phrase “CGI script” to mean an application executed by a web server, people tend to talk about CGI or mod\_perl. However, CGI is the way to get data to your mod\_perl handler, so it’s appropriate that recent versions of CGI.pm (2.92 and above) allow you to interact with mod\_perl.

Essentially, you can use CGI.pm in your mod\_perl handlers in almost the same way that you would in your CGI scripts. All you have to do is initialize your CGI object using the *Apache::RequestRec* object:

```
my $cgi = CGI->new( $r );
my $value = $cgi->param( 'key' );
```

*CGI::Cookie* can also be used in a mod\_perl environment—see the *CGI::Cookie* documentation for more information.

## Sensible Database Access

Presumably, you don’t want to be reconnecting to your database through DBI for each and every request—you want to create your database handle outside of the handler sub. But every Apache process will have its own copy of your handler library in memory, and presumably, you’re not going to want each of those to have an open connection to your database if you’re not using it.

This is where *Apache::DBI* comes in handy—use it before you start invoking DBI and it’ll transparently maintain a pool of database connections that all the different instances of your handler can use. It even works transparently with *Class::DBI*.

## Final Example

To bring all these ideas together, we’ll write a handler that potentially does something useful. Sites like <http://xrl.us/> and <http://www.tinyurl.com/> allow you to generate short URLs that link to longer ones. We’re going to write a handler that will provide redirects in this manner—when a URL is requested, we’ll extract the key, search the database for a stored URL, and redirect the user as appropriate.

The only thing we’ve not seen yet that we’re going to introduce with this example is the use of the *path\_info* method. What exactly this returns is somewhat complicated—but if we set up a handler to match the location “/x/,” and someone asks for “/x/abc,” the *path\_info* method will return “abc.” The code for this is shown in Listing 1.

Hopefully, this short tutorial will get you started writing your own mod\_perl handlers—for many applications, you will not need to venture outside of the toolbox presented here.

TPJ

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

### Listing 1

```
package TestModule::Redirect;
```

```
    use strict;
    use warnings;

    use Apache::DBI;
    use Apache::RequestRec ();

    # We only need two of the status code constants...

    use Apache::Const -compile => qw(
        NOT_FOUND
        REDIRECT
    );

    # Connect to the database

    my $dbh = DBI->connect(
        'dbd:mysql:redirect',
        'user',
        'password',
    );

    # Prepare our SQL query

    my $sql = $dbh->prepare("
        SELECT url FROM redirects WHERE key = ?
    ");

    # And finally our handler...

    sub handler {
```

```
        my $r = shift;

        # Get the part of the URL we want (see notes)

        my $key = $r->path_info;

        # Strip out anything we're not expecting...

        $key =~ s/[^a-z]//g;

        # Which might leave us without a key at all,
        # so we check and give an error if this is the
        # case

        return Apache::NOT_FOUND unless $key;

        # Grab the URL in the database corresponding
        # to the key

        my $result = $sql->execute( $key );
        my $url = $result->fetchrow_arrayref;

        # If there's no entry for the key in the database,
        # then let the user know

        return Apache::NOT_FOUND unless $url;

        # Set the new URL, and redirect appropriately

        $r->headers_out->set( Location => $url->[0] );
        return Apache::REDIRECT;

    }

    1;
```

TPJ

# Metamodeling with Perl and AMPL

In the 1980s, a significant conceptual breakthrough was accomplished in computational optimization with the introduction of universal optimization languages, such as the Algebraic, or Applied, Mathematical Programming Language (AMPL) and the General Algebraic Mathematical System (GAMS). AMPL and GAMS do not solve optimization problems. Instead, they provide the environment for users to describe their models in a standardized way, and call on separate solvers to solve the problem. The results are returned to human modelers via the same software.

Originally developed at AT&T Bell Labs, AMPL is a modeling language for optimization problems. With AMPL, users describe an optimization problem in the AMPL language, after which the AMPL software processes the problem and passes it to a separate optimization solver. AMPL supports numerous commercial and free solvers. For a full listing of available solvers, see <http://www.ampl.com/>. The Student Edition of AMPL (which limits the number of variables and constraints in the model) is included with *AMPL: A Modeling Language for Mathematical Programming*, Second Edition, by Robert Fourer, David M. Gay, and Brian W. Kernighan (Duxbury Press/Brooks/Cole Publishing, 2002; ISBN 0534388094). Windows, UNIX (including Linux), and Mac OS X versions of the Student Edition and compatible solvers are also available for free download from <http://www.ampl.com/>. The full version of AMPL can be purchased from ILOG at <http://www.ilog.com/>. ILOG also sells an AMPL-compatible version of CPLEX, which is one of the most powerful solvers for linear, mixed-integer, and quadratic optimization. Windows and UNIX (including Linux) binaries of AMPL and CPLEX are available. AMPL is also one of the modeling languages supported by the NEOS Server for Optimization (<http://www-neos.mcs.anl.gov/neos/>). The NEOS server lets users submit optimization problems via the Internet, solves them for free on a NEOS computer, and returns the results to the submitter.

Christian is president and cofounder of Elysium Digital LLC. Dessislava is an assistant professor of operations research at Babson College, MA. They can be contacted at [cbhicks@elys.com](mailto:cbhicks@elys.com) and [dpachamanova@babson.edu](mailto:dpachamanova@babson.edu), respectively.

A simple optimization example from the book *AMPL: A Modeling Language for Mathematical Programming* is as follows: A steel mill processes unfinished steel slabs into steel bands and steel coils. It can produce 200 tons of bands per hour and 140 tons of coils per hour, but cannot sell more than 6000 tons of bands per week or 4000 tons of coils per week. When sold, bands profit \$25.00 per ton, and coils profit \$30.00 per ton. How many tons of bands and how many tons of coils should be produced to maximize profit?

To solve this problem, you create the two input files for AMPL, like Examples 1(a) and 1(b). Of course, this is a simple problem, and AMPL can handle problems far more complex. But real-world optimization models can have infinite variety. Frequently, they require multiple refinements of the model and data, and therefore, multiple calls to optimization and data-processing software. Some optimization formulations involve solving sequences of subproblems. Other models require, in addition to optimization, the ability to analyze data through statistical techniques (to run regressions, for example), the ability to simulate scenarios, the ability

```
(a)
steel.mod:
set PROD;                # products
param rate {PROD} > 0;    # tons produced per hour
param avail >= 0;         # hours available in week
param profit {PROD};     # profit per ton
param market {PROD};     # limit on tons sold in week
var Make {p in PROD} >= 0, <= market[p]; #tons produced
maximize Total_Profit: sum (p in PROD) profit[p] * Make[p];
# Objective: total profits from all products
subject to Time: sum (p in PROD) (1/rate[p]) * Make[p] <= avail;
# Constraint: total of hours used by all
# products may not exceed available hours

(b)
steel.dat:
set PROD := bands coils;
param:      rate  profit  market  :=
bands      200    25      6000
coils      140    30      4000
param avail := 40;
```

Example 1: Input files.



to pass results from one stage of the computation as inputs for the next stages of the problem, and so on. We define such complex models as “optimization metamodels.”

We ran into the problem of solving optimization metamodels while creating sophisticated financial models that required simulation, statistical analysis, and multistage optimization. As we sought to devise a good system for handling optimization metamodels, we focused on leveraging AMPL because of its modern design and its support of a wide range of solvers.

## Little Languages and Optimization Systems

In a 1995 talk at Princeton University, Brian Kernighan, one of AMPL’s creators, described AMPL as a prime example of a “little language.” The term “little languages” was introduced for the first time by Jon Bentley (1986), who was at Bell Labs at the same time AMPL was designed. Bentley considered three different approaches to solving the problem of generating certain images:

- Interactive systems (such as PhotoShop) made drawing simple images easy, but did not allow for programmatic generation of complex pictures.
- Subroutine libraries worked well for generating images from big programs, but were awkward for small tasks.
- Little languages provided an attractive option, since they could be powerful enough to generate complex images in an automated way, but limited enough that users could master the syntax for the language in a reasonable amount of time.

Bentley’s reasoning applied perfectly to AMPL, which was designed to allow for the easy handling of both simple and complex optimization problems. We used Bentley’s reasoning (as well as our own) in creating a system for optimization metamodeling.

Interactive systems were unsuitable because of the complexity of optimization metamodels. The goal of an integrated system was not to help users create simple models—this problem had already been solved by other systems. We placed a premium on supporting complexity and automation.

The little languages approach adopted by AMPL had certainly produced an effective optimization tool, and could be extended to support optimization metamodeling. This would require adding new functionality to the little language for each task that the metamodeler might need; for example, loading data, running simulations, storing and processing results in fully flexible data structures, launching external programs to process intermediate results. In fact, some such capabilities have been added recently: Today, AMPL provides not only an environment for optimization formulations but also tools for data manipulation and database access. For example, AMPL’s table declaration lets you define explicit connections between sets, parameters, variables, and expressions in AMPL, and relational database tables maintained by other software, such as Microsoft Access and Excel. In addition, recently introduced command script features let you specify the number of times a model should be rerun, change values of parameters, and specify output format.

The problem with this continuous expansion of AMPL’s capabilities is that a metamodel might need a great variety of functionality, to the point that the little language might not be so little anymore. If AMPL were to encompass the entire functionality of C/C++, Perl, or Java, it would be difficult to learn, to say nothing of how difficult it would be to create.

Alternatively, you could allow a little language access to outside resources by providing a mechanism for the invocation of outside code. You can imagine an AMPL program that made calls to other code written in C/C++, Perl, or Java. This option provoked the question: If one language is to be used to call code written in a second language, is it not preferable for the calling language to be the broader, general-purpose language, and the called

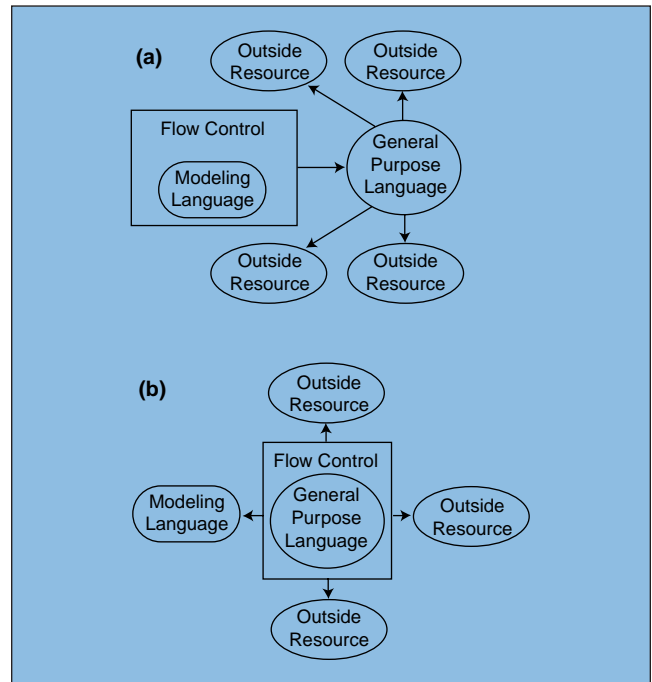


Figure 1: Language design options.

code to be written in the special-purpose language? The two options are illustrated in Figures 1(a) and 1(b). Again, Bentley’s paper shed light on the subject, in that he agrees that performance and logic suggest that flow-control elements, such as loops, are better implemented in general-purpose languages like Pascal, rather than in little languages. From a system design standpoint, we believe that the system illustrated in Figure 1(b) is the better option.

