

The Perl Review

Volume 0 Issue 7

January 1, 2003

Like this issue? Support *The Perl Review* with a donation! <http://www.ThePerlReview.com/>

Community News	i
Short Notes	ii
Jotto: The Five-Letter Word Game	1
<i>Kevin Jackson-Mead</i>	
Processing RSS Files with XSL-T	12
<i>Dr. A J Trickett</i>	
Separating code, presentation, and configuration	18
<i>brian d foy</i>	
Paying Homage to Perl (PHP)	25
<i>Ed Summers</i>	

Web Access <http://www.ThePerlReview.com/> **Email** letters@theperlreview.com **Publisher** brian d foy **Editor** Andy Lester **Technical Editors** Kurt Starsinic, Adam Turoff **Copy Editors** Jim Brandt **Contributors** brian d foy, Andy Lester, Kevin Jackson-Mead, Ed Summers, A J Trickett **Copyright** Format Copyright © 2003 *The Perl Review*, article content Copyright © 2003 by their respective authors.

Community News

Send us your news stories at news@theperlreview.com.

Keep up with Perl news at <http://use.perl.org> (not a part of *The Perl Review*)

Perl turns 15

<http://history.perl.org/PerlTimeline.html>

Perl turned 15 last month. Larry Wall released Perl 1.0 to alt.sources in 1987. You can get this version of Perl at <http://history.perl.org/src/perl-1.0.tar.gz>.

YAPC::Israel 2003

<http://www.perl.org.il/YAPC/2003/>

The first YAPC::Israel will be a one-day event on May 12 at the Weizmann Institute of Science in Rehovot. This YAPC is organized by the Rehovot.pm and the Israel.pm Perl Users Groups and sponsored by Perl Training Israel, the Weizmann Institute, and Apress. The Call For Papers ends January 5.

Parrot 0.0.9

<http://www.cpan.org/authors/id/S/SF/SFINK/parrot-0.0.9.tar.gz>

Parrot version 0.0.9, code named “Nazgul”, is available from CPAN. The Parrot teams can still use volunteers for documentation, support tools, platform-dependent bits, testing and coding.

ActivePerl 5.8.0

<http://www.activestate.com/activeperl/>

ActiveState released a new ActivePerl for perl 5.8.0. It may take some time for modules to update their ActivePerl packages for the new version. Randy Kobes has updated packages for some of the popular modules: <http://theoryx5.uwinnipeg.ca/>

MacPerl 5.6.1r2

<http://dev.macperl.org/>

MacPerl 5.6.1r2 is now available from CPAN in the ports/mac directory. This version has initial Mac OS X support and minor bug fixes, including all of the 5.6.1 maintenance patches. Keep up with MacPerl development at <http://macperl.sourceforge.net/>.

OSCon 2003

http://conferences.oreillynet.com/cs/os2003/create/e_sess

This year the O’Reilly Open Source Convention is in Portland, Oregon, home of Randal Schwartz, Tom Phoenix, Allison Randal, and many other Perl people. The conference is July 7-11. The Call For Participation is open until February 15.

New books

Computer Science & Perl Programming: Best of The Perl Journal

*Edited by Jon Orwant
O’Reilly & Associates;
0-596-00310-2; November 2002*

Embedding Perl in HTML with Mason

*Dave Rolsky & Ken Williams
O’Reilly & Associates;
0-596-00225-4; November 2002*

LDAP Programming

*Clayton Donley; Manning;
1-930110-82-0; December 2002*

Perl Graphics Programming

*Shawn Wallace
O’Reilly & Associates;
0-596-00219-X; December 2002*

Programming Web Services with Perl

*Randy J. Ray & Pavel Kulchenko
O’Reilly & Associates;
0-596-00335-8; December 2002*

Perl In Magazines

Know of another magazine with regular Perl content? Let us know at letters@theperlreview.com

The Perl Journal – <http://www.tpj.com>

Linux Magazine – <http://www.linux-mag.com/>

;login: – <http://www.usenix.org/publications/login/>

SysAdmin – <http://www.samag.com/>

Unix Review – <http://www.unixreview.com/>

Short Notes

If you have something that you want to show off without writing an entire article, like a cool Perl trick, a module you just released or something happening in the Perl community, send your short note, between 200 and 400 words, to short_notes@theperlreview.com.

PHP::Include

Ed Summers

While writing the article on PHP that appears in this issue, Kris Boulez wrote to the London.pm list about parsing simple PHP from Perl. That got me to thinking about a module that would allow me to include PHP files from Perl. This led to PHP::Include which is now available on CPAN.

