June 2003

The Perl Journal

LETTER FROM THE EDITOR

Publish or Perish

The Web has, unquestionably, had a democratizing effect on the software world. Open-source collaboration could never have reached its current level without it. It's hard to imagine the rise of Linux without the Web, not to mention tools like The Gimp, or window managers like KDE or Gnome. But arguably the biggest revolution brought about by the Web has been focused in publishing and broadcasting. The democratization of information delivery represents a shift in the balance of power. No, CNN and *The New York Times* aren't going away any time soon (no matter how hard they try), but they can't ignore the Web. Big generalist news organizations may be the purview of other media, but when it comes to niche information specialists, well, long live the Web.

Perl didn't create this online publishing revolution. Any such revolution must be credited to the widespread adoption of the Web itself, not the rise of Perl. But Perl has played a significant part. Perl's preeminence in CGI scripting has earned it a lasting place in the toolkit of dynamic content delivery. And without that dynamism, web publishing would quickly become unrealistic. It doesn't take very many updates of your static-HTML web site to realize that you need some programming logic behind it to ease your burden, and more often than not, it's Perl that comes to the rescue. Sure, other technologies—PHP, JavaScript, Python, and JSPs, to name a few—do the job, too, but Perl is still the heavy lifter of the Web.

It's not surprising, then, that Perl plays its part in Weblog-publishing software like Blosxom (http://www.raelity.org/apps/blosxom/), Greymatter (http://www.noahgrey.com/greysoft/), or Movable Type (http://www.movabletype.org/). There's been a tremendous amount of hype about blogs and many predictions that they would quickly become irrelevant. But darn it, they seem to have staying power. That's because blogs are just the latest incarnation of the original idea behind the Web: Lots of individuals all providing their own individual brand of information, and all linking to each other in a thoroughly democratic fashion. Blog software just automates this publishing process to a new degree and in a way that's easily adopted by the masses. It's content management for the rest of us.

But Perl has done more than just automate the Web. It helped to translate the information that already existed. Much of what is now published on the Web didn't start out as HTML, XML, or even plain text. It began life in some proprietary format. Some of these formats are incredibly opaque and resistant to conversion (Quark XPress comes to mind). There are huge repositories that have been converted to these formats by, you guessed it, Perl scripts. And there are uncounted Terabytes yet to convert. There's probably no better language for this than Perl.

Perl has been a force in bringing about this sea change in publishing. Now we're trying to cope with the result—more information than we can handle, and much of it from sources we're not sure we trust. The next revolution will need to be in sorting this deluge into usable signal and useless noise. This will need to be done on an individual basis—one person's signal is another's noise. Extraction and Reporting. Gosh, sounds like a job for Perl.

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, Randal Schwartz

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

INTERNET OPERATIONS
DIRECTOR
Michael Calderon
SENIOR WEB DEVELOPER
Steve Goyette
WEB DEVELOPER
Bryan McCormick
WEBMASTERS
Sean Coady, Joe Lucca, 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 CFO
John Day
EXECUTIVE VICE PRESIDENT AND COO
Steve Weitzner
EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND MARKETING
Jeff Patterson
CHIEF INFORMATION OFFICER
Mike Mikos
PRESIDENT, TECHNOLOGY SOLUTIONS GROUP
Robert Faletra
PRESIDENT, HEALTHCARE GROUP
Vicki Masseria
SENIOR VICE PRESIDENT, HUMAN RESOURCES AND COMMUNICATIONS
Leah Landro
VICE PRESIDENT, SOFTWARE DEVELOPMENT MEDIA GROUP
Peter Westerman
VICE PRESIDENT AND GENERAL COUNSEL
Sandra Grayson

Shannon Cochran

Perl News

MisterHouse's Neighborhood

The home-automation program MisterHouse, which began development in 1998, has been generating buzz with an updated release in April and a Slashdot mention in May. MisterHouse is written in Perl and relies on X10 equipment to regulate things such as household temperature, lighting, or operation of the VCR. The program can also record Caller ID and pager data to a web interface, deliver updates on web stats or daily computer usage, track the location of vehicles by communicating with a ham radio modem, monitor the weather outside, and turn lawn sprinklers on or off. MisterHouse is capable of speech output: for example, "Notice, the sun is bright at 32 percent, and it is cold outside at 24 degrees, so I am opening the curtains at 8:07AM," or "The front door has been left open."

Bruce Winter, the primary author of MisterHouse, makes his house's web interface available at http://www.misterhouse.net/. There, you can read the latest messages from MisterHouse, see who's been calling or e-mailing the Winters, find out where their cars are, or check out the traffic on their web page. There's also a weblog from Ron Klinkien, a MisterHouse user, at http://mrhousefromscratch.tk/.

A Day at the Links

A new season has begun in IRCNet's #perlgolf's minigolf system, where the "holes" are programming problems and the "golfers" compete to write the smallest script that will solve the problem. About three to five problems are posed every month; each contest lasts for a few days. Teams are allowed to compete as well as individuals. A "leaderboard" reports the current high scores throughout the contest period.

According to the Perl Golf rules, "The program may only use the Perl executable; no other executables on the system are allowed. (In particular, you must not use the programs implementing any other holes. The program may use itself though.) You may use any of the Perl 5.8.0 Standard core modules. Your solution must be portable in the sense that it should work on all official versions of Perl 5.8.0 everywhere. It's perfectly fine to abuse Perl 5.8.0 bugs. For Perl golf, the executable (not the documentation) defines the language."

The Perl Golf homepage is at http://terje.perlgolf.org/.

ActiveState Active Awards 2003

From June 3 to June 30, ActiveState will be accepting votes for this year's Programmers' Choice Active Awards. The awards are given in five categories—Perl, PHP, Python, XSLT, and Tcl—and are meant to honor those who have contributed significantly to open-source language programming. (ActiveState suggests authors of useful modules, maintainers of advocacy sites, dedicated bug fixers, "or simply someone who writes really cool code.") Winners of the Programmers' Choice awards are determined by an open vote; ActiveState also sponsors the Activators' Choice prizes, the winners of which are chosen by ActiveState's development team.

Last year, Matt Sergeant won the Programmers' Choice awards in both the Perl and XSLT categories, and Andy Dougherty won the Activators' Choice award for Perl development. This year's nominees for the Programmers' Choice awards were announced June 1 at http://www.activestate.com/Corporate/ActiveAwards/. Winners in all categories except Tcl will be announced at OSCON on July 8; the Tcl awards will be bestowed at the Tcl conference at the end of July.

A Dip in the POOL

Simon Cozens recently posted an article at perl.com describing POOL, "a handy 'little language' I recently created for templating object-oriented modules." POOL itself originally stood for the Perl Object Oriented Language, but after realizing that there's nothing Perl-specific about it, Cozens chose to invoke the recursive naming tradition, and dubbed it the "POOL Object Oriented Language." While Cozens himself describes it as "very, very ad hoc…bizarre and inconsistent," he hopes it will help programmers avoid tedium while writing OO classes in Perl.

The full article is at http://www.perl.com/pub/a/2003/04/22/pool.html.

VoiceTronix Phones It In

VoiceTronix, an Australian telephony company, has released "a full function, web-enabled small office PBX package" written in Perl, called OpenPBX. (PBX stands for private branch exchange, and refers to a telephone network set up within an organization: If you dial four digits to get a coworker, you're using a PBX.) OpenPBX joins Bayonne and Asterisk in the open-source PBX software arena.

VoiceTronix would like to hear from developers interested in working on new open-source telephony projects, such as porting the CT Library package from Perl to Python, porting the VoiceTronix drivers to new operating systems, or creating an SIP-based VOIP gateway. The company's web site is at http://www.voicetronix.com/.

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

Ala Qumsieh

Fuzzy Logic in Perl

Everyone has heard about how Fuzzy Logic (FL) is being used in many real-life applications such as traffic signal controls, automobile transmission control, cancer diagnosis, dam gate control for hydroelectric power plants, and elevator control. This article will show you how this is done, and will describe a new Perl module, *AI::FuzzyInference*, that allows you to write Perl programs that use Fuzzy Logic to make rational decisions.

In order to understand how the module works, it is helpful to know some of the theory behind Fuzzy Logic. What follows is a brief introduction to Fuzzy Logic and Fuzzy Inference Systems. A more detailed introduction can be found in an excellent tutorial by Jerry Mendel (http://sipi.usc.edu/~mendel/publications/FLS_Engr_Tutorial_Errata.pdf). Please note that multiple terminologies exist to describe the same thing. What I will use might differ from that in the aforementioned tutorial.

Fuzzy Logic

Fuzzy Logic is an extension of regular two-valued logic that was developed by Lotfi Zadeh (L.A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, pp. 338–353, 1965). In regular set theory, an element *x* either belongs to a set *S*, or it does not belong. For example, the element *apple* belongs to the set *Fruits*, but does not belong to the set *Vegetables*. We say that the element *apple* is a "member" of the set *Fruits*.

Problems start to arise when membership is not as easily determined. Consider, for example, the set of *Round Objects*. Is our apple a member of this set? Well, it depends on how round the apple is. In everyday life, we say "The apple is almost round." In order to take account of adverbs such as almost, sort of, and approximately, Zadeh extended regular set theory to allow for partial degrees of membership (DOM). This way, we can postulate that the degree of membership of our apple in the set of *Round Objects* is 0.8, while a pear's degree of membership is only 0.4, and that of a banana is 0.

Many people confuse fuzzy logic and probability. Although they look similar, they are completely different. As an example, consider two bottles filled with unknown liquids. The contents of bottle *A* have a probability of 0.9 of being a member of the set *Toxic Liquids*. The contents of bottle *B* have a fuzzy degree of membership of 0.9 in that same set. If I had to drink from one, I would choose bottle *A* because one time out of ten, the contents will not be toxic. Bottle *B* will always kill me because its contents are highly toxic (nine parts poison and one part water, say). Probability gives you the likelihood of the liquid being toxic. Fuzzy logic tells you how toxic it is.

Term Sets

It is very useful to be able to graphically depict what all this means. To do that, let's take another example. Let's try to categorize people based on their height. We can do so by defining three sets: *Short*, *Average*, and *Tall*. In FL, these three sets are called *Term Sets* of the variable *Height*, which is referred to as a "Linguistic Variable." Linguistic variables are variables that do not have defined numerical values, but are described by words or sentences. A 6' man is of average height, and a 6' woman is rather tall.

Let's define *Average* height as 6'. So, a 6' person has $DOM_{average}$ of 1. We can expect that for people progressively shorter than that, their DOM will be progressively smaller. Similarly for taller people. If we plot a graph of height versus DOM, it might look like Figure 1. This triangular shape of term sets is typical of Fuzzy Logic, and is mainly chosen for convenience. Other representations are possible, such as a bell-curve or Gaussian, but then computation becomes much more difficult. Corresponding term sets for *Short* and *Tall* can be similarly determined. Furthermore, term sets can overlap. A 6'3" person has DOM of 0.3 in the *Average* set, and a DOM of 0.6 in the *Tall* set.

An important thing to note here is that degrees of membership are highly dependent on the problem being investigated. For an

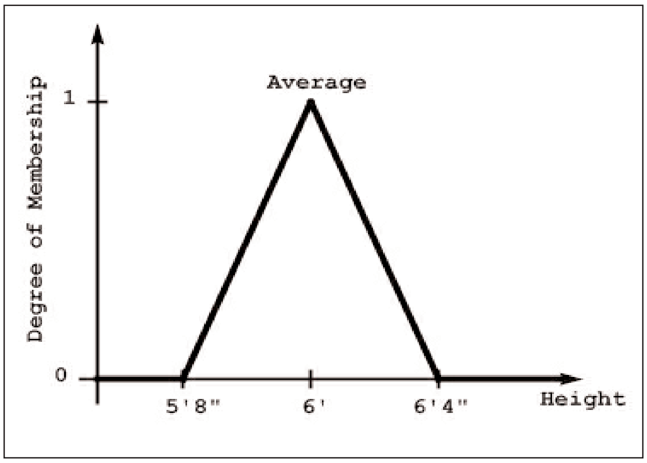


Figure 1: Graph of height versus DOM.

Ala works at NVidia Corp. as a physical ASIC designer. He can be reached at aqumsieh@cpan.org.

average person, a 6’5” person is tall, but for the set of *Basketball Players*, such a person is of average height. Thus, in different contexts, the same term sets might have different shapes (thinner or fatter), and can shift and overlap by different degrees.

Set Theoretic Operations

In regular set theory, sets can be combined and manipulated using the logical operations of *union*, *intersection*, and *complement*. The same operations are extended to handle fuzzy sets in the following manner:

Union: The union C of two sets A and B is the maximum of the two sets. This means that for every element x, $DOM_C(x)=max(DOM_A(x), DOM_B(x))$.

Intersection: The intersection C of two sets A and B is the minimum of the two sets. This means that for every element x, $DOM_C(x)=min(DOM_A(x), DOM_B(x))$.

Complement: The complement C of a set A is defined as $1–A$. So, for every element x, $DOM_C(x)=1–DOM_A(x)$.

Note that in the case of two-valued logic, those definitions collapse to their binary counterparts: OR, AND, and NOT, respectively. Another important thing to note here is that those are not the only definitions for the fuzzy set theoretic operations. We are free to choose any definition as long as it reduces to the binary form in the case of two-valued logic. As a matter of fact, Zadeh’s original definitions were:

Union: $DOM_A(x)+DOM_B(x)–DOM_A(x)DOM_B(x)$
Intersection: $DOM_A(x)DOM_B(x)$
Complement: $1–DOM_A(x)$

Other interpretations also exist.

Fuzzy Inference

Now we have just enough background to describe a Fuzzy Inference System (FIS). An FIS is a system that has a number of input variables and a number of output variables. Those variables are described by a number of term sets, and are related to each other via fuzzy rules. For example, a system with input variables *QualityOfFood* and *QualityOfService*, and output variable *AmountOfTip* might have the following rule:

```
If QualityOfFood is good AND QualityOfService is bad
THEN AmountOfTip is small
```

The IF part is called the “precedent” while the THEN part is called the “consequent.” Other rules can be defined for the remaining scenarios. For each variable (both input and output), term sets must be defined that span the whole universe of discourse of the variable, which is the range of possible values that the variable can take.

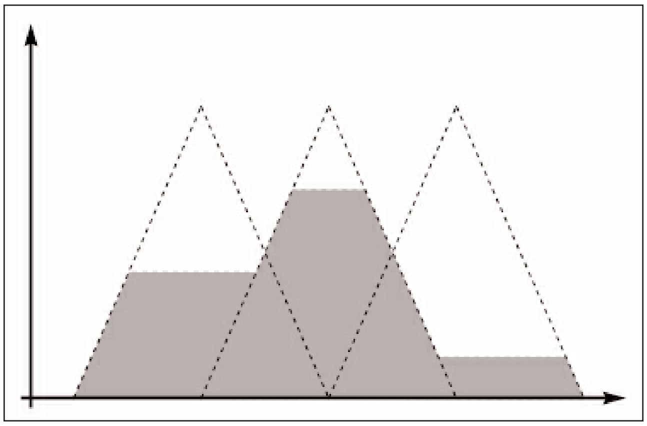


Figure 2: Union of three clipped term sets.