This analysis pushed us toward using a subroutine library packaged up in an API. Users can download the library, then call the functions contained in it from within his program. Inputs and outputs to an individual optimization model are variables within the main program, and can therefore be manipulated directly using the full facilities of the programming language.

An existing metamodeling system that we analyzed in this context was Matlab, a programming language that “provides core mathematics and advanced graphical tools for data analysis, visualization, and algorithm and application development” (<http://www.Mathworks.com/>). One of the available toolboxes for Matlab is the Optimization Toolbox, which enables general and large-scale optimization, including linear programming, quadratic programming, nonlinear least-squares, and nonlinear equations. Accordingly, Matlab could be viewed as an example of a language that provides access to optimization functionality via a subroutine library.

However, the evolution of Matlab showed us that it is more like a little language that has been expanded to include additional functionality, instead of an efficient general-purpose programming language with subroutine library access to special functionality. This shows in the performance of the Matlab language, especially in its slow handling of loops. Since we wanted to take advantage of good performance and flexibility in our master language, we shied away from Matlab’s solution.

Several candidates emerged for the programming language of our library:

- Microsoft Visual Basic is easy to learn, allows for the fast creation of attractive user interfaces for programs, and has mechanisms to interact with Excel and Access. However, it has inconsistent syntax, making programming frustrating at times.



Programs written in Visual Basic run slowly, and in our experience, they run unreliably, letting the same program behave differently when run repeatedly under the same conditions. This is unacceptable for many programming situations. Additionally, Visual Basic is nonportable, limiting users to Windows, and it is licensed, requiring a license for each programmer.

- C/C++ was an automatic candidate, given how widely the language is used for software development. A C/C++ compiler is available on almost every platform. Programs written in C/C++ tend to run fast (unless written poorly), and there are many existing subroutine libraries available to provide programmers with additional resources. The downside of C/C++ is that programming in the language is difficult and error prone, due to the way low-level memory management and addressing are left in the hands of the programmer. Also, while there are compilers on many platforms, code usually requires some customization for each platform.
- Java was a serious contender, with many strengths. The programming language is rigorously designed for large, object-oriented projects. Programs written in Java tend to be resistant to memory errors, thanks to an effective memory-management model that includes garbage collection. There are many existing Java subroutine libraries available to provide the programmer with additional resources. Finally, portability was a key goal in Java's design, and as a result, Java programs tend to run across several platforms. Java's downside relates to one of its upsides: The language's strict reliance on the object-oriented model makes learning the language difficult and writing small programs cumbersome.
- Perl has very strong built-in data parsing and reformatting functionality. Programs written in Perl are portable, running on many platforms, including all major UNIX variants and Windows versions. The language allows for easy procedural programming for smaller tasks, as well as object-oriented programming for larger projects. Also, there is a wide range of add-ons already available for Perl. This was important because it told us that programmers would have many resources available to them, and that creating such an add-on ourselves was likely to be easier and well documented. On the downside, for very large projects, Perl's object-oriented model is less rigorous than that of Java. Also, graphical user interfaces are harder to create in Perl than in Visual Basic or in some C/C++ toolkits.

After some consideration, Perl emerged as the winner. Visual Basic's design weaknesses and per-programmer licensing excluded it easily. C/C++ is too cumbersome. Java was a tempting option, but the ease with which researchers can throw together Perl programs made Perl our choice.

We implemented a Perl-AMPL library and packaged it as a module that is to be distributed via the Comprehensive Perl Archive Network (CPAN; <http://www.cpan.org/>). CPAN, a large collection of software and documentation that add extra functionality to Perl, is complemented by the Perl Automatic Upload Server (PAUSE). This server lets creators of Perl modules upload to CPAN, where their modules become available to Perl users for free. Because PAUSE makes it possible for Perl programmers around the world to contribute to CPAN, the archive contains thousands of Perl modules on a broad range of subjects: accessing different kinds of databases, communicating via different network protocols, interoperating with different data formats, and so on.

We need to mention here that there are some commercial products that fit the main requirements of the metamodeling system-design framework we have outlined in this article. For example, ILOG's OPL Studio (<http://www.ilog.com/products/oplstudio/>) includes the ability to build optimization models that are then accessed from a subroutine library using Visual Basic, Java, or C/C++.

The main program can generate data for the model dynamically, and the program can trigger the solving of the model and the generated data repeatedly. This indicates that OPL Studio includes the kind of functionality that we endorse for optimization meta-modeling; namely, allowing the use of a general-purpose programming language that can make optimization calls to a subroutine library. However, we believe that the existence of CPAN

---

## *The PerlAmpl module was designed as an object-oriented Perl class*

---

is an important reason for selecting Perl as the general-purpose programming language of the system. By creating a framework that lets a large number of programmers contribute code that they perceive as helpful, CPAN has created the open-source framework by which a product (in this case, Perl) can outgrow the imagination of its own original creators.

### **The PerlAmpl Module**

The *PerlAmpl* module was designed as an object-oriented Perl class. The *AMPL* object is an abstraction of an AMPL problem. The member functions let you set the problem's model, data, and options, after which member functions can be used to trigger the solver and access the results.

To use the API, include the module in your program with use *Math::Ampl*;, then initialize the library:

```
Math::Ampl::Initialize(inAmplDir, inAmplBin,
                      inTempDir, [inPreserveFiles]);
```

The argument *inAmplDir* is the directory containing the AMPL binary, *inAmplBin* is the name of the binary in that directory, *inTempDir* is a place to store temporary files, and *inPreserveFiles* is an optional argument (if 1, temp files are not deleted).

You next construct an instance:

```
$problem = new Math::Ampl;
```

Then, by using *\$problem->Function(Arguments)* syntax, you can use the following instance member functions:

- *Input\_Mod\_Append(inModText)* to append *inModText* to the *mod* data, which will be passed to AMPL as the model.
- *Input\_Mod\_Clear* to clear the *mod* data.
- *Input\_Dat\_Add\_Set\_Item(inSet, inDimension1, ...)* adds a set item to the *dat* data. *inSet* is the name of the set. *inDimension1* is the first dimension of the item, and so on.
- *Input\_Dat\_Add\_Param\_Item(inParam, inDimension1, ..., inValue)* to add a *param* item to the *dat* data. *inParam* is the name of the *param*. *inDimension1* is the first dimension of the item, and so on. *inValue* is the value.
- *Input\_Dat\_Clear* to clear the *dat* data.

Data file:						
U.S. TREASURY YIELD CURVE SUN, 4 JUL 1999, 11:32AM EDT						
Bills	Mat Date	Current Price/Yield	Previous Price/Yield	Yld Chg	Prc Chg	
3month	9/30/99	4.54(4.67)	4.54(4.67)	0.00	+0	
6month	12/30/99	4.77(4.97)	4.77(4.97)	0.00	+0	
1year	6/22/00	4.80(5.06)	4.80(5.06)	0.00	-0	
Notes/Bonds	Coupon	Current Mat Date	Previous Price/Yield	Price/Yield	Yld Chg	Prc Chg
2year	5.750	6/30/01	100-10+(5.57)	100-10+(5.57)	0.00	-0-00
5year	5.250	5/15/04	98-04(5.70)	98-04(5.70)	0.00	-0-00
10year	5.500	5/15/09	97-17(5.83)	97-17(5.83)	0.00	-0-00
30year	5.250	2/15/29	89-17+(6.01)	89-17+(6.01)	0.00	-0-00
[U.S. TREASURY YIELD CURVE]						
Inflation Indexed Treasury	Coupon	Current Mat Date	Previous Price/Yield	Price/Yield	Yld Chg	Prc Chg
5year	3.625	7/15/02	00 (.00)	99-06(3.91)	-3.91	-99-06
10year	3.875	1/15/09	00 (.00)	99-07+(3.97)	-3.97	-99-08
30year	3.875	4/15/29	00 (.00)	99-09+(3.91)	-3.91	-99-10
B(c) Copyright 1999, Bloomberg L.P. All Rights Reserved. B(c) Copyright 1999 USA TODAY, a division of Gannett Co. Inc.						

Figure 2: Example use of PerlAmpl parsing Treasury yield curve data.

$$\text{minimize } \left\{ \sum_{i=1}^n \sum_{j=1}^n \sigma_{ij} x_i x_j \mid \sum_{i=1}^n E[r_i] x_i \geq r_{\text{target}} \right\}$$

Example 2: Optimization problem.

- *Input\_Display\_Add(inDisplayItem)* to add an item to be retrieved from AMPL using *Display*. After solving, the values of all added items can be retrieved in Perl using *Output\_Display\_Get\_Text* or *Output\_Display\_Get\_Value*.
- *Input\_Display\_Clear* to clear the list of *Display* items to retrieve.
- *Input\_Option\_Add(inOption, inValue)* to add an option to be set during the *Solve* command, right before AMPL actually runs the solver. *inOption* is the option to set, *inValue* is the value.
- *Input\_Option\_Clear* to clear the options to set during *Solve*.
- *Solve* to run AMPL on the problem. It returns -1 if an error occurred, which prevented the solver from being called, and 0 if the solver was called but could not generate a solution. It returns 1 if the solver found a solution.
- *Solve\_Best\_Solver(inMinOrMax, inObjective, inSolver1, ...)* to run *Solve* for each *inSolver*, keeping track of the best results. *inMinOrMax* must be either "min" or "max." The function uses *inMinOrMax* to decide if one result is better than the next. After trying all the solvers, *Solve\_Best\_Solver* reruns the best solver to fill the *Display* variables with the best values (unless the best solver was already the last one, in which case, it does not need to be rerun). This function has been particularly useful for solving difficult nonlinear problems where the solution found may vary from solver to solver.
- *Output\_Display\_Get\_Value(inParamOrVar, inDimension1, ...)* to get the retrieved value of a variable or parameter. The *inParamOrVar* must have been added using *Input\_Display\_Add* before calling *Solve*.
- *Output\_Display\_Get\_Text(inParamOrVar)* to get the complete text block returned by AMPL for *inParamOrVar*. The *inParamOrVar* must have been added using *Input\_Display\_Add* before calling *Solve*.

The *PerlAmpl* module addresses many of the metamodeling issues mentioned here. The Perl environment facilitates the efficient handling of data and intermediate results, allows leveraging a wide variety of modules to maximize code reuse and minimize prod-

uct development time, and makes formatting end results easy. We illustrate the system's capabilities with several examples.

### Example: Handling Input

While the World Wide Web contains an enormous amount of data, it is frequently difficult to download them in the appropriate format. Perl modules, such as *Finance::Quote*, can be used directly to download stock price data from web sites, such as Yahoo Finance. Perl modules also exist for accessing spreadsheets and databases. For example, *DBD::Excel* lets you access the data within an Excel file from a database interface, while *Spreadsheet::Tie-Excel* lets you tie an array within a Perl program to an Excel spreadsheet.

When a module is not available from CPAN, Perl's strong regular expression engine makes parsing unformatted (or inconveniently formatted data) an easy task. Figure 2 and Listing 1 present an example use of *PerlAmpl* parsing Treasury yield curve data that was then used in a computational study of credit risky bonds. The subroutine *read\_file* reads in data on Treasuries downloaded from <http://www.bloomberg.com/> (see Figure 2) and stores the yields and maturities of the 3-month, 6-month, and 1-year bills in Perl lists *@gYieldList* and *@gPeriodList*, respectively (it ignores the other information).

