CMP
United Business Media

March 2003

The Perl Journal

LETTER FROM THE EDITOR

The Fractal World

I was a band geek in high school. That's right—I played trombone. Every Saturday during autumn, I dressed up in a silly green-and-gold uniform, complete with a gold feather in the cap, and marched onto the football field with a hundred other ridiculously dressed, like-minded aficionados and played my heart out.

In retrospect, I didn't really understand at the time the way music intersects with other disciplines, which might explain why I stopped playing by the time I got to college. Music is a science, as much as anything else, and at its core it is mathematics. If I'd known that, well, things might have been different. Nevertheless, I loved math and science. Still do.

Which is why I find Randy Kobes's article "Fractal Images and Music with Perl" in this issue so fascinating. If you've never listened to fractal music before, give Randy's code a try. Here, we see the direct link from mathematical formula to musical score. But does it sound good? Is it really music? I confess, I expected the answer to be "no"—but I was wrong. Fractal generators are capable of making some surprisingly beautiful arrangements. As Randy admits in the article, it does still require a certain amount of human inspiration to seed the process with the right values. But the degree to which the formulas take over and make music is astonishing.

Fractals also describe—if not govern—some of the basic processes of nature. From leaves to mollusk shells, the concept of self-similarity at different scales seems to be one of nature's more efficient means of encoding the instructions for life. Perhaps for this reason, fractals are particularly good at modeling the biological world. What is more surprising is that the geological world also seems to exhibit a fractal nature—this can easily be seen by looking at an aerial photo of a coastline.

This all just reinforces the idea that algorithms underlie much of the world around us. Algorithms form the intersections between disciplines that we wouldn't normally connect, like music and biology. For some, it's these intersections that make the world interesting. They give us a bridge between disciplines and hint at the possibility that there's a real system governing it all. I spent hours as a kid poring over those "How Things Work" books. I had a shelf full of them, and I think the attraction for me was seeing the clockwork behind familiar things. That's what made me love science.

I've recently started making music again. No, not with a trombone (for which my neighbors are thankful), but with a synthesizer and computer. I won't be quitting my day job anytime soon, but what I'm finding is that the reason I care about it now, when for so many years I didn't, is that now I see some of the clockwork behind the music.

Kevin Carlson

Kevin Carlson
Executive Editor
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to editors@tpj.com, faxed to (650) 513-4618, or mailed to *The Perl Journal*, 2800 Campus Drive, San Mateo, CA 94403.
THE PERL JOURNAL is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: $12.00 for one year. Payment may be made via Mastercard or Visa; see http://www.tpj.com/. Entire contents (c) 2003 by CMP Media LLC, unless otherwise noted. All rights reserved.

CMP
United Business Media

The Perl Journal

EXECUTIVE EDITOR
Kevin Carlson

MANAGING EDITOR
Della Song

ART DIRECTOR
Margaret A. Anderson

NEWS EDITOR
Shannon Cochran

EDITORIAL DIRECTOR
Jonathan Erickson

COLUMNISTS
Simon Cozens, brian d foy, Moshe Bar

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

INTERNET OPERATIONS
DIRECTOR
Michael Calderon
SENIOR WEB DEVELOPER
Steve Goyette
WEB DEVELOPER
Bryan McCormick
WEBMASTERS
Sean Coady, Joe Lucca, Rusa Vuong

MARKETING/ADVERTISING
PUBLISHER
Timothy Trickett
MARKETING DIRECTOR
Jessica Hamilton
GRAPHIC DESIGNER
Carey Perez

THE PERL JOURNAL
2800 Campus Drive, San Mateo, CA 94403
650-513-4300. http://www.tpj.com/

CMP MEDIA LLC
PRESIDENT AND CEO
Gary Marshall
EXECUTIVE VICE PRESIDENT AND COO
Steve Weitzner
EXECUTIVE VICE PRESIDENT AND CFO
John Day
CHIEF INFORMATION OFFICER
Mike Mikos
PRESIDENT, TECHNOLOGY SOLUTIONS GROUP
Robert Faletra
PRESIDENT, HEALTHCARE GROUP
Vicki Masseria
PRESIDENT, ELECTRONICS GROUP
Jeff Patterson
PRESIDENT, SPECIALIZED TECHNOLOGIES GROUP
Regina Starr Ridley
SENIOR VICE PRESIDENT, GLOBAL SALES AND MARKETING
Bill Howard
SENIOR VICE PRESIDENT, HUMAN RESOURCES AND COMMUNICATIONS
Leah Landro
PRESIDENT AND GENERAL COUNSEL
Sandra Grayson

Shannon Cochran

Perl News

Call for Talks: TPC 7

The seventh annual Perl Conference will, as in past years, be held in conjunction with the O'Reilly Open Source Convention (OSCON). The event, which also includes the Python 11 Conference and the third PHP Conference, will be held on July 7–11 in Portland, Oregon.

Ideas for lightning talks—five minute presentations that focus narrowly on one facet of an idea, or on a single example or technique—are now being solicited. "The point is that because the talk is only five minutes long, you don't have to take it so seriously," writes Mark Jason Dominus. "Just try to say something brief and interesting, and then get out in a hurry."

If you'd like to give a Perl-oriented lightning talk at OSCON, e-mail a four-sentence abstract of your talk to osc-lt-2003-perl@plover.com. Submissions will be accepted until June 7th.

The OSCON website is at http://conferences.oreillynet.com/os2003/, and more details about the lightning talks are online at http://perl.plover.com/lt/osc2003/. Questions about the Perl talks in particular should go to mjd-osc-lt-2003-perl@plover.com.

Perl Mongers Infiltrate Stockholm, Buffalo

With ninja-like speed and mystery, the influence of the far-flung and powerful Perl Mongers organization continues to spread. The cities of Stockholm, Sweden, and Buffalo, New York, have now sprouted user groups, which will undoubtedly spread the knowledge of advanced scripting techniques among the native populaces.

The Buffalo Perl Mongers have gone so far as to create a web site, located at http://buffalo.pm.org/, and to advertise their mailing list and gatherings. The activities of the Stockholm Perl Mongers, meanwhile, can be traced to http://stockholm.pm.org/meetings.html. Sources report that this users group is working in close association with other Perl Mongers to organize a grand cabal—the Scandinavian Perl Workshop (http://perlworkshop.dk/)—which will meet April 25–26 in Copenhagen.

Parrot vs. Python: This Time It's Personal

Ten dollars and a round of beer is riding on the question: Will Parrot or Python be faster at executing a pure Python benchmark? Dan Sugalski and Guido van Rossum are squaring off over the challenge, which was announced on the perl6-internals list. The details of the competition are to be determined at OSCON 2003, and the results announced at OSCON 2004. Sugalski has also raised the ugly specter of a "mudwrestling cage deathmatch," though both participants hope to avoid such a tragic eventuality.

POE 0.25 Released

In a separate thread, the POE framework for creating multitasking applications in Perl (in development since 1998) has announced its 0.25 release. POE divides execution time among sessions, which cooperate to perform multiple tasks. POE also has a component architecture, supported by medium- and low-level concurrency functions; components have been written to handle client, server, and peer networking functions, as well as other tasks. Graphical toolkits (Tk and GTK) are supported, as well as Curses, HTTP, and other user interfaces—or even several interfaces at once.

New features in the 0.25 release include support for ActivePerl 5.8.0 and Gentoo Linux; TCP client and server support for multiple session types; better TCP server handling of aborted connections; more configuration options for TCP clients and servers; and bug fixes. More information is available from http://poe.perl.org/.

Perl Idioms for Java

A new SourceForge project is aiming to bring "a dash of Perlishness into Java" by implementing Perl idioms in Java. *foreach*, *grep*, *join*, *map*, *pop*, *push*, *shift*, *split*, and *unshift* have been implemented, along with "a (very) rudimentary form of autovivication," and "a class representing Perl's notion of truth." (Project admin David Jantzen wrote on the Perl Monks site: "If only I could convey my glee at the realization that a *java.lang.Boolean* initialized to 'false' is true according to Perl's semantics!")

The project has yet to formally release any files, but the source code is available at http://sourceforge.net/projects/perlforjava/.

Fun with the Google API

Aaron Straup Cope has updated his *Net::Google* module, which provides an interface to the Google SOAP API. The Google methods can be used to get spelling suggestions, find cached documents, and, of course, get search results. On top of this, Darren Chamberlain has added a new module, *DBD::google*, which allows Perl authors to treat Google as a data source for DBI queries.

"For the most part," Chamberlain writes, "use *DBD::google* like you use any other DBD, except instead of going through the trouble of building and installing (or buying!) database software, and employing a DBA to manage your data, you can take advantage of Google's ability to do this for you. Think of it as outsourcing your DBA, if you like."

Both modules are available on CPAN.

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

Randy Kobes, Andrea Letkeman, and Olesia Shewchuk

Fractal Images and Music With Perl

You may remember taking an IQ test in which you are given some numbers: 1 4 7 10…, and are asked to predict the next number in the sequence. What you end up doing is, at least subconsciously, starting from the first number, looking for a pattern that subsequent numbers fit, and then using that pattern to predict the next (or any other) number in the series. In this article, we'll examine some sequences of numbers that exhibit a fractal nature, and from those, see how Perl can be used to generate images and sounds that represent aspects of this nature.

The Nature of Fractals

A simple example of a fractal sequence can be given by considering the binary form of the first few numbers:

```
printf("%b ", $_) for (0 .. 10);

0 1 10 11 100 101 110 111 1000 1001 1010
```

Next, form the sequence by counting the number of times "1" occurs in each number:

```
for (0 .. 20) {
  $b = sprintf("%b", $_);
  $n = $b =~ tr/1//;
  print "$n ";
}

0 1 1 2 1 2 2 3 1 2 2 3 2 3 3 4 1 2 2 3 2
```

Now take every other number in this sequence:

```
for (0 .. 20) {
  next unless $_ % 2 == 0;
  $b = sprintf("%b", $_);
  $n = $b =~ tr/1//;
  print "$n ";
}

0 1 1 2 1 2 2 3 1 2 2
```

We see that, up to the same number of terms considered, the two sequences are the same. This is also true if we had taken every 4th number of the original sequence, and every 8th number, and so on. This self-similarity is a fundamental property of what is known as a "fractal"—something that looks similar to itself on different scales. This is why, for example, when we look at a picture filled with clouds or one of a moonscape with many craters, it is hard to judge how far away the picture was taken—the fractal nature of these objects makes it hard for us to associate one particular length scale with them, as they look similar at many different scales. Fractals are also often used in generating patterns for screensavers, some of which are a true source of fascination.

A web search will quickly find the two most popular aspects of fractals—fractal images, which are especially associated with what are called the "Mandelbrot" and "Julia" sets, and fractal music. In this article, we'll describe how easily both of these can be made with Perl. What amazed early workers in the field, and is still a source of wonder today, is just how easily these can be generated, and despite the simplicity of the algorithms used, just how intricate and surprising the results can be.

In the following, we consider particular sequences of numbers to illustrate the results. We imagine the sequences to be stored in some array *@x*, and denote a particular member by *x[n]*. The sequence is generated, starting from an initial *x[0]*, through a relation *x[n+1]=F(x[n])* for some given function *F(x)*, where *n=0, 1, 2,…* What this notation means is that we start with some given *x[0]*, and then, with *n=0*, find *x[1]=F(x[0])*. Having found *x[1]*, we then set *n=1* to find *x[2]=F(x[1])*, and so on. For example, the sequence *1 4 7 10…* can be represented as *x[n+1]=x[n]+3*, with *x[0]=1*.

The *Fractal* Module

The actual work of making the images and sound files in what follows is done through some functions made available in the *Fractal* module (fractal.pm, available online at http://www.tpj.com/source/). This module contains three functions:

draw()—Takes a reference to an array of arrays (for example, *[[x1,y1,i1], [x2,y2,i2]]*), and produces an image using Lincoln Stein's *GD* module. The (*x,y*) values represent the horizontal and vertical coordinates of the points to be plotted, while the associated *i* value is used to help determine the color used to plot this point. This is done by first calculating an index based on the values of (*x,y,i*), and then using this index to calculate an *rgb* triplet (*r,g,b*) of numbers, each between 0 and 255, which specify the amounts of red, green, and blue to use. Default routines to do these are given in the module, or the user can specify his or her own routines by providing code references as *draw(rgbindex => \&my_rgbindex, rgb => \&my_rgb)*.

compose()—Takes a reference to an array of points and produces a midi sound file using Sean Burke's *MIDI* module. Which particular note is associated with each point is determined, as with *draw()*, by first calculating an index from the point under consideration and then from this index, associating a note. Again,

Randy researches and teaches physics at the University of Winnipeg. He can be reached at randy@theoryx5.uwinnipeg.ca. Andrea currently works for an oil company, and is studying towards her Royal Conservatory of Music Teacher's certification. She can be reached at andrea_letkeman@hotmail.com. Olesia is a student of science, computers, dance, and music. She can be reached at ochuchma@io.uwinnipeg.ca.

default routines to do this are contained in the module, or the user can specify her or his own as *compose(noteindex => \&my_noteindex, note => \&my_note)*.

min_max()—Takes either an array reference or a reference to an array of arrays and calculates the minimum and maximum values. For an array reference, two scalars representing the minimum and maximum values are returned, while for a reference to an array of arrays, two array references are returned containing the corresponding information. This routine is useful in determining the index used in both *draw()* and *compose()*, and is also used in *draw()* in converting the (*x,y*) points passed to the routine into pixel coordinates based on the size of the image specified.

Fractal Images

With this, we begin by considering fractal images, for which there are two main algorithms. (One can mathematically relate the two algorithms, but it is calculationally convenient to consider them separately.) The first algorithm uses what is called the "escape-time" method. Consider the sequence of numbers (*x[n],y[n]*):

```
x[n+1] = x[n]² – y[n]² + cx
y[n+1] = 2 x[n] y[n] + cy
```

where *cx* and *cy* are constants. (This unlikely looking sequence is more natural in the context of complex numbers, where it can be written as $z[n+1]=z[n]^2+c$, where *z[n]* and *c* are complex numbers having a real and an imaginary part (see the documentation on *Math:: Complex* for details). Starting from some initial points (*x[0],y[0]*), two distinct possibilities can arise—the numbers in the sequence all remain finite or, at some stage, they start to diverge towards infinity. To avoid dealing with very large numbers, a breakout condition on the size of the numbers is imposed that when met, ends the generation of further numbers in the sequence. This defines the following escape-time algorithm: For each point (i.e., pixel) on a graph of interest, set up the initial conditions and then (up to some maximum number of iterations):

1. Find the next point in the sequence.
2. If the breakout condition on the size of the numbers is met, end the iterations.
3. Plot the point according to some criteria based on when the breakout condition was met.

A typical breakout condition used is to calculate $x[n]^2+y[n]^2$ for each point, and then to end the sequence if this is larger than some number.

There are two major types of images typically used in illustrating this algorithm, the difference being in their choices of initial points (*x[0],y[0]*) and the constants (*cx,cy*). For the Mandelbrot set, one sets *x[0]=0=y[0]* and chooses for (*cx,cy*) the coordinates of the pixel under consideration. For the Julia set, one sets (*x[0],y[0]*) to the coordinates of the pixel under consideration, and chooses (*cx,cy*) equal to fixed constants. Listing 1 is a script for generating the Mandelbrot set, while Listing 2 is for generating the Julia set. Running these scripts results in the images in Figures 1(a) and 1(b), respectively.

Apart from producing interesting images, the use of colors gives information about the behavior of these sequences. For the Mandelbrot set, points in the interior (pink, in this case) never satisfy the breakout condition, while points on the edges (shades of green, in this case) quickly do. The interesting region is around the jagged edges, where one can see evidence of the fractal nature through self-similarity. Indeed, if one blows up a region around the edges by changing (*xmin,xmax,ymin,ymax*) in the mandelbrot.pl script, similar-looking images result. The Julia set shows similar behavior and, in fact, is closely related to the Mandelbrot set—interesting images for the Julia set are found by choosing the constants (*cx,cy*) near the jagged edges of the Mandelbrot set.