Now, given numerical values for its input variables, an FIS will use the fuzzy rules to compute a crisp numerical value for each of the output variables. This operation can be broken into four distinct steps.

Fuzzification

In this step, the values of the input variables are used to compute their degrees of membership into each term set. This is simply done by drawing a vertical line on the graph of each term set at the input variable’s value, and noting where it intersects the graph. For example, if *QualifyOfFood==0.6*, then we calculate:

Fuzzy Logic is an extension of regular two-valued logic that was developed by Lotfi Zadeh

```
DOM_qof_bad(0.6) = 0
DOM_qof_good(0.6) = 0.9
DOM_qof_excellent(0.6) = 0.2
```

and if *QualityOfService == 0.3*, then we calculate:

```
DOM_qos_bad(0.3) = 0.7
DOM_qos_good(0.3) = 0.2
DOM_qos_excellent(0.3) = 0
```

Inference

Now, we examine all the defined rules, and for each rule, we compute a degree of support, which indicates the firing strength of that rule. This is done by looking at the precedent of the rule, and using fuzzy logic operations to combine the values of all its constituent parts to produce a single number. So, for the rule:

```
If QualityOfFood is good AND QualityOfService is bad
THEN AmountOfTip is small
```

Degree of support will be *0.9* AND *0.7=min(0.9, 0.7)=0.7*. This value is then used to implicate the term set *small* of the variable *AmountOfTip*. Implication modifies the shape of the term set. There are various methods, but the two most popular ones are scaling and clipping. In scaling, the whole term set is multiplied by the degree of support (0.7, in this case) to yield another term set. In clipping, the term set is clipped at the degree of support value to yield a trapezoidal term set.

Aggregation

In this step, all the implicated fuzzy term sets of the output variable are combined using fuzzy logic operations to yield one fuzzy set. For our *SizeOfTip* variable, this means that its three implicated term sets (corresponding to the original *Big*, *Average*, and *Small* term sets) will be combined to create one big set. A simple way to do that would be to take the maximum of the three sets at each point (see Figure 2).

Defuzzification

Finally, the aggregated term set of the last step is used to compute a single crisp value for our output variable. The most widely used method is to take the centroid of the term set as its crisp value. Other methods are possible including taking the maximum value, or the average of the peaks.

We can see here that if a rule is not satisfied very well, then its degree of support will be close to 0. This, then, will implicate its term set by a large amount, reducing the contribution of this rule to the aggregated fuzzy set. This will, in turn, reduce its effect on the computed values of the output variables.

Systems with multiple output variables can be treated as multiple systems, each with a single output variable.

Getting and Installing the Module

You can grab a copy of *AI::FuzzyInference* from your local CPAN mirror at http://search.cpan.org/author/aqumsieh. The latest version, as of this writing, is 0.03. You can install it using the traditional method:

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

If you're on Windows, you can simply type "ppm install AI::FuzzyInference" at a command prompt. Alternatively, since it's all in pure Perl, you can unpack it in any place where *perl* will find it.

An Example: Balancing Act

Armed with our knowledge of FL and FIS's, we can now proceed to use *AI::FuzzyInference* in an example. Let's assume we have a solid horizontal 10-meter-long rod hinged exactly in its middle, with a ball placed on top of it. Furthermore, let's simplify things by assuming our world is two-dimensional, so we won't worry about the *z*-axis. We would like to be able to control our rod so as to balance the ball on it, and prevent it from falling off.

First, we have to create our *AI::FuzzyInference* object. That's easy since the constructor takes no arguments:

```
use AI::FuzzyInference
my $fis = new AI::FuzzyInference;
```

Now, we have to define our input and output variables. We need variables to capture the states of the ball and the rod. For the ball, we define two variables: *$velBall* and *$posBall*. *$velBall* specifies the current velocity of the ball, where negative values indicate motion to the left. *$posBall* indicates the current position of the ball on the rod with 0 being the center of the rod and positive values to the right. Those will be our input variables. For the rod, we define a variable *$thRod* that is the angle the rod currently makes with the *x*-axis, where positive values indicate clockwise displacement. This is our output variable. This means that the angle that the rod makes with the horizontal is a function of the velocity and position of the ball.

To complete our variable definitions, we have to define the term sets associated with them. Those will be as defined in Figure 3. The way you pass this information to the *AI::FuzzyInference* object is as follows:

```
$fis->inVar(posBall    => -5, 5,
         far_left   => [-4, 1, -2, 0],
         left       => [-4, 0, -2, 1, 0, 0],
         center     => [-2, 0, 0, 1, 2, 0],
         right      => [0, 0, 2, 1, 4, 0],
         far_right => [2, 0, 4, 1],
        );
```

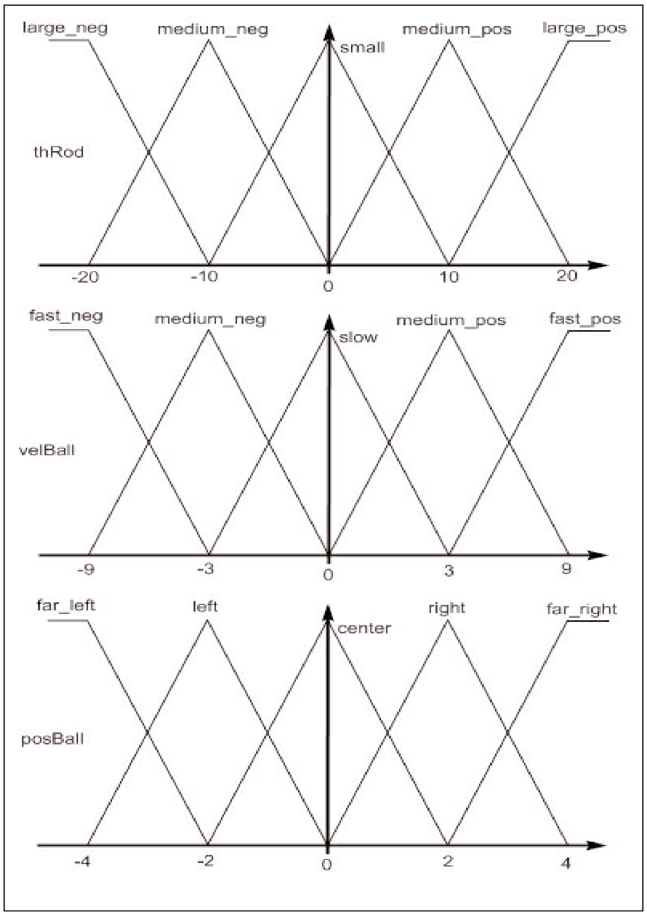


Figure 3: Term sets associated with the variables* $thRod*,* $velBall *and* $posBall*.

The first argument of the *inVar()* method is the name of the input variable. The next two arguments define the limits of the values the variable can take (aka, the universe of discourse). The rest of the arguments are key-value pairs that define each term set along with its coordinates. Those coordinates are simply lists of *x*, *y* values. Successive points are connected together by a straight line. If the given coordinates of a term set do not span the whole universe of discourse, then *AI::FuzzyInference* will automatically extrapolate, by extending the first and last points horizontally, to complete the definition.

Similarly, we define our other input variable, $*velBall*. Our output variable, $*thRod*, is defined using the *outVar()* method, which has the exact same arguments as *inVar()*.

The next step is to define the rules. Here, we use our common sense to decide what to do in each scenario in order to keep the ball on the rod. One rule can be:

```
If   $posBall is far_left     AND   # ball is close to left edge of rod
     $velBall is slow         AND   # ball almost stationary
THEN $thRod   is medium_pos         # lower the right edge by small
                                    # amount
```

In general, if all input variables have the same number of term sets, the number of possible rules is TS^{VARS} where *TS* is the number of term sets for each variable, and *VARS* is the number of input variables. We use the method *addRule()* to define our rules. So, for the rule declared above, we write:

```
$fis->addRule(
              'position = far_left   &
```

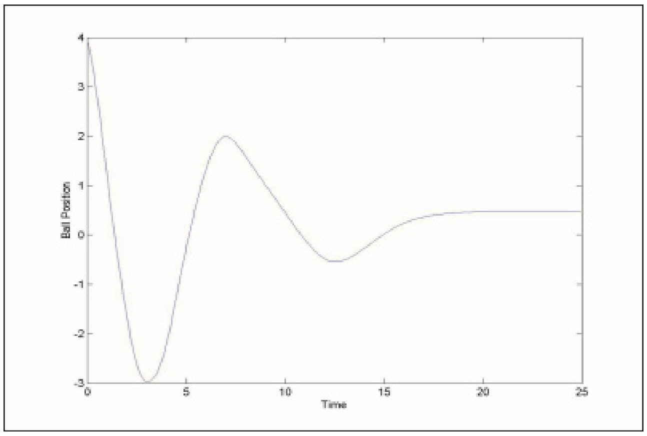


Figure 4: Ball position versus time.

```
                    velocity = slow' => 'thRod = medium_pos',
                    );
```

The arguments of *addRule()* are key-value pairs. We can define multiple rules in the same call to *addRule()*, or in multiple calls. The key of each pair is the precedent string, and the value is the consequent string. Spaces are completely optional and will be ignored. Each precedent and consequent string is composed of *variable = term_set*. Multiple precedents are combined using *&* and | for *AND* and *OR*, respectively. The equality can be reversed by using the *!* symbol, such as *position = !far_left*.

We round off our program with a subroutine, *calcNewData()*, which calculates the position and velocity of the ball after one time step has elapsed. And that is it—we are now ready to relinquish control of the rod to our FIS. In each time step, we calculate the current position and velocity of the ball, and we pass them on to our FIS, which will spit out a new value for the angle of the rod. We do this by using the *compute()* method, which takes as input key value-pairs where the keys are the input variable names, and the values are the corresponding numerical values:

```
  $fis->compute(posBall => 0.9,  #current ball position
                velBall => -1.5, #current ball speed
               );
```

This is the procedure that does all the work going through the *Fuzzification* -> *Inference* -> *Aggregation* -> *Defuzzification* loop. After that, the *value()* method is used to get the newly computed value for the output variable *thRod*:

```
  $thRod = $fis->value('thRod');
```

Then we loop back to the next time step. To better visualize this, the Perl program uses Tk to draw the rod with the ball balanced on top of it at each time step.

Running the program from random initial conditions, we can observe the behavior shown in Figure 4. The *x*-axis is time, and the *y*-axis is the position of the ball on the rod. As we would expect, the ball oscillates from one side to the other as our system struggles to control it. As the system gains more control of the ball, the oscillations die down.

Concluding Remarks

Rules. You do not have to define rules for ALL possible combinations of input variables. Any rule you do not define will not produce any change in the system when its associated scenario is encountered. Having said that, it is useful to specify as many rules as possible, since this will constrain your system more. In our case, we specify all the rules (see Listing 1; also available for download at http://www.tpj.com/source/), which enhances our chances of controlling the ball. Removing some rules might cause more oscillation before the ball stops, or even prevent the system from controlling the ball altogether.

Moreover, choosing the correct response for every combination of input variables might not be a very easy thing to do in itself. One problem is that those rules are not set in stone. So, what constitutes common sense to one person might be considered ridiculous to another. Another problem is the sheer number of rules needed as the number of variables increases. This easily becomes the case if we have four or five input variables or more. Therefore, most people design their FISs with the smallest number of input variables possible to keep things under control. Of course, this hasn't deterred other researchers from using various techniques and heuristics to tackle such problems. Some interesting approaches use evolutionary and biologically inspired techniques such as neural networks and genetic algorithms to make the system "evolve" and "learn" its own rules. But this is a topic for another article.

Variables. In this example, I chose a relatively simple system where the output is a function of only two variables. Some systems are even simpler and contain just a single input variable. Indeed, in our case, we could find rules that define the angle of the rod in terms of the ball's position only. Conversely, some systems are much more complicated. Initially, I designed the system to have three input variables: *ball position*, *ball speed*, and *rod angle*. The output variable was the change in the angle of the rod. I opted for the simpler version (yet not too simple) to avoid unnecessary complications. I would be interested in hearing from anyone who uses *AI::FuzzyInference* to design a system with more than two variables.

The *AI::FuzzyInference* Module

Finally, keep in mind that the module does not have too many checks integrated yet. So, if in your rules you use term sets or variables that are not defined (due to a typo for example), then the module will not warn you, and you might get "use of uninitialized value" warnings. Please check your code before you send in those bug reports :) Comments and suggestions regarding the module are greatly appreciated.