### Example: Handling Intermediate Results and Formatting Output

We now show an example from finance—computing and plotting the mean-variance efficient frontier of a portfolio of three stocks—to illustrate how you could use the *PerlAmpl* module to run optimization problems multiple times, perform statistical analysis of data, plot graphs, and format output. To generate the efficient frontier, we need to solve the optimization problem in Example 2, where  $n$  is the number of stocks in the portfolio,  $x_i$  are decision variables corresponding to the allocation in each of the  $n$  stocks,  $r_{\text{target}}$  is the target portfolio return, and  $E[r_i]$  and  $\sigma_{ij}$  are the expected values and the covariances of asset returns  $i, j, i=1, \dots, n, j=1, \dots, n$ , respectively.  $\sigma_{ii}$  equals the variance of return  $i$ .

Lists of expected returns, target returns, and a covariance matrix are passed to the optimization problem formulation using the *PerlAmpl* module. The forecasts for expected returns could be generated in Perl, for example, by running regressions or using time series techniques. A number of statistics Perl modules, such as *Statistics::Regression*, available free from <http://www.cpan.org/>, can help with the statistical analysis of data. Also,

```

newgraph
xaxis
  label : Standard Deviation
yaxis
  label : Expected Portfolio Return
newcurve marktype none linetype solid linethickness 1
  label : Efficient Frontier
  pts 0.105347520141672 0.08 0.105347520141672
0.085 0.105347520141672 0.09 0.107279541386044
0.095 0.118133822421862 0.1 0.136400513195515
0.105 0.159553752697954 0.11 0.185775671173596
0.115 0.213940645974532 0.12
legend defaults
x 0.17 y 0.10

```

Example 3: Text contained in the file *efficientFrontier.jgr*.

Perl modules, such as *Math::Matlab*, provide additional capabilities by allowing for calling outside statistical software and collecting the results.

The portfolio optimization problem is nonlinear, so the FSQP solver is called from within AMPL. The results are stored in a hash in Perl. JGraph (a free graphing software project under Linux; <http://sourceforge.net/projects/jgraph/>) plots the results dynamically. The advantage of this type of organization is that the whole metamodel can be run efficiently from beginning to end while keeping the same output format. Output formats that are frequently convenient, for example, are Latex table, Excel, or HTML. Listing 2 contains a subroutine for printing the optimal standard deviation for each level of target portfolio return directly in a Latex table format.

The output of this program is a file *efficientFrontier.jgr* and this table:

```

Expected Return & Standard Deviation \\
0.080 & 0.105 \\
0.085 & 0.105 \\
0.090 & 0.105 \\
0.095 & 0.107 \\
0.100 & 0.118 \\
0.105 & 0.136 \\
0.110 & 0.160 \\
0.115 & 0.186 \\
0.120 & 0.214 \\

```

The *efficientFrontier.jgr* file contains the text in Example 3. Using the command *jgraph efficientFrontier.jgr > efficientFrontier.ps*, the file can be converted to a picture (Figure 3), which can then be used, for example, in a Latex file. The conversion of the Jgraph file to a PostScript file can be automated by including it directly in the Perl program.

### Example: Handling Intermediate Results And Formatting Output

We successfully used the PerlAmpl module in computational comparisons of the efficiency of multiperiod versus single-period portfolio optimization techniques. The module can be similarly used in other optimization metamodels that involve simulations at multiple stages.

Consider an example of a multiperiod optimization problem: Given a portfolio of stocks and information about their future expected values at multiple times in the future, solve two different portfolio optimization problem formulations to determine optimal portfolio allocations for each time period. Then run simulations to test which of the two allocations results in a better final period portfolio return. To compare the two optimization formulations, you would need to compare cumulative returns at the end of the time horizon. At every point in time, you need to

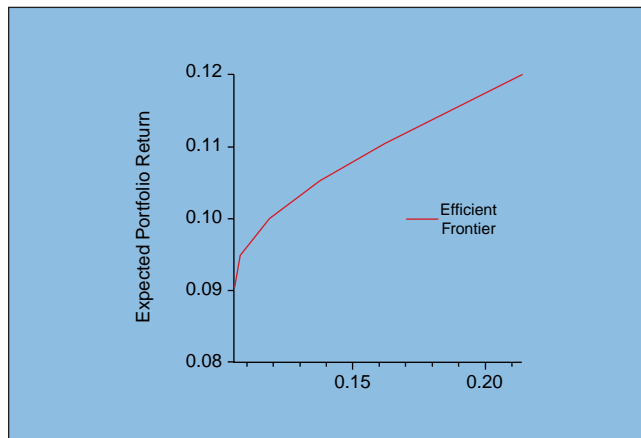


Figure 3: Converting file to a picture.

keep track of the portfolio value, rebalance the portfolio, simulate returns for one period ahead, and recompute the value of the portfolio.

AMPL contains several built-in random generator functions, such as *Beta(a,b)*, *Cauchy()*, *Exponential()*, *Normal(mean, standard deviation)*, *Poisson(parameter)*, and *Uniform(lower limit, upper limit)*. All of the previous random distributions can also be generated in Perl by using, for example, the *Math::Cephes* Perl module, which contains over 150 mathematical functions. In some contexts, such as the multiperiod portfolio optimization example we mention, it is significantly more convenient to run the simulations within the main Perl program because of the need to process intermediate optimization results. Moreover, Perl offers a wider variety of random generator functions. Listing 3 is Perl code for a simplified simulation example in which the asset returns in a portfolio are drawn from a multivariate normal distribution with prespecified expected values vector and covariance matrix. Currently, there is no provision for generating correlated random variables from within AMPL. We use the Perl library *random\_multivariate\_normal.pl* (<http://www.cpan.org/>).

### Conclusion

While the PerlAmpl module is effective and useful in its current state of development, possible enhancements include parallelizing the execution of solvers to take advantage of multiprocessors, as well as networking the PerlAmpl module to allow client computers to solve optimization metamodels by leveraging software on server computers.

TPJ





## Listing 1

```
%kMonthHash = ("Jan" => 1, "Feb" => 2, "Mar" => 3, "Apr" => 4, "May" => 5,
               "Jun" => 6, "Jul" => 7, "Aug" => 8, "Sep" => 9, "Oct" => 10,
               "Nov" => 11, "Dec" => 12);
%kConvertHash = ("3month" => 90, "6month" => 180, "1year" => 360,
                 "2year" => 720, "5year" => 1800, "10year" => 3600,
                 "30year" => 10800);

sub read_file
{
    my($inFile) = @_;
    my($state, $temp, $period, $yield);
    open(INPUT, $inFile) || die("Cannot open $inFile");
    $state = 0;
    while (<INPUT)
    {
        if ($state == 0)
        {
            #if (/U.S. Treasury yield curve/)
            if (/^U.S. Treasuries$/)
            {
                $state = 1;
            }
        }
        elsif ($state == 1)
        {
            #READ IN CURRENT DATE
            # Sun, 4 Jul 1999, 11:32am EDT
            if (/[A-Z][a-z][a-z]\s*([0-9]+)\s*([A-Z][a-z][a-z])\s*([0-9]+)/)
            {
                $gDay = $1;
                #assign a numerical value to the month according to %kMonthHash
                $gMonth = $kMonthHash{$2};
                $gYear = $3;
                $state = 2;
            }
            else
            {
                die("cannot parse date");
            }
        }
        elsif ($state == 2)
        {
            #if (/Bills/)
            if (/Prc Chg/)
            {
                $state = 3;
            }
        }
        elsif (($state == 3) || ($state == 5))
        {
            if ((/Notes/) || (/U.S. Treasury Yield Curve/))
            {
                $state++;
            }
            elsif (/^(\s+)\s+([^\s]+)\s+$/) #READ IN RELEVANT DATA
            {
                $period = $kConvertHash{$1};
                push(@gPeriodList, $period);
                $yield = $2;
                push(@gYieldList, $yield);
            }
            elsif (/s*[+-][0-9+-]\s*$/)
            {
                # do nothing
            }
            else
            {
                die("cannot parse data: $_");
            }
        }
        elsif ($state == 4)
        {
            #if (/Bonds\s+Coupon\s+Mat\s+Date/)
            if (/Prc Chg/)
            {
                $state = 5;
            }
        }
    }
    close(INPUT);
}
```

## Listing 2

```
use Math::Ampl;
use strict;

#SPECIFICATION OF INPUT DATA:
my($NumStocks) = 3;
#vector of expected returns
my(@ExpectedReturnsList) = (0.08, 0.09, 0.12);
#vector of standard deviations for each stock
my(@StdDeviationsList) = (0.15, 0.20, 0.22);
#vector of target portfolio returns
my(@TargetReturnsList) = (0.08, 0.085, 0.09, 0.095, 0.10, 0.105,
                          0.11, 0.115, 0.12);

#RESULTS STORAGE:
#hash table to store optimal portfolio standard deviation results after
#solving all optimization problems
my(%OptimalStdDeviationsHash) = ();
#hash table to store optimal portfolio holdings after solving the
#portfolio optimization problem for each value of TargetReturn
my(%OptimalHoldingsHash) = ();

#OUTPUT FILES:
my($kOutputFile) = "efficientFrontier.out"; #to store Latex table
my($kFrontierGraphFile) = "efficientFrontier.jgr"; #to store graph

#DECLARATION OF AMPL PROBLEM INSTANCE:
my($gProblem);

#optimization model problem to be solved: minimize portfolio variance
#subject to constraints on portfolio expected return

#optimization model file to be passed to AMPL using PerlAmpl

sub setup_ampl_mod
{
    my($mod);
    $mod =<<EODATA;

    param NumStocks;
    param ExpectedReturns[1..NumStocks];
    param StdDeviations[1..NumStocks];
    param TargetReturn;

    var holdings[1..NumStocks];

    minimize portfolio_variance:
    sum{i in 1..NumStocks}
        (StdDeviations[i]*StdDeviations[i]*holdings[i]*holdings[i]);

    subject to portfolio_target_return:
    sum{i in 1..NumStocks} (ExpectedReturns[i]*holdings[i]) >= TargetReturn;

    subject to portfolio_total:
    sum{i in 1..NumStocks} (holdings[i]) = 1;
EODATA
    $gProblem->Input_Mod_Append($mod);
}

#optimization data file to be passed to AMPL using PerlAmpl
sub setup_ampl_dat
{
    #target portfolio return passed for this instance of the problem
    my($inTargetReturn) = @_;
    my($iStock);

    $gProblem->Input_Dat_Add_Param_Item("NumStocks", $NumStocks);
    $gProblem->Input_Dat_Add_Param_Item("TargetReturn", $inTargetReturn);
    for($iStock = 1; $iStock <= $NumStocks; $iStock++)
    {
        $gProblem->Input_Dat_Add_Param_Item("ExpectedReturns",
                                           $iStock, $ExpectedReturnsList[$iStock-1]);
        $gProblem->Input_Dat_Add_Param_Item("StdDeviations",
                                           $iStock, $StdDeviationsList[$iStock-1]);
    }
}

#request for AMPL to keep track of variables of interest

sub setup_ampl_display
{
    $gProblem->Input_Display_Clear;
    $gProblem->Input_Display_Add("solve_result");
    $gProblem->Input_Display_Add("portfolio_variance");
    $gProblem->Input_Display_Add("holdings");
}

#script for running the problem in AMPL and obtaining the results
```

```

sub solve_problem
{
    #target portfolio return passed for this instance of the problem
    my($inTargetReturn) = @_;
    my($solved);

    $gProblem->Input_Mod_Clear;
    $gProblem->Input_Dat_Clear;
    setup_ampl_dat($inTargetReturn);
    setup_ampl_mod();
    setup_ampl_display();
    $solved = $gProblem->Solve;
    return $solved;
}

#PRINT GRAPH OF EFFICIENT FRONTIER DYNAMICALLY
#the output is a .jgr file which can then be converted to a .ps file

sub print_graph
{
    my($graph);
    my($targetReturn); #portfolio target return, read from list
    my($portfolioVariance); #optimal result from optimization problem
    my($portfolioStdDeviation); #to be computed from portfolio variance
    my(@currentHoldings); #obtained from AMPL output
    my($iStock); #counter

    open(GRAPH, ">$kFrontierGraphFile") ||
        die ("Cannot open file \"$kFrontierGraphFile\" for graph: $!");
    $graph = "newcurve marktype none linetype solid linethickness 1";
    $graph .= " label : Efficient Frontier\n";
    $graph .= "\tpts ";
    foreach $targetReturn (@TargetReturnsList)
    {
        solve_problem($targetReturn);
        $portfolioVariance =
            $gProblem->Output_Display_Get_Value("portfolio_variance");
        $portfolioStdDeviation = sqrt($portfolioVariance);
        $graph .= " $portfolioStdDeviation $targetReturn ";
    }
    #store optimal standard deviations if necessary
    $OptimalStdDeviationsHash{"$targetReturn"} = $portfolioStdDeviation;
    #if necessary, store also the optimal holdings for each value of TargetReturn
    for($iStock = 1; $iStock <= $NumStocks; $iStock++)
    {
        $CurrentHoldings[$iStock-1] =
            $gProblem->Output_Display_Get_Value("holdings",$iStock);
        $OptimalHoldingsHash{"$targetReturn,$iStock"} =
            $CurrentHoldings[$iStock-1];
    }
}

print GRAPH "newgraph\n xaxis\n label : Standard Deviation\n";
print GRAPH "yaxis \n ";
print GRAPH "label : Expected Portfolio Return\n";
print GRAPH "$graph\n\n";
print GRAPH "legend defaults\n x 0.17 y 0.10\n";
close GRAPH;
}

#PRINT A TABLE WITH RESULTS IN LATEX TABLE FORMAT
sub print_table
{
    my($portfolioStdDeviation);
    my($targetReturn);

    open(OUTPUT, ">$kOutputFile") ||
        die ("Cannot open file \"$kOutputFile\" with results: $!");
    printf OUTPUT "%s %s ", "Expected Return", "&";
    printf OUTPUT "%s %s \n", "Standard Deviation", "\\\\";

    foreach $targetReturn (@TargetReturnsList)
    {
        $portfolioStdDeviation = $OptimalStdDeviationsHash{$targetReturn};
        printf OUTPUT "%2.3f %s ", $targetReturn, "&";
        printf OUTPUT "%2.3f %s \n", $portfolioStdDeviation, "\\\\";
    }
    close OUTPUT;
}

#MAIN
#initialization
Math::Ampl::Initialize($kAmplDir, $kAmplBin, $kTempDir, 1);
$gProblem = new Math::Ampl;

print_graph();
print_table();