Another type of fractal image is formed from what is called an "iterated function system." For this one, consider sequences of the form:

```
x[n+1] = a[i] x[n] + b[i] y[n] + e[i]
y[n+1] = c[i] x[n] + d[i] y[n] + f[i]
```

where (*a[i],b[i],c[i],d[i],e[i],f[i]*), with *i=1..N*, are sets of constant numbers. (Although theoretically the constants can be chosen arbitrarily, constraints on their magnitude exist if one wishes to generate images of a finite size.) To generate an image from these sequences, one begins with some initial point (*x[0],y[0]*) and then implements the following algorithm for some number of iterations:

1. Choose, by some criteria, one of the sets of numbers labeled by the index *i* to generate the next point in the sequence.
2. Find the next point, and plot it.

Listings 3 and 4 produce the images that appear in Figure 2.

As well as the coefficients used in generating the sequences, these differ according to how the index *i* is chosen. For the snowflake, an *i* is picked randomly from those available, while for the fern, a weighted set of probabilities is used. For many cases, this latter way of choosing the index produces more complete images in fewer iterations.

Fractal Music

We now come to a second way of viewing fractals—through musical scores. Like art, music is a very subjective and personal thing—what one person passionately likes, another may equally detest.

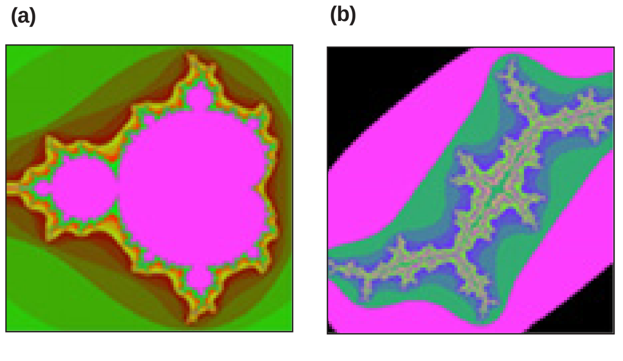


Figure 1: Fractal images drawn using the escape-time algorithm: (a) Mandelbrot set; (b) Julia set.

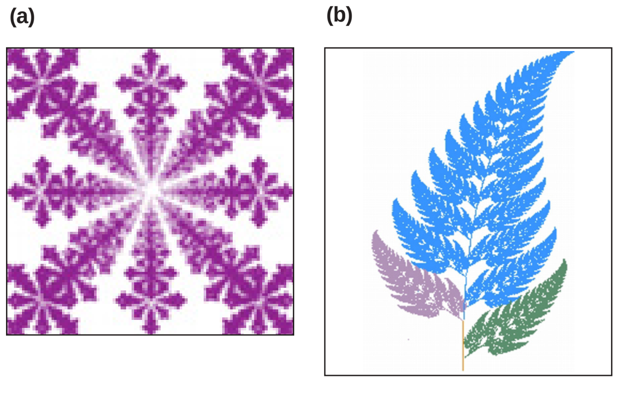


Figure 2: Fractal images drawn using an iterated function system: (a) snowflake; (b) fern.

However, there are some common aspects about music that span across genres and time, and one of those is a fractal nature. (The degree of "fractalness" is related to the information content, and entropy, of a system. Lovers of classical music will take pride in knowing that classical music is more "fractal" than most forms of today's popular music; we'll let the reader draw his or her own conclusions.) This again is related to the property of self-similarity—patterns of notes occurring within a few bars are often repeated throughout a score, and also occur (qualitatively) on a larger scale.

Perhaps even more significantly, what makes music interesting and enjoyable is an aspect that was also evident in the images just studied—that of their relative complexity. Humans like to find patterns—a random collection of colored pixels or notes on a scale is unpleasant—but we also like to be surprised. In music, phrases in harmony lead to what is termed "consonance," while the introduction of suspense, or tension, leads to "dissonance." Music needs elements of both to be enjoyable. Fractals, in a sense, live on the border between random noise and dull predictability.

For a simple example of fractal music, consider what is known as the "logistic map":

```
x[n+1] = A x[n] (1 – x[n])
```

where *A* is a constant and each *x[n]* lies between 0 and 1. (Historically, this sequence was proposed as a simple model of population growth, where *x[n]* represents the fraction of the maximum that a population attains in year *n*. The term *Ax[n]* increases the population, through reproduction, and the $-Ax[n]^2$ term decreases the population, through predators and disease.) The behavior of this sequence depends on the parameter *A*—for *A* less than about 3, the sequences of numbers converges to some fixed value; while for *A* greater than about 3.5 up until 4.0, the numbers appear to be random. (Actually, the behavior is *chaotic*.) The key word here is "appear"—for this range of *A*, there is actually a fractal nature present.

Sequences such as these are the basic ingredient in making fractal music—the numbers of the sequence are simply mapped to notes of a musical scale. In Listing 5, we give an example of how this is done for the logistic map; while in Listing 6, we do so for another example called the "Hénon map":

```
x[n+1] = a - x[n]² + b y[n]
y[n+1] = x[n]
```

for which we use only the *x[n]* in generating the notes. For the logistic score, we employ the default methods of the *Fractal* module to map numbers in the sequence to notes, while for the Hénon score, we specify these within the script.

When played, one can sense these scores as being musical— recognizable patterns emerge, and yet there are elements of unpredictability. (If one had to classify fractal scores, a possible candidate might be "new" or "modern" classical music, such as that of Schönberg, in which greater dissonance and a movement away from a fixed key signature plays a prominent role.) One might think it easy to create such scores from an almost arbitrary sequence, but it is really those with a fractal nature that are most pleasing. Try creating a score from just a random sequence of numbers (in the logistic map, this corresponds to setting the parameter *A* to 4)—the difference is quite noticeable. Of course, at this level, fractal music is missing many components of real music such as dynamics, changing tempo, and varied note durations and rests. This, in principle, can be added to the scripts; one learns from these studies just how multifaceted and complicated music really is.

We can compose potentially more interesting scores by obtaining, as for the Mandelbrot set, two related sequences of numbers (*x[n],y[n]*), which can then be used to make two simultaneous tracks in the score. This is done using the *sync()* routine of the *MIDI* module; an example, using a point on the ragged boundary of the Mandelbrot set, appears in Listing 7. Apart from producing richer scores, an interesting philosophical aspect of doing this is the potential for being able to "hear" images and, reversing the process, being able to "see" musical scores.

Conclusion

What do we learn from such studies? The main goal here was to have fun. There's much variety in these algorithms—using other sequences, changing the ranges of the coordinates used, playing with the indexing routines used to map numbers to pixel colors and notes, and so on. Discovering an unexpectedly rich image or melodic score is a source of amazement, and if one learns some Perl along the way, that can't be a bad thing. This isn't to say, though, that the main use of fractals is in entertainment—the fern image, for example, indicates there is something "real" about fractals. More generally, science is just discovering that many diverse phenomena, such as weather patterns, fluid flow, heart rates, data- transmission noise, and commodity prices, have a common fractal nature. But even in the context of fractal images and music, something pretty profound is going on here. We are taking a mathematical set of numbers and producing aesthetically pleasing pictures and scores. However, quite a bit of programmer insight is still needed even to create these—simply using the "wrong" coloring index can turn a beautiful image into an eyesore. Perhaps the lesson to take away from this is that while we are crossing, at least tentatively, into understanding aspects of creativity, we are just beginning to understand how complicated, and wondrous, this creativity is.

Acknowledgments

We'd like to thank the Natural Sciences and Engineering Research Council of Canada for support.

(All listings for this article are also available online at http://www.tpj.com/source/.)

TPJ

Listing 1

```
#!/usr/bin/perl
# mandelbrot.pl - draw the Mandelbrot set
use strict;
use warnings;
use Fractal qw(draw min_max);

# xmin, xmax, ymin, and ymax specify the range of coordinates
# steps is the number of points used between (xmin .. xmax),
#   [which is the same number used between (ymin .. ymax)]
# maxit is the maximum number of iterations
my ($xmin, $xmax, $ymin, $ymax, $steps, $maxit) =
  (-1.6, 0.6, -1.2, 1.2, 300, 200);

my $points = mandelbrot();

draw(pts => $points, file => 'mandelbrot.png', size => $steps);

# Mandelbrot set, as defined by
#    x[n+1] = x[n]*x[n] - y[n]*y[n] + c_x
#    y[n+1] = 2*x[n]*y[n] + c_y
# where (c_x, c_y) are the pixel coordinates
# and the iterations begin at (x, y) = (0, 0)

sub mandelbrot {
  my $pts = [];
# determine step size between (xmin .. xmax) and (ymin .. ymax)
  my $xstep = ($xmax - $xmin) / $steps;
  my $ystep = ($ymax - $ymin) / $steps;

  for (my $cx=$xmin; $cx<=$xmax; $cx+=$xstep) {
    for (my $cy=$ymin; $cy<=$ymax; $cy+=$ystep) {
      my ($oldx, $oldy, $newx, $newy) = (0,0,0,0); # start at x = y = 0
      my $i=1;  # iteration number before breakout condition is met
      for ($i=1; $i<$maxit; $i++) {
           $newx = $oldx*$oldx - $oldy*$oldy + $cx;
           $newy = 2*$oldx*$oldy + $cy;
```

```
          last if sqrt($newx*$newx + $newy*$newy) > 2; # breakout condition
          ($oldx, $oldy) = ($newx, $newy);
      }
      push @$pts, [$cx, $cy, $i];
    }
  }

  return $pts;
}
```

Listing 2

```
#!/usr/bin/perl
# julia.pl - draw the Julia set
use strict;
use warnings;
use Fractal qw(draw min_max);

# xmin, xmax, ymin, and ymax specify the range of coordinates
# steps is the number of points used between (xmin .. xmax)
# the same number is used between (ymin .. ymax)
# maxit is the maximum number of iterations
my ($xmin, $xmax, $ymin, $ymax, $steps, $maxit) =
  (-1.3, 0.9, -1.2, 1.2, 300, 200);

my ($cx, $cy) = (-0.11, -0.9); # used to define the Julia set

my $points = julia();
my ($min, $max) = min_max($points); # used in my_rgbindex and my_rgb
draw(pts => $points, file => 'julia.png', size => $steps,
     rgbindex => \&my_rgbindex, rgb => \&my_rgb);

# Julia set, as defined by
#    x[n+1] = x[n]*x[n] - y[n]*y[n] + c_x
#    y[n+1] = 2*x[n]*y[n] + c_y
# where (c_x, c_y) are fixed constants
# and the iterations begin at the pixel coodinates

sub julia {
  my $pts = [];
# determine step size between (xmin .. xmax) and (ymin .. ymax)
  my $xstep = ($xmax - $xmin) / $steps;
  my $ystep = ($ymax - $ymin) / $steps;

  for (my $x=$xmin; $x<=$xmax; $x+=$xstep) {
    for (my $y=$ymin; $y<=$ymax; $y+=$ystep) {
      my ($oldx, $oldy, $newx, $newy) = ($x, $y, 0, 0); # initialization
      my $i=1; # iteration number before breakout condition is met
      for ($i=1; $i<$maxit; $i++) {
          $newx = $oldx*$oldx - $oldy*$oldy + $cx;
          $newy = 2*$oldx*$oldy + $cy;
          last if sqrt($newx*$newx + $newy*$newy) > 2; # breakout
          ($oldx, $oldy) = ($newx, $newy);
      }
      push @$pts, [$x, $y, $i];
    }
  }

  return $pts;
}

sub my_rgbindex {
  my $p = shift;
  return int($p->[2]/$max->[2]*100);
}

sub my_rgb {
  my $index = shift;
  return map{hex} unpack "a2a2a2",
    sprintf("%02X", int($index*255**3));
}
```

Listing 3

```
#!/usr/bin/perl
# snowflake.pl - draw a snowflake
use strict;
use warnings;
use Fractal qw(draw min_max);

# specify the maximum number of iterations used,
# and the size of the desired image
my ($maxit, $size) = (500000, 300);

# specify the transformations
#     x[n+1] = a[i]*x[n] + b[i]*y[n] + e[i]
#     y[n+1] = c[i]*x[n] + d[i]*y[n] + f[i]
# where "i" labels the particular transformation

my @a = (.75, .5, .25, .25, .25, .25);
my @b = (0, -0.5, 0, 0, 0, 0);
my @e = (.125, .5, 0, .75, 0, .75);
my @c = (0, .5, 0, 0, 0, 0);
my @d = (.75, .5, .25, .25, .25, .25);
my @f = (.125, 0, .75, .75, 0, 0);

# where to begin the iterations at
my ($xstart, $ystart) = (0.6, 0.5);

my $points = ifs_random();
my ($min, $max) = min_max($points);
draw(pts => $points, file => 'snowflake.png', size => $size,
     rgbindex => \&my_rgbindex, rgb => \&my_rgb);

# routine to find the points of an ifs by
# choosing the transformation used randomly
sub ifs_random {
  my $pts = [];
  push @$pts, [$xstart, $ystart, 0];
  my ($oldx, $oldy, $newx, $newy) = ($xstart, $ystart, 0, 0);
  for (my $count = 1; $count < $maxit; $count++) {
# choose a transformation randomly
    my $index = int(rand @a);
    $newx = $a[$index]*$oldx + $b[$index]*$oldy + $e[$index];
    $newy = $c[$index]*$oldx + $d[$index]*$oldy + $f[$index];
    push @$pts, [$newx, $newy, $index];
    ($oldx, $oldy) = ($newx, $newy);
  }
  return $pts;
}

sub my_rgbindex {
  my $p = shift;
  return int($p->[2]/$max->[2]*100);
}

sub my_rgb {
  my $index = shift;
  return (128, 0, 128);
}
```

Listing 4

```
#!/usr/bin/perl
# fern.pl - draw a fern
use strict;
use warnings;
use Fractal qw(draw min_max);

# specify the maximum number of iterations used,
# and the size of the desired image
my ($maxit, $size) = (100000, 300);

# specify the transformations
#     x[n+1] = a[i]*x[n] + b[i]*y[n] + e[i]
#     y[n+1] = c[i]*x[n] + d[i]*y[n] + f[i]
# where "i" labels the particular transformation

my @a = (0.85, 0.20, -0.15, 0);
my @b = (0.04, -0.26, 0.28, 0);
my @e = (0.075, 0.4, 0.575, 0.5);
my @c = (-0.04, 0.23, 0.26, 0);
my @d = (0.85, 0.22, 0.24, 0.16);
my @f = (0.18, 0.045, -0.086, 0);

# specify the probablility for which to choose each transformation
# and check that they all add up to 1
my %prob = (0 => 0.77, 1 => 0.12, 2 => 0.1, 3 => 0.01);
my $sum = 0;
$sum += $_ for values %prob;
die 'The probablilities have to add up to 1'
  unless ( abs($sum - 1) < 0.001);

# where to begin the iterations at
my ($xstart, $ystart) = (0, 0);

my $points = ifs_specified();
my ($min, $max) = min_max($points);
draw(pts => $points, file => 'fern.png', size => $size,
     rgbindex => \&my_rgbindex, rgb => \&my_rgb);

# routine to find the points of an ifs by choosing the
# transformation as specified by a user-set probability
sub ifs_specified {
  my $pts = [];
  push @$pts, [$xstart, $ystart, 0];
  my ($oldx, $oldy, $newx, $newy) = ($xstart, $ystart, 0, 0);

# set up bounds to use in test for deciding which
# transformation to use
  my %bounds;
  my $last = 0;
  my @indices = sort {$a <=> $b} keys %prob;
  for (@indices) {
    $last += $prob{$_};
    $bounds{$_} = $last;
  }
  for (my $count=1; $count<$maxit; $count++) {
# decide which transformation to use
    my $index;
    my $r = rand;
    for (@indices) {
      if ($r < $bounds{$_}) {
           $index = $_;
           last;
      }
    }

    $newx = $a[$index]*$oldx + $b[$index]*$oldy + $e[$index];
    $newy = $c[$index]*$oldx + $d[$index]*$oldy + $f[$index];
    push @$pts, [$newx, $newy, $index];
    ($oldx, $oldy) = ($newx, $newy);
  }
  return $pts;
}

sub my_rgbindex {
  my $p = shift;
  return int($p->[2]/$max->[2]*255);
}

sub my_rgb {
  my $index = shift;
  return ($index*20, 128, (255 - $index*10));
}
```