TPJ

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

Listing 1

```
#!perl -w

use strict;
use Tk;
use Tk::LabEntry;
use AI::FuzzyInference;

use constant PI => 3.1415927;
use constant G  => 9.81;

my $halfLenRod   = 5;
my $timeStep     = 0.05;

my $thRod;    # between -30 and 30 degrees.
my $velBall;  # ball's velocity. From -15 .. 15 m/s
my $posBall;  # ball's position. From -5 .. 5 m
my $time = 0;

# initialize.
$thRod   = -10;
$velBall = -2;
$posBall = 4;

# create the FIS.
my $fis = new AI::FuzzyInference;
```

```
# define the input variables.
$fis->inVar(posBall  => -5, 5,
        far_left  => [-4, 1, -2, 0],
        left      => [-4, 0, -2, 1, 0, 0],
        center    => [-2, 0, 0, 1, 2, 0],
        right     => [0, 0, 2, 1, 4, 0],
        far_right => [2, 0, 4, 1],
       );

$fis->inVar(velBall   => -15, 15,
        fast_neg   => [-9, 1, -3, 0],
        medium_neg => [-9, 0, -3, 1, 0, 0],
        slow       => [-3, 0, 0, 1, 3, 0],
        medium_pos => [0, 0, 3, 1, 9, 0],
        fast_pos   => [3, 0, 9, 1],
       );

# define the output variable.
$fis->outVar(thRod      => -30, 30,
         large_neg  => [-20, 1, -10, 0],
         medium_neg => [-20, 0, -10, 1, 0, 0],
         small      => [-10, 0, 0, 1, 10, 0],
         medium_pos => [0, 0, 10, 1, 20, 0],
         large_pos  => [10, 0, 20, 1],
       );

# now define the rules.
$fis->addRule(
          'posBall=far_left  & velBall=fast_neg'   => 'thRod=large_pos',
          'posBall=far_left  & velBall=medium_neg' => 'thRod=large_pos',
          'posBall=far_left  & velBall=slow'       => 'thRod=medium_pos',
          'posBall=far_left  & velBall=medium_pos' => 'thRod=medium_pos',
          'posBall=far_left  & velBall=fast_pos'   => 'thRod=medium_pos',

          'posBall=left      & velBall=fast_neg'   => 'thRod=large_pos',
          'posBall=left      & velBall=medium_neg' => 'thRod=medium_pos',
          'posBall=left      & velBall=slow'       => 'thRod=medium_pos',
          'posBall=left      & velBall=medium_pos' => 'thRod=medium_pos',
          'posBall=left      & velBall=fast_pos'   => 'thRod=medium_pos',

          'posBall=center    & velBall=fast_neg'   => 'thRod=large_pos',
          'posBall=center    & velBall=medium_neg' => 'thRod=medium_pos',
          'posBall=center    & velBall=slow'       => 'thRod=small',
          'posBall=center    & velBall=medium_pos' => 'thRod=medium_neg',
          'posBall=center    & velBall=fast_pos'   => 'thRod=large_neg',

          'posBall=right     & velBall=fast_neg'   => 'thRod=medium_neg',
          'posBall=right     & velBall=medium_neg' => 'thRod=medium_neg',
          'posBall=right     & velBall=slow'       => 'thRod=medium_neg',
          'posBall=right     & velBall=medium_pos' => 'thRod=medium_neg',
          'posBall=right     & velBall=fast_pos'   => 'thRod=large_neg',

          'posBall=far_right & velBall=fast_neg'   => 'thRod=medium_pos',
          'posBall=far_right & velBall=medium_neg' => 'thRod=medium_neg',
          'posBall=far_right & velBall=slow'       => 'thRod=medium_neg',
          'posBall=far_right & velBall=medium_pos' => 'thRod=large_neg',
          'posBall=far_right & velBall=fast_pos'   => 'thRod=large_neg',
         );

drawGUI();

MainLoop;

# this subroutine calculates the new values of the ball's position
# and velocity after a period of time $timeStep.
# Friction is not modeled.
sub calcNewData {
  my $acc  = G * sin ($thRod * PI / 180);
  my $Vnew = $velBall + $acc * $timeStep;
  my $dist = $velBall * $timeStep + 0.5 * $acc * $timeStep * $timeStep;

  $velBall  = $Vnew;
  $posBall += $dist;

  $velBall =  15 if $velBall >  15;
  $velBall = -15 if $velBall < -15;

  $time += $timeStep;
}

# This sub draws the gui.
sub drawGUI {
   my $mw = new MainWindow;

   my $canvas = $mw->Canvas(qw/-bg black -height 400 -width 600/)->pack;

   $canvas->createLine(0, 0, 0, 0,   qw/-width 2 -fill white -tags ROD/);
   $canvas->createOval(0, 0, 50, 50, qw/-fill green -tags BALL/);

   my $f = $mw->Frame->pack(qw/-fill x/);

   my $dth;

   $f->Button(-text    => 'run',
         -command => sub {
        my $id;
        $id = $canvas->repeat(100 => sub {
                    # update the ball's data.
                    calcNewData();

                    # check for termination conditions.

                    # stop if ball is almost stationary and the rod
                    # is almost flat.
                    if (abs($velBall) < 0.005 && abs($thRod) < 0.001) {
                      print "Simulation ended.\n";
                      $canvas->afterCancel($id);
                      return;
                    }

                    # stop if ball fell off the rod.
                    if ($posBall > $halfLenRod or $posBall < -$halfLenRod)
{
                      print "Ball fell off the rod!\n";
                      $canvas->afterCancel($id);
                      return;
                    }

                    # compute the new angle of the rod.
                    $fis->compute(posBall => $posBall,
                              velBall => $velBall);
                    $thRod = $fis->value('thRod');

                    # update our drawing.
                    updateCanvas($canvas);
                      });
         })->pack(qw/side left -ipadx 10/);

   $f->LabEntry(-label => 'Ball Pos',
        -textvariable => \$posBall,
           )->pack(qw/-side left -padx 10/);

   $f->LabEntry(-label => 'Ball Speed',
        -textvariable => \$velBall,
           )->pack(qw/-side left -padx 10/);

   $f->LabEntry(-label => 'Rod Angle',
        -textvariable => \$thRod,
           )->pack(qw/-side left -padx 10/);

   updateCanvas($canvas);
}

# This subroutine draws the rod at its current angle, and
# the ball at its current position.
sub updateCanvas {
  my $c = shift;

  my $ly = 200;
  my $dy = int(40 * $halfLenRod * tan(PI * $thRod / 180));
  $c->coords(ROD => 100, $ly - $dy, 500, $ly + $dy);

  my $by = 150 + $posBall * $dy / $halfLenRod;

  my $bx = 100 + (5 + $posBall) * 40;
  $c->coords(BALL => $bx - 25, $by, $bx + 25, $by + 50);
}

# tangent sub.
sub tan { sin($_[0]) / cos($_[0])  }
```

TPJ

Brian Ingerson

Creating Module Distributions with *Module::Install*

There is no doubt that Perl has the best code reuse infrastructure of all the open-source programming languages. This is due to the wonder known as the "Perl module." Almost every modern programming language worth a second look has a modular extension mechanism. But that's not what I'm getting at. The Perl module is different.

The reason has every bit as much to do with community and heroics as it does with technology and programming. Some very dedicated people have spent many years of their lives collectively nurturing the Perl module and, in the process, it has become something of a superstar of modern programming. The Perl module has a lot going for it:

- **Good Documentation.** Perl modules have an exceptionally high quality of documentation, and the ubiquitous POD format converts nicely into all sorts of other formats including UNIX manpages, LaTeX, and HTML.
- **Object Orientation.** Object-oriented interfaces are available for a large portion of modern modules.
- **Consistency of Installation.** 99.9 percent of all Perl modules install the same way: *perl Makefile.PL && make install*. This has made it possible to create sophisticated installation tools.
- **Testability.** Perl modules have a standard for running unit tests that allow them to be tested by users with a single *make test*.
- **Huge Repository.** CPAN has over 5500 modules. No other programming language can top that.

The Perl forefathers spent a lot of time, thought, and hard work developing good standards for Perl modules. Because of this, Perl was able to excel in this domain while other languages have struggled.

But the Perl module has been in a bit of a rut lately, due to weaknesses that have begun to appear in the module installation process, some of which I'll discuss in this article. Perl has had many accomplishments, but the next big development in Perl may hinge on this installation question. However, the Perl community is (justifiably) wary of large overhauls that may cause more turmoil than they are worth.

In this article, I'll describe a new module called *Module::Install*, coauthored by Autrijus Tang and myself. It looks at things from a whole new perspective and breaks down many common barriers to module authoring. It also honors the three guiding principles of Perl:

- There's More Than One Way To Do It (TMTOWTDI).
- Do What I Mean (DWIM).
- It Just Works!

There were other key ideas that affected the design of *Module::Install*:

1. *Module::Install* should reduce the effects of module dependencies. If an author of a module (let's call it module X) incorporates *Module::Install* into a module, *Module::Install* itself need not even be available on CPAN to the users of module X.
2. The average module user should never need to directly install *Module::Install*.
3. Even if *Module::Install* becomes widely used, a need for backward compatibility should never be a limitation.
4. *Module::Install* should get smarter over time. You should be able to teach it new tricks every time you use it.

MakeMaker

In the beginning, it was decided that the best way to build and manipulate Perl modules was with the UNIX "*make*" utility. This program is nearly ubiquitous and is a sharp tool for automating all sorts of processes, especially those having to do with building and configuring software. That means every Perl module (or more accurately, module distribution) needs a "makefile"—the file that contains all the build rules.

The problem is that you can't create a makefile that is portable enough to work on all systems. This has to do both with differences in the system command processors (shells) and differences in the various implementations of make itself. The forefathers got around this problem in a clever way: They wrote a Perl module called *ExtUtils::MakeMaker* that could generate a proper makefile on any system. All it needed was a set of simple directives from the module author.

The convention has always been to put these directives in a Perl program called *Makefile.PL*. This has the advantage that an author can write any Perl code they want in order to set the directives. Often, this involves inspecting the local system, or prompting

Brian has been programming for 20 years, the last five of those in Perl. He lives in Portland, Oregon, and can be contacted at ingy@ttul.org.

the person installing the module. In its most basic form, a *Makefile.PL* looks like this:

```
use ExtUtils::MakeMaker;
WriteMakefile
    NAME => 'Acme::Pie',
    VERSION => '3.14';
```

When anybody runs the command:

```
perl Makefile.PL
```

ExtUtils::MakeMaker will generate a rather large makefile for their system. This Makefile is capable of all sorts of tasks relating to Perl module building. If you dare to study the *MakeMaker* sources, you'll find them to be an arcane, nonextensible behemoth of a code base. In short, enough spaghetti code to feed the Italian Army. The code basically works, but it ain't gonna get much better. This ugliness leaves us with the good and the bad.

The good things about *ExtUtils::MakeMaker* are:

- It can produce a makefile that will work on all Perl platforms from Windows to UNIX to VMS to Mac OS (the old one!).
- The makefile can do everything that a Perl module needs to do—run tests, build and link libraries, create documentation, and even generate a module distribution ready for CPAN.
- It has been shipped with Perl for a long time. That means a module author can expect it to be on virtually every installation of Perl that they would ship their module to.

The bad things about *ExtUtils::MakeMaker* are:

- It relies on *make*. Why would such a powerful language need such a crufty little old UNIX utility to do what amounts to little more than invoking other commands?
- A *make* utility isn't always available on all platforms by default. Win32 is a common problem spot as it requires "nmake" to be installed.
- It can't realistically be overhauled. There are literally millions of Perl installations in the world with all sorts of administrative policies. How would you get them all to upgrade? It's like rolling a snowball up a mountain.
- It doesn't do what it doesn't do. Authors of modules with a complex build process end up writing lots of extra custom code inside *Makefile.PL* to make (no pun intended) up for the missing features of *ExtUtils::MakeMaker*.

The Cure

So the Perl Module is kind of stuck in its legacy. Or is it?

Module::Install offers some fairly clever solutions to the problems listed above. Its *modus operandi* is "embrace and extend," but in a better, more Perlish sense of that phrase. *Module::Install* simply makes use of all the goodness offered by *ExtUtils::MakeMaker*, and adds any extras as needed. Let's look at it in action:

```
use inc::Module::Install;
name        ('Acme::Pie');
version     ('3.14');
check_nmake ();
&Makefile->write;
```

This does the same thing as the previous example, but in a completely different way. Let's see how: The first thing you'll notice is that we didn't use *Module::Install* at all. We used *inc::Module::Install*—and therein lies the core of *Module::Install*'s subtle trickery.

Authors and Users

There are three types of people in the world—well, the Perl world, anyway:

- Perl Module Authors.
- Perl Module Users.
- Perl Module Users who are also Authors.

Module::Install is different from most Perl modules because it is only installed by module authors. When the author runs *perl Makefile.PL*, the module called *inc::Module::Install* copies the module called *Module::Install* into the *./inc/* directory, if it doesn't already exist. Therefore, the new file has the path *./inc/Module/Install.pm*.

Next, *inc::Module::Install* adds *./inc/* to *@INC*, Perl's module-path lookup variable. Finally *inc::Module::Install* does a *require Module::Install*. The module that gets loaded is the one that was just copied. And the *Makefile.PL* proceeds.

There are several (potentially dozens) of submodules that *Module::Install* might need to make use of. Whenever one of these modules is needed, it goes through the same process of copying to the local directory and then loading into memory. Understanding this is the first key to understanding *Module::Install*.

All of the modules that get copied into the local *./inc/* directory need to be added to the MANIFEST file so that they will be shipped to the user. This can be done with the *make manifest* command, and *Module::Install* will warn you if you try to do a *make dist* without doing this first.

When the user gets the module, all of *Module::Install*'s parts will already be packaged right inside the module—and that's the whole point. When the user runs the *Makefile.PL,* it will load *inc::Module::Install* directly from the local directory. This happens because Perl always has "." in the *@INC* path, so no user ever has to install *Module::Install* directly.

If the user also happens to be an author with his own (and possibly incompatible) version of *Module::Install*, his copy of *inc::Module::Install* will load the local copy from *./inc/*. This way, there can never be a conflict between different versions of *Module::Install*. The one that the module shipped with will always be the one used, so backward compatibility need not be maintained.

Features on Demand

Back to the example *Makefile.PL*. The next two lines are:

```
name        ('Acme::Pie');
version     ('3.14');
```

These do exactly what you would think. They set the *NAME* and *VERSION* parameters so that *Module::Install* can call *ExtUtils::MakeMaker::WriteMakefile()* internally for you. But don't assume they aren't special. They are.

name and *version* are not exported subroutines as you might think. They are simply two of myriad subroutines that exist in some unknown module. As long as that unknown module's name begins with *Module::Install::*, the subroutine will be found.

Module::Install uses Perl's AUTOLOAD facility to catch the unknown subroutines, locate them in some module, and make sure that this module is inside the local *./inc/* directory. *Module::Install* only adds the modules that you actually need for your *Makefile.PL*. Note that when a module gets bundled into *./inc/* it is "pod stripped." In other words, all the documentation is removed. Since the *./inc/* modules never get installed by the user, the pod is just a waste of space.

Next we have:

```
check_nmake ();
```

This only comes into play when a user is installing your module on an MSWin32 system. It will check to see if nmake is installed. If not, it will download it from the Microsoft web site. Of course, it will ask the user first, just to be polite. This solves the common problem of people not having nmake on Windows.

Finally we have:

```
&Makefile->write;
```

Makefile is a subroutine that returns a *Module::Install::Makefile* object. This object holds all the info that is needed to create a Makefile by invoking *ExtUtils:MakeMaker::WriteMakefile*. The reason we don't just say *write* is because there are several modules that have *write* functions, as we'll see shortly. The *name* and *version* subroutines also happen to come from *Module::Install::Makefile*, so we could have written things like this:

```
use inc::Module::Install;
&Makefile->name    ('Acme::Pie');
&Makefile->version ('3.14');
&Makefile->write;
```

Reducing Dependencies

Module::Install works independently. My recent CPAN module, *only.pm*, has been on CPAN for over two months, and it uses *Module::Install*. But as I am writing these words, *Module::Install* hasn't even been released. It simply doesn't need to be because all the parts of *Module::Install* that *only.pm* needs are shipped along with *only.pm*.

In fact, Autrijus and I have started moving all of our modules over to *Module::Install*. We're testing our own creation. The whole *Module::Install* idea stems from a module that I released last year called *CPAN::MakeMaker*. Autrijus figured out how to accomplish all my goals in a simpler, more extendable way. When *Module::Install* is released, it will completely replace the need for *CPAN::MakeMaker*, which I will likely remove from CPAN.

Module::Install should be released on CPAN by the time you read this. If not, keep an eye out for it.

Bundles of Joy

Module:Install has another keen feature: bundling. You can actually convince *Module::Install* to pull in entire modules that it needs for building things. A great example is Michael Schwern's *Test::More*. It is great for writing tests, but some people don't use it because it doesn't exist by default on older versions of Perl. It's a tradeoff of whether or not to burden the users for the sake of writing better tests that, in the end, the users probably don't care about.

Module::Install makes it a nonissue. It can simply pull in the author's version of *Test::More* right into the author's module. Here's an example:

```
include_deps    ('Test::More', 5.004);
build_requires  ('Test::More');
```

The first line instructs *Module::Install* to bundle all the parts of your local *Test::More* into *./inc/*, as long as they support Perl 5.004 and higher. If they don't support 5.004, no bundling will take place.