```

### Listing 3

#Generates one path, and returns a List of Lists indexed by [time  
#period, asset number]. The entries equal single-period returns for  
#each stock. Single-period returns are multivariate normal random variables.

```

sub create_scenario
{
    #pass number of stocks in portfolio and number of time periods ahead
    my($inNumStocks, $inNumPeriods) = @_;

    my(@SimulatedReturnsList); #single time period simulated returns
    my(@ScenarioLoL); #list of lists of asset returns for each time period
    my($iT); #time period counter

    for ($iT = 0; $iT < $inNumPeriods; $iT++)
    {
        @SimulatedReturnsList =
            random_multivariate_normal($inNumStocks,
                @ExpectedReturnsList, @CovarianceMatrixList);
        #add simulated returns for time period iT to scenario path
        @ScenarioLoL[$iT] = [ @SimulatedReturnsList ];
    }
    return (@ScenarioLoL);
}

```

TPJ

## Subscribe Now To

# The Perl Journal

For just \$18/year,  
you get the latest in  
Perl programming  
techniques in e-zine  
format from the  
world's best Perl  
programmers.

<http://www.tpj.com/>

# Perl, VMWare, and Virtual Solutions

**K**rang is an open-source content-management system that runs on a variety of operating systems, including various Linux and BSD flavors. To make Krang easy to install, we produce binary distributions for each supported platform. For example, if you're running Redhat Linux 9 with Perl 5.8.4, you would download this file to install Krang:

```
krang-1.020-Redhat9-perl5.8.4-i686-linux.tar.gz
```

A binary distribution contains the Krang code, written in Perl, an Apache/mod\_perl web server, and all the external Perl modules needed to run Krang. All you need to supply is a base operating-system installation and MySQL 4.0.13 or later. If there are other requirements on your platform, they're listed in a platform-specific README file.

As we ported Krang to more and more platforms, generating these binary distributions for each release became a real chore. Each build would have to be performed by hand on each target platform. Worse, each distribution needs to be tested to make sure Krang still works on all our platforms.

My solution is the Krang Farm (<http://krang.sf.net/>), an automated build and test system created using VMWare GSX Server and Perl. With the Krang Farm, I can enter a single command to build and test Krang across all our supported platforms, all on a single physical machine. In this article, I'll explain how the farm works and how you can create your own virtual farms using VMWare GSX Server and Perl.

## VMWare Background

VMWare (<http://vmware.com/>) produces virtual-machine software. A virtual machine is basically a simulated machine that runs its own operating system. Each virtual machine has its own virtual hardware (processor, hard drive, video card, and so on), and is completely separate from the host system. The real machine that runs the VMWare software is known as the "host," while the virtual machines are "guests." Thus, the term "guest operating system" refers to the operating system running inside a virtual machine.

VMWare's software isn't an emulator like Bochs or Virtual PC. Instead, VMWare uses the built-in virtualization hardware in all x86 processors. This lets the virtual machines run nearly as fast as software that runs directly on the real hardware. However, it does have one major drawback—the virtual machines are all x86 systems. VMWare won't run PowerPC or Sparc virtual machines, for example. (See the sidebar "Why Not Build a Real Farm?")

The first VMWare product I used was VMWare Workstation, which I use to run Microsoft Windows on my Linux desktop.

---

*Sam is a Perl programmer at the PIRT Group (<http://thepirgroup.com/>) and author of Writing Perl Modules for CPAN (Apress, 2002). He can be contacted at [sam@tregar.com](mailto:sam@tregar.com).*

When I need to test a web application under IE or open an Excel spreadsheet that won't work with OpenOffice, VMWare Workstation is indispensable. A coworker uses VMWare Workstation in the reverse situation—to do development on a Linux guest running VMWare on a Windows host.

I've also used VMWare Workstation to do some software development for the Bricolage project. I set up virtual machines running a few guest operating systems supported by Bricolage: FreeBSD, Debian Linux, and Redhat Linux. I used these machines to test the Bricolage installation system while it was under development.

However, for this project, VMWare Workstation just won't work. First, it requires a local X server and won't work over a network connection. Second, it doesn't offer a scripting interface to automate operations on the virtual machines.

VMWare's entry-level server product, VMWare GSX Server, remedies both these deficiencies. It works on a headless server and provides a complete scripting system via the VMPerl and VMCOM APIs. This is the product I used to produce the Krang Farm. The rest of the article will refer to VMWare GSX Server simply as "VMWare."

VMWare can be controlled through two interfaces—the web-based VMWare Management Interface and the Virtual Machine Console rich client. Figure 1 shows the VMWare Management Interface running in the Firefox web browser. This interface lets you provision new machines, reconfigure existing ones, start machines, and stop machines. It also has links to download the Virtual Machine Console, which runs as a native GUI on Linux and Windows.

The Virtual Machine Console is the interface used to set up a new machine because it gives you access to the virtual console. Users of VMWare Workstation will be at home in the interface as it is virtually identical. The only significant difference is the need to log in to the server before accessing the guest systems. Figure 2 shows the Virtual Machine Console accessing a Redhat Linux 9 guest at the beginning of the installation process.

## Scripting VMWare

VMWare offers two APIs—VMPerl for Perl scripting on Linux and Windows, and VMCOM for COM programming on Windows only. Since I'm primarily a Perl programmer, I went straight to the VMPerl API.

Example 1 shows a simple script called "enumerate\_vms.pl" that lists all available virtual machines. Example 2 shows the result of running this script on my VMWare server.

As you can see, VMPerl is an object-oriented API. Example 1 works by creating a `VMware::VmPerl::Server` object and calling `connect()` on it. The `connect()` method takes a `VMware::VmPerl::ConnectParams` object, which I created using the `new()` method. Not passing any arguments to `new()` causes `VMware::VmPerl::ConnectParams` to use all the default values for hostname, port, username,



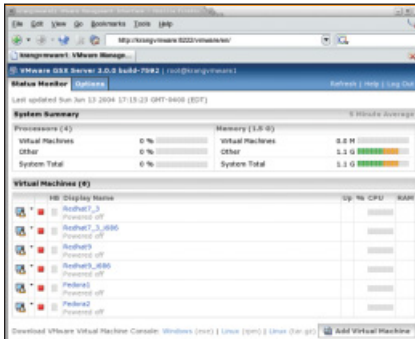


Figure 1: VMWare Management Interface running in the Firefox browser.

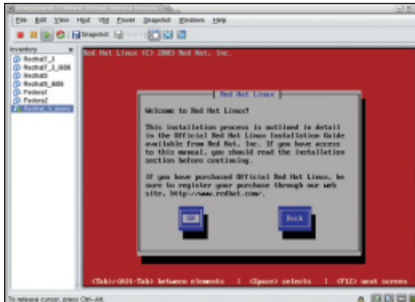


Figure 2: Virtual Machine Console accessing a machine installing Redhat Linux 9.

and password. If this script were run on a different machine from the VMWare installation, then these values would have to be filled in. Finally, a call is made to the `registered_vm_names()` method on the server object. This returns a list of virtual machine identifiers, which are in fact the paths of configuration files.

The VMPerl API contains methods for starting and stopping machines, as well as a mechanism for exchanging data with programs running inside the guest operating system. Throughout, I found the API to be well designed and easy to pick up. The excellent documentation and plentiful examples that come with the software are a great help.

## Time Travel

Software testing becomes much harder if the results of one test run can effect the results of the next run. It is very hard to ensure this won't happen because bugs have a way of defying your specifications—very hard, that is, unless you can travel back in time. Imagine setting up your system in a “known-good state.” Then, every time you want to run a test, you just travel back in time and run your test on your system in that state.

VMWare lets you do just that. VMWare's virtual storage system has a “nonpersistent” mode. This means that every time the virtual machine is booted, its hard drive has the contents it had when it went into nonpersistent mode, the contents of the known-good state. During the run, changes are made on disk, but nothing lasts beyond the next boot. This gives each test run a clean slate. No matter how badly it bombs, there's no way it can affect the next run.

## Running Commands On Virtual Machines

So far, I've described how new machines are created and how they can be controlled programmatically via the VMPerl API. The final piece needed to run automated tests and builds on virtual machines is a way to run commands and get output from software running on the machines.

The simplest way to solve this problem is to use a network connection to control a shell session running on the virtual machine. Telnet and RSH would have worked fine, but I used SSH because most modern operating systems have SSH running after installation.