Listing 5

```
#!/usr/bin/perl
# logistic.pl - hear the logistic map
use strict;
use warnings;
use Fractal qw(compose);
# where to start the map at, the parameter mu, and the number of notes
my ($xstart, $A, $maxit) = (0.4, 3.82, 256);

my $points = logistic();
compose(pts => $points, file => 'logistic.mid');

sub logistic {
  my $pts = [];
  my ($oldx, $newx) = ($xstart, 0);
  push @$pts, $xstart;
  for (my $i=1; $i<$maxit; $i++) {
    $newx = $A*$oldx*(1-$oldx);
    push @$pts, $newx;
    $oldx = $newx;
  }
  return $pts;
}
```

Listing 6

```
#!/usr/bin/perl
# henon.pl - hear the henon map
use strict;
use warnings;
use Fractal qw(compose min_max);
# where to start the map at
my ($xstart, $ystart) = (0.4, 0.2);
# the parameters a and b, and the number of notes
my ($a, $b, $maxit) = (1.4, 0.3, 256);

# use a chromatic scale
my @notes = (55 .. 89);

my $points = henon();
my ($min, $max) = min_max($points);
compose(pts => $points, file => 'henon.mid',
           tempo => 100000, patch => 44,
           noteindex => \&my_noteindex, note => \&my_note);

sub henon {
  my $pts = [];
  my ($oldx, $oldy, $newx, $newy) = ($xstart, $ystart, 0, 0);
  push @$pts, $xstart;
  for (my $i=1; $i<$maxit; $i++) {
    $newx = $a - $oldx*$oldx + $b*$oldy;
    $newy = $oldx;
    push @$pts, $newx;
    ($oldx, $oldy) = ($newx, $newy);
  }
  return $pts;
}

sub my_noteindex {
  my $x = shift;
  $x = ($x - $min) / ($max - $min);
  return int($x * $#notes);
}

sub my_note {
  my $index = shift;
  return $notes[$index];
}
```

Listing 7

```
#!/usr/bin/perl
# two_track.pl - hear a point of the Mandelbrot set
use strict;
use warnings;
use Fractal qw(min_max);
use MIDI::Simple;

my $maxit = 200; # maximum number of notes
my ($cx, $cy) = (-0.1, -0.9); # pick a point on the Mandelbrot set
my $file = 'mandelbrot.mid';
my ($tempo, $patch_0, $patch_1) = (90000, 33, 66);

my @xnotes = (55, 57, 59, 60, 62, 64, 66, 67);

my @ynotes = (67, 69, 71, 72, 74, 76, 78, 79);

my $points = mandelbrot();
my ($min, $max) = min_max($points);
my ($xmin, $xmax, $ymin, $ymax) =
  ($min->[0], $max->[0], $min->[1], $max->[1]);

new_score();
patch_change 0, $patch_0;
patch_change 1, $patch_1;
set_tempo($tempo);
my @subs = (\&track_0, \&track_1);
my ($x, $y);

foreach (@$points) {
  ($x, $y) = ($_->[0], $_->[1]);
  synch(@subs);
}
write_score($file);

sub mandelbrot {
  my $pts = [];
  my ($oldx, $oldy, $newx, $newy) = (0,0,0,0); # start at x = y = 0
  push @$pts, [$oldx, $oldy];
  my $i=1;  # iteration number before breakout condition is met
  for ($i=1; $i<$maxit; $i++) {
    $newx = $oldx*$oldx - $oldy*$oldy + $cx;
    $newy = 2*$oldx*$oldy + $cy;
           last if sqrt($newx*$newx + $newy*$newy) > 2; # breakout
                                                         # condition
    ($oldx, $oldy) = ($newx, $newy);
    push @$pts, [$newx, $newy];
  }
  return $pts;
}

sub track_0 {
  my $it = shift;
  my $note = xnote();
  $it->n('c0', 'hn', $note);
}

sub track_1 {
  my $it = shift;
  my $note = ynote();
  $it->n('c1', 'hn', $note);
}

sub xnote {
  my $xn = ($x - $xmin) / ($xmax - $xmin);
  my $index = int($xn * $#xnotes);
  return $xnotes[$index];
}

sub ynote {
  my $yn = ($ymax - $y) / ($ymax - $ymin);
  my $index = int($yn * $#ynotes);
  return $ynotes[$index];
}
```

TPJ

Stas Bekman and Eric Cholet

Writing Multilingual Sites With mod_perl and Template Toolkit

Before you search for a solution for your multilingual site, you have to figure out what kind of service you are going to provide: dynamic or static. If the pages are static, you need to evaluate whether there will be many pages to maintain or just a few. If you have only a few pages, the easiest solution is to just prepare each page in each language and forget about it.

If you have many pages, it's pretty much the same whether your pages are dynamic or static: Manual maintenance of many pages is time consuming and error prone—in a word, ineffective. Therefore, the correct solution is to approach the problem as if it were a dynamic site, and generate static pages. Template Toolkit (http://www.template-toolkit.org/) provides a utility program called "*ttree*" that creates static pages from dynamically generated output. From now on, we will assume that you are developing a dynamic site.

User Language Detection

Another important question is the process of figuring out what language should be used when presenting the content. The following algorithm tries to answer this question:

1. Separate users into two groups: those who are visiting the site for the first time, and those who have previously visited the site.
2. If you use cookies to track users (or some other mechanism that stores the information on the client side), and a user connects from the same machine/account that was used when previously accessing your service, you should already know the language preferences: This bit of information can be stored in the cookie, thereby answering the question of language detection for the second group of users.
3. If you don't use cookies or some other mechanism to track users, it probably doesn't matter whether they have accessed the site beforehand because you have no way to tell what their language preferences are.
4. If users have registered with your service, their language preference will be known after they have logged in, since preferences can be stored on the server side. However, the problem is that you need to know the user's language to display the login form.
5. You can try to figure out the user's language by deducing it from the user's country. One way to determine the user's country is by doing a reverse DNS lookup on the client IP address. This yields the user's computer name. You can then use the top level domain (TLD) to make some reasonable assumptions about the language: Chances are, most users in the .fr domain can read French, for example. Since many hosts do not have correct reverse DNS mapping, you might also be tempted to deduce the country from the IP address itself. However, this approach is due to fail in many cases: There are plenty of users whose visible IP address is outside their country; for example, AOL users worldwide use AOL proxies located in the United States.

Eric and Stas are the authors of the upcoming book Practical mod_perl *(O'Reilly and Associates). Eric runs his own consulting business, Logilune, in Paris and can be reached at cholet@logilune.com. Stas is sponsored by TicketMaster to work on mod_perl development and can be reached at stas@stason.org.*

So, we are back to the first group of users—those we know nothing about. We must provide them a way to choose a language. The best method is to present a page with all available languages, with each language name written in that language. Each name is linked to the version of the service with content presented in that language.

We can go a little bit further and try to make an intelligent guess of the preferred language. This guess is made by looking at the *Accept-Language* header sent by most browsers. Localized versions of modern browsers set up the preferred language at install time. If users knows that it's possible to adjust the language preference in their browser, there is a chance that they will. For example they might set the following preferences:

```
German
English-US
French
```

which might mean the following: My preferred language is German. I also understand American English to some extent, and I know a little bit of French. (Of course, a user might know all three languages perfectly, but still prefer one language over another).

When a browser sends a request to a server, it generates the following header:

```
Accept-Language: de,en-us;q=0.7,fr;q=0.3
```

where each language is separated by a comma. In some browsers, the preference level can also be specified for alternate languages. So in this example, I've marked American English as 70 percent and French as 30 percent.

You can parse this header manually, but a better approach is to use standard CPAN modules. If you are using mod_perl, you can use the *Apache::Language* module; otherwise, use the all-purpose *HTTP::Negotiate* module bundled into the *libwww* distribution.

Browsers bundled with OS or ISP packages are usually preconfigured with the language of the country the package was issued in. So if users aren't computer savvy, chances are that the default language setting will be correct. If you have accepted this header, you may want to try your luck and present the top of the first page using the language derived from the header. But you still have to give users an option to change the language, since the browser setting might be incorrect for the particular user.

One of the big questions is how to build your database so that it will accommodate the site's multilingual capability

Remember that *Accept-Language* is useful for making your service more user friendly and sparing yourself the hassle of picking the right language, but it doesn't come as a replacement for the standard way of presenting the available languages.

At this point, we know the user's preferred language. In the case of a dynamic site, we proceed with generating the content. Otherwise, we simply direct the user to the right static content.

Generating the Content

When dynamic content is generated, at least two basic ingredients are used:

- Invariant data: page headers and footers, side navigation, and other table information that is always the same.
- Variant data: data that is not known *a priori*, which depends on the user or some other input.

When a site is generated in a single language, these two items are easily implemented: Either use templates for invariant data or hardcode it into the code, and retrieve the variant data from the database or through some other method. When the requirements include multilingualism, these tasks become more complex. I'm going to talk about each of them separately.

Fetching Dynamic Data

We separate the dynamic data requests into two groups: those that require user input and those that don't. A site search feature falls into the first category, whereas browsing the site belongs to the second one.

Searching

Let's use a movie server and a user whose preferred language is French as an example. Our user searches a movie by entering search keywords in a search box.

French includes accented characters. This is common in many other languages, too. An accented character usually uses some nonaccented character as a base. For example, characters: *â*, *à*, and *á* are all based on character *a*.

Because not all software supports accented characters, or appropriate keyboard maps are not always available, the user might generate input using only the base characters, without accents. In fact, even with proper software and hardware support, most French users will type keywords with no accents. The server is still expected to interpret this input correctly as if the accented characters were used.

We cannot guess which characters were inserted incorrectly, therefore, the obvious solution is to make the search index free of accented characters. This means that you'll have to keep two versions of the text; one version adjusted for the search, and the original unaltered version. You need the original because you still have to output the correct text, regardless of the user's input limitations.

In this article, I'll use the ISO-8859-1 character set, which is used by many Western European languages.

Listing 1 allows you to convert accented characters into their base characters. The code generates two methods: *iso_8859_1_lc()*, which turns any input using ISO-8858-1 into a lowercase, accent-free version; and *iso_8859_1_uc()*, which yields an accent-free uppercase version of its input.

For example, when you call:

```
$stripped_lc = $charsets{'iso-8859-1'}{lc}->('Bienvenüe');
```

$stripped_lc will be set to:

```
bienvenue
```

These functions are used twice: first, when creating the search index, and second, when accepting the search string, before the actual search is performed. Usually, using the lower case for searching is the accepted technique.

In addition, these functions can be used for sorting. Consider the following function:

```
sub cmp_nocase{
  return My::Language::lc($_[0], $_[1])
          cmp
         My::Language::lc($_[0], $_[2]);
}
```

which can then be used as:

```
my @correctly_sorted =
    sort { My::Language::cmp_nocase($lang,$a,$b) } @data;
```

where *$lang* is the currently used language (e.g., *fr*).

Browsing

When a user browses the site, preset data inputs are used (be careful to make sure that inputs you assume to be nonchangeable really can't be changed by users). For example, after a search has successfully completed and matches one or more records, you may list all the matched results or a subset of them. From now on, the user clicks on one of the links to get to the full record.

At this point, you may want to use the original text version using all the characters, but these should be encoded because when the link is clicked, it's possible that the browser will interpret the request incorrectly. To accomplish this, you can use *URI::Escape::uri_escape()* or *Apache::Util::escape_uri()* (a much faster implementation under mod_perl).

The text itself should be encoded as well so the browser will not mess it up. The *HTML::Entities::encode()* or *Apache::Util::escape_html()* functions can be used for that.

Data Retrieval

One of the big questions is how to build your database so that it will accommodate the site's multilingual capability. Obviously, you should avoid creating language-specific fields in every table that includes multilingual data. For example:

```
table movies:
_____________
title_fr
title_en
title_es
description_fr
description_en
description_es
....
```

is a bad idea because, as you can see, the table will require many columns. Don't forget that the number of columns will actually be doubled, since you need to duplicate all the columns for the searchable version of the text. Every time you want to support a new language, you'll have to alter the table and add many columns. A better solution is to place all language-specific data in one table:

```
id
lang
real_text
search_text
```

where *lang* specifies the language, *real_text* the real text, and *search_text* holds the version of the text adjusted for searching. *id* is needed to map every record into the table the data belongs to. This link back to the actual data table can be more complex and comprised of several fields. In one project, we used three fields to represent a unique data ID:

```
orig_table
orig_column
orig_id
```

The concatenation of these three fields gives us a unique mapping from a record in the text table to the data table, and the record it belongs to. For example:

```
SELECT * FROM lang where orig_table='movies'
AND orig_column='description' AND lang='fr'
AND search_text LIKE '%foo%'
```

will search only the description columns in the *movies* table. If we want to retrieve all language fields tied to some record, we can do:

```
SELECT * FROM lang where orig_table='movies'
AND orig_id=123456
```

If your particular database implementation cannot cope with all the textual data in all languages in one table, you may consider using one table per language.

Invariant Data

Finally, let's talk about invariant data. Data that doesn't change is either hardcoded in the code or, better yet, placed in a template. Let's take, for example, a search feature. The template will look something like Example 1. A simple mod_perl script that will parse this template and produce output is shown in Example 2.

So, these are the template and code used in a single-language site. When adding multilingualism to the site, we face this question: Should we use a template per language, or one template for all languages?

If you decide to go with the first option, you'll end up with many templates. However, keeping them synchronized will be a nightmare as changes must carefully be applied to each language template. A better approach is to keep all languages in one template. Modifications are easier because all strings are stored in the same place.

We have to find a way to parse this file and extract only the text in the requested language. Therefore, we've chosen to use XML tags, which will then be parsed by Template Toolkit so that texts in the right language will be selected.

We have used tag *<text>* for the text sections, and two-letter code tags for language-specific sections. Example 3 shows the search input template after applying these definitions, and Example 4 shows what the search results output template would now look like.

Template Toolkit allows us to provide our own template-parsing method. We use this feature to preprocess templates, turning *<text>* sections into conventional *[% IF %]*, *[% ELSE %]*, and *[% END %]* Template Toolkit directives. The module in Listing 2 overrides the default Template Toolkit parsing method.

This parser assumes that a template variable named *lang* holding the current language code will exist at request time when the template is processed. We then use Template Toolkit to process the template and generate the output, using the following code:

```
use Template;
use My::Template::Parser ();
my $r = shift;
$r->send_http_header('text/html; charset=ISO-8859-1');
my $t = Template->new();
$t->_init(PARSER       => My::Template::Parser->new,
         INCLUDE_PATH => '/search/path',
        ) or die $t->error();
$t->process('search.ttml',
           { input         => 'foo',
             total_results => 15,
             lang          => 'fr',
           },
           $r
          ) or die $t->error();
```

Handling Dates

Date and time need to be formatted according to the locale. Different countries have different conventions for date and time presentation. Where an American user will read:

```
Thursday March 22, 2001 2:25pm
```

a French user will expect:

```
Jeudi 22 Mars 2001 14h25
```

In this article, we will assume that these conventions are tied to languages, rather than countries. This is incorrect in reality, but this assumption is good enough to be used as an example. In reality, you may want to tie the conventions to countries and not languages. In this case, you would need to ask the user for country preferences.