The second line simply says that *Test::More* is required for building this module. By the way, the bundled *Test::More* modules are POD-stripped so the extra weight is minimal.

What About *Module::Build*?

Anybody who has been following the Perl module authors' scene in the past couple years should know about the *Module::Build* project. This is actually Ken Williams's attempt to do the impossible: replace *MakeMaker*. And he's doing a darn good job of it, too.

The main goal of *Module::Build* is to do everything important that *MakeMaker* does, but without using *make*. Ken has also done a fabulous job of keeping the code object oriented and clean as a whistle. That makes it easy for others to contribute to the project.

One of the hurdles Ken faces is getting people to use his module. Like *Test::More*, authors will have to make a choice between using the ubiquitous *MakeMaker* or the riskier new upstart, *Module::Build*.

Module::Install will be a great ally for Ken. That's because *Module::Install* can use the same *Makefile.PL* that was created for *MakeMaker* to use *Module::Build* instead. Here's how to convert our *Makefile.PL* to use *Module::Build*:

```
use inc::Module::Install;
name        ('Acme::Pie');
version     ('3.14');
check_nmake ();
&Build->write;
```

All I did was change *Makefile* to *Build*. Everything else remains the same! If I really wanted to cover my bases, I could ship with both methods in place like this:

```
use inc::Module::Install;
name        ('Acme::Pie');
version     ('3.14');
check_nmake ();
&Makefile->write;
&Build->write;
```

If we also bundle in the *Module::Build* software, then there is almost no risk in using *Module::Build*. And then one day when *Module::Build* has replaced *MakeMaker* everywhere, authors can start dropping *MakeMaker* support.

No Big Thing

You may think that it's not a wise thing to add extra weight to all the CPAN modules. I personally wouldn't lose too much sleep over it. In the general case, you're only adding a few kilobytes once everything is gzip compressed. And much of this added weight is directly beneficial to your module.

Of course, it is possible to overdo it. I wouldn't suggest bundling the *Tk* module, for instance. But as long as you employ a modicum of wisdom, you (and the rest of the CPAN community) should be just fine.

Metadata Support

The future of a better CPAN lies in the ability of CPAN to easily obtain good metadata about each module. For instance, it would be immensely beneficial for CPAN to be able to report the license that each module is distributed under. Unfortunately, there is no good way to determine this unless the module author declares it somewhere.

Ken Williams realized this and created the META.yml standard file for holding module metadata. His *Module::Build* project encourages the use of META.yml. Michael Schwern has also added support for generating a META.yml file.

Module::Install has very easy-to-use metadata support. Just add as many metadata fields as you wish to the *Makefile.PL* and *Module::Install* will make sure they get into the META.yml file.

Here is the *Makefile.PL* file for *only.pm*:

```
use inc::Module::Install;
name          ('only');
version_from  ('lib/only.pm');
abstract      ('Load specific module versions; Install many');
author        ('Brian Ingerson <ingy@cpan.org>');
```

```
license        ('perl');
include_deps   ('Test::More', 5.004);
build_requires ('Test::More', 0);
clean_files(qw(lib/only/config.pm));
clean_files(qw(t/lib t/site t/distributions t/version t/alternate));
create_config_module();
check_nmake();
&Meta->write;
&Makefile->write;
```

As you can see, creating a META.yml is no harder than creating a makefile. Here's what the META.yml looks like:

```
name: only
version: 0.26
abstract: Load specific module versions; Install many
author: Brian Ingerson <ingy@cpan.org>
license: perl
distribution_type: module
build_requires:
  Test::More: 0
private:
  directory:
    - inc
```

CPAN for Scripts

Traditionally, CPAN has been for distributing Perl modules, but not really for distributing Perl scripts. Some modules such as *LWP* and *YAML* come with a few helper scripts, but finding a distribution that only installs a Perl script in your path is rare indeed.

This distinction is somewhat artificial, in my opinion. I think the real reason that people don't generally do this is that it isn't easy to figure out how to do it. Here's a recipe for doing it with *Module::Install*:

First, create a script called "Hello World" that looks like this:

```
#!/usr/bin/perl -w
use strict;
print ;"Hello, world\n";
__END__
=head1 NAME - Hello World Script
Don't forget your documentation!
=cut
```

Then write the *Makefile.PL*:

```
use inc::Module::Install;
name('Hello-World');
version('1.23');
install_script('hello-world');
&Makefile->write;
```

It's that easy. Run these commands:

```
perl Makefile.PL
make manifest
make dist
```

And upload the tarball to CPAN. Well, actually there's a bit more to it than that: You still need to create some tests and a README and a Changes file. And, yes, you need to write a more useful script with better documentation. But that's about all.

Extending *Module::Install*

The best part of *Module::Install* might not come from Autrijus or myself, but from you. You can easily create your own extensions to *Module::Install*. The trick is to put your functionality into a module called *Module::Install::PRIVATE::FooBar*. By using the private namespace, you'll avoid any possible collisions with future *Mode::Install::* modules.

Say you had a *Makefile.PL* that looked like this:

```
use inc::Module::Install;
name('Acme::Pie');

print "Are you 18 years or older? ";
if (<> !~ /^Y/i) {
    die "Can't install this module for minors\n";
}

&Makefile->write;
```

You could hide the query in a module, *Module::Install::PRIVATE::Assertions*:

```
package Module::Install::PRIVATE::Assertions;
use base 'Module::Install::Base';

sub check_18 {
    print "Are you 18 years or older? ";
    if (<> !~ /^Y/i) {
        die "Can't install this module for minors\n";
    }
}

1;
```

Then your *Makefile.PL* would look like this:

```
use inc::Module::Install;
name('Acme::Pie');
check_18();
&Makefile->write;
```

That's so much cleaner. And you can use it over and over again for all your modules.

Conclusion

Module::Install is just coming onto the scene. We hope that its fresh outlook on building Perl modules will help to keep Perl's future bright.

Module::Install is just one of several *Module::** modules that we hope will reinvent the way Perl authors do business. One of these will be *Module::Test*. Although it might not make its debut for a couple of months, its mission is clear—to do for Perl testing what *Module::Install* does for Perl modules.

TPJ

Moshe Bar

Integrating OS X Aqua and Perl

With millions of Mac OS X (and they do insist you pronounce the X as "ten") installations around the world, it finally is possible to have both the stability, performance, and configurability of UNIX and the ease of use, good looks, and multimedia capability of the Mac.

One of OS X's most intriguing features is its elaborate developer framework. One can write OS X applications using the Cocoa, Carbon, Java, or Classic frameworks. The Classic framework allows older pre-OS X programs (i.e., Mac OS 9) to continue to run under OS X.

Carbon provides more traditional methods to access Mac OS X features. The Carbon APIs can be used to write Mac OS X applications that also run on previous versions of the Mac OS (8.1 or later).

The Cocoa application environment is designed specifically for Mac OS X-only native applications. It is comprised of a set of object-oriented frameworks that support rapid development and are designed to promote high productivity. The Cocoa frameworks include a full-featured set of classes designed to create robust and powerful Mac OS X applications. The object-oriented design simplifies application development and debugging.

Cocoa provides developers starting new Mac OS X-only projects the fastest way to full-featured implementations. Applications from UNIX and other OS platforms can also be brought to Mac OS X quickly by using Cocoa to build Aqua user interfaces while retaining most existing core code.

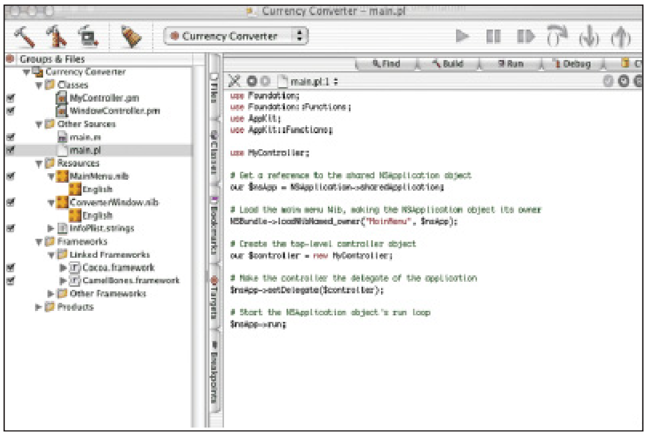
One can use Apple's standard Project Builder integrated development environment (IDE) to create, edit, compile, and debug Cocoa applications. One of the main attractions of Cocoa is its tight integration with the Aqua user interface. Aqua is a class framework/API that allows you to write programs inheriting the full power of the OS X GUI, including its expressive icons, vibrant color, and fluid motion. Aqua includes a number of innovative time- and work-saving features that help you navigate and organize your system. Therefore, it is possible to write a simple program and let it inherit the looks and ease of use of OS X.

In Aqua, title bars of windows are no longer the old and boring Mac OS platinum gray, but a subtle blending of translucent stripes. Instead of a border around a window, a drop shadow adds depth to your perception. This shadow grows larger for the window in the foreground, adding a visual cue to how many windows you have open. As you move windows around, the contents move also, instead of just the outline. Aqua also provides basic functionality such as file opening and saving from within your application. When one of these save dialogs is open, a popup menu offers access to your favorite and most recently used folders. To save the document somewhere else, simply disclose the full filesystem browser. It also offers spring-loaded folders that snap open when you drag things into them in the Finder view.

The sample Objective C program in Listing 1 shows how easy it is to use OS X features. I have been using OS X for half a year now and one of the first questions I had was how to integrate Cocoa with the ease of use and power of Perl.

A fast search on the Internet revealed that this integration had already been achieved with an open-source integration tool called "CamelBones" (see http://camelbones.sourceforge.net/). CamelBones is a framework that allows many types of Cocoa programs to be written entirely in Perl. It also provides a high-level object-oriented wrapper around an embedded Perl interpreter, so that Cocoa programs written in Objective-C can easily make use of code and libraries written in Perl.

An example of a currency converter using the Aqua interface and the Cocoa framework shows how easy it is to tightly integrate Perl in OS X. In Figure 1, you can see the seamless integration of Perl code and Cocoa in the IDE tool.



```
use Foundation;
use Foundation::Functions;
use AppKit;
use AppKit::Functions;

use MyController;

# Get a reference to the shared NSApplication object
our $nsApp = NSApplication->sharedApplication;

# Load the main menu Nib, making the NSApplication object its owner
NSBundle->loadNibNamed_owner("MainMenu", $nsApp);

# Create the top-level controller object
our $controller = new MyController;

# Make the controller the delegate of the application
$nsApp->setDelegate($controller);

# Start the NSApplication object's run loop
$nsApp->run;
```

Figure 1: CamelBones integrated into the Cocoa framework. Notice the Frameworks tab, which now includes CamelBones.framework.

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

Listing 2 shows how to call Cocoa methods from within Perl using the CamelBones framework. Notice how Perl is used to interact with the user on the screen:

```
# Outlets
     $self->{'RateField'} = undef;
     $self->{'DollarField'} = undef;
     $self->{'TotalField'} = undef;
     $self->{'Window'} = undef;

     bless ($self, $class);

     $self->{'NSWindowController'} =
              NSWindowController->alloc>initWithWindowNibName_owner(
                                                 "ConverterWindow", $self);
     $self->{'NSWindowController'}->window;
```

The *Window*, *TotalField*, *DollarField*, and *RateField* widgets need to be created in the OS X Interface Builder (which is included with the Developer Tools package along with Project Builder) and are an instance of the *NSTextField* class. *NSTextField*, like most GUI widget classes, is a subclass of *NSControl*, and inherits the *setStringValue* method from that class. Instance methods are called for GUI widgets by treating the outlets connected to them as object references. So, the sample *sayHello* method should look like this:

```
sub sayHello {
           my ($self, $sender) = @_;
           $self->{'TextLabel'}->setStringValue("Hello");
     }
```

If you don't know what *$self* is, this would be a great time to read Tom Christiansen's excellent OO tutorial, found in the *perltoot* POD document.

Event Handling

Any interactive application (like the currency converter used here) needs to handle events, such as buttons pressed. With CamelBones, you can (and should in fact) register event handlers, too.

There are some rough edges in CamelBones. The integration with the GUI still needs to be done through Objective-C. That's because the GUI resource is "owned" by an Objective-C class, rather than one that's defined in Perl. Handling menu selections is a bit different than handling actions sent by other GUI widgets.

To handle a menu selection action, you connect the action to the "First Responder" in Project Builder, instead of the "File's Owner." This will allow you to register your own Perl event handler.

CamelBones is a framework that allows many types of Cocoa programs to be written entirely in Perl

In Project Builder, the Perl method to handle the menu action should be added to the *MyApp* class, instead of the *MyWindowController* class.

Listing 3 shows the sample event controller for the currency converter.

Conclusion

Though it still has a few rough edges, the CamelBones framework is a good start. It allows a relatively easy integration of Perl and Cocoa, which should get you started in connecting your Perl code to the OS X Aqua interface.