I used the *Expect* module to interact with shell sessions running inside the guest operating systems (<http://www.cpan.org/authors/id/R/RG/RGIERSIG/>). *Expect* is a Perl implementation of the venerable *Expect* TCL system created by Don Libes and it operates much the same way. The Perl implementation was writ-

```
#!/usr/bin/perl -w
use VMware::VmPerl::Server;
use VMware::VmPerl::ConnectParams;

# connect to the server using all the default settings
$server = VMware::VmPerl::Server::new();
$server->connect(VMware::VmPerl::ConnectParams::new()) or
die "Could not connect to server: ", ($server->get_last_error())[1];

# get a list of virtual machines
@vm_list = $server->registered_vm_names();
die "Could not get list of VMs from server: ", ($server->get_last_error())[1]
unless @vm_list;

# print them out
print "$_\n" for @vm_list;
```

Example 1: `enumerate_vms.pl` lists all virtual machines.

ten by Austin Schutz and Roland Giersig. You provide *Expect* a command to spawn and it handles setting up a pseudoterminal (pty) for the new process. Then you can run pattern matches on the output of the command and send the command input.

Example 3 shows via ssh how to run the *date* command on another machine (it could be virtual or real). When prompted, the script provides the user's password. The script then captures the output from *date* and prints it. When I run this script against my Redhat Linux 9 virtual machine (conveniently named Redhat9), I see:

```
$ ./get_date.pl
The date on Redhat9 is Thu Jun 17 08:41:24 EDT 2004
```

The VMPerl API includes an alternative mechanism for communicating with software running on the guest OS, using `set_guest_info()` and `get_guest_info()`. I did not use these methods for two reasons: First, they lack sufficient flexibility to control an interactive process easily. Second, to use them, you must install the VMWare Tools software on the guest operating system. This would add an extra step to the process of setting up a new machine, one that is otherwise unnecessary.

## Putting It All Together

With the VMPerl API to start and stop machines and the *Expect* module to run commands via ssh, the rest is, as a former boss used to say, merely a simple matter of programming.

I started by designing a configuration file format to contain a description of all the machines in the farm. The format follows the same conventions as the Apache web server, made possible by the *Config::ApacheFormat* module created as part of the Krang project (<http://www.cpan.org/authors/id/S/SA/SAMTREGAR/>). Example 4 is a single machine's configuration from the file, *farm.conf*. Each machine in the farm gets a <Machine> block listing the username and password used to log in to the machine, a short description used in script output, and a list of all the Perl binaries on the machine paired with the Krang builds they generate.

One thing you might have expected to see in *farm.conf* that isn't there is the IP address of the machine. I decided to store that information in */etc/hosts* because it is convenient to be able to ssh to the machine manually to debug problems. For example, the machine in Example 4 has a corresponding entry in */etc/hosts* like this:

```
192.168.1.10 Redhat7_3_i686
```

With configuration out of the way, I created a class to encapsulate operations on the machines called *KrangFarm::Machine*. Example 5 shows a script that will start each machine on the farm and execute the *date* command. Notice how *KrangFarm::Machine* completely abstracts interaction with the VMPerl API and the farm configuration file.

The actual scripts in the Krang Farm system, *krang\_farm\_build* and *krang\_farm\_test*, aren't much more complicated than Example 5. The build script, *krang\_farm\_build*, transfers a source tar-ball to

each machine, runs *make build* and *make dist* and fetches the resulting binary distribution. The test script transfers a binary distribution to each machine, installs it, runs *make test*, and parses the output to determine success or failure.

Building and testing on all configured platforms is as simple as:

```
krang_farm_build --
from-cvs && krang_farm_test -- version 1.020
```

## Plans for the Future

The Krang Farm is working well, but there's always room for improvement. Now that building and testing are automated, I plan to add a script to perform test runs automatically every night against the latest source in CVS. If the tests fail, the script will send mail to the Krang development mailing list.

```
$ perl enumerate_vms.pl
/var/lib/vmware/Virtual Machines/Redhat7_3-0/Redhat7_3.vmx
/var/lib/vmware/Virtual Machines/Redhat7_3_i686/Redhat7_3_i686.vmx
/var/lib/vmware/Virtual Machines/Redhat9/Redhat9.vmx
/var/lib/vmware/Virtual Machines/Redhat9_i686/Redhat9_i686.vmx
/var/lib/vmware/Virtual Machines/Fedora1/Fedora1.vmx
/var/lib/vmware/Virtual Machines/Fedora2/Fedora2.vmx
```

Example 2: Output from *enumerate\_vms.pl*.

```
#!/usr/bin/perl -w
use Expect;

# connection parameters
$SERVER = 'Redhat9';
$USER   = 'krang';
$PASS   = 'krang';

# spawn the date command on $SERVER running as $USER
my $spawn = Expect->spawn(qq(ssh $USER@$SERVER date))
or die "Unable to spawn ssh.\n";
$spawn->log_stdout(0);

# provide the password when prompted, waiting up to 5 seconds
if ($spawn->expect(5, 'password:')) {
    $spawn->send($PASS . "\n");
}

# wait for the date and print it out
if ($spawn->expect(5, -re => qr/^\d{4}\r?\n/)) {
    print "The date on $SERVER is " . $spawn->match();
}
```

Example 3: *get\_date.pl* gets the date via ssh.

```
# each machine gets a Machine block
<Machine Redhat7_3_i686>

# a reminder of what's on this machine
Description "Redhat 7.3 Server w/ custom Perls for i686"

# the user and password the farm system will use to login, needs sudo
User krang
Password krang
# the Perl binaries and the builds they generate
Perls /usr/bin/perl Redhat7_3-perl5.6.1-i686-linux \
/usr/local/bin/perl5.6.2 Redhat7_3-perl5.6.2-i686-linux \
/usr/local/bin/perl5.8.3 Redhat7_3-perl5.8.3-i686-linux \
/usr/local/bin/perl5.8.4 Redhat7_3-perl5.8.4-i686-linux
</Machine>
```

Example 4: A single machine's configuration block.

```
#!/usr/bin/perl -w
use lib '/home/sam/krang-farm/lib';
use KrangFarm::Machine;

# loop through all configured machines
foreach $name (KrangFarm::Machine->list()) {
    $machine = KrangFarm::Machine->new(name => $name);
    $machine->start();

    # call the date command and extract the output
    $spawn = $machine->spawn(command => 'date');
    if ($spawn->expect(5, -re => qr/^\d{4}\r?\n/)) {
        print "The date on $name is " . $spawn->match();
    }

    # stop the machine
    $machine->stop();
}
```

Example 5: Script that starts each machine on the farm.

Another possible enhancement would be a system to test upgrades from one version of Krang to another. Krang includes an upgrade facility that lets users move from one version of Krang to another without losing their data. Testing upgrades from an arbitrarily old version of Krang to the latest release is a time-consuming process, and automating it could help us find upgrade bugs faster.

This work may well be done by the time you read this article; drop by the Krang web site to find out, or even to lend a hand! Like Krang, the Krang Farm software is 100 percent open source.

## Problems

For all the cheerleading in this article, the project wasn't without problems. One of Krang's supported platforms, Fedora Linux, isn't officially supported by VMWare. While Fedora Core 1 worked great, Fedora Core 2 ran extremely slowly under VMWare. Compiling Krang took around an hour versus 10 minutes on the rest of the machines. I eventually solved this problem by compiling a new kernel on the Fedora Core 2 machine with a lower value for HZ (100 versus the default of 1000), a tactic I found on the VMWare community message boards.

Additionally, I am unable to start machines as a nonroot user in the Virtual Machine Console. Judging by the documentation, I'm sure this is supposed to work, but I get a fatal error whenever I try it. However, starting machines using the VMPerl API as a nonroot user works fine.

TPJ

## Why Not Build a Real Farm?

The biggest reason to build a virtual farm rather than a real farm is cost. At my company, we can host and administrate a single machine for approximately \$2400 per year. The cheapest new machine that meets our hosting requirements costs at least \$3000. Thus, to build a farm of six machines costs \$18,000 up-front and \$14,400 per year. Each new machine will cost another \$3000 plus \$2400 per year.

Contrast this with the virtual farm. The machine running the Krang farm costs \$4275; it's a ProLiant DL360 G3 from Hewlett-Packard with dual 2.8-GHz Intel Xeon processors and 1.5 GB of RAM. The VMWare GSX Server software costs \$2500. It costs \$2400 per year to host and administrate, just like any other machine on our network. Thus, the total cost to set up the virtual farm is \$6775 and \$2400 per year. That's already much lower than setting up six machines in a real farm. But even better, adding a new machine to the virtual farm is free. It doesn't require any new hardware or administration overhead.

Another reason to build a virtual farm is the added flexibility. Adding new machines to a real farm takes time; a new machine must be ordered and set up on the network. Adding a new machine to the virtual farm can be done in just a few hours, depending only on how long the operating-system installation takes to run.

However, a real farm has advantages over a virtual farm. First, it's likely to be faster. Because all the machines in the farm can run independently, it will scale better as machines are added. Since run-time performance isn't very important for a build and test farm, this wasn't an issue for the Krang Farm project. Second, a single machine means a single point of failure. It's much more important to have a working backup system when all your eggs are in one basket!

—S.T.



# Sorting Out the Linguistic Mess

*Simon Cozens*

I have a friend who works for a web consultancy putting together all kinds of web sites and applications for clients. One of his latest projects was an internationalized dictionary. He was having some problems with the Japanese terms, so he wanted me to have a look at it.

Well, we got the terms sorted out, but sensing the opportunity of a subcontract, I said “You know, a Japanese dictionary wouldn’t be in this order.” And it wouldn’t. He had sorted the words with a simple *sort* routine, which would sort them according to their Unicode codepoint. Japanese dictionaries don’t work like that; Japanese people don’t have a mental mapping of characters to Unicode codepoints they can consult when they want to look up a word.

But Japanese is an extreme case: Even European languages sort in ways we might not expect. The point is that, in general, *sort* just isn’t good enough for sorting. Collation (the technical term for sorting) of nonEnglish text is tricky, but we’ll look at a few ways to make it easier.

## ArbBiLex—Simple Two-Level Sort

How complicated your sorting needs to be depends on the languages you’re dealing with. We’ll look first at a nice, simple language like Spanish. Spanish sorts just like English, except that it has “ch” as a separate letter after “c,” “ll” after “l,” “ñ” after “n,” and “rr” after “r.”

To be honest, this ordering is pretty purist. In 1994, the Real Academia Española officially dropped the separate diglyph letters “ch,” “ll,” and “rr,” partly because it made sorting difficult for computer programmers. We’re now going to prove them wrong.

There are two approaches we could take to this collation problem. We could replace all the new characters with sentinels: We replace “ch” with “c{” or something similar, meaning “chorizo” will become “c{orizo.” Now we do a standard lexicographic sort, and “c{orizo” will sort after “cz” but before “d.” Then once we’ve finished, we can replace them all back again. This is messy, and requires us to pick our sentinels carefully so that they don’t interfere with the rest of our text.

The alternate approach is to write our own *sort* routine. However, that’s sufficiently boring so Sean Burke has written a module to write *sort* routines. *Sort::ArbBiLex* allows you to construct efficient *sort* functions that understand nonASCII orders and multi-character glyphs.

---

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

The easiest way to use *Sort::ArbBiLex* is to specify on the import line the name of a subroutine that you’d like it to create, and then a space-separated list of characters that makes up your alphabet.

So, for our Spanish alphabet, we can say:

```
use Sort::ArbBiLex ( spanish_sort =>
"a A b B c C ch CH d D e E f F g G h H i I j J k
K l L ll LL m M n N ñ o O p P q Q r R rr RR s
S t T u U v V x X y Y z Z");
```

This will import a function called *spanish\_sort* into our namespace, which will act just like the built-in *sort*, but will sort in the order described.