We specify the following data set for each language, as shown in Listing 3. Then we use this data to produce the dates and times in the correct language using the correct format. These are handy compile-time constants that are used in the date and time generators:

```
use enum qw(YEAR MONTH DAY);
use enum qw(HOUR MINUTE);
```

Listing 4 shows some useful macros used in the formats above (they are derived from the format used by the *strftime()* function).

```
<!-- results set -->
[% IF input %]
  [% IF total_results %]
      A total of [% total_results %] movies were found.
      Displaying results
 [% ELSE %]
      No Results.
 [% END %]
[% END %]
<!-- input form -->
<form>
<input type="text"   name="input"  value="[% input %]" size="32">
<input type="submit" name="search" value="Search">
</form>
```

Example 1: Typical template with invariant data.

```
use Template;
my $r = shift;
$r->send_http_header('text/html');
my $t = Template->new(INCLUDE_PATH => '/templates/path');
$t->process('search.ttml',
          { input         => 'foo',
            total_results => 15,
          },
          $r
         ) or die $t->error();
```

Example 2: mod_perl script to parse the template in Example 1.

```
<form>
<input type="text"   name="input"  value="[% input %]" size="32">
<input type="submit" name="search"
      value="<text>
                 <en>Search</en>
                 <fr>Chercher</fr>
                 <it>Cercare</it>
             </text>">
</form>
```

Example 3: Input template after applying XML tags.

```
[% IF input %]
  <text>
       <en>Search Results</en>
       <fr>Résultats de la recherche</fr>
       <it>Risultati della ricerca</it>
  </text>
  [% IF total_results %]
      <text>
          <en>A total of [% total_results %] movies was found.</en>
          <fr>[% total_results %]
              [% IF total_results > 1 %]
                 films trouvés
              [% ELSE %]
                 film trouvé
              [% END %].</fr>
          <it>[% total_results %]
              [% IF total_results > 1 %]
                 movies trovati
              [% ELSE %]
                 movie trovato
              [% END %].</it>
      </text>
 [% ELSE %]
      <text>
          <en>No Results.</en>
          <fr>Aucun résultat.</fr>
          <it>Nessun risultato.</it>
      </text>
 [% END %]
[% END %]
```

Example 4: Search results output template.

Generating Correct *Charset* Headers

When the page is produced, it's important to specify a correct charset, so the browser will do the right thing when rendering the output. There are two techniques to accomplish that:

- The preferred method to specify the character set is to use the *charset* parameter of the 'Content-Type' HTTP header. For example, to specify that an HTML document uses ISO-8859-1, a server would send the following header:

  ```
  Content-Type: text/html; charset=ISO-8859-1
  ```

  With mod_perl, you can do that with:

  ```
  my $r = shift;
  $r->send_http_header('text/html; charset=ISO-8859-1');
  ```

- A less preferable method of setting the character encoding is by using the following tag in the 'HEAD' section of an HTML document:

  ```
  <META HTTP-EQUIV="Content-Type" CONTENT="text/html;
          charset=ISO-8859-1">
  ```

  This method requires that ASCII characters stand for themselves until after the <META> tag and often causes an annoying redraw with Netscape. The META HTTP-EQUIV method should only be used if you cannot set the *charset* parameter using the server.

Conclusion

We've discussed the following multilingual site development issues:

- It's almost always better to develop a dynamic site rather than a static one.
- Language selection is done by asking the user about it and/or looking at the *Accept-Language* header.
- Storing user preference is best done via cookies or by making the user log in.
- We have seen that generated output is comprised from semistatic template text and dynamic database content.
- We have seen how site browsing is different from site searching in terms of multilingual input processing.
- We have discussed ways that language-specific data can be stored in the database.
- We have seen how multilingual variants of text can coexist in the same template and have the code deal with that.
- We have seen how the presentation of dates and times can be adjusted to the user preferences.
- Finally, we have learned how to tell client browsers to render the output using a correct language encoding.

Resources

- mod_perl home page: http://perl.apache.org/
- mod_perl guide: http://perl.apache.org/guide/
- Template Toolkit home page: http://www.template-toolkit.org/
- Internationalization/Localization—Charset parameter: http://www.w3.org/International/O-HTTP-charset.html
- "A tutorial on character code issues" by Jukka Korpela: http://www.cs.tut.fi/~jkorpela/chars.html
- "Localizing Your Perl Programs" by Sean Burke and Jordan Lachler. *The Perl Journal*, Issue 13, Spring 1999.
- CPAN modules mentioned in this article:
  HTTP::Negotiate: http://search.cpan.org/search?dist=libwww-perl
  HTML::Entities: http://search.cpan.org/search?dist=libwww-perl
  Apache::Util: http://search.cpan.org/search?dist=mod_perl
  Apache::Language: http://search.cpan.org/search?dist=Apache-Language

(Code for this article is also available online at http://www.tpj.com/source/.)

TPJ

(Note: For best results, download these listings from http://www.tpj.com/source. Cutting and pasting from this PDF file may result in incorrect text encoding for the accented characters.)

Listing 1

```
package My::Language;

my %iso_8859_1_accents = (
    a => [ qw(à á â ã ä å À Á Â Ã Ä Å) ],
    c => [ qw(ç Ç) ],
    e => [ qw(è é ê ë È É Ê Ë) ],
    i => [ qw(ì í î ï Ì Í Î Ï) ],
    n => [ qw(ñ Ñ) ],
    o => [ qw(ò ó ô õ ö ø Ò Ó Ô Õ Ö Ø) ],
    u => [ qw(ù ú û ü Ù Ú Û Ü) ],
    y => [ qw(ý Ý) ],
);
# build translation strings
my (%in, %out);
for my $letter ('a'..'z') {
    my $uletter = CORE::uc $letter;
    # translate non-accented letters
    $in{uc}  .= $letter;
    $out{lc} .= $letter;
    $in{lc}  .= $uletter;
    $out{uc} .= $uletter;
    if (my $ra_accented = $iso_8859_1_accents{$letter}) {
        my $in = join '', @$ra_accented;
        $in{lc} .= $in;
        $in{uc} .= $in;
        $out{lc} .= $letter  x @$ra_accented;
        $out{uc} .= $uletter x @$ra_accented;
    }
}
# build translation subroutines
for my $type (qw(lc uc)) {
    my $sub = qq!
        sub iso_8859_1_$type {
           (my \$s = shift) =~ tr/$in{$type}/$out{$type}/;
           \$s
        }
    !;
    eval $sub;
}

# character sets
my %charsets = (
  'iso-8859-1'    => { lc => \&iso_8859_1_lc,
                       uc => \&iso_8859_1_uc,
                     },
);
```

Listing 2

```
package My::Template::Parser;
use strict;
use base qw(Template::Parser);

use constant LANG_RE        => qr{<([a-z]{2})>(.*?)</\1>}s;

######################
# constructor
sub new {
  my ($class, $options) = @_;
  my $self = $class->SUPER::new();
  $self->init($options);
  return $self;
}
######################
sub init {
  my ($self, $options) = @_;
  $self->{$_} = $options->{$_} for keys %$options;
}

######################
sub parse {
  my ($self, $text) = @_;
```

```
  # tokenize
  $self->_tokenize($text);

  # replace C<text> language sections with Template Toolkit
  # directives which will pick up the correct language text
  # at request processing time.
  $text = '';
  for my $section (@{$self->{sections}}) {
      my $translated = $section->{text};
      $translated =~ s{@{[LANG_RE]}}
                      {\[% IF lang=='$1' %\]$2\[% END %\]}gs
          if $section->{lang};
      $text .= $translated;
  }
  # proceed with standard Template Toolkit parsing
  return $self->SUPER::parse ($text);
}
########################
sub _tokenize {
  my ($self, $text) = @_;
  return unless defined $text && length $text;

  # extract all sections from the text
  $self->{sections} = [];
  while ($text =~ s!
         ^(.*?)                # $1 - start of line up to start tag
          (?:
              <text>                       # start of tag
              (.*?)                        # $3 - tag contents
              </text>                      # end of tag
          )
          !!sx) {
      push @{$self->{sections}}, { text => $1 } if $1;
      push @{$self->{sections}}, { lang => 1, text => $2||'' }
          if defined $2;
  }
  push @{$self->{sections}}, { text => $text } if $text;
}
########################
1;
__END__
```

Listing 3

```
# all languages
%languages = (
  en => { name        => 'english',
          charset     => 'iso-8859-1',
          months      => [qw(January February March April May
                             June July August September October
                             November December)],
          days        => [qw(Sunday Monday Tuesday Wednesday
                             Thursday Friday Saturday)],
          date        => {short  => '%m/%d/%Y',
                          medium => '%B %e, %Y',
                          long   => '%A %B %e, %Y',
                         },
          time        => {short  => '%H:%M',
                         },
        },
 it => { name        => 'italiano',
         charset     => 'iso-8859-1',
         months      => [qw(gennaio febbraio marzo aprile maggio
                            giugno luglio agosto settembre ottobre
                            novembre dicembre)],
         days        => [qw(domenica lunedì martedì mercoledì
                            giovedì venerdì sabato)],
         date        => {short  => '%d/%m/%Y',
                         medium => '%e %B %Y',
                         long   => '%A %e %B %Y',
                        },
         time        => {short  => '%H:%M',
                        },
       },

  fr => { name        => 'français',
          charset     => 'iso-8859-1',
          months      => [qw(Janvier Février Mars Avril Mai
                             Juin Juillet Août Septembre Octobre
                             Novembre Décembre)],
          days        => [qw(Dimanche Lundi Mardi Mercredi
                             Jeudi Vendredi Samedi)],
          date        => {short  => '%d/%m/%Y',
                          medium => '%e %B %Y',
                          long   => '%A %e %B %Y',
                         },
          time        => {short  => '%Hh%M',
                         },
        },
 es => { name        => 'español',
         charset     => 'iso-8859-1',
         months      => [qw(Enero Febrero Marte Abril Maio Junio
                            Julio Agosto Septiembre Octubre Noviembre
                            Diciembre)],
         days        => [qw(Domingo Lunes Martes Miércoles Jueves Viernes
                            Sábado)],
         date        => {short  => '%d/%m/%Y',
                         medium => '%e %B %Y',
                         long   => '%A %e %B %Y',
                        },
         time        => {short  => '%Hh%M',
                        },
       },
);
```

Listing 4

```
######################################################################
# strftime compatible specifiers
# %A    full weekday name
# %B    full month name
# %d    day of the month (01-31)
# %e    day of the month (1-31)
# %H    hour (00-23)
# %I    hour, 12-hour clock (01-12)
# %k    hour (0-23)
# %l    hour, 12-hour clock (1-12)
# %M    minute (00-59)
# %m    month (01-12)
# %p    AM/PM
# %S    second (00-60)
# %Y    year with century
# %y    year without century (00-99)
# %%    %
use vars qw(%strftime);
%strftime =
  (
   A => q/@{$languages{$lang}{days}||[]}[week_day($date)]/,
   B => q/@{$languages{$lang}{months}||[]}[$date->[MONTH]-1]/,
   d => q/sprintf('%02d',$date->[DAY])/,
   e => q/sprintf('%d',$date->[DAY])/,
   H => q/sprintf('%02d',$time->[HOUR])/,
   I => q/sprintf('%02d',$time->[HOUR]?$time->[HOUR]>12?$time->[HOUR]-
           12:$time->[HOUR]:12)/,
   k => q/sprintf('%d',$time->[HOUR])/,
   l => q/$time->[HOUR]?$time->[HOUR]>12?$time->[HOUR]-12:$time->[HOUR]:12/,
   M => q/sprintf('%02d',$time->[MINUTE])/,
   m => q/sprintf('%02d',$date->[MONTH])/,
   p => q/$time->[HOUR] && $time->[HOUR]<13 ? 'AM' : 'PM'/,
   S => q/sprintf('%02d',$s)/,
   Y => q/sprintf('%04d',$date->[YEAR])/,
   y => q/$date->[YEAR]%100/,
  '%' => q/\%/,
  );

These are two functions that show that accept either current date or
time, the language and the requested format:

# $time_str = format_time($lang,$format_type,$time)
# $time == [$HH,$SS]
################
sub format_time{
  my ($lang,$format_type,$time) = @_;
  # either can be zero
  return '' unless defined $time->[HOUR] and defined $time->[MINUTE];

  my $format = $languages{$lang}{time}{$format_type};
  warn("unknown time format: $format"), return '' unless $format;

  $format =~ s/\%(.)/$strftime{$1}/gee;
  return $format;
}

# $date_str = format_date($lang,$format_type,$date)
# where $date = [$YY,$MM,$DD]
################
sub format_date{
  my ($lang,$format_type,$date) = @_;

  return '' unless $date->[YEAR] and $date->[MONTH] and $date->[DAY];
  my $format = $languages{$lang}{date}{$format_type};
  warn("unknown date format: $format"), return '' unless $format;

  $format =~ s/\%(.)/$strftime{$1}/gee;
  return $format;
}
```

TPJ

Moshe Bar

Perl in High Performance Computing Environments

Linux has brought High Performance Computing (HPC) to the masses. Until a few years ago, only government agencies and big corporations could afford to crunch numbers with Crays and other supercomputers. Today, small companies or even individuals can build a cheap cluster of commodity Linux boxes and run compute-intensive applications.

Two distinct clustering paradigms exist for HPC: Beowulf-style and Single-System-Image-style (SSI). SSI clusters like openMosix (http://www.openMosix.org/) or openSSI (http://www.openSSI.org/) connect *n* Linux boxes to look like one giant, single computer with *n* CPUs. In the November *TPJ,* I showed how to make use of the simple but powerful Perl parallel fork manager module to create parallel environments. The parallel fork manager works best in SSI environments because each new forked child is automatically sent to a new cluster node for execution.

For more classic computing problems, such as fast Fourier transformations (FFTs), it is often more efficient to use the Beowulf approach. Beowulf technologies such as message passing interface (MPI) or parallel virtual machines (PVM) are based on external libraries implementing a virtual parallel computer paradigm. Programmers need to modify their applications to use parallelization directives to split up computational loops across multiple nodes.

Traditionally, programming languages such as Fortran and C have been used for number-crunching applications, in part due to the multitude of mathematical and engineering libraries available to programmers. In fact, HPC applications are quite often written ad hoc; that is, they are written by scientists for a particular, temporary problem and then discarded to be replaced by new ad hoc programs, taking the output of the previous program as input for the next step.

To these developers, speed of development is of prime importance. Fortran and C, however, do not lend themselves easily to fast development or ad hoc programming. Perl, on the other hand, is ideal for prototyping and ad hoc programming. Naturally, people have been devising ways to use Perl for number crunching.

Perl and/or scripting language opponents will be quick to point out that interpreted languages have, by definition, a speed disadvantage compared to compiled languages. In my own personal experience, this is only true where pure mathematical performance is concerned. However, as soon as you add some I/O and other outer-world interaction, Perl quickly gains ground in comparison. Additionally, given the ample computing power available on today's CPUs, a few percentage points advantage in cycle efficiency is not going to make such a big difference in execution time.

There are a number of parallel computing modules out there for Perl. The most widely used are the *Parallel::Pvm* and *Parallel::Mpi* modules from CPAN.

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

Using the Parallel Modules

Before hacking away on a number-crunching application, one should first install either PVM and/or MPI, and the corresponding Perl modules from CPAN. For some help in setting up the PVM environment, see http://help-site.com/c.m/prog/lang/perl/cpan/05/parallel/parallel-pvm/_d11729. For help with MPI setup, see http://www.jics.utk.edu/MPI/MPIguide/MPIguide.html.

Once your cluster is installed, say with PVM, you start the virtual parallel computer with simple statements like this:

```
Use Parallel::Pvm;
Parallel::Pvm::addhosts("foo", "bar");
```

Once your cluster is up and running, you register your program to be a PVM implicitly by calling any PVM function or explicitly by doing something like this:

```
$mytask = Parallel::Pvm::mytask;
```

The next step would be to let an executable run on, say, 16 virtual hosts. In PVM lingo, you call that "spawning" executables. You do that by executing a line like this:

```
($ntasks, &tids) = Parallel::Pvm::spawn("executable', 16,,argv);
```

In this example, the scalar *$ntasks* will hold the number of children spawned and *$tids* will hold the task ID's of the children. There are several arguments for the spawn function, obviously. The documentation of the module goes into great lengths to explain every single function's use.

Just like in MPI, you may want to send messages to instances of an executable running on a remote node. In *Parallel::Pvm,* you do that with a code sequence:

```
Parallel::Pvm::initsend ;
Parallel::Pvm::pack(1.333,"sample_message");
Parallel::Pvm::pack(100);
Parallel::Pvm::send($dtid,999);
```

The first statement makes sure we have a *send* buffer container, and then we fill it with a double, string, and integer value, respectively. Once we fill the container, we send it to a particular task, *$tid*, and we tag this message with *999*.

On the receiving end of this message, you unpack the content in reverse order by executing the following statements:

```
Return_code = Parallel::Pvm::recv;
$int_t = Parallel::Pvm::unpack;
($double_t,$str_t) = Parallel::Pvm::unpack;
```

There are dozens of other options and functions in *Parallel::Pvm*. You can build-in fault tolerance by respawning lost children, for instance. You can even provide parallel I/O or send nonblocking messages.