TPJ

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

Listing 1

```
#include <Carbon/Carbon.h>
#include "CocoaBundle.h"
enum
{
    kOpenCocoaWindow = 'COCO'
};
CFBundleRef bundleRef = NULL;
static OSStatus
handleBundleCommand(int commandID) {
    OSStatus osStatus = noErr;
    if (commandID == kEventButtonPressed) {
        OSStatus (*funcPtr)(CFStringRef message);
                funcPtr = CFBundleGetFunctionPointerForName(bundleRef,

CFSTR("changeText"));
                require(funcPtr, CantFindFunction);
                osStatus = (*funcPtr)(CFSTR("button pressed!"));
                require_noerr(osStatus, CantCallFunction);
    }
CantFindFunction:
CantCallFunction:
        return osStatus;
}
static void
myLoadPrivateFrameworkBundle(CFStringRef framework, CFBundleRef *bundlePtr)
{
    CFBundleRef appBundle = NULL;
    CFURLRef baseURL = NULL;
    CFURLRef bundleURL = NULL;
        appBundle = CFBundleGetMainBundle();
        require(appBundle, CantFindMainBundle);
            baseURL = CFBundleCopyPrivateFrameworksURL(appBundle);
            require(baseURL, CantCopyURL);
                bundleURL =
CFURLCreateCopyAppendingPathComponent(kCFAllocatorSystemDefault, baseURL,
                                        CFSTR("CocoaBundle.bundle"),
false);
                require(bundleURL, CantCreateBundleURL);
                    bundleRef = CFBundleCreate(NULL, bundleURL);
                        CFRelease(bundleURL);
CantCreateBundleURL:
                            CFRelease(baseURL);
CantCopyURL:
CantFindMainBundle:
                            return;
}

static OSStatus
appCommandHandler(EventHandlerCallRef inCallRef, EventRef inEvent, void*
userData) {
    HICommand command;
    OSStatus (*funcPtr)(void *);
    OSStatus (*showPtr)(void);
    OSStatus err = eventNotHandledErr;

    if (GetEventKind(inEvent) == kEventCommandProcess) {
        GetEventParameter( inEvent, kEventParamDirectObject, typeHICommand,
                           NULL, sizeof(HICommand), NULL, &command );
        switch ( command.commandID ) {
            case kOpenCocoaWindow:

                myLoadPrivateFrameworkBundle(CFSTR("CocoaBundle.bundle"),

&bundleRef);
                require(bundleRef, CantCreateBundle);
                                // call function to initialize bundle
                funcPtr = CFBundleGetFunctionPointerForName(bundleRef,
                                              CFSTR("initializeBundle"));
                require(funcPtr, CantFindFunction);
                err = (*funcPtr)(handleBundleCommand);
                require_noerr(err, CantInitializeBundle);
                                // call function to show window
                showPtr = CFBundleGetFunctionPointerForName(bundleRef,
                                              CFSTR("orderWindowFront"));
                require(showPtr, CantFindFunction);
                err = (*showPtr)();
                require_noerr(err, CantCallFunction);
                CantCreateBundle:
                CantCallFunction:
                CantInitializeBundle:
                CantFindFunction:
                break;
            default:
                break;
        }
    }
        return err;
}
int main(int argc, char* argv[])
{
    IBNibRef   nibRef;
    WindowRef   window;
        OSStatus  err;
            EventTypeSpec cmdEvent = {kEventClassCommand,
kEventCommandProcess};
                // Create a Nib reference passing the name of the nib file
                // (without the .nib extension)
                // CreateNibReference only searches into the application
bundle.
            err = CreateNibReference(CFSTR("main"), &nibRef);
            require_noerr( err, CantGetNibRef );
                // Once the nib reference is created, set the menu bar.
                // "MainMenu" is the name of the menu bar
                // object. This name is set in InterfaceBuilder when the
nib is
                // created.
            err = SetMenuBarFromNib(nibRef, CFSTR("MenuBar"));
            require_noerr( err, CantSetMenuBar );
                // Then create a window. "MainWindow" is the name of the
window
                // object. This name is set in
                // InterfaceBuilder when the nib is created.
            err = CreateWindowFromNib(nibRef, CFSTR("MainWindow"),
&window);
            require_noerr( err, CantCreateWindow );
                // We don't need the nib reference anymore.
            DisposeNibReference(nibRef);
                // The window was created hidden so show it.
            ShowWindow( window );

InstallApplicationEventHandler(NewEventHandlerUPP(appCommandHandler), 1,
                               &cmdEvent, 0, NULL);
                    // Call the event loop
                RunApplicationEventLoop();
                CantCreateWindow:
CantSetMenuBar:
CantGetNibRef:
                    return err;
}
```

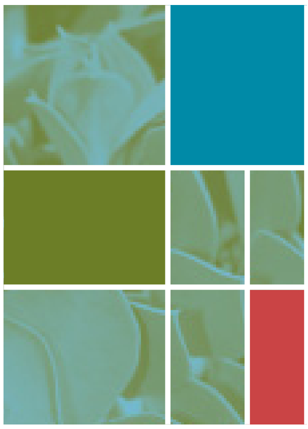
Listing 2

```
package WindowController;

use Foundation;
use Foundation::Functions;
use AppKit;
use AppKit::Functions;

@ISA = qw(Exporter);

sub new {
    # Typical Perl constructor
    # See 'perltoot' for details
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = {};

    # Outlets
    $self->{'RateField'} = undef;
    $self->{'DollarField'} = undef;
    $self->{'TotalField'} = undef;
    $self->{'Window'} = undef;

    bless ($self, $class);

    $self->{'NSWindowController'} =
                NSWindowController->alloc->initWithWindowNibName_owner(
                                                        "ConverterWindow",
$self);
    $self->{'NSWindowController'}->window;

    return $self;
}

sub convertButtonClicked {
    my $self = shift;

    my $rate = $self->{'RateField'}->floatValue;
    my $amount = $self->{'DollarField'}->floatValue;
    my $total = $rate * $amount;

    $self->{'TotalField'}->setFloatValue($total);

    return 1;
}

1;
```

Listing 3

```
package MyController;

use Foundation;
use Foundation::Functions;
use AppKit;
use AppKit::Functions;

use WindowController;

@ISA = qw(Exporter);

our $wc = undef;

sub new {
    # Typical Perl constructor
    # See 'perltoot' for details
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = {
    };
    bless ($self, $class);

    return $self;
}

sub applicationWillFinishLaunching {
    my ($self, $notification) = @_;

    # Create the new controller object
    $wc = new WindowController;

    return 1;
}

1;
```

TPJ



Managing Newsletters with Perl

Simon Cozens

It ought to have been very simple. I needed to produce a newsletter. The content was going to be created offline, and then uploaded to a processing script that was going to distribute it electronically. And Perl was going to help me.

It ought to have been very simple, but like so many things, it ended up being a little more complex than that. Complex enough, I hope, that there are one or two things involved in the process of creating the newsletter that we can all learn from.

The first idea I had was to produce a newsletter in HTML, which people could look at on the Web, or print off and distribute to less Internet-aware friends. And because I hate producing HTML by hand, I used the Template Toolkit to template it out. Let's begin by looking at how I did that.

Template Toolkit

Andy Wardley's Template Toolkit is a fantastically useful suite of Perl modules that implement a parser and interpreter for a little templating language. Templating languages are most often used to fill values computed by a program into some text. For instance, we could have a template like this:

```
[% today %]

[% title %] [% forename %] [% surname %]
[% address %]

Dear [% title %] [% surname %],
    Thank you for your letter dated [% their_date %]. This is to
confirm that we have received it and will respond with a more
detailed response as soon as possible. In the mean time, we
enclose more details of ...
```

We tell Template Toolkit what the various values of *today*, *title*, and so on ought to be, and it fills out the template.

Of course, as we're about to find out with our newsletter project, things that start out nice and simple have a way of getting bigger and more complex. Template Toolkit supports a lot more than just filling scalars into a form: It has support for arrays, hashes and objects, the ability to include templates inside templates, declare macros, run blocks multiple times, filter text through various functions, and much more. Thankfully, we're only going to use a small amount of this functionality in the newsletter.

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.*

The HTML page we're constructing is slightly tricky. It uses CSS to lay out text in three columns. We'll have a header, a column describing generally what the newsletter is about, a main column of news, and then a further column of other information—how to get in touch with me, and so on. At the bottom, we'll put some information about how to make sure people have the latest edition of the newsletter.

The top of the HTML is static, so we pass that out to a separate file. Let's assume we have a file called "Head" that contains all the HTML header and the initial <*body*> tag.

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>Simon's Newsletter</title>
...
</head>
<body>
<div class="box-wrap">
```

Similarly, we have a static "Foot" file:

```
</div>
</body>
</html>
```

Now we can forget about most of the mucky business of HTML and concentrate on the content:

```
[% INCLUDE Head %]
<p> My newsletter will appear here. </p>
[% INCLUDE Foot %]
```

We can process this with the Template Toolkit using the following bit of Perl:

```
use Template;
my $template = Template->new();
$template->process("newsletter.thtml");
```

(I use the extension "thtml" to remind myself that this is templated HTML.)

With that little program, the generated HTML page gets spat out to standard output. Great. In fact, Template Toolkit comes with a handy little utility called *tpage*, which is functionally equivalent to our Perl program above. You can just say *tpage newsletter.thtml*, and Template Toolkit will process the template in the same way.

So far so good. But of course, we haven't used any template variables yet. Let's add some now, by giving the newsletter an issue number and date:

```
[% INCLUDE Head %]
<div class="box-header">
<h1 align="center">Simon's Newsletter</h1>
<p align="right"><I>Issue [% issue %] - [% date %]</I></p>
</div>
<p> My newsletter will appear here. </p>
[% INCLUDE Foot %]
```

Now we need a way to tell the template what the values of these variables are. We do this by passing in a hash reference as the second parameter of *process*.

```
use Template;
my $template = Template->new();
my $vars = {
    issue => 1,
    date => scalar localtime
};
$template->process("newsletter.thtml", $vars);
```

Once again, the newsletter will be produced on standard output. As it happens, we can still use *tpage*, even if we're adding template variables. We can say:

```
tpage —define issue=1 —date="May 17th" newsletter.thtml
```

tpage is a very handy tool for prototyping your templates in the way we're doing here.

Now we have our header out of the way. Let's move on to our three columns. We'll handle them in order of complexity. The left-hand column is just static text, so we can dispose of that trivially:

```
<div class="column-two">
<div class="column-two-content">
<h2>WEC Trek to Japan</h2>
[% INCLUDE trek %]
</div>
</div>
```

The right-hand column will be partially static text, but will also contain an array of brief news items. I'm going to omit all the *div* and other extraneous tags for the time being so that we can concentrate on the content.

```
<h2>In brief</h2>
<ul>
[% FOREACH point = brief %]
<li>[% point %]</li>
[% END %]
</ul>

<h2>Contact Details</h2>
[% INCLUDE contact %]
```

brief is going to be an array of pieces of news. Just like in Perl, we use FOREACH to iterate over that array; the template code is equivalent to this in Perl:

```
for my $point (@brief) {
```

This makes the local variable *point* contain the text for each news item.

The middle column is very similar, but slightly more complex. We'll have a number of more substantial news items, separated by horizontal lines. These will be passed in as an array of hash references, and we let Template do the work of sorting it all out. Here's what the template looks like for the news column.

```
<h2 align="center">News</h2>

[% FOREACH item = news %]
    <h3>[%item.title%]</h3>

    [% item.content %]
    <P ALIGN="right"><I>- [%item.when%]</I></P>

    <BR>
    <HR WIDTH="80%" ALIGN="center">
    <BR>
[% END %]
```

What this says is that it expects an array called "*news*," and will iterate over the array (that's the familiar FOREACH), putting each element in a temporary variable called *item*. *item* will itself be a hash reference, and we extract the elements called *title*, *content*, and *when* from it.

Template's *dot* operator is a little like Perl 5's *arrow* operator (and Perl 6's *dot* operator), minus the worry about brackets: It can be used to retrieve elements from hashes or arrays and also call methods on objects. Template Toolkit knows how to look at *item* and do the right thing with it—if we put an object inside our *news* array with *when*, *content*, and *title* methods, we'd get the same results.

This works well, but there's a little bit of a bug: We want the items separated by lines, but it looks slightly ugly to have a line right at the end after the last item. So we tell Template to output the *HR* tag unless we're on the last item:

```
[% '<HR WIDTH="80%" ALIGN="center">' UNLESS item == news.last %]
```

Template Toolkit provides special "virtual methods" on Perl values, which allow us to do clever things like this: Arrays have methods like *first* and *last*, which are equivalent to *.[0]* and *.[–1]* respectively. There are also methods that allow you to call Perl functions such as *split* or *join* on template variables.

This completes the template part of the newsletter—the complete template is given in Listing 1. (All code for this article is also available online at http://www.tpj.com/source/.)

Enter Blosxom

Now let's start thinking about how we want to get these values into our template. We will have our "in brief" news points stored in a file, one point per line, to make it very easy to read those into an array:

```
open BRIEF, "inbrief" or die $!;
my @brief = <BRIEF>;
close BRIEF;

$vars = {
    issue => $issue,
    brief => \@brief,
    date => $date
};

$template->process("newsletter.thtml", $vars, "newsletter-$is
                   sue.html");
```

This time, we've used a third argument to *process*, which tells Template Toolkit not to write to standard output, but to save the output to the named file.

What about the main news items? Well, this is where the story starts to get a bit more complicated. I want the news articles to also appear on my blog (http://blog.simon-cozens.org/) as I upload them. My blog uses a piece of software called *blosxom*, written by Rael Dornfest at O'Reilly. I like blosxom because it has a UNIX nature—I put my blog items as plain-text files in a directory and it sorts them all out. So a blog entry could be a file called "1234.txt" containing this:

```
Head Goes Here

<p> Here is the text of today's blog entry </p>
```

Blosxom looks at the first line of the file and uses that as the entry's heading. The rest of the file is HTML text that is added verbatim into the blog page which is being constructed. Blosxom also takes a look at the file's timestamp in the filesystem and uses that as the date of the entry. Note that the name of the file ("1234.txt") is arbitrary, and isn't used in building up the entry at all.

Now, because I wanted the news articles to appear on my blog, I thought it would be sensible to use blosxom format for the articles. That way, once they've been processed into the newsletter, they can be moved across to the blog data directory and be picked up there, too. So let's read in these files the same way blosxom does:

```
use File::stat;
use File::Copy;

my @news;
for my $file (<*txt>) {
    my $item = {};
    $item->{when} = localtime(stat($file)->mtime);
    $item->{when} =~ s/\d+:\d+:\d+ //;

    open IN, $file or die "$file: $!";
    $item->{title} = <IN>;

    local $/;
    $item->{content} = <IN>;
    close IN;
    push @news, $item;
    copy $file, "/opt/blog/$file";
}
```

We look for all the "txt" files in the current directory, and process each one of them. First, we look at the last-modified time of the file and convert that to a string. We remove the time, leaving only the date, and use that as the *when* element of our array. Now we can open up the file, read the first line as the *title*, and everything else goes into *content*. Once we've finished reading the file, we stick the item onto the array of news items and copy the entry over to the blog data directory for blosxom to pick it up.

Now we have all the data we need…or most of it at least.

Unpacking Archives

A further wrinkle comes from the fact that I only want to create one file offline and let my processing program do the right thing with it. This actually works to our advantage because we can produce a tar file that contains all the data and metadata we need in one directory.

We'll stipulate that the tar file comes in with a known filename and known format: Each issue should be contained in a file called "issueX.tar.gz" and this should contain a directory issueX/. We can now use *Archive::Tar* to extract the files:

```
use Archive::Tar;

my $filename = shift;
my $tar = Archive::Tar->new;
$tar->read($filename, 1);
$tar->extract;
```

And we can grab our issue number and the directory where we expect to find our files from the name of the file:

```
my $dir = $filename;
$dir =~ s/\.tar\.gz//;
$dir =~ /issue(-?\d+)$/
    or die "Directory name not in correct format";
my $issue = $1;
```

The *-?* is there because I wanted to produce "pre" issues of the newsletter, whimsically called "Issue –2" and "Issue –1." Because these preissues were monthly and the real issues will be weekly, I wanted to specify the date manually: "Issue –2" should have a date of "May," rather than "July 10–17" or whatever. So we read in the date of the newsletter from a file called date in our data directory:

```
open DATE, "$dir/date" or die "Can't open date file";
my $date = <DATE>;
```

So now we have all we need to produce the HTML version of the newsletter: a way to untar the input, find the issue number, look at the date, read the brief news items, and also read in the blosxom entries.

Uploading the HTML

We have an HTML file, but it's not much good just sitting on our filesystem. We need to get it out onto the Web. My personal web site is currently externally hosted, so I have to use FTP to transfer the files up to the site. No problem—Perl has the *Net::FTP* module to handle this for me:

```
use Net::FTP;
$ftp = Net::FTP->new("simon-cozens.org");
$ftp->login("simon",$password) or die $!.$@;

print "Creating HTML version...\n";
my $output = "newsletter$issue.html";
$template->process("newsletter.thtml", $vars, $output);

print "Uploading $output...\n";
$ftp->put($output, "public_html/newsletter/$output");
```

And I also upload it once again calling it "latest.html" so people can make sure they're reading the most recent version.