That was the easy bit. We call it a one-level sort, since you’re only interested in looking at one character at a time. I forgot to say that Spanish also has some accent marks, which may well appear in your text. These don’t actually affect the sorting at all: “á” sorts the same as “a.”

What we want to be able to do is group the accented and/or capitalized letters together for the purposes of sorting. This produces a two-level sort. We could describe this with the following Perl data structure:

```
[qw/ a A á Á /],
[qw/ b B /],
[qw/ c C /],
[qw/ ch CH /],
[qw/ d D /],
[qw/ e E é É /],
...
]
```

Fortunately, instead of giving *Sort::ArbBiLex* a plain string as a description of an alphabet, we can also pass an array of arrays just like this:

```
use Sort::ArbBiLex ( spanish_sort =>
[
[qw/ a A á Á /],
[qw/ b B /],
[qw/ c C /],
[qw/ ch CH /],
[qw/ d D /],
[qw/ e E é É /],
...
]
);
```



That's one language (and many others like it) out of the way.

## Sorting Thai

Unfortunately, not all languages are this easy. Thai, for instance, is an alphabetic language—that is, each character represents a sound—but it has some odd sorting rules. Initial vowels before the first consonant are not counted when sorting Thai words, and tonal accents should be ignored.

---

*In general, sort just isn't good enough for sorting*

---

In order to sort Thai words, then, we're going to have to use one of the best known Perl sorting techniques, the Schwartzian transform. This comes in handy every time you have to transform a value in some complex way to sort it, and then get the original value back. First we construct a list of pairs: the original value, and the transformation. Let's say we're going to sort a list of English words but discount all their vowels:

```
@pairs = map {  
    my ($vowelless = $_) =~ tr/aeiou//d;  
    [ $_, $vowelless ]  
} @words;
```

We use *map*, naturally, to transform a list. Now, if we wanted to get the original words back, we could say:

```
@originals = map { $_->[0] } @pairs
```

picking up the first element of each pair. But before we do that, we could sort the list on the transformed value:

```
@sorted = map { $_->[0] }  
    sort { $a->[1] cmp $b->[1] } @pairs
```

using the second element.

But *@pairs* is just a temporary variable used for the transformation, so we could say this:

```
@sorted = map { $_->[0] }  
    sort { $a->[1] cmp $b->[1] } @pairs  
    map { ($copy = $_) =~ tr/aeiou//d;  
        [ $_, $copy ]  
    } @words;
```

This *map-sort-map* idiom is the hallmark of the Schwartzian. We can generalize this for our linguistic sorting: If we have a function *sort\_key* that transforms a word into some kind of regular, sortable string, then we can sort any language, like so:

```
@sorted = map { $_->[0] }  
    sort { $a->[1] cmp $b->[1] } @pairs  
    map { [ $_, sort_key($_) ] } @words;
```

All we need to do now is find an adequate *sort\_key* for Thai. Here's one I made earlier. It starts with some definitions gleaned from the Unicode Standard:

```
my $thai_tone = qr/[x{0e48}-x{0e4b}]/;  
my $thai_vowel = qr/[x{0x40}-x{0e45}]/;
```

and now:

```
sub sort_key {  
    my $word = shift;  
    $word =~ s/$thai_tone//g;  
    $word =~ s/^(($thai_vowel)+//;  
    return $word;  
}
```

(Another Thai sorting algorithm, more useful for languages without Perl's string handling, can be found at <http://linux.thai.net/~thep/tsort.html>.)

## Sorting Han Languages

The languages we've looked at so far have been alphabetic—each character has a well-defined “reading.” Unfortunately, not all languages are like that. The so-called Han languages—Chinese, Japanese, and to a much lesser extent, Korean and Vietnamese—use the Chinese character set “hanzi” (“kanji” in Japanese). There are between forty and eighty thousand kanji, and their readings are dependent on context: The characters can have multiple readings and can also be used to convey a meaning as well as a sound.

This naturally makes it rather difficult to sort words: If you want to sort them by sound, you have to work out what that sound is, and this can't necessarily be looked up in a table because multiple sounds are possible. There are two ways around this; the Chinese way and the Japanese way.

The Chinese way is very simple: Don't even bother trying to sort by sound; sort by some other property. So some indexes of Chinese books sort their terms based on the number of strokes used to write each hanzi character. There's a freely available kanji dictionary that contains stroke-count information, so if we want to do the same, we can use the *Lingua::JP::KanjiDic* module to interface to it:

```
use Lingua::JP::KanjiDic;  
my $reader = Lingua::JP::KanjiDic->new;  
  
sub sort_key {  
    my $word = shift;  
    my $key;  
    for my $char (split //, $word) {  
        my $kanji = $reader->lookup($char);  
        next unless $kanji;  
        $key .= chr(96 + $kanji->strokes);  
    }  
    return $key;  
}
```

Here, we look up each character in the dictionary, which returns a kanji object. The object contains all sorts of information, including the stroke count. We convert it into a letter: Character 97 is “a,” so a one-stroke hanzi would be “a.” There aren't any characters with more than 26 strokes, so we have a nice readable key representing the number of strokes in each character of the word.

The Japanese method, however, is a bit more difficult. The Japanese use two alphabets as well as kanji, and sorts words—even kanji—based on the phonetic order used in these alphabets. So, in order to sort a list of Japanese words, you have to find out how they're pronounced, and really, there's no easy way to do this.

All is not lost. The folks at Nara University have produced a piece of free software called *chasen*; this is a morphological analyzer that takes apart sentences and words, puts them in their uninflected dictionary form, and also reports their pronunciation. Even better, they have produced a Perl interface to it, *Text::ChaSen*. We can use this to turn kanji words into hiragana, one of the phonetic alphabets:

```
use Encode;
my $kanji = qr/\x{4e00}-\x{9fff}\/;

sub kanji_to_kana {
    my $string = shift;
    return $string unless /$kanji/;
    require Text::ChaSen;
    my $string_euc = encode("euc-jp", $string);
    Text::ChaSen::getopt_argv('chasen-perl', '-F',
                             '%a0');
    $string = decode("euc-jp",
                     Text::ChaSen::sparse_tostr($string_euc))
    || return $string;
    chomp $string;
    # Turn any katakana found into hiragana.
    $string =~
        tr/\x{30a1}-\x{30ff}/\x{3041}-\x{309f}/;
    $string;
}
```

There are two things to note here. First, *chasen* expects its input in the Japanese EUC character set, not in Unicode, so we use *Encode* to convert between the two. Second, *chasen* returns its output as katakana strings, whereas we want to deal with a single phonetic alphabet. We settled on hiragana because that's more often used for nonkanji words as well. A simple translation maps between the two alphabets.

There are several other rules that need to be followed before this turns into a usable sort key. For instance, some characters, even in the phonetic alphabet, are phonetic mutations of other characters: “ga,” for instance, is a softer form of “ka,” and these are sorted as identical.

All of the rules are wrapped up in the *Lingua::JA::Sort::ReadableKey* module: This exports a *japanese\_sort\_order* subroutine that we can use in our Schwartzian transform.

## Putting It All Together

To solve the problem of the multilingual dictionary, we used a variety of sort key routines, one for each language we were concerned with. We also defined the order in which each language would appear in a full list, so that if we were sorting lists containing words from more than one language, they would be sorted by language first. The array looked something like this:

```
my @languages =
    qw(English French German Thai Chinese Japanese);
my %language_order =
    map { $languages[$_] => $_ } 0..$#languages;
```

*%language\_order* is a hash where we can look up, say, Chinese, and get the order 4. Then we had a correlation between each language and a subroutine that would generate a key for a word in that language:

```
my %key_generators;
@key_generators{@languages} = (
    sub { $_[0] },
    \&french_key,
    \&german_key,
```

```
    \&thai_key,
    \&chinese_key,
    \&japanese_sort_order
);
```

The final sort received not just a list of words, but a list of word objects containing the word and its language. The sorting process then went like this:

```
sub lexicographic_sort {
    map { $_->[0] }
    sort {
        $a->[2] <=> $b->[2]
        ||
        $a->[1] cmp $b->[1]
    }
    map {
        [
            $_,
            $key_generators{$_->language}->($_->word),
            $language_order{$_->language}
        ]
    } @_;
}
```

Now, this is not necessarily obvious, so we'll look at it a step at a time. The overall idiom is our Schwartzian transform again. However, this time, instead of a pair, we've created an array of three-element arrays. This is because we're sorting on two factors: the order of the language of the word and the key generated by the key generator.

For each object, we look for the key generator for its language, and then pass the word to it; this will generate a key in the appropriate language. Then we look up the language order, so that we can sort first by language. You'll see the idiom

```
$a->{some_property} cmp $b->{some_property}
||
$a->{another_property} cmp $b->{another_property}
```

quite often to do multiple-level sorting. It means that if the comparison with the first property is a tie (the comparison returns 0), then the `||` operator does not return a result, but moves on to the second comparison and returns its result instead. In our case, we're testing the language, then the key within that language. Finally, we transform the three-element array back into the original object, and the job is done.

## More About Sorting

We've deliberately looked at some of the nastier languages to sort, and you may not be dealing with some of these. If your sorting needs are relatively simple, the standard Unicode Collation Algorithm may be good enough. This is detailed in Unicode Technical Report 10, at <http://www.unicode.org/reports/tr10/>.

The UCA does not deal with all possible languages; instead, it has particular translation tables that can be slotted into the algorithm to explain how a particular language sorts words. There's a Perl implementation of this in the *Unicode::Collate* CPAN module. It takes tables such as the Default Unicode Collation Element Table, and uses these to sort and provide comparison operators for any kind of Unicode strings.

It doesn't, however, deal with nonalphabetic scripts like Chinese or Japanese. If you're having to deal with these, you still need some of the techniques we've discussed in this article to sort them.

One thing's for certain, though—*sort* won't do it.

TPJ



# Making New Distributions

*brian d foy*

Lately, I've been casting about for a better way to start a new module, which I have to do quite often for clients. Perl has tools to do this already, but none of them did what I needed them to do. At the moment, I'm using the `tpage` and `ttree` programs from Template Toolkit, and it's working nicely.

I've created modules in many different ways over the years, and it shows. If you look in the BackPAN (the historic record of all CPAN uploads), you can watch my modules develop and know where I learned about a certain feature or fell in love with a particular structure. For instance, I just started adding `META.yml` and `README` files to my distributions. I can get around the mishmash of styles if I create distributions the same way every time. Perl has several tools to do this.

The `h2xs` program was the first one that I used to create new distributions, and the mantra at the time was "Start with `h2xs`." Indeed, as of Perl 5.8.4, that phrase is still in *perlnewmod*. The `h2xs` utility started as a way to convert C header files to XS modules so a module could connect a Perl script to C code. As time went on and there were fewer C libraries to convert, people used `h2xs` to create module distribution skeletons.

```
$ h2xs -AX -n Stonehenge::Foo
Writing Stonehenge/Foo/Foo.pm
Writing Stonehenge/Foo/Makefile.PL
Writing Stonehenge/Foo/README
Writing Stonehenge/Foo/t/1.t
Writing Stonehenge/Foo/Changes
Writing Stonehenge/Foo/MANIFEST
```

I used `h2xs` for several years, and I went through the pain of converting this skeleton into what I actually wanted: moving the `*.pm` files into a `lib` directory, replacing the skeleton `*.t` file (with its non-descript name), and then editing all of the files. I really wasn't saving any time. Although I haven't used `h2xs` for a couple of years, it would be even more work now since it uses newer features that aren't compatible with older versions of Perl (*our()*, *use warnings*).