If you need to write a number-crunching application fast, then *Parallel::Pvm* is certainly worth considering. It's easy to use and powerful.

Parallel::MPI::Simple

In MPI (similar to PVM), the central idea is to have several instances of the same executable running on various nodes, and use message passing for coordination among the instances. Just as with PVM, installing the MPI libraries is as easy as typing "rpm –install" under Linux (or through similar means under other operating systems). Perl programs making use of the *Parallel::MPI::Simple* module should be launched with the standard MPI command:

```
mpirun –np 2 perl script.pl
```

MPI is very simple. These six functions allow you to write many programs:

```
MPI_Init
MPI_Finalize
MPI_Comm_size
MPI_Comm_rank
MPI_Send
MPI_Recv
```

In fact, in most cases, you don't even need communication between the nodes. Take, for instance, a simple program to calculate π on any number of nodes available. For that, we use a discrete integration (Simpson's rule) under the curve $x*x+y*y=1$, a circle of radius 1 for $0<x<1$, and multiply by 4.

Using MPI (see Listing 1), you first initialize MPI, then indicate the number of divisions. Now, within the integration loop, each node can compute a constant cycle of divisions. Finally, we assemble the results at the end at one of the instances.

Conclusion

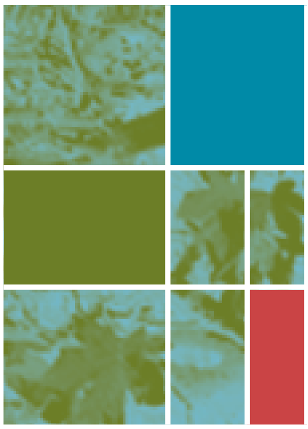
For quick and not-so-dirty development of number-crunching applications, Perl (with the appropriate modules) can be an intelligent choice. Because relatively few people use this technique as of yet, there is still a lot of room for research in this field, particularly in the benchmarking area. Please get back to me with samples of your own parallel Perl programs, as I plan to return to this subject in a future article.

TPJ

Listing 1

```
#!/usr/bin/perl

use lib qw(/usr/local/perlmod/Parallel-MPI-0.03/contrib/cpi/../../blib/arch
/usr/local/perlmod/Parallel-MPI-0.03/contrib/cpi/../../blib/lib);

$|=1;
use Parallel::MPI qw(:all);

sub f {
    my ($a) = @_;
    return (4.0 / (1.0 + $a * $a));
}

my $PI25DT = 3.141592653589793238462643;

MPI_Init();
$numprocs = MPI_Comm_size(MPI_COMM_WORLD);
$myid =     MPI_Comm_rank(MPI_COMM_WORLD);

#printf(STDERR "Process %d\n", $myid);

$n = 0;
while (1) {
    if ($myid == 0) {
        if ($n==0) { $n=100; } else { $n=0; }
        $startwtime = MPI_Wtime();
    }

    MPI_Bcast(\$n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    last if ($n == 0);

    $h   = 1.0 / $n;
    $sum = 0.0;

    for ($i = $myid + 1; $i <= $n; $i += $numprocs) {
        $x = $h * ($i - 0.5);
        $sum += f($x);
    }
    $mypi = $h * $sum;

    MPI_Reduce(\$mypi, \$pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if ($myid == 0) {
        printf("pi is approximately %.16f, Error is %.16f\n",
               $pi, abs($pi - $PI25DT));
        $endwtime = MPI_Wtime();
        printf("wall clock time = %f\n", $endwtime - $startwtime);
    }
}

MPI_Finalize();
```

TPJ



Something for Nothing

Simon Cozens

Everyone knows that the object of Perl modules is to make life easier for the programmer—to reduce the amount of code you end up writing. More correctly, you can think of CPAN modules as reducing the amount of auxiliary code in your programs, leaving you free to get on with the specific algorithms you wish to implement.

In this article, I'm going to embrace and extend Mark-Jason Dominus's concept of "structural code." When Mark talks about structural code in his Red Flags tutorial, he means code that doesn't get you any closer to doing what you want to do, but is required to keep the compiler happy or the code looking sane. For instance, in

```
sub remove_duplicates {
    my @list = @_;
    my %seen;
    return grep { !$seen{$_}++ } @list;
}
```

most of the code is structural. The first three lines of that code do nothing towards removing duplicates from a list— they fulfill no functional role, merely a structural one. This implementation is better, but still contains a lot of structural code that you can't avoid when you're programming in Perl:

```
sub remove_duplicates {
   my %seen;
   grep { !$seen{$_}++ } @_;
}
```

As I said, I'm going to extend that concept. In this article, structural code is anything that is generic to programming and is not essential to the specific algorithms and functionality of the program you're writing.

I'd like to introduce four Perl modules—three of mine, and one originally written by Michael Schwern—and show how they can be combined to reduce the amount of structural code in an application to nearly zero. We'll first take a brief look at the four modules, then we'll show how they worked in a recent application of mine.

Config::Auto

Almost every single application needs to store a user's configuration settings. The end result is that every single application generally includes some code for dealing with whatever configuration format the programmer chose. Such a case is a prime candidate for modularization, and indeed there are a number of modules that can deal with various formats: *XML::Simple* for the ever-popular XML, *Config::IniFiles* for Windows-style INI, and several others. For UNIX applications, the standard configuration formats are either a variant of *key* = *value* (from lynx):

```
# all cookies.
accept_all_cookies=off

# bookmark_file specifies the name and
# location of the default bookmark file
# into which the user can paste links for
# easy access at a later date.

bookmark_file=lynx_bookmarks.html
```

or colon separated (as in /etc/groups and friends):

```
nobody:*:-2:
nogroup:*:-1:
wheel:*:0:root
daemon:*:1:root
```

Or maybe space separated (this one from is "gltron," a rather enjoyable OpenGL light-bikes game):

```
iset show_help 0
iset show_fps 1
iset show_wall 1
iset show_glow 1
iset show_2d 1
```

I've had to write code to deal with all of these different formats many times over, and I finally gave up: I had what I call a "once and for all" moment. I wanted to sit down and crunch out some

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

code that would just handle whatever I threw at it, and know that I would never ever have to tackle this problem again in my Perl programming career. Try it. It's tremendously freeing.

So I wrote *Config::Auto*, which parses all of the above formats and more. The idea is not that it gives the user a complete free-for-all. Ideally, you would specify what format you were prepared to read, what sort of data structure you expected to get at the end of it, and then you would know that the parser would be able to handle it with no additional work needed.

In its most basic use, you would say:

```
use Config::Auto;
my $config = Config::Auto::parse("~/.myapprc", format => "equal");
```

There are a number of things that every novice Perl program reinvents, despite there being perfectly round wheels out there

to parse an equals-separated configuration file such as the .lynxrc above. But if we're trying to avoid extraneous code, why not have the parser work out what sort of configuration file it's been handed?

```
use Config::Auto;
my $config = Config::Auto::parse("~/.myapprc");
```

And now it'll take a long look at your *rc* file and determine what format it looks like it's in.

And actually, there's no reason why, assuming standard naming conventions, you should have to tell it where the configuration file is, anyway. If your program is called *myapp*, then it's a reasonable guess that if the user has a ~/.myapprc file, those are the configuration settings. *Config::Auto* also tries a few other standard locations, to leave us with:

```
use Config::Auto;
my $config = Config::Auto::parse();
```

Configuration file handled! Structural code: zero. (Well, near zero. Future versions of *Config::Auto* may well declare and populate a *$main::config* variable for you on import. But maybe there is such a thing as Too Much Magic.)

Attribute::Persistent

The next area that requires too much code is storing persistent data. As I've mentioned in a previous column, even with *AnyDBM_File* and *MLDBM*, handling persistent variables is still a pain in the neck. *Attribute::Persistent* just takes all the pain away, once and for all, with no *tie* and no structural code at all.

```
use Attribute::Persistent;
my %hash :persistent; # And that's all.
```

Persistent storage handled! Structural code: zero.

Getopt::Auto

In a recent comp.lang.perl.moderated thread, it was pointed out that there are a number of things that every novice Perl program reinvents, despite there being perfectly round wheels out there; a command-line options processing system was one of them. Here, I disagree.

There are a number of different styles of command-line options: —*long*, —short, and the CVS-style "bare" command. In a similar vein to *Config::Auto*, I wanted a system that handled all of them.

But then I realized there was a more serious problem. If you're implementing something with an interface similar to CVS (that is, a single executable that can perform various commands, although it could be argued that this is not the UNIX Way), then you'll end up with a horrific piece of code that would look something like this:

```
my $command = shift @ARGV;
if ($command eq "add") {
   do_add(@ARGV);
} elsif ($command eq "subtract") {
   do_subtract(@ARGV);
} ...
```

The two equally dissatisfying alternatives look slightly better:

```
my %commands = ( add => \&do_add,
                 subtract => \&do_subtract,
                 ...
               );
my $what = shift;
if (exists $commands{$what}) {
   $commands{$what}->(@ARGV)
}
else { do_help() }
```

Or even

```
no strict 'refs';
my $what = shift;
&{"do_$what"}(@ARGV);
```

But they have two problems: First, you still need to handle things like —*help* and —*version* separately, and your —*help* text will generally repeat all the possible arguments over again. Opponents of structural code will know that repetition is to be avoided at all costs: This is a specific case of the Prime Rule of programming and user-interface design—"You should never tell the computer anything it already knows or can reasonably be expected to work out." If you're a manager, you might like to contemplate the fact that programmer time is expensive and computer time is cheap. Who should be doing the boring work?

The bigger problem is that all this is structural code once again. Dispatching to the appropriate routine is useful, but it's not as useful as actually doing the work of your program. So I had another once-and-for-all moment and decided that something else should be implementing this structural code. That "something else" is *Getopt::Auto*. As you can probably tell, I'm pretty fond of the idea that computers should do things *auto*matically—it is, after all, what they're for.

With *Getopt::Auto*, you simply declare what commands you're willing to process, maybe give some help text for them, and the module does the rest. For instance:

```
use Getopt::Auto (
   [ "—add", "Add two numbers together", \&do_add ],
    [ "—subtract",
      "Subtract one number from another",
      \&do_subtract
    ],
      ...
);
```

With no further code, *yourapp —add 3 5* will call *do_add(3,5)*. And, as an added bonus, you get *—version* and *—help* free of charge:

```
% yourapp —help
yourapp —help - This text
yourapp —version - Prints the version number

yourapp —add - Add two numbers together
yourapp —subtract - Subtract one number from another
```

Of course, you may not like GNU-style *—long* options. Let's try again with CVS-style options without specifying the subroutines explicitly:

```
use Getopt::Auto (
  [ "add", "Add two numbers together" ],
  [ "subtract", "Subtract one number from another" ],
    ...
);
```

This time, *yourapp add 3 5* will call *add(3,5)*; *help* will still work and will now spit out the commands in the new bare style. You write the specification, and *Getopt::Auto* takes care of the rest.

The more alert of you may well be asking "Isn't this specification structural code?" Well, yes; I thought of that. What would be really nice is if you could say:

```
use Getopt::Auto;
```

and it would just work. Well, with one proviso, it does. The proviso is that you must provide POD documentation for each subroutine you want to turn into a command. But of course, all of your subroutines are documented anyway, so that shouldn't be a problem.

Here's our fully automated calculator example:

```
use Getopt::Auto;
our $VERSION = "1.0";

=head2 add - Adds two numbers together

   calc add x y

Adds x and y together and prints the result.

=cut

sub add { print $_[0] + $_[1], "\n" }

=head2 subtract - Subtracts one number from another

   calc subtract x y

Subtracts y from x.

=cut

sub subtract { print $_[0]-$_[1], "\n" }
```

Now we can say:

```
% calc —add 3 5
8

% calc —help
This is calc, version 1.0

calc —help - This text
calc —version - Prints the version number

calc —add - Adds two numbers together[*]
calc —subtract - Subtracts one number from another[*]

More help is available on the topics marked with [*]
Try calc —help —foo
```

And if we follow its suggestion:

```
% calc —help —add
This is calc, version 1.0

calc —add - Adds two numbers together

       calc add x y

Adds x and y together and prints the result.
```

Options processing and subroutine dispatch handled! Structural code: zero.

Class::DBI

The final module is not one of my own, but it's so efficient at removing structural code in database-backed applications that it absolutely has to be mentioned. Database applications with the DBI are breeding grounds for structural code: Either you spend a lot of time handling the various *select*, *insert*, *update*, and *delete* calls yourself, or you use some kind of abstraction layer that does some of the work for you.

Class::DBI is like this abstraction layer, except that in most cases, it does almost all of the work for you. With *Class::DBI*, you set up one subclass that represents your database:

```
package Myapp::DBI;
use base 'Class::DBI';
```

and then tell it what your DBI parameters are:

```
Myapp::DBI->set_db('Main', 'dbi:mysql:myapp');
```

No connecting, no disconnecting, no mucking about with handles. But how do you get at the data? Well, you need a class for each of the tables you want to play with:

```
package Myapp::Person;
use base 'Myapp::DBI';
Myapp::Person->table("person");
```

Next, tell it the columns you're interested in, starting with the primary key:

```
Myapp::Person->columns(All => qw(
                                      id
                                      name
                                      department
                                      salary
                                 )
                       );
```

and away you go: Your class now has *create*, *retrieve*, and *search* methods to return *Person* objects, and you also have accessor methods for each of the columns.

```
# 3% raise for all programmers!
for my $person (Myapp::Person->search({
                 department  => "programming"}) {
   $person->salary($person->salary()*1.03);
}
```

There are good tricks for handling relationships between tables and between database and nondatabase objects; I refer you to Tony Bowden's article on *Class::DBI* for perl.com at http://www.perl.com/pub/a/2002/11/27/classdbi.html.

While this removes most of the rigmarole of handling data in databases, it still violates the Prime Rule because we're having to tell the computer about the columns in our database tables. In the vast majority of cases, the database can tell *us* what columns it has. Unfortunately, the way it tells us is generally database specific. So *Class::DBI* has certain database-specific add-on modules, such as *Class::DBI::mysql*. (It's only a matter of time before someone combines them all…)

Now we can tell our *Myapp::DBI* to inherit from this:

```
package Myapp::DBI;
use base 'Class::DBI::mysql';
...
```

and the need to detail the columns goes away:

```
package Myapp::Person;
use base 'Myapp::DBI';
__PACKAGE__->set_up_table('person');
```

(*Class::DBI* folk tend to use *__PACKAGE__* instead of repeating the class name; this is slightly related to the Prime Rule. If you ever need to change the class's name, you only want to be changing it in one place.)

But even this code is reasonably structural! The computer not only knows what columns it has in its database tables, but it also knows what tables it has. With *Class::DBI::Loader*, we can get it down to:

```
use Class::DBI::Loader;
Class::DBI::Loader->new( dsn => "dbi:mysql:myapp",
                          namespace => "MyApp");
```

and now we can use *MyApp::Person* as before.

Database access is handled with very little structual code indeed.

Putting It All Together

We've seen four tools that give us a great deal of functionality for very little cost in code. With all of these modules, what we gain in brevity, we sacrifice in flexibility. For instance, to make absolutely full use of *Class::DBI* requires some investment, in terms of tuning access to the columns of each table and declaring the various relationships between columns longhand.

In the code that I write from day to day, I try to strike a balance; the last thing you really want are classes and variables magicking themselves into existence without your really being aware of them. So, for instance, I don't use *Class::DBI::Loader*. I prefer to declare each table's class manually.

Well, not exactly "manually." That wouldn't be a very good use of my time. Instead, I have a little script that produces an application template—a basis for an application that uses many of the aforementioned techniques. I spend most of my preparation time working out the best database schema, and then I type something like:

```
appgen PerlBooks
```

Anyone who bears the scars of the old dBase III+ application generator will recognize the name and the concept; *appgen* goes away, examines the database, and spits out a number of skeleton files, which I will turn into my eventual application.

So, first we take the name of the namespace (*PerlBooks*), turn it into our database name (*perlbooks*), and try to use *Class::DBI::Loader* on that database:

```
use Class::DBI::Loader;

my $namespace = shift;
my $database  = lc $namespace;

my $loader = Class::DBI::Loader->new(
   dsn       => "dbi:mysql:$database",
   namespace => $namespace,
);
```

(The application generator itself doesn't need to be portable to multiple databases—although its output must be!—since, for better or worse, I do all my development on MySQL.)