```
print "Uploading as latest.html...\n";
$ftp->put($output, "public_html/newsletter/latest.html");
```

A Final Flourish

So far we have the newsletter available as an HTML file on the Web, and also as entries on my weblog. But both of these are "pull" media—people have to keep checking the site to see if there's something new. Some people expressed a desire to have the news available as "push," where they are informed every time there's an update. The obvious way to do this is by e-mail. (Another way is via RSS, but that raises the bar a little—everyone knows how e-mail works.) And of course, I would rather die than knowingly send HTML e-mail.

Easy enough, I thought—I'll just knock up another template that will generate a plain-text e-mail and send that out to a mailing list I had set up. This template was very similar to the HTML one, but obviously, much simpler:

```
Issue [% issue %] - [% date -%]
______________

News
====

[% FOREACH item = news -%]
[%item.title%]

[%- item.content -%]

- [%item.when%]
[%- '—-' UNLESS item == news.last %]
[%- END %]

...
```

But when I processed this, I realized a slight problem. All of the news items are designed to be on the Web, in blosxom format—in HTML. I had to de-HTMLify these items before putting them through the processor. The *HTML::TreeBuilder* and *HTML::FormatText* modules came to my rescue here:

```
use HTML::TreeBuilder;
use HTML::FormatText;
for (@news) {
    my $text = $_->{content};
    $tree = HTML::TreeBuilder->new->parse($text) or die $!;
    $formatter = HTML::FormatText->new(leftmargin => 1, right
                                              margin => 75);
    $_->{content} =$formatter->format($tree);
}
```

This replaces each *content* with a plain-text equivalent, ready to be processed by our e-mail template.

Now it's a very simple matter of using *Mail::Mailer* to send out the processed e-mail:

```
$template->process("email.template", $vars, "email.txt");

use Mail::Mailer;
$mailer = new Mail::Mailer 'smtp', Server => "localhost";
$fh = $mailer->open({Subject => "Newsletter Issue $issue",
                          To     => 'wectrek2003@lists.netthink.co.uk');
print "Sending...\n";
open LET, "email.txt" or die $!;
print $fh <LET>;
$fh->close;
```

And we're done. 76 lines of Perl code and seven modules later, we have a system that allows me to take a file full of news and metadata, say

```
% process-newsletter issue3.tar.gz
```

and magically have a web site and weblog updated and a newsletter sent out via e-mail. The whole process-newsletter program can be found in Listing 2.

What It's All About

Larry is wise, and strong. But remember how his one regret was he didn't get to a Christian missionary? Guess what Ruby's creator used to be? A missionary in Hiroshima, Larry. In Hiroshima.

–Dave Green, NTK
http://www.ntk.net/index.cgi?b=02001-02-16&l=160#l

So far I've been very coy about what this newsletter is all about. Next month, I'm planning to even Larry's old score—I'll be going out on a short-term mission trip working with churches around the Shiga area of western Japan. In the field, I may not have excellent Internet connectivity, so I wanted something that would allow me to do as much of the work offline as possible; this is why I wanted only to have to deal with one file and have the processing system do the rest.

In the process of writing the newsletter system, we've seen examples of how to use Template Toolkit, how to unpack tarballs with *Archive::Tar*, how to upload files with *Net::FTP*, how to turn HTML into plain text, and how to send out mail, all from Perl. By putting in the time to create this admittedly complex processor, I'll now be able to spend less time creating the newsletter and more time creating news to go in it.

This is, I believe, exactly the kind of laziness Larry had in mind when he created Perl—laziness that requires a reasonable investment of time and effort up front, but then allows me to keep in touch with those back home, yet still have more time away from the computer, doing good things with good people.

You can keep up to date with my trip at http://simon-cozens.org/mission/latest.html, where you'll see the output of this very system.

TPJ

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

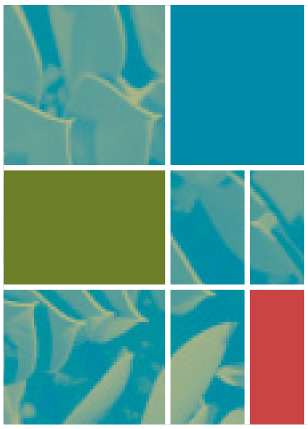
Listing 1

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />

<title>Newsletter</title>
<link rel=stylesheet type="text/css"
href="http://simon-cozens.org/mission/pl.css" title="myStyle">
</head>

<body>

<div class="box-wrap">

<div class="box-header">
<h1 align="center">Simon Cozens newsletter</h1>
<p align="right"><I>Issue [% issue %] - [% date %]</I></p>
</div>
<div class="columns-float">

<div class="column-one">
<div class="column-one-content">
<h2 align="center">News</h2>
[% FOREACH item = news %]
<h3>[%item.title%]</h3>
[% item.content %]
<P ALIGN="right"><I>- [%item.when%]</I></P>
<BR>
[% '<HR WIDTH="80%" ALIGN="center">' UNLESS item == news.last %]
<BR>
[% END %]
</div>
</div>

<div class="column-two">
<div class="column-two-content">
<h2>WEC Trek to Japan</h2>
[% INCLUDE trek %]
</div>
</div>

</div><!-- close columns-float -->

<div class="column-three">
<div class="column-three-content">
<h2>In Brief</h2>
<ul>
[% FOREACH point = brief %]
<li>[% point %]</li>
[% END %]
</ul>
[% INCLUDE phone %]
</div>
</div>

</div><!-- close box-wrap -->
</body>

</html>
```

Listing 2

```
use Template;
use File::stat;
use Net::FTP;
$ftp = Net::FTP->new("simon-cozens.org", Debug => 0);
      $ftp->login("simon","xxx") or die $!.$@;
use Archive::Tar;

my $filename = shift;
my $tar = Archive::Tar->new;
$tar->read($filename, 1);
$tar->extract;

my $dir = $filename;
$dir =~ s/\.tar\.gz//;
$dir =~ /issue(-?\d+)$/ or die "Directory name not in correct format";
my $issue = $1;
open DATE, "$dir/date" or die "Can't open date file";
open BRIEF, "$dir/brief"
    or die "Can't open brief news file $dir/brief: $!";
my @points = <BRIEF>;

# Now, the rest will be in blosxom format
my @news;
for my $file (<$dir/*txt>) {
    my $item = {};
    $item->{when} = localtime(stat($file)->mtime);
    $item->{when} =~ s/\d+:\d+:\d+ //;
    open IN, $file or die "$file: $!";
    $item->{title} = <IN>;
    local $/;
    $item->{content} = <IN>;
    close IN;
    push @news, $item;
}


my $template = Template->new();
my $vars = {
    issue => $issue,
    news => \@news,
    brief => \@points,
    date => <DATE>
};

print "Creating HTML version...\n";

my $output = "newsletter$issue.html";
$template->process("newsletter.thtml", $vars, $output);
print "Uploading $output...\n";
$ftp->put($output, "public_html/japan/$output");
print "Uploading as latest.html...\n";
$ftp->put($output, "public_html/japan/latest.html");

print "Processing email version...\n";
# Now process the email version, and send it to the list.
use HTML::TreeBuilder;
use HTML::FormatText;
for (@news) {
    my $text = $_->{content};
    $tree = HTML::TreeBuilder->new->parse($text) or die $!;
    $formatter = HTML::FormatText->new(leftmargin => 1, rightmargin => 75);
    $_->{content} =$formatter->format($tree);
}


$template->process("email.template", $vars, "email.txt");

use Mail::Mailer;
$mailer = new Mail::Mailer 'smtp', Server => "localhost";
$fh = $mailer->open({Subject => "Newsletter Issue $issue",
                         To      => 'wectrek2003@lists.netthink.co.uk');
print "Sending...\n";
open LET, "prayer-email.txt" or die $!;
print $fh <LET>;
$fh->close;
```

TPJ



HTML Filtering in Perl, Part 2

Randal Schwartz

Last month I presented an HTML stripper that tackled the problem of removing unwanted HTML from posts in web-based bulletin board systems. I outlined some of the risks in allowing arbitrary HTML in such posts. Left unfiltered, such HTML can be exploited to allow cross-site scripting and the triggering of browser bugs, which can lead to denial-of-service attacks or usurped credentials, or it can simply mangle the display of your pages.

The code for the stripper that I described last month is in Listing 1. This month, I'll explain the code for the *My_HTML_Filter* module, on which the HTML stripper depends. So how does *My_HTML_Filter* work? Let's go to Listing 2 for the details.

Line 3 pulls in the *XML::LibXML* module and line 4 creates a *$PARSER* object, using the default settings. This parser can be shared among many individual filters, so we've made it a class variable.

Lines 6 through 10 provide the constructor, which simply captures the *$permitted* parameter as an instance (member) variable, and returns the blessed object.

The real meat begins in line 12: the *strip* instance method. Lines 13 and 14 grab the instance variable and the input HTML, respectively.

Line 16 parses the HTML into a DOM. That's it. The result is an *XML::LibXML::Document* object from which we can get nice, clean HTML, or even XHTML if we choose. But we'll want to strip out the ugly stuff first. Line 17 caches the *$permitted* hashref into a simple scalar for quick access.

Line 19 establishes the "cursor" or "current node," conveniently and ambiguously called *$cur*. We'll do a walk of the DOM tree by moving this pointer around. We'll start by dropping down to the first child of the document node, and we'll end when the value hits *undef*, dropping out of the loop beginning in line 20.

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

Line 21 establishes a *$delete* flag. If this flag is true at the bottom of this loop, the current node must be deleted after we've computed the next node.

The comments in lines 23 and 24 reflect my feelings about how this is a bad design. Any design that requires you to ask an object for its type is generally a maintenance nightmare. Instead, a common protocol should have been established, where I'd merely have to query the properties and abilities of each thing within the tree using a Boolean-returning query method. But not here. So I grit my teeth and use *ref* a lot, hoping that some future version fixes all of this before it breaks all of this.

If it's an element, we'll note that in line 25, and then proceed to see whether the element type (the *nodeName* in line 27) is one of our permitted elements. If so, *$ok_attr* will then be a hashref of the permitted attributes, and we'll continue into line 28.

Lines 29 to 32 remove any attribute that is not permitted. Each attribute is queried for its *nodeName*, which is then checked against a list of permitted attributes, and removed if not permitted.

Line 34 moves our cursor down into the first child node, if it exists. For example, if we're looking at a *td* element, it will almost certainly have some content that we then have to scan. Some permitted elements (like *br*) won't have any content, so *$next* will be *undef*, and we'll use the normal "move forward" logic at the bottom of the loop.

That handles the permitted elements, but when we have a forbidden element, we need to remove it and reparent the orphaned children, using the code beginning in line 39.

Line 42 caches the parent node of the node to be deleted. Lines 43–45 move all of the node's children up to follow the node. This must be done in reverse order so that the order is retained following the current node, since we're always inserting immediately following the current node.

Finally, line 47 notes that the *$cur* node must be deleted after we've computed the following node at the bottom of the loop.

Lines 50 through 52 retain any existing text or *cdata* sections. The *cdata* section results when a *script* tag is used. Although our permitted list will probably cause the *script* element to be removed, the hoisted data is in a *cdata* element, and should be treated like text.

Lines 53 through 57 flag comments and DTDs as needing to be deleted. The former is dangerous (possibly hiding JavaScript). The latter is unnecessary, since we'll likely be including this text as part of a larger HTML page anyway.

Lines 58 to 60 attempt to flag anything else that I didn't see in my testing. I have no idea if I covered all of the nodes permitted in an HTML document, but I'm hoping I did.

Lines 62 to 73 compute the "next" node to be visited. I want the equivalent of the *XPath* expression *following::node()[1]*, without paying the price of parsing an *XPath* each time through the loop. This expression looks for the next node of any type, either at the same level or at any higher level. Child nodes are not considered.

Line 63 initially sets this "next" node to be the current node. Lines 65 to 68 determine if the next sibling node is available. If so, that's our selection, and we drop out of the "naked block" defined in lines 64 through 72.

If the node has no next sibling, then we need to pop up a level in the tree and look for that node's following node. Line 70 tries this, restarting the naked block if successful. If we're already at the top-level node, then the *undef* value is left in *$next*, which will end the outer loop started in line 20.

Lines 76 and 77 delete the current node if needed by requesting that the parent node forget about the current node. Line 79 advances the current node to the next node as the last step of this outer loop.

All that's left now is to spit out the modified DOM as HTML, in line 82, and then toss away the initial DTD in line 83. (In my tests, this was always the first line up to a newline, but this may change in future releases of the library, so this is a bit risky.)

Line 88 provides the mandatory true value for all files brought in with *require* or *use*.

And there you have it! A configurable HTML stripper that is fast and thorough. Now there's no excuse for letting someone start a comment tag or bold tag in your guestbook, messing up the rest of your display. Until next time, enjoy!