Surely other people must have the same problem. There are enough Perl programmers out there that one of them must have hated `h2xs` enough to create their own module-starter tool.

The `ExtUtils::ModuleMaker` module came along around 2001. R. Geoffrey Avery had the same complaints about `h2xs`, so he created something to fix it. It got rid of all of the XS and Autoloader functionality, so if you used the `-AX` switches with `h2xs`, you could use `ExtUtils::ModuleMaker`. It eventually added a modulemaker program to use interactively.

---

*brian has been a Perl user since 1994. He is founder of the first Perl Users Group, NY.pm, and Perl Mongers, the Perl advocacy organization. He has been teaching Perl through Stonehenge Consulting for the past five years, and has been a featured speaker at The Perl Conference, Perl University, YAPC, COMDEX, and Builder.com. Contact brian at comdog@panix.com.*

Unfortunately, modulemaker was too much work for me. It does a good job organizing and collecting the information it needs to collect, but my name and other information stay the same, so I don't want to enter it every time I run the program. I could use the `ExtUtils::ModuleMaker` module if I wanted to write a program or add to the module file, but there are easier ways to do that.

Most of my module distributions look the same at the start, or at least I want them, to. I use the same POD structure, the same contact information (just my e-mail address), and many other things. There really isn't much programming there, I just need to change a few things in the same, basic template.

For a short time, I created my own skeleton structure, and out of laziness (the bad kind, not the virtue), I would simply copy that entire directory to a new directory, then go through it to edit things. The downside is apparent, and it was apparent to me even as I was doing it: I had to ensure that I edited everything I needed to edit. I was manually processing templates. Leave it to a human to do something and it's not going to get done most of the time.

Last year, I wrote (and wrote about in *TPJ*) the `scriptdist` program, which I created to build distributions for scripts. I needed to build distributions around a lot of standalone scripts, and I created `scriptdist` to do that. I didn't make it flexible enough to create module distributions, though. While I was looking around for other code that might do the same thing, I ran into "mmm" (my module maker) by Mark Fowler. It's a short program that uses Template Toolkit to build the right files. Mark even steals a little bit from `ExtUtils::ModuleMaker`.

Before I can use `mmm`, though, I need to create a directory of templates that it can process. Once I have that, I run `mmm` from the command line to get my processed module directory. But if I'm going to create a directory of templates, I really don't need `mmm`.

```
./modules
./modules/lib
./modules/lib/Foo.pm
./modules/Makefile.PL
./modules/README
./modules/t
./modules/t/load.t
./modules/t/pod.t
./modules/t/prereq.t
./modules/t/test_manifest
```

My templates are very simple, and most only need to replace very simple tokens, such as my name, e-mail address, and the name of the distribution or module. My initial `README` template is about as complicated as it gets. The placeholders in `[% %]` are in Template Toolkit syntax:

```
$Id: README,v 1.1 2004/09/08 00:25:41 comdog Exp $
```

You can install this using in the usual Perl fashion:



```
perl Makefile.PL
make
make test
make install
```

The documentation is in the module file. Once you install the file, you can read it with `perldoc`.

```
perldoc [% namespace %]
```

If you want to read it before you install it, you can use `perldoc` directly on the module file.

```
perldoc lib/[% dist-name %]
```

This module is also in CVS on SourceForge

```
http://sourceforge.net/projects/brian-d-foy/
```

Enjoy,

```
[% name %], [% email %]
```

The Template Toolkit comes with a wonderful tool called “`ttree`” that turns a directory of templates into a new directory of processed files. I put into a file everything I need to configure, including the source directory.

A configuration file has everything I need, and is just a Template Toolkit configuration. I tell it what to copy and what to ignore. I also define values for the template placeholders:

```
lib      = includes
ignore   = .DS_Store
ignore   = \b(CVS|RCS)\b

copy     = \.(png|gif|jpg|ico|pdf)$
src      = /Users/brian/templates/modules/stonehenge
dest     = /Users/brian/Desktop

define email=bdfoy@cpan.org
define name="brian d foy"
```

I can create different configuration files, even using different source directories (whereas `mmm` assumes a single source, which made it less attractive). I can have one set of templates for my public work that I release to CPAN, and another set that I use for Stonehenge (or even separate directories for each client):

```
ttree -f local-config --depend name=Local::Baz
ttree -f cpan-config --depend name=CPAN::Bar
ttree -f stonehenge-config
      --depend name=Stonehenge::Foo
```

I can shorten these with little shell scripts to do most of the typing for me:

```
#!/bin/sh
ttree -f stonehenge-config --depend namespace=$1
```

I save that in a script named “`shdist`.” When I need a new module for some Stonehenge work, I run my shell script

```
shdist Stonehenge::Foo
```

The only thing left is additional files that I add to a distribution. When I need to generate a single file, I can use Template

Toolkit’s `tpage` program. Indeed, the code example in the `tpage` program shows it processing a template file for a new module:

```
tpage --define author="Andy Wardley" skeleton.pm
      > MyModule.pm
```

I can do that, too, with a little script:

```
#!/bin/sh
tpage --define namespace=$1
/Users/brian/templates/modules/stonehenge/lib/Foo.pm
```

That’s not even good enough, though. Why should I have a bunch of different scripts to do the same thing?

```
#!/usr/bin/perl

use File::Basename qw(basename);

my $invoked_as = basename( $0 );

my %configs = (
    shmod => 'stonehenge-config',
    mymod  => 'my-config',
    cpanmod => 'cpan-config',
    default => 'cpan-config',
);

my $config = $configs{$invoked_as}
             || $configs{default};

exec "/usr/local/bin/ttree", "-f $config",
     "--depend namespace=$ARGV[0]\n";
```

I save this script, then create symlinks to it. The name of the script that I invoked shows up in `$0`. If I use one of the symlink names to invoke it, the symlink name shows up in `$0`, so I can tell how I invoked the program. I get the basename of that and use it to pull a configuration file out of the `%configs` hash. I then use `exec()` to turn my Perl process into a `ttree` process. The module name is the first argument and shows up in `$ARGV[0]`. This way, the command

```
shmod Foo::Bar
```

is really just

```
/usr/local/bin/ttree -f stonehenge-config
--depend namespace=Foo::Bar
```

So far, nothing in my solution really cares what is in the templates directory, so this process doesn’t care why I’m processing the templates. I can use the same thing, with different templates and configuration files, to do something else, such as processing a template-based website, writing a book, or generating customer invoices. I don’t need the tools that only generate modules when I have Template Toolkit.

## References

*ExtUtils::ModuleMaker*:

<http://search.cpan.org/dist/ExtUtils-ModuleMaker>.

Template Toolkit: <http://search.cpan.org/dist/Template>.

`h2xs`: <http://www.perldoc.com/perl5.8.4/bin/h2xs.html>.

`mmm`: <http://unixbeard.net/svn/mark/homedir/bin/mmm>.



# Perl 6 and Parrot Essentials

Jack J. Woehr

**P**arrot (<http://www.parrotcode.org/>), the new language-independent virtual machine that provides the runtime execution for Perl 6 (and for Ponie, an implementation of Perl 5 running on Parrot), is about 43 MB of somewhat-difficult-to-compile source. The Perl 6 language itself, a community rewrite of Perl, was the chicken that came after the egg, the idea of separating the execution engine from the language parser already having gained a foothold in the Perl community prior to the notion firmly taking root that Perl semantics need a major, code-breaking overhaul. (“Code-breaking” may be a little harsh: The promise is that Perl 5 programs will be mechanically translatable into Perl 6.)

*Perl 6 and Parrot Essentials*, 2nd Edition is an update of the recent *Perl 6 Essentials* (ISBN 0596004990). From the title shift, you’d suspect that Parrot is the difference between the two books; but Parrot, covered here for about 160 pages, was covered to the tune of about 110 pages in the previous edition. The remainder of the 80-odd pages added between the first and second editions are devoted to a more detailed and better factored discussion of the Perl 6 language. Perl 6 promises to be a “new-and-improved” Perl architected along cleaner lines. For example, I find it amusing and gratifying that in Perl 6, all `{ }` blocks are now closures. This is nice for me because I always reflexively assumed they already were closures and thus my code was, as you might guess, occasionally subject to surprising results.

*Perl 6 and Parrot Essentials* is concise and exhibits the tidiness of a theoretical work. Achieving such pristine beauty was simplicity itself for the authors since they can’t tell you all the gory details about Perl 6. Perl 6 hasn’t been fully invented yet: It’s a work in progress. Alongside filling you in on all truths so far revealed, the book gives organizational details of how to participate in the ongoing work of this largish, open-source project. Coauthor Dan Sugalski, the chief architect of Parrot, is in a position to know such things, as likewise are coauthors Leo Tötsch, the Parrot pumping (as our witty community styles its release coordinator), and Allison Randal, Perl 6 core development-team project manager.

Virtual machines being somewhat more concrete in nature (virtually concrete?) than language semantics, there is already more definition to Parrot than to Perl 6. Well, let’s put it this way: Parrot is up and running already, and Perl 6 is not fully so yet. Virtual machines as a mechanism of interpretive language execution were pioneered in practice by languages such as Pascal, Forth, and Prolog (remember the WAM, the Warren Abstract Machine?), and later Java. It’s gratifying to see that project participants have seen fit to make an historical tribute by implementing a sample Forth engine in Parrot. Parrot Forth is not fully ANS-compliant Forth, but that’s a Forth tradition in itself. And if anyone doubts that the Forth tradition of “programming religion” still lives, they will reach satori when they learn from the book how Perl advances via what Perl creator Larry Wall and the team term “Apocalypses” and “Exigeses.”

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

## *Perl 6 and Parrot Essentials, 2nd Edition*

Allison Randal, Dan Sugalski,  
and Leopold Tötsch  
O’Reilly Press, 2004  
256 pp., \$29.95  
ISBN 059600737X

While the metamorphosis of Perl into Perl 6 is arguably the most useful coverage in the book, Parrot is by far the more nerdily entertaining. Parrot ends up taking the Perl programmer for a dive deeper than most high-level language jockeys usually dare to descend.

The focus of *Perl 6 and Parrot Essentials* is on the goals, development process, and design of the revised language and its underlying language-independent runtime engine. It might have been nice, albeit a little off-message, if the authors had given some information about working with the Perl 6 and Parrot source code tree as it currently exists. It is not obvious to the newcomer how to build and install Parrot from source, despite the presence of a “configure/make/make install” procedure. The procedure is understandably rough and buggy, and the result is anything but tidy. Actually, using Parrot to run what exists so far of a prototype of Perl 6 turns out to be something of an existential conundrum. As Leo Tötsch said on one of the lists, “Perl 6 people are preparing a parser currently, should arrive RSN. But Perl 6 hacking is a bit hidden from the public still.”

Overall, *Perl 6 and Parrot Essentials* is a notable volume, significant for the moment as an invitation to an ongoing worldwide Perl community effort, and for some time to follow as a monument to the clarity of purpose and speed and excellence of execution of the development effort itself. Surely there will be at least a third edition in a year or two as the work on Perl 6 and Parrot reaches its fulfillment and a definitive tome becomes a possibility and a necessity.

Thus, you ask why a working Perl programmer like yourself, well-steeped (as in boiling water) in Perl 5, would have any pressing interest in reading now about Perl 6? Perl 6 will not be ready-as-in-ready-and-stable for a year or two. When finally ready, Perl 6 will present a naggingly incompatible environment to your current code corpus. Aside from givens such as personal anxiety to keep current in the field and irrepressible programmer’s curiosity, the best answer as to why you should participate in Perl 6 now is the same answer as to why you should participate in the political system; that is, as a form of self defense.