Now we do a little ugly messing about. First, we want our own copy of the database handle so we can prod the database, and this allows us to ask it for its tables. Instead of repeating the DSN in the DBI connection, we ask *$loader* what DSN it used:

```
my $dbh = DBI->connect(
    @{ $loader->_datasource }
) or croak($DBI::errstr);
my %tables = map { $_ => 1 } $dbh->tables;
```

Now for each table, we want to spit out a module representing that table in the ordinary *Class::DBI* way:

```
foreach my $table (keys %tables) {
   my $class = $loader->_table2class($table);
   my $ref   = $dbh->selectall_arrayref(
                              "DESCRIBE $table"
                                         );
```

Most of this code is cobbled together from bits of *Class::DBI::mysql* and *Class::DBI::Loader*. Here, we turn the table name (say, *account*) into the appropriate class name, *PerlBooks::Account*, using *Class::DBI::Loader*'s built-in method, and then get a description of the database table.

Now we want to know what the primary key is, so we *grep* that out of the table's description:

```
my ( @cols, $primary );
foreach my $row (@$ref) {
   my ($col) = $row->[0] =~ /(\w+)/;
   push @cols, $col;
   next unless $row->[3] eq "PRI";
   die "$table has composite primary key" if $primary;
   $primary = $col;
}
die "$table has no primary key" unless $primary;
```

This gives us *$primary* and a list of columns in *@cols*. At this point, we can write our class:

```
my $file = $class; $file =~ s{::}{/}g;
open OUT, ">$file.pm" or die $!;
print OUT <<EOF;
package $class;
use base '${namespace}::DBI';
__PACKAGE__->table($table);
__PACKAGE__->columns( Primary => q{$primary} );
__PACKAGE__->columns( All     => qw{@cols} );
EOF
```

We do something that *Class::DBI::Loader* doesn't do, which is to guess the "has-a" relationships in each table. For instance, if we have a column in *transaction* called *account*, we guess this is a reference to the primary key in the *account* table:

```
for (@cols) {
   if (exists $tables{$_}) {
      print OUT "__PACKAGE__->has_a($_ => q{".
         $loader->_table2class($_)."});\n";
   }
}
```

This spits out something like:

```
__PACKAGE__->has_a(account => q{PerlBooks::Account});
```

Then the *account* method in our *PerlBooks::Transaction* will no longer produce a numeric ID, but will instead produce a *PerlBooks::Account* object. Finally, our generator finishes off the current class:

```
print OUT <<EOF;

1;
EOF
   close OUT;
}
```

Now we can get onto the main *PerlBooks* module, which has to load up the others, and any other modules we might want to use:

```
open OUT, ">$namespace.pm" or die $!;
print OUT "package $namespace;\n\n";
print OUT "use Config::Auto\n";
print OUT "use ".$loader->_table2class($_).";\n"
                                  for keys %tables;
print OUT "\n1;\n";
close OUT;
```

Our *PerlBooks::DBI* class is generated next, but this needs to be done a little carefully. As we've seen, *Class::DBI* expects the main class that subclasses it to tell it the connection parameters, including the username and password. Typically, though, we don't want to store the username and password in our main program files, so we bring them in from a *PerlBooks::Config* class:

```
open OUT, ">$namespace/DBI.pm" or die $!;
print OUT <<EOT;
package ${namespace}::DBI;
use ${namespace}::Config;
use base 'Class::DBI';
__PACKAGE__->set_db('Main',
   'dbi:'.\$${namespace}::Config::dbd.
   ':'.\$${namespace}::Config::db,
   \$${namespace}::Config::username,
   \$${namespace}::Config::password);
__PACKAGE__->autocommit(1);
1;
```

```
EOT
close OUT;
```

Finally, we write out a skeleton version of that *PerlBooks::Config* class to be overwritten by the real values of the username and password by our application's installer:

```
open OUT, ">$namespace/Config.pm" or die $!;
print OUT <<EOT
package ${namespace}::Config;
our (\$dbd, \$db, \$username, \$password) =
    ("mysql", "$database", "", "");

1;
EOT
```

This is as far as I've currently progressed with the application generator, and already it has saved me a lot of work. But as I look at it now, there's a lot more it ought to do. For instance, it could easily spit out an *ExtUtils::MakeMaker*-based installation program, which would prompt for the correct username and password and write the *::Config* module. As Alan Perlis said, "Programs that write programs are the happiest programs of all"—this is a program that writes a program that writes a program!

The other obvious task for my application generator is to spit out the main application file *perlbooks*, containing at least:

```
use PerlBooks;
use Getopt::Auto;


...
```

But this may be overkill, and currently I'm sufficiently happy with the ability to point my application generator at a database and come out with most of what I need to start writing database-driven application code that is relatively free from structural code.

In Closing

There are a number of things you could take away from this article. You might think that I've created three really interesting modules that you should go and have a look at—but then, I know who you'll come to for help with them, so maybe that's not such a good idea.

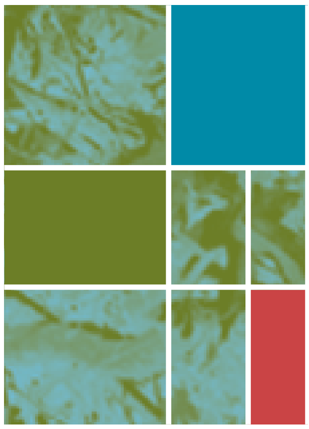
You might take away the Prime Rule—never tell a computer what it already knows or can be reasonably expected to find out for itself. If you do, I promise it'll radically impact the way you think about user interfaces.

You could take away the fact that, with CPAN modules, there may well be More Than One Way To Do It, but there's almost always an easier way.

But what I really want you to take away is that programming really ought to be fun. If you find that your programming is becoming a drudge, see if there isn't a way you can abstract away the drudge, whether there's already a module out there that does it all for you or whether you should sit down and tackle it in a once-and-for-all moment.

Doing so will free you from banging out code for the sake of code, and allow you to get on with the interesting bit of your job—having ideas, working out the best way to get things done, solving problems—and my fervent hope is that it'll make programming fun for you once again.

TPJ



Creating Perl Application Distributions

brian d foy

Most Perl users have run a Makefile.PL file when installing a module, and module authors know how to use one to make their module distributions, but most people do not know that they can use the same mechanism to distribute their scripts. The Makefile.PL file can help configure, test, and install scripts as well as modules.

The Makefile.PL file is just a Perl script that invokes the *ExtUtils::MakeMaker::WriteMakefile* function, which creates a Makefile that actually does all of the work (the *make* utility comes with most UNIX platforms and is available for Windows and other platforms). The typical installation sequence consists of four commands:

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

The first line, *perl Makefile.PL*, creates the Makefile, which controls all of the action. The *make* line runs the Makefile with the default target ("all"), which copies the right files to the right places—usually a subdirectory of "blib" (build library) to prepare for installation. The *make test* line is optional, and runs either a test.pl file or the t/*.t files, which test the script. Other people have written plenty about testing; I'll focus on the other parts in this article. The *make install* command takes the files in blib and puts them in the right place based on various configuration directives and defaults.

Usually, the end user needs to put the script in a particular place and set the script file permissions—the Makefile can do this automatically. To illustrate this, I start with a fictitious Perl script called "buster."

I create a directory for this script and the files that will go with it, and create a simple Makefile.PL:

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

```
prompt$ ls
Makefile.PL  buster
```

To create the Makefile, Makefile.PL loads *ExtUtils::MakeMaker* and calls the *WriteMakefile()* function, which takes a list of key-value pairs that describes how it should create the Makefile. Each key is explained in the *ExtUtils::MakeMaker* documentation, which I have printed and keep near my desk.

```
use ExtUtils::MakeMaker;

WriteMakefile(
   'NAME'          => 'buster',
   'VERSION'       => '0.10',

   'EXE_FILES'     =>  [ 'buster' ],

   'PREREQ_PM'     => {},

   'INSTALLSCRIPT' => "$ENV{HOME}/bin",
   );
```

The NAME key gives a name to the distribution and does not have to be the name of the script. I'll show why the NAME key is important later. I set the version of the distribution with the VERSION key (modules typically get their distribution from a module file with VERSION_FROM). The EXE_FILES key has an anonymous array of file names as its value. When I run *make*, the Makefile will move those files into the blib/script directory. The PREREQ_PM key has an anonymous hash as its value and lists all of the modules on which the script depends. Installers like CPAN.pm can automatically fetch and install these modules. The INSTALLSCRIPT key names the directory in which to install the script. I hard code a value for INSTALLSCRIPT so the script will show up in my personal bin directory.

Once I have my Makefile.PL, I can go through the steps I listed earlier. The script ends up in my personal bin directory (/Users/brian/bin/), and the default file permissions are 0555, meaning the owner, group, and everyone has read and executed permissions. The script is in the right place and ready to run, and the end user did not have to learn about copying files or setting permissions; see Example 1.

If I want to install buster in a different location, I have to tell Makefile.PL where to install it. I specify an alternate value for INSTALLSCRIPT on the command line when I run Makefile.PL. The value on the command line overrides the one in *WriteMakefile()*.

```
$ perl Makefile.PL INSTALLSCRIPT=/usr/local/bin
```

The Makefile gives me many other benefits. I use it to automatically create a distribution file that I can give to other people so they can install my script easily. Before I create the distribution, I need to create the list of files that should go into the distribution. The *make manifest* command creates a file named MANIFEST and adds files from the current directory to it. It has a default list of files it ignores (like Makefile), so it only puts the relevant files in MANIFEST.

```
prompt[3036]$ make manifest
/usr/bin/perl "-MExtUtils::Manifest=mkmanifest" -e mkmanifest
Added to MANIFEST: MANIFEST
Added to MANIFEST: Makefile.PL
Added to MANIFEST: buster
```

I run the *make dist* command to create the distribution. It looks at the values I specified in the NAME and VERSION keys when I ran *WriteMakefile* and puts them together with a hyphen (-), then creates a temporary directory with that name. It looks in MANIFEST and copies the listed files into the new directory, *tar*s and *gzip*s the directory, and finally removes the temporary directory.

```
prompt$ make dist
rm -rf buster-0.10
/usr/bin/perl "-MExtUtils::Manifest=manicopy,maniread"
      -e "manicopy(maniread(),'buster-0.10', 'best');"
mkdir buster-0.10
tar cvf buster-0.10.tar buster-0.10
buster-0.10
buster-0.10/buster
buster-0.10/Makefile.PL
buster-0.10/MANIFEST
rm -rf buster-0.10
gzip –best buster-0.10.tar
```

If I want to use ZIP instead of *tar* and Gnu *zip*, I run *make zipdist*:

```
prompt$ make zipdist
rm -rf buster-0.10
/usr/bin/perl "-MExtUtils::Manifest=manicopy,maniread"
```

```
prompt$ perl Makefile.PL
Writing Makefile for buster

prompt$ make
cp buster blib/script/buster
/usr/bin/perl "-MExtUtils::MY" -e "MY->fixin(shift)"
blib/script/buster

prompt$ make test
No tests defined for buster extension.

prompt$ make install
Installing /Users/brian/bin/buster
Writing /usr/local/lib/perl5/darwin/auto/buster/.packlist
Appending installation info to
/usr/local/lib/perl5/darwin/perllocal.pod
```

Example 1: The make ***process for buster.***

```
      -e "manicopy(maniread(),'buster-0.10', 'best');"
mkdir buster-0.10
zip -r buster-0.10.zip buster-0.10
  adding: buster-0.10/ (stored 0%)
  adding: buster-0.10/buster (stored 0%)
  adding: buster-0.10/Makefile.PL (deflated 25%)
  adding: buster-0.10/MANIFEST (deflated 2%)
rm -rf buster-0.10
```

I am not satisfied with that, though. I want to create some tests for my script so I can see the damage that I do when I work on it. The Makefile gives me the testing framework for free through *Test::Harness*. So far *Test::Harness* has simply told me that there are no tests to run.

I create a directory named “t”. *Test::Harness* will look in this directory for files matching the pattern “*.t” and run those as test files. In this example, I make a simple test to ensure that the script compiles. I call this test “compile.t”. In the test script, I load the *Test::More* module, which insulates me from most of the testing details. I tell *Test::More* that I have one test to run. In the next line, I call *perl -c* in backticks so Perl runs itself with the -c switch on the buster script in the blib/script directory (which is what *make* did). The backticks return the output, which I store in *$output*. The *Test::More* module provides a *like()* function that compares the first argument to the regular expression in the second argument. If the regex matches, the test passes; if not, it fails. The third argument to *like()* is the name I choose for the test.

```
use Test::More tests => 1;

my $output = 'perl -c blib/script/buster 2>&1';

like( $output, qr/syntax OK$/, 'script compiles' );
```

The *make test* output looks much different now; see Example 2.

To see the test fail, which is always a good idea so I know it will catch failures, I introduce a small syntax error into buster. Once I make the change, the buster script is different than the one in the blib/script. The Makefile catches this and copies the updated version into blib/script and then runs the tests. The *Test::More* module shows me the output that it actually got, and what it expected (“syntax OK”); see Listing 1.

So now my distribution has its first test file, and I need to add it to MANIFEST. I use *make manifest* again, which recognizes the new files and adds them to MANIFEST. Sometimes extra files get into MANIFEST, especially if I have been doing other things in that directory (in which case I edit them out of MANIFEST.)

```
prompt$ make manifest
/usr/bin/perl "-MExtUtils::Manifest=mkmanifest" \
         -e mkmanifest
Added to MANIFEST: t/compile.t
```

```
prompt$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e"
"test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/compile....ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.13 cusr +  0.06 csys =  0.19
              CPU)
```

Example 2: The make test ***output after adding a compile test.***

After testing my script, I want to create a new distribution, but I first use the *make disttest* command, which creates a distribution, then unwraps it in its own directory and runs *make test* on it. This tests the distribution to make sure all of the tests pass when it only has the files from the distribution (those in MANIFEST), rather than all of the files I have in my working directory. This test catches omissions from MANIFEST.

```
three_brian[3064]$ make disttest
rm -rf buster-0.10
/usr/bin/perl \
   "-MExtUtils::Manifest=manicopy,maniread" \
   -e "manicopy(maniread(),'buster-0.10', 'best');"
mkdir buster-0.10
mkdir buster-0.10/t
cd buster-0.10 && /usr/bin/perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for buster
cd buster-0.10 && make LIB="" LIBPERL_A="libperl.a"
      LINKTYPE="dynamic" PREFIX="/usr/local" OPTI
      MIZE="" PASTHRU_DEFINE="" PASTHRU_INC=""
cp buster blib/script/buster
/usr/bin/perl “-MExtUtils::MY" -e "MY->fixin(shift)"
      blib/script/buster
cd buster-0.10 && make test LIB=""
      LIBPERL_A="libperl.a" LINKTYPE=”dynamic" PRE
      FIX="/usr/local" OPTIMIZE="" PASTHRU_DEFINE=""
      PASTHRU_INC=""
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Com
      mand::MM" "-e" "test_harness(0, 'blib/lib',
      'blib/arch')" t/*.t
t/compile....ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.16 cusr +
      0.04 csys =  0.20 CPU)
```

The *ExtUtils::Makemaker* framework provides much more functionality than I have covered here—these are just the basics to get you started. My buster script now has a full-fledged, full-featured distribution framework. I can create distributions, test them, and easily distribute them. Other users can easily install them because the distribution uses the familiar Perl installation sequence.