TPJ

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

Listing 1

```
=0=     ###### LISTING ONE (main program) ######
=1=     #!/usr/bin/perl
=2=     use strict;
=3=     $|++;
=4=
=5=     use My_HTML_Filter;
=6=
=7=     ## from http://www.perlmonks.org/index.pl?node_id=29281
=8=
=9=     my %PERMITTED =
=10=      map { my($k, @v) = split; ($k, {map {$_, 1} @v}) }
=11=      split /\n/, <<'END';
=12=    a href name target class title
=13=    b
=14=    big
=15=    blockquote class
=16=    br
=17=    center
=18=    dd
=19=    div class
=20=    dl
=21=    dt
=22=    em
=23=    font size color class
=24=    h1
=25=    h2
=26=    h3
=27=    h4
=28=    h5
=29=    h6
=30=    hr
=31=    i
=32=    li
=33=    ol type start
=34=    p align class
=35=    pre class
=36=    small
=37=    span class title
=38=    strike
=39=    strong
=40=    sub
=41=    sup
=42=    table width cellpadding cellspacing border bgcolor class
=43=    td width align valign colspan rowspan bgcolor height class
=44=    th colspan width align bgcolor height class
=45=    tr width align valign class
=46=    tt class
=47=    u
=48=    ul
=49=    END
=50=
=51=    use Test::More qw(no_plan);
=52=
=53=    my $f = My_HTML_Filter->new(\%PERMITTED) or die;
=54=    isa_ok($f, "My_HTML_Filter");
=55=
=56=    is($f->strip(qq{Hello}),
=57=       qq{<p>Hello</p>\n},
=58=       "basic text gets paragraphed");
=59=    is($f->strip(qq{<p><bogus>Thing}),
=60=       qq{<p>Thing</p>\n},
=61=       "bogons gets stripped");
=62=    is($f->strip(qq{<a href="foo">bar</a>}),
=63=       qq{<a href="foo">bar</a>\n},
=64=       "links are permitted");
=65=    is($f->strip(qq{<a href=foo>bar</a>}),
=66=       qq{<a href="foo">bar</a>\n},
=67=       "attributes get quoted");
=68=    is($f->strip(qq{<a href=foo bogus=place>bar</a>}),
=69=       qq{<a href="foo">bar</a>\n},
=70=       "bad attributes get stripped");
=71=    is($f->strip(qq{<p>What do <!-- comment -->you say?}),
=72=       qq{<p>What do you say?</p>\n},
=73=       "comments get stripped");
=74=    is($f->strip(qq{<table><tr><td>Hi!}),
=75=       qq{<table><tr><td>Hi!</td></tr></table>\n},
=76=       "tags get balanced");
=77=    is($f->strip(qq{<b><i>bold italic!}),
=78=       qq{<b><i>bold italic!</i></b>\n},
=79=       "b/i tags get balanced");
=80=    is($f->strip(qq{<b><i>bold italic!</b></i>}),
=81=       qq{<b><i>bold italic!</i></b>\n},
=82=       "b/i tags get nested properly");
=83=    is($f->strip(qq{<B><I>bold italic!</I></B>}),
=84=       qq{<b><i>bold italic!</i></b>\n},
=85=       "tags get lowercased");
=86=    is($f->strip(qq{<h1>hey</h1>one<br>two}),
=87=       qq{<h1>hey</h1>\n<p>one<br>two</p>\n},
=88=       "br comes out as HTML not XHTML");
=89=
=90=    use Benchmark;
=91=    my $homepage = do { open my $f, "homepage.html"; join "", <$f> };
=92=
=93=    timethese
=94=      (-1,
=95=       {
=96=        strip_homepage => sub { $f->strip($homepage) }
=97=       });
```

Listing 2

```
=0=     ###### LISTING TWO (My_HTML_Filter.pm) ######
=1=     package My_HTML_Filter;
=2=     use strict;
=3=     require XML::LibXML;
=4=     my $PARSER = XML::LibXML->new;
=5=
=6=     sub new {
=7=       my $class = shift;
=8=       my $permitted = shift;
=9=       return bless { permitted => $permitted }, $class;
=10=    }
=11=
=12=    sub strip {
=13=      my $self = shift;
```

```
  my $html = shift;

  my $dom = $PARSER->parse_html_string($html) or die "Cannot parse";
  my $permitted = $self->{permitted};

  my $cur = $dom->firstChild;
  while ($cur) {
    my $delete = 0;             # default to safe

    ## I really really hate switching on class names
    ## but this is a bad interface design {sigh}
    if (ref $cur eq "XML::LibXML::Element") {
      ## "that which is not explicitly permitted is forbidden!"
      if (my $ok_attr = $permitted->{$cur->nodeName}) {
       ## so this element is permitted, but what about its attributes?
        for my $att ($cur->attributes) {
          my $name = $att->nodeName;
          $cur->removeAttribute($name) unless $ok_attr->{$name};
        }
        ## now descend if any kids
        if (my $next = $cur->firstChild) {
          $cur = $next;
          next;                 # don't execute code at bottom
        }
      } else {
        ## bogon - delete!
        ## we must hoist any kids to be after our current position
in
       ## reverse order, since we always inserting right after old node
       my $parent = $cur->parentNode or die "Expecting parent of $cur";
        for (reverse $cur->childNodes) {
          $parent->insertAfter($_, $cur);
        }
        ## and flag this one for deletion
        $delete = 1;
        ## fall out
      }
    } elsif (ref $cur eq "XML::LibXML::Text"
             or ref $cur eq "XML::LibXML::CDATASection") {
      ## fall out
    } elsif (ref $cur eq "XML::LibXML::Dtd"
             or ref $cur eq "XML::LibXML::Comment") {
      ## delete these
      $delete = 1;
      ## fall out
    } else {
      warn "[what to do with a $cur?]"; # I hope we don't hit this
    }

    ## determine next node ala XPath "following::node()[1]"
    my $next = $cur;
    {
      if (my $sib = $next->nextSibling) {
        $next = $sib;
        last;
      }
      ## no sibling... must try parent node's sibling
      $next = $next->parentNode;
      redo if $next;
    }
    ## $next might be undef at this point, and we'll be done

    ## delete the current node if needed
    $cur->parentNode->removeChild($cur)
      if $delete;

    $cur = $next;
  }

  my $output_html = $dom->toStringHTML;
  $output_html =~ s/.*\n//;     # strip the doctype

  return $output_html;
}

1;
```

TPJ

Sign Up For BYTE.com!

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

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

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

ONLY $19.95 FOR ANNUAL ACCESS!

SPECIAL BONUS OFFER

SAVE 40%!!

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

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

CMP United Business Media

BYTE

Web Development with Apache and Perl

Jack J. Woehr

Web Development with Apache and Perl, by Theo Petersen, is a nice, tight book about classic CGI for interactive web sites. It's a pretty comprehensive overview of what we have been doing in Perl for almost a decade now to create and handle forms and dynamic content. This volume is eminently worthwhile for anyone who is not already a complete expert in the domain.

I've been enjoying this one and reading it cover to cover because I'm not a complete expert in the domain. I jumped into Java very early (1996) in the Web era and never looked back. While I have coded trivial CGI in the past, it's nice to revisit the topics covered by this book at the height of maturity of this programming niche and its code base. In addition, it's much easier to grasp the Perl-coded CGI conceptual framework than it was before module CGI and all the other fine CPAN stuff referenced in the work went online.

As is typical with Manning Publications, Petersen has his own page on the Manning web site (http://www.manning.com/petersen/) at which one can find reviews, source code, a sample chapter, and the author participating in a forum with his readers online.

The book is divided as follows:

Part 1: Web site basics
1. Open Source
2. The web servers
3. CGI scripts

Part 2: Tools for web applications
4. Databases
5. Better scripting
6. Security and users
7. Combining Perl and HTML

Part 3: Example sites
8. Virtual communities
9. Intranet applications
10. The web storefront

Part 4: Site management
11. Content management
12. Performance management

with many subsections to the sections shown above.

There is circumstantial evidence that this book ran late in its publishing cycle and was rushed a bit in the final stages. Despite the 2002 copyright date, the table of contents listing on the web site (apparently directly from the author's original outline) doesn't literally match the above, apparently due to last-minute changes, which left some inconsistencies in typographical styles. Aside from minor flaws of this sort, the book flows nicely and is well conceived, well designed, and well executed.

Petersen requires little of the reader aside from a familiarity with Perl, which he thankfully does not waste page space attempting to teach in this appropriately sized volume. He assumes the reader may need an introduction to open source and web servers. This will serve most readers well, but additionally places this book squarely on the "must-read" shelf for a multitude of OS/400 and z/OS programmers, whose platforms support Perl and Apache, but who may be less familiar with them.

Web Development with Apache and Perl
Theo Petersen
Manning, 2002
560 pp., $44.95
ISBN 1-930110-06-5

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/.*

When moving up from the study of component techniques to the study of complete interactive web sites, the book does more than plough through its own excellent and feature-filled examples. The analysis includes real-world sites such as slashdot.org and imdb.com. The web storefront example could serve as a template for creating your own online order taker. But perhaps closer to most readers' experience is the virtual communities chapter, which shows how to create online forums. You might rather download one of the complete forum packages found around the open-source world, but it's probably a good exercise to build one yourself.

The book doesn't include the word "Apache" in the title in vain. Much space is devoted to examining, explaining, and fine tuning the interaction of Apache and Perl. I haven't encountered a better introductory-to-intermediate discussion of this subject elsewhere. The performance-management chapter at the end of the book is quite juicy with insight.

Perhaps the least satisfactory part of the book, through no fault of the author, is the section on Perl access to databases. What is present is perfectly adequate for the author's task of bringing the reader along to the creation of complete CGI-driven sites, but this subject drips with complexity and begs for a complete book or bookshelf on its own.

If there's a unifying theme to the book, it's the modularity of Perl coding. It's impossible to become productive in lone-wolf mode in this modern world: Being adept at Perl software engineering is almost identical to being adept at using CPAN modules. Many readers may take for granted the object coordination of contemporary Perl practice, but I still marvel at the accomplishment of the community in building up such a complete repertory with a level of integration and teamwork idealized but almost unthinkable in the 1980s.

Theo Petersen is clearly an excellent Perl coder and an accomplished tech writer who has managed to sum up an entire programming discipline in a pleasingly concise fashion. If you are writing CGI-driven interactive web sites using Perl, and you're not sure whether you need this book, you probably do need it.

TPJ

Source Code Appendix

Ala Qumsieh "Fuzzy Logic in Perl"

Listing 1

```
#!perl -w

use strict;
use Tk;
use Tk::LabEntry;
use AI::FuzzyInference;

use constant PI => 3.1415927;
use constant G  => 9.81;

my $halfLenRod   = 5;
my $timeStep     = 0.05;

my $thRod;    # between -30 and 30 degrees.
my $velBall;  # ball's velocity. From -15 .. 15 m/s
my $posBall;  # ball's position. From -5 .. 5 m
my $time = 0;

# initialize.
$thRod   = -10;
$velBall = -2;
$posBall = 4;

# create the FIS.
my $fis = new AI::FuzzyInference;

# define the input variables.
$fis->inVar(posBall  => -5, 5,
        far_left  => [-4, 1, -2, 0],
        left      => [-4, 0, -2, 1, 0, 0],
        center    => [-2, 0, 0, 1, 2, 0],
        right     => [0, 0, 2, 1, 4, 0],
        far_right => [2, 0, 4, 1],
       );

$fis->inVar(velBall   => -15, 15,
        fast_neg   => [-9, 1, -3, 0],
        medium_neg => [-9, 0, -3, 1, 0, 0],
        slow       => [-3, 0, 0, 1, 3, 0],
        medium_pos => [0, 0, 3, 1, 9, 0],
        fast_pos   => [3, 0, 9, 1],
       );

# define the output variable.
$fis->outVar(thRod      => -30, 30,
         large_neg  => [-20, 1, -10, 0],
         medium_neg => [-20, 0, -10, 1, 0, 0],
         small      => [-10, 0, 0, 1, 10, 0],
         medium_pos => [0, 0, 10, 1, 20, 0],
         large_pos  => [10, 0, 20, 1],
       );

# now define the rules.
$fis->addRule(
          'posBall=far_left  & velBall=fast_neg'   => 'thRod=large_pos',
          'posBall=far_left  & velBall=medium_neg' => 'thRod=large_pos',
          'posBall=far_left  & velBall=slow'       => 'thRod=medium_pos',
          'posBall=far_left  & velBall=medium_pos' => 'thRod=medium_pos',
          'posBall=far_left  & velBall=fast_pos'   => 'thRod=medium_pos',

          'posBall=left      & velBall=fast_neg'   => 'thRod=large_pos',
          'posBall=left      & velBall=medium_neg' => 'thRod=medium_pos',
          'posBall=left      & velBall=slow'       => 'thRod=medium_pos',
          'posBall=left      & velBall=medium_pos' => 'thRod=medium_pos',
          'posBall=left      & velBall=fast_pos'   => 'thRod=medium_pos',

          'posBall=center    & velBall=fast_neg'   => 'thRod=large_pos',
          'posBall=center    & velBall=medium_neg' => 'thRod=medium_pos',
          'posBall=center    & velBall=slow'       => 'thRod=small',
          'posBall=center    & velBall=medium_pos' => 'thRod=medium_neg',
          'posBall=center    & velBall=fast_pos'   => 'thRod=large_neg',

          'posBall=right     & velBall=fast_neg'   => 'thRod=medium_neg',
          'posBall=right     & velBall=medium_neg' => 'thRod=medium_neg',
          'posBall=right     & velBall=slow'       => 'thRod=medium_neg',
          'posBall=right     & velBall=medium_pos' => 'thRod=medium_neg',
          'posBall=right     & velBall=fast_pos'   => 'thRod=large_neg',

          'posBall=far_right & velBall=fast_neg'   => 'thRod=medium_pos',
          'posBall=far_right & velBall=medium_neg' => 'thRod=medium_neg',
          'posBall=far_right & velBall=slow'       => 'thRod=medium_neg',
          'posBall=far_right & velBall=medium_pos' => 'thRod=large_neg',
          'posBall=far_right & velBall=fast_pos'   => 'thRod=large_neg',
         );

drawGUI();

MainLoop;

# this subroutine calculates the new values of the ball's position
# and velocity after a period of time $timeStep.
# Friction is not modeled.
sub calcNewData {
  my $acc  = G * sin ($thRod * PI / 180);
  my $Vnew = $velBall + $acc * $timeStep;
  my $dist = $velBall * $timeStep + 0.5 * $acc * $timeStep * $timeStep;

  $velBall  = $Vnew;
  $posBall += $dist;

  $velBall =  15 if $velBall >  15;
  $velBall = -15 if $velBall < -15;

  $time += $timeStep;
}

# This sub draws the gui.
sub drawGUI {
   my $mw = new MainWindow;

   my $canvas = $mw->Canvas(qw/-bg black -height 400 -width 600/)->pack;

   $canvas->createLine(0, 0, 0, 0,   qw/-width 2 -fill white -tags ROD/);
   $canvas->createOval(0, 0, 50, 50, qw/-fill green -tags BALL/);

   my $f = $mw->Frame->pack(qw/-fill x/);
   my $dth;

   $f->Button(-text    => 'run',
          -command => sub {
        my $id;
        $id = $canvas->repeat(100 => sub {
                    # update the ball's data.
                    calcNewData();

                    # check for termination conditions.

                    # stop if ball is almost stationary and the rod
                    # is almost flat.
                    if (abs($velBall) < 0.005 && abs($thRod) < 0.001) {
                      print "Simulation ended.\n";
                      $canvas->afterCancel($id);
                      return;
                    }

                    # stop if ball fell off the rod.
                    if ($posBall > $halfLenRod or $posBall < -$halfLenRod) {
                      print "Ball fell off the rod!\n";
                      $canvas->afterCancel($id);
                      return;
                    }

                    # compute the new angle of the rod.
                    $fis->compute(posBall => $posBall,
                              velBall => $velBall);
                    $thRod = $fis->value('thRod');

                    # update our drawing.
                    updateCanvas($canvas);
                      });
          })->pack(qw/side left -ipadx 10/);

   $f->LabEntry(-label => 'Ball Pos',
        -textvariable => \$posBall,
           )->pack(qw/-side left -padx 10/);

   $f->LabEntry(-label => 'Ball Speed',
        -textvariable => \$velBall,
           )->pack(qw/-side left -padx 10/);

   $f->LabEntry(-label => 'Rod Angle',
        -textvariable => \$thRod,
           )->pack(qw/-side left -padx 10/);

   updateCanvas($canvas);
}

# This subroutine draws the rod at its current angle, and
# the ball at its current position.
sub updateCanvas {
  my $c = shift;

  my $ly = 200;
  my $dy = int(40 * $halfLenRod * tan(PI * $thRod / 180));
  $c->coords(ROD => 100, $ly - $dy, 500, $ly + $dy);

  my $by = 150 + $posBall * $dy / $halfLenRod;

  my $bx = 100 + (5 + $posBall) * 40;
  $c->coords(BALL => $bx - 25, $by, $bx + 25, $by + 50);
}

# tangent sub.
sub tan { sin($_[0]) / cos($_[0])  }
```

Moshe Bar "Integrating OS X Aqua and Perl"