The publisher’s web page for Perl 6 and Parrot Essentials is at <http://www.oreilly.com/catalog/059600737X/index.html>. As an alternative to buying the print book, you can read it online at Safari (<http://safari.oreilly.com/?XmllId=059600737X>). The preview is free, but to read the whole book, you must subscribe.

TPJ

---

# Source Code Appendix

---

## Peter Sergeant “A mod\_perl 2 Primer”

### Listing 1

```
package TestModule::Redirect;

    use strict;
    use warnings;

    use Apache::DBI;
    use Apache::RequestRec ();

# We only need two of the status code constants...

    use Apache::Const -compile => qw(
        NOT_FOUND
        REDIRECT
    );

# Connect to the database

my $dbh = DBI->connect(
    'dbd:mysql:redirect',
    'user',
    'password',
);

# Prepare our SQL query

my $sql = $dbh->prepare("
    SELECT url FROM redirects WHERE key = ?
");

# And finally our handler...

sub handler {

    my $r = shift;

# Get the part of the URL we want (see notes)

    my $key = $r->path_info;

# Strip out anything we're not expecting...

    $key =~ s/[^a-z]//g;

# Which might leave us without a key at all,
# so we check and give an error if this is the
# case

    return Apache::NOT_FOUND unless $key;

# Grab the URL in the database corresponding
# to the key

    my $result = $sql->execute( $key );
    my $url = $result->fetchrow_arrayref;

# If there's no entry for the key in the database,
# then let the user know

    return Apache::NOT_FOUND unless $url;

# Set the new URL, and redirect appropriately

    $r->headers_out->set( Location => $url->[0] );
    return Apache::REDIRECT;

}

1;
```

## Christian Hicks and Dessislava Pachamanova “Metamodeling with Perl and AMPL”

### Listing 1

```
%kMonthHash = ("Jan" => 1, "Feb" => 2, "Mar" => 3, "Apr" => 4, "May" => 5,
               "Jun" => 6, "Jul" => 7, "Aug" => 8, "Sep" => 9, "Oct" => 10,
```



```

        "Nov" => 11, "Dec" => 12);
%kConvertHash = ("3month" => 90, "6month" => 180, "1year" => 360,
                "2year" => 720, "5year" => 1800, "10year" => 3600,
                "30year" => 10800);
sub read_file
{
    my($inFile) = @_;
    my($state, $temp, $period, $yield);
    open(INPUT, $inFile) || die("Cannot open $inFile");
    $state = 0;
    while (<INPUT>)
    {
        if ($state == 0)
        {
            #if (/U.S. Treasury yield curve/)
            if (/^U.S. Treasuries$/)
            {
                $state = 1;
            }
        }
        elsif ($state == 1)
        {
            #READ IN CURRENT DATE
            # Sun, 4 Jul 1999, 11:32am EDT
            if (/^[A-Z][a-z][a-z],\s*([0-9]+)\s*([A-Z][a-z][a-z])\s*([0-9]+),/)
            {
                $gDay = $1;
                #assign a numerical value to the month according to %kMonthHash
                $gMonth = $kMonthHash{$2};
                $gYear = $3;
                $state = 2;
            }
            else
            {
                die("cannot parse date");
            }
        }
        elsif ($state == 2)
        {
            #if (/Bills/)
            if (/Prc Chg/)
            {
                $state = 3;
            }
        }
        elsif (($state == 3) || ($state == 5))
        {
            if (/Notes/ || (/^U.S. Treasury Yield Curve/))
            {
                $state++;
            }
            elsif (/^(\S+)\s+^[^(\s)]+\s+$/) #READ IN RELEVANT DATA
            {
                $period = $kConvertHash{$1};
                push(@gPeriodList, $period);
                $yield = $2;
                push(@gYieldList, $yield);
            }
            elsif (/^\s*[+-][0-9+-]\s+$/)
            {
                # do nothing
            }
            else
            {
                die("cannot parse data: $_");
            }
        }
        elsif ($state == 4)
        {
            #if (/Bonds\s+Coupon\s+Mat\s+Date/)
            if (/Prc Chg/)
            {
                $state = 5;
            }
        }
    }
    close(INPUT);
}

```

## Listing 2

```

use Math::Ampl;
use strict;

```

#SPECIFICATION OF INPUT DATA:

```

my($NumStocks) = 3;
#vector of expected returns
my(@ExpectedReturnsList) = (0.08, 0.09, 0.12);
#vector of standard deviations for each stock
my(@StdDeviationsList) = (0.15, 0.20, 0.22);
#vector of target portfolio returns
my(@TargetReturnsList) = (0.08, 0.085, 0.09, 0.095, 0.10, 0.105,
                          0.11, 0.115, 0.12);

#RESULTS STORAGE:
#hash table to store optimal portfolio standard deviation results after
#solving all optimization problems
my(%OptimalStdDeviationsHash) = ();
#hash table to store optimal portfolio holdings after solving the
#portfolio optimization problem for each value of TargetReturn
my(%OptimalHoldingsHash) = ();

#OUTPUT FILES:
my($kOutputFile) = "efficientFrontier.out"; #to store Latex table
my($kFrontierGraphFile) = "efficientFrontier.jgr"; #to store graph

#DECLARATION OF AMPL PROBLEM INSTANCE:
my($gProblem);

#optimization model problem to be solved: minimize portfolio variance
#subject to constraints on portfolio expected return

#optimization model file to be passed to AMPL using PerlAmpl

sub setup_ampl_mod
{
    my($mod);
    $mod =<<EODATA;

    param NumStocks;
    param ExpectedReturns{1..NumStocks};
    param StdDeviations{1..NumStocks};
    param TargetReturn;

    var holdings{1..NumStocks};

    minimize portfolio_variance:
    sum{i in 1..NumStocks}
        (StdDeviations[i]*StdDeviations[i]*holdings[i]*holdings[i]);

    subject to portfolio_target_return:
    sum{i in 1..NumStocks} (ExpectedReturns[i]*holdings[i]) >= TargetReturn;

    subject to portfolio_total:
    sum{i in 1..NumStocks} (holdings[i]) = 1;
EODATA
    $gProblem->Input_Mod_Append($mod);
}
#optimization data file to be passed to AMPL using PerlAmpl
sub setup_ampl_dat
{
    #target portfolio return passed for this instance of the problem
    my($inTargetReturn) = @_;
    my($iStock);

    $gProblem->Input_Dat_Add_Param_Item("NumStocks", $NumStocks);
    $gProblem->Input_Dat_Add_Param_Item("TargetReturn", $inTargetReturn);
    for($iStock = 1; $iStock <= $NumStocks; $iStock++)
    {
        $gProblem->Input_Dat_Add_Param_Item("ExpectedReturns",
                                           $iStock,$ExpectedReturnsList[$iStock-1]);
        $gProblem->Input_Dat_Add_Param_Item("StdDeviations",
                                           $iStock,$StdDeviationsList[$iStock-1]);
    }
}
#request for AMPL to keep track of variables of interest

sub setup_ampl_display
{
    $gProblem->Input_Display_Clear;
    $gProblem->Input_Display_Add("solve_result");
    $gProblem->Input_Display_Add("portfolio_variance");
    $gProblem->Input_Display_Add("holdings");
}
#script for running the problem in AMPL and obtaining the results
sub solve_problem
{
    #target portfolio return passed for this instance of the problem
    my($inTargetReturn) = @_;
    my($solved);

```

```

$gProblem->Input_Mod_Clear;
$gProblem->Input_Dat_Clear;
setup_ampl_dat($inTargetReturn);
setup_ampl_mod();
setup_ampl_display();
$solved = $gProblem->Solve;
return $solved;
}
#PRINT GRAPH OF EFFICIENT FRONTIER DYNAMICALLY
#the output is a .jgr file which can then be converted to a .ps file

sub print_graph
{
    my($graph);
    my($targetReturn); #portfolio target return, read from list
    my($portfolioVariance); #optimal result from optimization problem
    my($portfolioStdDeviation); #to be computed from portfolio variance
    my(@CurrentHoldings); #obtained from AMPL output
    my($iStock); #counter

    open(GRAPH, ">$kFrontierGraphFile" ||
        die ("Cannot open file \"$kFrontierGraphFile\" for graph: $!"));
    $graph = "newcurve marktype none linetype solid linethickness 1";
    $graph .= " label : Efficient Frontier\n";
    $graph .= "\tpts ";
    foreach $targetReturn (@TargetReturnsList)
    {
        solve_problem($targetReturn);
        $portfolioVariance =
            $gProblem->Output_Display_Get_Value("portfolio_variance");
        $portfolioStdDeviation = sqrt($portfolioVariance);
        $graph .= " $portfolioStdDeviation $targetReturn ";
    }
    #store optimal standard deviations if necessary
    $OptimalStdDeviationsHash{"$targetReturn"} = $portfolioStdDeviation;
    #if necessary, store also the optimal holdings for each value of TargetReturn
    for($iStock = 1; $iStock <= $NumStocks; $iStock++)
    {
        $CurrentHoldings[$iStock-1] =
            $gProblem->Output_Display_Get_Value("holdings",$iStock);
        $OptimalHoldingsHash{"$targetReturn,$iStock"} =
            $CurrentHoldings[$iStock-1];
    }
    }
    print GRAPH "newgraph\n xaxis\n label : Standard Deviation\n";
    print GRAPH "yaxis \n ";
    print GRAPH "label : Expected Portfolio Return\n";
    print GRAPH "$graph\n\n";
    print GRAPH "legend defaults\n x 0.17 y 0.10\n";
    close GRAPH;
}
#PRINT A TABLE WITH RESULTS IN LATEX TABLE FORMAT
sub print_table
{
    my($portfolioStdDeviation);
    my($targetReturn);

    open(OUTPUT, ">$kOutputFile" ||
        die ("Cannot open file \"$kOutputFile\" with results: $!"));
    printf OUTPUT "%s %s ", "Expected Return", "&";
    printf OUTPUT "%s %s \n", "Standard Deviation", "\\\\";

    foreach $targetReturn (@TargetReturnsList)
    {
        $portfolioStdDeviation = $OptimalStdDeviationsHash{"$targetReturn"};
        printf OUTPUT "%2.3f %s ", $targetReturn, "&";
        printf OUTPUT "%2.3f %s \n", $portfolioStdDeviation, "\\\\";
    }
    close OUTPUT;
}
#MAIN
#initialization
Math::Ampl::Initialize($kAmplDir, $kAmplBin, $kTempDir, 1);
$gProblem = new Math::Ampl;

print_graph();
print_table();

```

### Listing 3

#Generates one path, and returns a List of Lists indexed by [time  
#period, asset number]. The entries equal single-period returns for  
#each stock. Single-period returns are multivariate normal random variables.

```
sub create_scenario
```

---

```

{
    #pass number of stocks in portfolio and number of time periods ahead
    my($inNumStocks, $inNumPeriods) = @_;

    my(@SimulatedReturnsList); #single time period simulated returns
    my(@ScenarioLoL); #list of lists of asset returns for each time period
    my($iT); #time period counter

    for ($iT = 0; $iT < $inNumPeriods; $iT++)
    {
        @SimulatedReturnsList =
            random_multivariate_normal($inNumStocks,
                                      @ExpectedReturnsList, @CovarianceMatrixList);
        #add simulated returns for time period iT to scenario path
        @ScenarioLoL[$iT] = [ @SimulatedReturnsList ];
    }
    return (@ScenarioLoL);
}

```

***TPJ***