TPJ

Listing 1

```
prompt$ make test
cp buster blib/script/buster
/usr/bin/perl "-MExtUtils::MY" -e "MY->fixin(shift)" blib/script/buster
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e"
          "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/compile....NOK 1#     Failed test (t/compile.t at line 5)
# 'String found where operator expected at blib/script/buster line 6, near
# "yprint "Hello World!\n""
#       (Do you need to predeclare yprint?)
# syntax error at blib/script/buster line 6, near "yprint "Hello World!\n""
# blib/script/buster had compilation errors.
# '
#     doesn't match '(?-xism:syntax OK$)'
# Looks like you failed 1 tests of 1.
t/compile....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/1 tests, 0.00% okay
Failed Test Stat Wstat Total Fail  Failed  List of Failed
-----------------------------------------------------------
t/compile.t    1   256     1    1 100.00%  1
Failed 1/1 test scripts, 0.00% okay. 1/1 subtests failed, 0.00% okay.
make: *** [test_dynamic] Error 2
```

TPJ

Graphics Programming in Perl

Jack J. Woehr

Graphics Programming in Perl, by Martien Verbruggen, is a very fun book. It's fun because Perl is fun, and it's about doing something pretty tricky in Perl—manipulating image data (for example, dynamically generating images like charts or figures as part of a CGI in response to a web request). It covers all of the graphics Perl modules significant at the time of authorship (see Verbruggen's comments on *Imager* below).

Graphics Programming in Perl is also fun in that it's fun to read, well-written, and well-produced. I hesitate to say what follows because I don't want to make the author or potential authors overly self-conscious, but I think it helped the book that the author's native language is not English. He doesn't blather on—he expresses things economically in the tone you'd encounter in a helpful newsgroup post.

Before tackling this book, I recommend a good familiarity with

- Perl.
- Perl module installation.
- Downloading, configuring, compiling, and installing C/C++-coded open-source software libraries.
- Using a draw program.

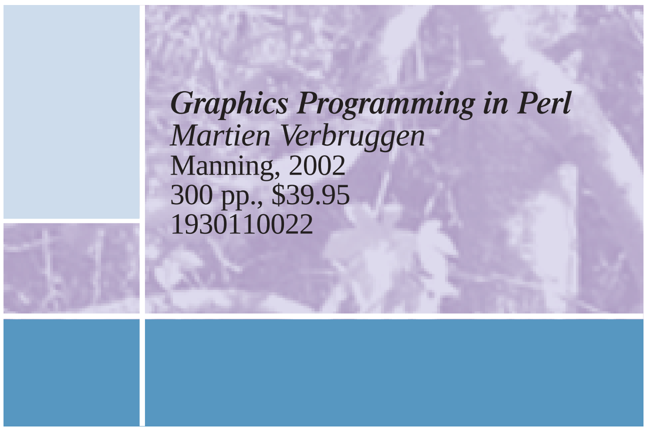
The introductory discussion of graphics is at an intermediate level. The targeted reader either already understands the basic elements of computer graphics or is quite good at linear algebra.

The web site for the book is http://www.manning.com/verbruggen/ where you can:

- Read the table of contents.
- Download sample chapters.
- Purchase the paper or ebook edition.
- Download source code.
- Download errata (none reported yet).
- Participate in the author's forum.

I chatted online with author Martien Verbruggen from his home in Turramurra, a northern suburb of Sydney, Australia.

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



Graphics Programming in Perl
Martien Verbruggen
Manning, 2002
300 pp., $39.95
1930110022

TPJ: Is Sydney a hi-tech mecca?
MV: Not too much. Though Australia has a very respectable technical history in early computing, as well as some brilliant minds nowadays—names that come to mind are Andrew Tridgell and Damian Conway. You know Andrew from Samba and Rsync, and Damian is currently redesigning Perl with Larry Wall. The historical bits I am not too sharp on; after all, I am not even Australian! I've been here since September 1996. I used to be in the Netherlands before then, which is where I was born, raised, and educated.

TPJ: What university did you attend?
MV: I went to the Eindhoven University of Technology. I did Chemical Engineering there. When I moved here, I chose a different direction.

TPJ: So your computer science interest dates from your move to Australia?
MV: Not really, I have been playing with computers since the late '70s, early '80s: Apple II, Commodore Pet, 32, 64, Amiga, etc. Then, at the university, I programmed a lot because I had a knack for it. I mainly worked in Fortran and Pascal in the early years, until I discovered C, and a little later, Perl. I was made system admin by default of the DEC Ultrix cluster of the Chem department, so my interests started shifting. I just made it permanent when I moved here. But my career "officially" started in November 1996.

TPJ: So what was your new responsibility in '96 that got you coding?
MV: I started working for the Trading Post Australia as a web programmer for this new web site that they created. I had no real qualifications, of course, except a few things that were floating around on the open-source platform. No CS degree or previous programming employment, but I got hired anyway and have been with them ever since. My responsibilities have shifted and grown a bit. I'm now development manager for two of the five business units in the company, which is now nationwide (it used to be three companies, a merger made it one). I don't really do much web stuff anymore and, unfortunately, not enough hands-on coding anymore, either. I keep fighting for less paperwork and more coding time, but the company fights back :) I still try to keep active by doing stuff at home and in my free time, and by keeping up with the newsgroups that I think are interesting.

TPJ: What is it you are developing?
MV: Trading Post publishes classified advertising periodicals. I think there are Trading Posts in the USA as well. There are differently sized Trading Posts: The Sydney one is about 200 pages weekly. The Melbourne TP was founded in 1966, and the Sydney one in 1968. My department takes care of all software in the particular business units we are responsible for. This includes the ad-taking systems (back-end, front-end, reporting), accounts receivable, billing, other reporting, outbound telemarketing. Most of this stuff runs with a Sybase RDBMS as a back-end. The main front-end is an application written in Java. There are many maintenance and reporting tools, most of which are written in Perl. We also have exporting tools for our web site, which are in C.

I guess we spend most of our time on deciding what the right design is, and implementing the correct tables and stored procedures for things, mainly SQL work and design, and higher level application and data design. And business analysis, trying to work out what the business wants, is also a very big part of my/our time.

TPJ: So what got you so hot on graphics? Your knowledge is very good!
MV: I guess it's just something that interested me. When at the university, I needed a few visualization things. My research was in mixing high-viscosity fluids, so pretty pictures were important. I also had some colleagues who did research on particles in suspension and needed pictures, so I started looking at how things could be drawn with computers, and dabbled with PostScript, various libraries, various terminals, and output devices, and slowly gathered information. In early '94, I installed an early release of the NCSA web server, and fiddled with a bundle of tools for that. One of them, about a year or so later, was a log parser; and later, some tools to make pretty pictures from the log file. That tool used gnuplot early on, which had limitations, so I wrote some Perl stuff to generate charts, and that survived as *GD::Graph*.

When Manning contacted me a few years ago to see whether I'd be interested to write about graphics in Perl, I didn't actually think there'd be enough to write about, or that I knew enough about it. I promised to look into it a bit and actually found out that there was more than enough material, and that what I didn't know, I could learn about. So I started writing, and I probably learned as much about graphics while writing as I already knew! The largest job was determining what to put in, and in which order. The actual assimilation of knowledge wasn't that hard :)

TPJ: Why did Manning contact you?
MV: They asked some of the authors they already knew to see whether anyone could come up with a name of someone who might be able to write a book about Perl and graphics. Because of *GD::Graph* and my participation on the comp.lang.perl.* newsgroups, my name popped into someone's head and was passed on to Manning.

TPJ: Your book interested me especially because, at the time I received it for review, I was working on a book about playing concertina. It has many diagrams that really are the same diagram with minor changes. So I was filled with angst as I tried to decide whether to go on making the diagrams by hand or delve into generating them algorithmically!
MV: I'd probably do them programmatically. I once helped someone write a tab/chord formatter and we coded the chord pictures as well, instead of storing them. The PostScript generated would actually draw them. This is one of the places where programming comes in handy: many pictures, all subtly, but determinedly different. But, did the book answer your question and/or help you out?

TPJ: Yes, it answered my question, enough so I could do the cost/benefits analysis and conclude that, for the given instance, it was more work to program than to do them by hand!
MV: I tend to find that when I do this sort of thing, I can often use the scaffolding of the programs in other projects. But I don't play an instrument, not really, so I suspect that for me, programming is still the most fun I can have on my own. However, I do make the point in the first chapter that certain things are still better done by sitting down at a terminal and interactively creating the pictures you need.

TPJ: I found your discussion of graphic file formats quite good. You really did spend a lot of time examining these things.
MV: Well, that particular one is a bit of a pet peeve of mine. I dislike it when people use inappropriate file formats. But of course, I don't think that I spent enough time on that section. There is still much more to say on it! There are other areas the book needs more work on, in retrospect. My biggest regret is not taking the time to add *Imager* to it. When I wrote the original material, *Imager* was very immature. However, it has turned out to be a very good and usable module. It has many capabilities that *Image::Magick* has, and many that *GD* has, and it certainly deserves a place between them. If I had more time, I'd write addenda to the book to be put on the web site.

TPJ: Are you going to write any more books?
MV: If anything, writing a book is much more work than I thought it would be. Maybe I'll become independently rich soon. If that happens, I'll be writing again.

TPJ

Source Code Appendix

Randy Kobes, Andrea Letkeman, and Olesia Shewchuk
"Fractal Images and Music With Perl"

Listing 1

```
#!/usr/bin/perl
# mandelbrot.pl - draw the Mandelbrot set
use strict;
use warnings;
use Fractal qw(draw min_max);

# xmin, xmax, ymin, and ymax specify the range of coordinates
# steps is the number of points used between (xmin .. xmax),
#   [which is the same number used between (ymin .. ymax)]
# maxit is the maximum number of iterations
my ($xmin, $xmax, $ymin, $ymax, $steps, $maxit) =
  (-1.6, 0.6, -1.2, 1.2, 300, 200);

my $points = mandelbrot();

draw(pts => $points, file => 'mandelbrot.png', size => $steps);

# Mandelbrot set, as defined by
#    x[n+1] = x[n]*x[n] - y[n]*y[n] + c_x
#    y[n+1] = 2*x[n]*y[n] + c_y
# where (c_x, c_y) are the pixel coordinates
# and the iterations begin at (x, y) = (0, 0)

sub mandelbrot {
  my $pts = [];
# determine step size between (xmin .. xmax) and (ymin .. ymax)
  my $xstep = ($xmax - $xmin) / $steps;
  my $ystep = ($ymax - $ymin) / $steps;

  for (my $cx=$xmin; $cx<=$xmax; $cx+=$xstep) {
    for (my $cy=$ymin; $cy<=$ymax; $cy+=$ystep) {
      my ($oldx, $oldy, $newx, $newy) = (0,0,0,0); # start at x = y = 0
      my $i=1;  # iteration number before breakout condition is met
      for ($i=1; $i<$maxit; $i++) {
           $newx = $oldx*$oldx - $oldy*$oldy + $cx;
           $newy = 2*$oldx*$oldy + $cy;
           last if sqrt($newx*$newx + $newy*$newy) > 2; # breakout condition
           ($oldx, $oldy) = ($newx, $newy);
      }
      push @$pts, [$cx, $cy, $i];
    }
  }

  return $pts;
}
```

Listing 2

```
#!/usr/bin/perl
# julia.pl - draw the Julia set
use strict;
use warnings;
use Fractal qw(draw min_max);

# xmin, xmax, ymin, and ymax specify the range of coordinates
# steps is the number of points used between (xmin .. xmax)
# the same number is used between (ymin .. ymax)
# maxit is the maximum number of iterations
my ($xmin, $xmax, $ymin, $ymax, $steps, $maxit) =
  (-1.3, 0.9, -1.2, 1.2, 300, 200);

my ($cx, $cy) = (-0.11, -0.9); # used to define the Julia set

my $points = julia();
my ($min, $max) = min_max($points); # used in my_rgbindex and my_rgb
draw(pts => $points, file => 'julia.png', size => $steps,
     rgbindex => \&my_rgbindex, rgb => \&my_rgb);

# Julia set, as defined by
#    x[n+1] = x[n]*x[n] - y[n]*y[n] + c_x
#    y[n+1] = 2*x[n]*y[n] + c_y
# where (c_x, c_y) are fixed constants
# and the iterations begin at the pixel coodinates

sub julia {
  my $pts = [];
# determine step size between (xmin .. xmax) and (ymin .. ymax)
  my $xstep = ($xmax - $xmin) / $steps;
  my $ystep = ($ymax - $ymin) / $steps;

  for (my $x=$xmin; $x<=$xmax; $x+=$xstep) {
    for (my $y=$ymin; $y<=$ymax; $y+=$ystep) {
      my ($oldx, $oldy, $newx, $newy) = ($x, $y, 0, 0); # initialization
      my $i=1; # iteration number before breakout condition is met
      for ($i=1; $i<$maxit; $i++) {
           $newx = $oldx*$oldx - $oldy*$oldy + $cx;
           $newy = 2*$oldx*$oldy + $cy;
           last if sqrt($newx*$newx + $newy*$newy) > 2; # breakout
           ($oldx, $oldy) = ($newx, $newy);
      }
      push @$pts, [$x, $y, $i];
    }
  }

  return $pts;
}

sub my_rgbindex {
  my $p = shift;
  return int($p->[2]/$max->[2]*100);
}

sub my_rgb {
  my $index = shift;
  return map{hex} unpack "a2a2a2",
    sprintf("%02X", int($index*255**3));
}
```

Listing 3

```
#!/usr/bin/perl
# snowflake.pl - draw a snowflake
use strict;
use warnings;
use Fractal qw(draw min_max);

# specify the maximum number of iterations used,
# and the size of the desired image
my ($maxit, $size) = (500000, 300);

# specify the transformations
#     x[n+1] = a[i]*x[n] + b[i]*y[n] + e[i]
#     y[n+1] = c[i]*x[n] + d[i]*y[n] + f[i]
# where "i" labels the particular transformation

my @a = (.75, .5, .25, .25, .25, .25);
my @b = (0, -0.5, 0, 0, 0, 0);
my @e = (.125, .5, 0, .75, 0, .75);
my @c = (0, .5, 0, 0, 0, 0);
my @d = (.75, .5, .25, .25, .25, .25);
my @f = (.125, 0, .75, .75, 0, 0);

# where to begin the iterations at
my ($xstart, $ystart) = (0.6, 0.5);

my $points = ifs_random();
my ($min, $max) = min_max($points);
draw(pts => $points, file => 'snowflake.png', size => $size,
     rgbindex => \&my_rgbindex, rgb => \&my_rgb);

# routine to find the points of an ifs by
# choosing the transformation used randomly
sub ifs_random {
  my $pts = [];
  push @$pts, [$xstart, $ystart, 0];
  my ($oldx, $oldy, $newx, $newy) = ($xstart, $ystart, 0, 0);
  for (my $count = 1; $count < $maxit; $count++) {
# choose a transformation randomly
    my $index = int(rand @a);
    $newx = $a[$index]*$oldx + $b[$index]*$oldy + $e[$index];
    $newy = $c[$index]*$oldx + $d[$index]*$oldy + $f[$index];
    push @$pts, [$newx, $newy, $index];
    ($oldx, $oldy) = ($newx, $newy);
  }
  return $pts;
}

sub my_rgbindex {
  my $p = shift;
  return int($p->[2]/$max->[2]*100);
}

sub my_rgb {
  my $index = shift;
  return (128, 0, 128);
}
```