Listing 1

```
#include <Carbon/Carbon.h>
#include "CocoaBundle.h"
enum
{
    kOpenCocoaWindow = 'COCO'
};
CFBundleRef bundleRef = NULL;
static OSStatus
handleBundleCommand(int commandID) {
    OSStatus osStatus = noErr;
    if (commandID == kEventButtonPressed) {
        OSStatus (*funcPtr)(CFStringRef message);
                funcPtr = CFBundleGetFunctionPointerForName(bundleRef,
                                                    CFSTR("changeText"));
                require(funcPtr, CantFindFunction);
                osStatus = (*funcPtr)(CFSTR("button pressed!"));
                require_noerr(osStatus, CantCallFunction);
    }
CantFindFunction:
CantCallFunction:
        return osStatus;
}
static void
myLoadPrivateFrameworkBundle(CFStringRef framework, CFBundleRef *bundlePtr)
{
    CFBundleRef appBundle = NULL;
    CFURLRef baseURL = NULL;
    CFURLRef bundleURL = NULL;
        appBundle = CFBundleGetMainBundle();
        require(appBundle, CantFindMainBundle);
            baseURL = CFBundleCopyPrivateFrameworksURL(appBundle);
            require(baseURL, CantCopyURL);
                bundleURL = CFURLCreateCopyAppendingPathComponent(kCFAllocatorSystemDefault, baseURL,
                                      CFSTR("CocoaBundle.bundle"), false);
                require(bundleURL, CantCreateBundleURL);
                    bundleRef = CFBundleCreate(NULL, bundleURL);
                        CFRelease(bundleURL);
CantCreateBundleURL:
                            CFRelease(baseURL);
CantCopyURL:
CantFindMainBundle:
                            return;
}

static OSStatus
appCommandHandler(EventHandlerCallRef inCallRef, EventRef inEvent, void* userData) {
    HICommand command;
    OSStatus (*funcPtr)(void *);
    OSStatus (*showPtr)(void);
    OSStatus err = eventNotHandledErr;

    if (GetEventKind(inEvent) == kEventCommandProcess) {
        GetEventParameter( inEvent, kEventParamDirectObject, typeHICommand,
                           NULL, sizeof(HICommand), NULL, &command );
        switch ( command.commandID ) {
            case kOpenCocoaWindow:

                myLoadPrivateFrameworkBundle(CFSTR("CocoaBundle.bundle"),
                                                        &bundleRef);
                require(bundleRef, CantCreateBundle);
                                // call function to initialize bundle
                funcPtr = CFBundleGetFunctionPointerForName(bundleRef,
                                              CFSTR("initializeBundle"));
                require(funcPtr, CantFindFunction);
                err = (*funcPtr)(handleBundleCommand);
                require_noerr(err, CantInitializeBundle);
                                // call function to show window
                showPtr = CFBundleGetFunctionPointerForName(bundleRef,
                                              CFSTR("orderWindowFront"));
                require(showPtr, CantFindFunction);
                err = (*showPtr)();
                require_noerr(err, CantCallFunction);
                CantCreateBundle:
                CantCallFunction:
                CantInitializeBundle:
                CantFindFunction:
                break;
            default:
                break;
        }
    }
        return err;
}
int main(int argc, char* argv[])
{
    IBNibRef   nibRef;
    WindowRef   window;
        OSStatus  err;
            EventTypeSpec cmdEvent = {kEventClassCommand, kEventCommandProcess};
                // Create a Nib reference passing the name of the nib file
                // (without the .nib extension)
                // CreateNibReference only searches into the application bundle.
            err = CreateNibReference(CFSTR("main"), &nibRef);
            require_noerr( err, CantGetNibRef );
                // Once the nib reference is created, set the menu bar.
                // "MainMenu" is the name of the menu bar
                // object. This name is set in InterfaceBuilder when the nib is
                // created.
            err = SetMenuBarFromNib(nibRef, CFSTR("MenuBar"));
            require_noerr( err, CantSetMenuBar );
                // Then create a window. "MainWindow" is the name of the window
                // object. This name is set in
                // InterfaceBuilder when the nib is created.
            err = CreateWindowFromNib(nibRef, CFSTR("MainWindow"), &window);
            require_noerr( err, CantCreateWindow );
                // We don't need the nib reference anymore.
            DisposeNibReference(nibRef);
                // The window was created hidden so show it.
            ShowWindow( window );
                InstallApplicationEventHandler(NewEventHandlerUPP(appCommandHandler), 1,
                               &cmdEvent, 0, NULL);
                    // Call the event loop
                RunApplicationEventLoop();
                CantCreateWindow:
CantSetMenuBar:
CantGetNibRef:
                    return err;
}
```

Listing 2

```
package WindowController;

use Foundation;
use Foundation::Functions;
use AppKit;
use AppKit::Functions;

@ISA = qw(Exporter);

sub new {
    # Typical Perl constructor
    # See 'perltoot' for details
    my $proto = shift;
    my $class = ref($proto) || $proto;

    my $self = {};

    # Outlets
    $self->{'RateField'} = undef;
    $self->{'DollarField'} = undef;
    $self->{'TotalField'} = undef;
    $self->{'Window'} = undef;

    bless ($self, $class);

    $self->{'NSWindowController'} =
                    NSWindowController->alloc->initWithWindowNibName_owner(
                                                "ConverterWindow", $self);
    $self->{'NSWindowController'}->window;

    return $self;
}

sub convertButtonClicked {
    my $self = shift;

    my $rate = $self->{'RateField'}->floatValue;
    my $amount = $self->{'DollarField'}->floatValue;
    my $total = $rate * $amount;
    $self->{'TotalField'}->setFloatValue($total);

    return 1;
}

1;
```

Listing 3

```
package MyController;

use Foundation;
use Foundation::Functions;
use AppKit;
use AppKit::Functions;

use WindowController;

@ISA = qw(Exporter);

our $wc = undef;

sub new {
    # Typical Perl constructor
    # See 'perltoot' for details
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = {
    };
    bless ($self, $class);

    return $self;
}

sub applicationWillFinishLaunching {
    my ($self, $notification) = @_;

    # Create the new controller object
    $wc = new WindowController;

    return 1;
}

1;
```

Simon Cozens "Managing Newsletters with Perl"

Listing 1

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />

<title>Newsletter</title>
<link rel=stylesheet type="text/css"
href="http://simon-cozens.org/mission/pl.css" title="myStyle">
</head>

<body>

<div class="box-wrap">

<div class="box-header">
<h1 align="center">Simon Cozens newsletter</h1>
<p align="right"><I>Issue [% issue %] - [% date %]</I></p>
</div>
<div class="columns-float">

<div class="column-one">
<div class="column-one-content">
<h2 align="center">News</h2>
[% FOREACH item = news %]
<h3>[%item.title%]</h3>
[% item.content %]
<P ALIGN="right"><I>- [%item.when%]</I></P>
<BR>
[% '<HR WIDTH="80%" ALIGN="center">' UNLESS item == news.last %]
<BR>
[% END %]
</div>
</div>

<div class="column-two">
<div class="column-two-content">
<h2>WEC Trek to Japan</h2>
[% INCLUDE trek %]
</div>
</div>
```

```
</div><!-- close columns-float -->

<div class="column-three">
<div class="column-three-content">
<h2>In Brief</h2>
<ul>
[% FOREACH point = brief %]
<li>[% point %]</li>
[% END %]
</ul>
[% INCLUDE phone %]
</div>
</div>

</div><!-- close box-wrap -->
</body>

</html>
```

Listing 2

```
use Template;
use File::stat;
use Net::FTP;
$ftp = Net::FTP->new("simon-cozens.org", Debug => 0);
      $ftp->login("simon","xxx") or die $!.$@;

use Archive::Tar;

my $filename = shift;
my $tar = Archive::Tar->new;
$tar->read($filename, 1);
$tar->extract;

my $dir = $filename;
$dir =~ s/\.tar\.gz//;
$dir =~ /issue(-?\d+)$/ or die "Directory name not in correct format";
my $issue = $1;
open DATE, "$dir/date" or die "Can't open date file";
open BRIEF, "$dir/brief"
    or die "Can't open brief news file $dir/brief: $!";
my @points = <BRIEF>;

# Now, the rest will be in blosxom format
my @news;
for my $file (<$dir/*txt>) {
    my $item = {};
    $item->{when} = localtime(stat($file)->mtime);
    $item->{when} =~ s/\d+:\d+:\d+ //;
    open IN, $file or die "$file: $!";
    $item->{title} = <IN>;
    local $/;
    $item->{content} = <IN>;
    close IN;
    push @news, $item;
}


my $template = Template->new();
my $vars = {
    issue => $issue,
    news => \@news,
    brief => \@points,
    date => <DATE>
};

print "Creating HTML version...\n";

my $output = "newsletter$issue.html";
$template->process("newsletter.thtml", $vars, $output);
print "Uploading $output...\n";
$ftp->put($output, "public_html/japan/$output");
print "Uploading as latest.html...\n";
$ftp->put($output, "public_html/japan/latest.html");

print "Processing email version...\n";
# Now process the email version, and send it to the list.
use HTML::TreeBuilder;
use HTML::FormatText;
for (@news) {
    my $text = $_->{content};
    $tree = HTML::TreeBuilder->new->parse($text) or die $!;
    $formatter = HTML::FormatText->new(leftmargin => 1, rightmargin => 75);
    $_->{content} =$formatter->format($tree);
}


$template->process("email.template", $vars, "email.txt");
```

```
use Mail::Mailer;
$mailer = new Mail::Mailer 'smtp', Server => "localhost";
$fh = $mailer->open({Subject => "Newsletter Issue $issue",
                           To      => 'wectrek2003@lists.netthink.co.uk');
print "Sending...\n";
open LET, "prayer-email.txt" or die $!;
print $fh <LET>;
$fh->close;
```

Randal Schwartz “HTML Filtering in Perl, Part 2”

Listing 1

```
###### LISTING ONE (main program) ######
#!/usr/bin/perl
use strict;
$|++;

use My_HTML_Filter;

## from http://www.perlmonks.org/index.pl?node_id=29281

my %PERMITTED =
  map { my($k, @v) = split; ($k, {map {$_, 1} @v}) }
  split /\n/, <<'END';
a href name target class title
b
big
blockquote class
br
center
dd
div class
dl
dt
em
font size color class
h1
h2
h3
h4
h5
h6
hr
i
li
ol type start
p align class
pre class
small
span class title
strike
strong
sub
sup
table width cellpadding cellspacing border bgcolor class
td width align valign colspan rowspan bgcolor height class
th colspan width align bgcolor height class
tr width align valign class
tt class
u
ul
END

use Test::More qw(no_plan);

my $f = My_HTML_Filter->new(\%PERMITTED) or die;
isa_ok($f, "My_HTML_Filter");

is($f->strip(qq{Hello}),
   qq{<p>Hello</p>\n},
   "basic text gets paragraphed");
is($f->strip(qq{<p><bogus>Thing}),
   qq{<p>Thing</p>\n},
   "bogons gets stripped");
is($f->strip(qq{<a href="foo">bar</a>}),
   qq{<a href="foo">bar</a>\n},
   "links are permitted");
is($f->strip(qq{<a href=foo>bar</a>}),
   qq{<a href="foo">bar</a>\n},
   "attributes get quoted");
is($f->strip(qq{<a href=foo bogus=place>bar</a>}),
   qq{<a href="foo">bar</a>\n},
   "bad attributes get stripped");
is($f->strip(qq{<p>What do <!-- comment -->you say?}),
   qq{<p>What do you say?</p>\n},
   "comments get stripped");
is($f->strip(qq{<table><tr><td>Hi!}),
   qq{<table><tr><td>Hi!</td></tr></table>\n},
    "tags get balanced");
is($f->strip(qq{<b><i>bold italic!}),
    qq{<b><i>bold italic!</i></b>\n},
    "b/i tags get balanced");
is($f->strip(qq{<b><i>bold italic!</b></i>}),
    qq{<b><i>bold italic!</i></b>\n},
    "b/i tags get nested properly");
is($f->strip(qq{<B><I>bold italic!</I></B>}),
    qq{<b><i>bold italic!</i></b>\n},
    "tags get lowercased");
is($f->strip(qq{<h1>hey</h1>one<br>two}),
    qq{<h1>hey</h1>\n<p>one<br>two</p>\n},
    "br comes out as HTML not XHTML");

use Benchmark;
my $homepage = do { open my $f, "homepage.html"; join "", <$f> };

timethese
  (-1,
   {
    strip_homepage => sub { $f->strip($homepage) }
   });
```

Listing 2

```
###### LISTING TWO (My_HTML_Filter.pm) ######
package My_HTML_Filter;
use strict;
require XML::LibXML;
my $PARSER = XML::LibXML->new;

sub new {
  my $class = shift;
  my $permitted = shift;
  return bless { permitted => $permitted }, $class;
}

sub strip {
  my $self = shift;
  my $html = shift;

  my $dom = $PARSER->parse_html_string($html) or die "Cannot parse";
  my $permitted = $self->{permitted};

  my $cur = $dom->firstChild;
  while ($cur) {
    my $delete = 0;             # default to safe

    ## I really really hate switching on class names
    ## but this is a bad interface design {sigh}
    if (ref $cur eq "XML::LibXML::Element") {
      ## "that which is not explicitly permitted is forbidden!"
      if (my $ok_attr = $permitted->{$cur->nodeName}) {
       ## so this element is permitted, but what about its attributes?
        for my $att ($cur->attributes) {
          my $name = $att->nodeName;
          $cur->removeAttribute($name) unless $ok_attr->{$name};
        }
        ## now descend if any kids
        if (my $next = $cur->firstChild) {
          $cur = $next;
          next;                 # don't execute code at bottom
        }
      } else {
        ## bogon - delete!
        ## we must hoist any kids to be after our current position in
       ## reverse order, since we always inserting right after old node
       my $parent = $cur->parentNode or die "Expecting parent of $cur";
        for (reverse $cur->childNodes) {
          $parent->insertAfter($_, $cur);
        }
        ## and flag this one for deletion
        $delete = 1;
        ## fall out
      }
    } elsif (ref $cur eq "XML::LibXML::Text"
             or ref $cur eq "XML::LibXML::CDATASection") {
      ## fall out
    } elsif (ref $cur eq "XML::LibXML::Dtd"
             or ref $cur eq "XML::LibXML::Comment") {
      ## delete these
      $delete = 1;
      ## fall out
    } else {
      warn "[what to do with a $cur?]"; # I hope we don't hit this
    }

    ## determine next node ala XPath "following::node()[1]"
    my $next = $cur;
    {
      if (my $sib = $next->nextSibling) {
        $next = $sib;
        last;
      }
      ## no sibling... must try parent node's sibling
      $next = $next->parentNode;
      redo if $next;
    }
    ## $next might be undef at this point, and we'll be done

    ## delete the current node if needed
    $cur->parentNode->removeChild($cur)
      if $delete;

    $cur = $next;
  }

  my $output_html = $dom->toStringHTML;
  $output_html =~ s/.*\n//;     # strip the doctype

  return $output_html;
}

1;
```