Listing 4

```
#!/usr/bin/perl
# fern.pl - draw a fern
use strict;
use warnings;
use Fractal qw(draw min_max);

# specify the maximum number of iterations used,
# and the size of the desired image
my ($maxit, $size) = (100000, 300);

# specify the transformations
#     x[n+1] = a[i]*x[n] + b[i]*y[n] + e[i]
#     y[n+1] = c[i]*x[n] + d[i]*y[n] + f[i]
# where "i" labels the particular transformation

my @a = (0.85, 0.20, -0.15, 0);
my @b = (0.04, -0.26, 0.28, 0);
my @e = (0.075, 0.4, 0.575, 0.5);
my @c = (-0.04, 0.23, 0.26, 0);
my @d = (0.85, 0.22, 0.24, 0.16);
my @f = (0.18, 0.045, -0.086, 0);

# specify the probablility for which to choose each transformation
# and check that they all add up to 1
my %prob = (0 => 0.77, 1 => 0.12, 2 => 0.1, 3 => 0.01);
my $sum = 0;
$sum += $_ for values %prob;
die 'The probablilities have to add up to 1'
  unless ( abs($sum - 1) < 0.001);

# where to begin the iterations at
my ($xstart, $ystart) = (0, 0);

my $points = ifs_specified();
my ($min, $max) = min_max($points);
draw(pts => $points, file => 'fern.png', size => $size,
     rgbindex => \&my_rgbindex, rgb => \&my_rgb);

# routine to find the points of an ifs by choosing the
# transformation as specified by a user-set probability
sub ifs_specified {
  my $pts = [];
  push @$pts, [$xstart, $ystart, 0];
  my ($oldx, $oldy, $newx, $newy) = ($xstart, $ystart, 0, 0);

# set up bounds to use in test for deciding which
# transformation to use
  my %bounds;
  my $last = 0;
  my @indices = sort {$a <=> $b} keys %prob;
  for (@indices) {
    $last += $prob{$_};
    $bounds{$_} = $last;
  }
  for (my $count=1; $count<$maxit; $count++) {
# decide which transformation to use
    my $index;
    my $r = rand;
    for (@indices) {
      if ($r < $bounds{$_}) {
           $index = $_;
           last;
      }
    }

    $newx = $a[$index]*$oldx + $b[$index]*$oldy + $e[$index];
    $newy = $c[$index]*$oldx + $d[$index]*$oldy + $f[$index];
    push @$pts, [$newx, $newy, $index];
    ($oldx, $oldy) = ($newx, $newy);
  }
  return $pts;
}

sub my_rgbindex {
  my $p = shift;
  return int($p->[2]/$max->[2]*255);
}

sub my_rgb {
  my $index = shift;
  return ($index*20, 128, (255 - $index*10));
}
```

Listing 5

```
#!/usr/bin/perl
# logistic.pl - hear the logistic map
use strict;
use warnings;
use Fractal qw(compose);
```

```
# where to start the map at, the parameter mu, and the number of notes
my ($xstart, $A, $maxit) = (0.4, 3.82, 256);

my $points = logistic();
compose(pts => $points, file => 'logistic.mid');

sub logistic {
  my $pts = [];
  my ($oldx, $newx) = ($xstart, 0);
  push @$pts, $xstart;
  for (my $i=1; $i<$maxit; $i++) {
    $newx = $A*$oldx*(1-$oldx);
    push @$pts, $newx;
    $oldx = $newx;
  }
  return $pts;
}
```

Listing 6

```
#!/usr/bin/perl
# henon.pl - hear the henon map
use strict;
use warnings;
use Fractal qw(compose min_max);
# where to start the map at
my ($xstart, $ystart) = (0.4, 0.2);
# the parameters a and b, and the number of notes
my ($a, $b, $maxit) = (1.4, 0.3, 256);

# use a chromatic scale
my @notes = (55 .. 89);

my $points = henon();
my ($min, $max) = min_max($points);
compose(pts => $points, file => 'henon.mid',
           tempo => 100000, patch => 44,
           noteindex => \&my_noteindex, note => \&my_note);

sub henon {
  my $pts = [];
  my ($oldx, $oldy, $newx, $newy) = ($xstart, $ystart, 0, 0);
  push @$pts, $xstart;
  for (my $i=1; $i<$maxit; $i++) {
    $newx = $a - $oldx*$oldx + $b*$oldy;
    $newy = $oldx;
    push @$pts, $newx;
    ($oldx, $oldy) = ($newx, $newy);
  }
  return $pts;
}

sub my_noteindex {
  my $x = shift;
  $x = ($x - $min) / ($max - $min);
  return int($x * $#notes);
}

sub my_note {
  my $index = shift;
  return $notes[$index];
}
```

Listing 7

```
#!/usr/bin/perl
# two_track.pl - hear a point of the Mandelbrot set
use strict;
use warnings;
use Fractal qw(min_max);
use MIDI::Simple;

my $maxit = 200; # maximum number of notes
my ($cx, $cy) = (-0.1, -0.9); # pick a point on the Mandelbrot set
my $file = 'mandelbrot.mid';
my ($tempo, $patch_0, $patch_1) = (90000, 33, 66);

my @xnotes = (55, 57, 59, 60, 62, 64, 66, 67);

my @ynotes = (67, 69, 71, 72, 74, 76, 78, 79);

my $points = mandelbrot();
my ($min, $max) = min_max($points);
my ($xmin, $xmax, $ymin, $ymax) =
  ($min->[0], $max->[0], $min->[1], $max->[1]);

new_score();
patch_change 0, $patch_0;
patch_change 1, $patch_1;
set_tempo($tempo);
```

```
my @subs = (\&track_0, \&track_1);
my ($x, $y);

foreach (@$points) {
  ($x, $y) = ($_->[0], $_->[1]);
  synch(@subs);
}
write_score($file);

sub mandelbrot {
  my $pts = [];
  my ($oldx, $oldy, $newx, $newy) = (0,0,0,0); # start at x = y = 0
  push @$pts, [$oldx, $oldy];
  my $i=1;  # iteration number before breakout condition is met
  for ($i=1; $i<$maxit; $i++) {
    $newx = $oldx*$oldx - $oldy*$oldy + $cx;
    $newy = 2*$oldx*$oldy + $cy;
           last if sqrt($newx*$newx + $newy*$newy) > 2; # breakout condition
    ($oldx, $oldy) = ($newx, $newy);
    push @$pts, [$newx, $newy];
  }
  return $pts;
}

sub track_0 {
  my $it = shift;
  my $note = xnote();
  $it->n('c0', 'hn', $note);
}

sub track_1 {
  my $it = shift;
  my $note = ynote();
  $it->n('c1', 'hn', $note);
}

sub xnote {
  my $xn = ($x - $xmin) / ($xmax - $xmin);
  my $index = int($xn * $#xnotes);
  return $xnotes[$index];
}

sub ynote {
  my $yn = ($ymax - $y) / ($ymax - $ymin);
  my $index = int($yn * $#ynotes);
  return $ynotes[$index];
}
```

Stas Bekman and Eric Cholet
"Writing Multilingual Sites With mod_perl and Template Toolkit"

Listing 1

```
package My::Language;

my %iso_8859_1_accents = (
    a => [ qw(à á â ã ä å À Á Â Ã Ä Å) ],
    c => [ qw(ç Ç) ],
    e => [ qw(è é ê ë È É Ê Ë) ],
    i => [ qw(ì í î ï Ì Í Î Ï) ],
    n => [ qw(ñ Ñ) ],
    o => [ qw(ò ó ô õ ö ø Ò Ó Ô Õ Ö Ø) ],
    u => [ qw(ù ú û ü Ù Ú Û Ü) ],
    y => [ qw(ý Ý) ],
);
# build translation strings
my (%in, %out);
for my $letter ('a'..'z') {
    my $uletter = CORE::uc $letter;
    # translate non-accented letters
    $in{uc}  .= $letter;
    $out{lc} .= $letter;
    $in{lc}  .= $uletter;
    $out{uc} .= $uletter;
    if (my $ra_accented = $iso_8859_1_accents{$letter}) {
        my $in = join '', @$ra_accented;
        $in{lc} .= $in;
        $in{uc} .= $in;
        $out{lc} .= $letter  x @$ra_accented;
        $out{uc} .= $uletter x @$ra_accented;
    }
}
# build translation subroutines
for my $type (qw(lc uc)) {
    my $sub = qq!
        sub iso_8859_1_$type {
           (my \$s = shift) =~ tr/$in{$type}/$out{$type}/;
           \$s
        }
    !;
    eval $sub;
}

# character sets
my %charsets = (
  'iso-8859-1'    => { lc => \&iso_8859_1_lc,
                       uc => \&iso_8859_1_uc,
                     },
);
```

Listing 2

```
package My::Template::Parser;
use strict;
use base qw(Template::Parser);

use constant LANG_RE        => qr{<([a-z]{2})>(.*?)</\1>}s;

#######################
# constructor
sub new {
  my ($class, $options) = @_;
  my $self = $class->SUPER::new();
  $self->init($options);
  return $self;
}
#######################
sub init {
  my ($self, $options) = @_;
  $self->{$_} = $options->{$_} for keys %$options;
}

#######################
sub parse {
  my ($self, $text) = @_;

  # tokenize
  $self->_tokenize($text);

  # replace C<text> language sections with Template Toolkit
  # directives which will pick up the correct language text
  # at request processing time.
  $text = '';
  for my $section (@{$self->{sections}}) {
      my $translated = $section->{text};
      $translated =~ s{@{[LANG_RE]}}
                      {\[% IF lang=='$1' %\]$2\[% END %\]}gs
          if $section->{lang};
      $text .= $translated;
  }
  # proceed with standard Template Toolkit parsing
  return $self->SUPER::parse ($text);
}
#######################
sub _tokenize {
  my ($self, $text) = @_;
  return unless defined $text && length $text;

  # extract all sections from the text
  $self->{sections} = [];
  while ($text =~ s!
         ^(.*?)               # $1 - start of line up to start tag
          (?:
               <text>                       # start of tag
               (.*?)                        # $3 - tag contents
               </text>                      # end of tag
          )
          !!sx) {
      push @{$self->{sections}}, { text => $1 } if $1;
      push @{$self->{sections}}, { lang => 1, text => $2||'' }
          if defined $2;
  }
  push @{$self->{sections}}, { text => $text } if $text;
}
#######################
1;
__END__
```

Listing 3

```
# all languages
%languages = (
  en => { name        => 'english',
          charset     => 'iso-8859-1',
          months      => [qw(January February March April May
                                 June July August September October
                                 November December)],
          days        => [qw(Sunday Monday Tuesday Wednesday
                                 Thursday Friday Saturday)],
          date        => {short  => '%m/%d/%Y',
                           medium => '%B %e, %Y',
                           long   => '%A %B %e, %Y',
                          },
          time        => {short  => '%H:%M',
                          },
        },
 it => { name        => 'italiano',
         charset     => 'iso-8859-1',
         months      => [qw(gennaio febbraio marzo aprile maggio
                             giugno luglio agosto settembre ottobre
                             novembre dicembre)],
         days        => [qw(domenica lunedì martedì mercoledì
                             giovedì venerdì sabato)],
         date        => {short  => '%d/%m/%Y',
                          medium => '%e %B %Y',
                          long   => '%A %e %B %Y',
                          },
         time        => {short  => '%H:%M',
                         },
       },

  fr => { name        => 'français',
          charset     => 'iso-8859-1',
          months      => [qw(Janvier Février Mars Avril Mai
                              Juin Juillet Août Septembre Octobre
                              Novembre Décembre)],
          days        => [qw(Dimanche Lundi Mardi Mercredi
                              Jeudi Vendredi Samedi)],
          date        => {short  => '%d/%m/%Y',
                           medium => '%e %B %Y',
                           long   => '%A %e %B %Y',
                          },
          time        => {short  => '%Hh%M',
                          },
        },
 es => { name        => 'español',
         charset     => 'iso-8859-1',
         months      => [qw(Enero Febrero Marte Abril Maio Junio
                             Julio Agosto Septiembre Octubre Noviembre Diciembre)],
         days        => [qw(Domingo Lunes Martes Miércoles Jueves Viernes Sábado)],
         date        => {short  => '%d/%m/%Y',
                          medium => '%e %B %Y',
                          long   => '%A %e %B %Y',
                          },
         time        => {short  => '%Hh%M',
                         },
       },
);
```

Listing 4

```
####################################################################
# strftime compatible specifiers
# %A    full weekday name
# %B    full month name
# %d    day of the month (01-31)
# %e    day of the month (1-31)
# %H    hour (00-23)
# %I    hour, 12-hour clock (01-12)
# %k    hour (0-23)
# %l    hour, 12-hour clock (1-12)
# %M    minute (00-59)
# %m    month (01-12)
# %p    AM/PM
# %S    second (00-60)
# %Y    year with century
# %y    year without century (00-99)
# %%    %
use vars qw(%strftime);
%strftime =
  (
   A => q/@{$languages{$lang}{days}||[]}[week_day($date)]/,
   B => q/@{$languages{$lang}{months}||[]}[$date->[MONTH]-1]/,
   d => q/sprintf('%02d',$date->[DAY])/,
   e => q/sprintf('%d',$date->[DAY])/,
   H => q/sprintf('%02d',$time->[HOUR])/,
   I => q/sprintf('%02d',$time->[HOUR]?$time->[HOUR]>12?$time->[HOUR]-12:$time->[HOUR]:12)/,
   k => q/sprintf('%d',$time->[HOUR])/,
   l => q/$time->[HOUR]?$time->[HOUR]>12?$time->[HOUR]-12:$time->[HOUR]:12/,
   M => q/sprintf('%02d',$time->[MINUTE])/,
   m => q/sprintf('%02d',$date->[MONTH])/,
   p => q/$time->[HOUR] && $time->[HOUR]<13 ? 'AM' : 'PM'/,
   S => q/sprintf('%02d',$s)/,
   Y => q/sprintf('%04d',$date->[YEAR])/,
   y => q/$date->[YEAR]%100/,
  '%' => q/\%/,
  );
```

These are two functions that show that accept either current date or time, the language and the requested format:

```
# $time_str = format_time($lang,$format_type,$time)
# $time == [$HH,$SS]
################
sub format_time{
  my ($lang,$format_type,$time) = @_;
  # either can be zero
  return '' unless defined $time->[HOUR] and defined $time->[MINUTE];

  my $format = $languages{$lang}{time}{$format_type};
  warn("unknown time format: $format"), return '' unless $format;

  $format =~ s/\%(.)/$strftime{$1}/gee;
  return $format;
}

# $date_str = format_date($lang,$format_type,$date)
# where $date = [$YY,$MM,$DD]
################
sub format_date{
  my ($lang,$format_type,$date) = @_;

  return '' unless $date->[YEAR] and $date->[MONTH] and $date->[DAY];
  my $format = $languages{$lang}{date}{$format_type};
  warn("unknown date format: $format"), return '' unless $format;

  $format =~ s/\%(.)/$strftime{$1}/gee;
  return $format;
}
```

Moshe Bar "Perl in High Performance Computing Environments"

Listing 1

```
#!/usr/bin/perl

use lib qw(/usr/local/perlmod/Parallel-MPI-0.03/contrib/cpi/../../blib/arch
/usr/local/perlmod/Parallel-MPI-0.03/contrib/cpi/../../blib/lib);

$|=1;
use Parallel::MPI qw(:all);

sub f {
    my ($a) = @_;
    return (4.0 / (1.0 + $a * $a));
}

my $PI25DT = 3.141592653589793238462643;

MPI_Init();
$numprocs = MPI_Comm_size(MPI_COMM_WORLD);
$myid =     MPI_Comm_rank(MPI_COMM_WORLD);

#printf(STDERR "Process %d\n", $myid);

$n = 0;
while (1) {
    if ($myid == 0) {
        if ($n==0) { $n=100; } else { $n=0; }
        $startwtime = MPI_Wtime();
    }

    MPI_Bcast(\$n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    last if ($n == 0);

    $h   = 1.0 / $n;
    $sum = 0.0;

    for ($i = $myid + 1; $i <= $n; $i += $numprocs) {
        $x = $h * ($i - 0.5);
        $sum += f($x);
    }
    $mypi = $h * $sum;

    MPI_Reduce(\$mypi, \$pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if ($myid == 0) {
        printf("pi is approximately %.16f, Error is %.16f\n",
               $pi, abs($pi - $PI25DT));
        $endwtime = MPI_Wtime();
        printf("wall clock time = %f\n", $endwtime - $startwtime);
    }
}

MPI_Finalize();
```

brian d foy “Creating Perl Application Distributions”

Listing 1

```
prompt$ make test
cp buster blib/script/buster
/usr/bin/perl "-MExtUtils::MY" -e "MY->fixin(shift)" blib/script/buster
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t
t/compile....NOK 1#     Failed test (t/compile.t at line 5)
# 'String found where operator expected at blib/script/buster line 6, near # "yprint "Hello World!\n""
#       (Do you need to predeclare yprint?)
# syntax error at blib/script/buster line 6, near "yprint "Hello World!\n""
# blib/script/buster had compilation errors.
# '
#     doesn't match '(?-xism:syntax OK$)'
# Looks like you failed 1 tests of 1.
t/compile....dubious
         Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
         Failed 1/1 tests, 0.00% okay
Failed Test Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------
t/compile.t    1   256     1    1 100.00%  1
Failed 1/1 test scripts, 0.00% okay. 1/1 subtests failed, 0.00% okay.
make: *** [test_dynamic] Error 2
```