At the moment it only allows me to include very simple PHP, but it could be useful for Perl applications that need to share configuration data with PHP. The real work is done by Filter::Simple and Parse::RecDescent. I designed it so that the more adventurous or knowledgeable could extend the grammar to parse more of PHP.

With a simple Perl function:

```
use PHP::Include;
include_php_vars( 'file.php' );
```

I take a file with PHP variables:

```
<?php
$robot = 'Book Agent';
$hosts = Array(
    'www.amazon.com' => 'Amazon',
    'www.bookpool.com' => 'BookPool' );
$times = Array( 10,12,14,16,18 );
?>
```

and turn it into the equivalent Perl source:

```
my $robot = 'Book Agent';
my %hosts = (
    'www.amazon.com' => 'Amazon',
    'www.bookpool.com' => 'BookPool' );
my @times = ( 10,12,14,16,18 );
```

Embedding Perl in HTML with Mason

reviewed by Andy Lester, andy@petdance.com

I am a sucker for practical instruction. Theory is fun for a while, but even in my geekiest moments, I am usually trying to get something accomplished. I have read more than enough chapters on inheritance that use shapes (“squares have an is-a relationship with polygons”), or trivial web counter scripts that do not even address file locking. *Embedding Perl In HTML With Mason* is one of the most practical books I have read in a while.

Dave Rolsky and Ken Williams are two of the main developers of Mason, a Perl-based web templating system that powers many web applications, including RT, Bricolage and Alzabo, and sites like Stamps.com and Salon. The book reflects the years of real-world applications that Mason has been used for, and is aimed at the Perl programmer with sites to create.

After a quick overview of Mason and its alternatives, such as Embperl and HTML::Template, Rolsky and Williams assume that you know Perl already and jump straight into Mason syntax. From the beginning, there is an emphasis on Mason’s component structure and reusability across sites.

Two chapters are standout examples—“Building A Mason Site” is a 50 page discussion of building the apprentice.perl.org web site. It explains design decisions and overview of the site, and gives annotated code listings of selected components. The “Recipes” chapter is 30 page catch-all of various tools to perform common tasks like authentication and database connection management.

Should you buy the book since masonhq.com has extensive documentation? If you are new to Mason, it is well worth the money to have a guided introduction. For the experienced Mason developer, there is still an advantage to seeing Mason code written the way that the authors intended, or learning a new idiom. The book uses a Mason version 1.10, so developers who cut their teeth on the older versions will get something new, too. The book collects and organizes a great deal of the conventional wisdom learned over the past three years of Mason development.

Embedding Perl in HTML with Mason, by Dave Rolsky & Ken Williams, O’Reilly & Associates, 0-596-00225-4, <http://www.masonbook.com>

Jotto: The Five-Letter Word Game

Kevin Jackson-Mead, kevin@aq.org

Abstract

My obsession with a word game got me started in learning to program Perl. One project led to the next, and I learned more and more about Perl with each project.

I have always been obsessed with games. I was weaned on card games like Euchre, Hearts, and Spades. I was the one begging my family to play Monopoly as a kid. I demo games for Looney Labs. I have hosted a Boggle tournament. But the game I have been most obsessed with is Jotto. Jotto got me collecting five-letter words in high school. I played Jotto over a 2400 baud modem. I made my brother and sister play Jotto with me to stay awake on long car trips. And, most recently, Jotto pulled me into Perl programming.

1 The Game

Jotto is a two-player game where each player attempts to guess the other's secret five-letter word. A player scores a guess based on the number of letters it has in common with the secret word. This is a sample game:

```
trios 1
false 3
slang 2
swell 2
passe 3
abase 2
pleat 4
paler 5
pearl correct
```

The sample game has the property that each guess could possibly be the secret word based on the previous guesses and scores. For example, 'swell' scores 1 with trios, 3 with 'false', and 2 with 'slang'. Based on the information from the first three guesses, 'swell' could have been the secret word.

When I first learned Jotto, my guesses usually just changed one letter of the previous guess in order to get information about specific letters. So, with a first guess of 'trios' scoring 1, I might have made my second guess 'trips'. If 'trips' scored 2, I would know that the secret word had a 'p' but no 'o'. If 'trips' scored 0, I would know the secret word had an 'o' but no 'p'. If 'trips' scored 1, I would know that either the secret word had neither 'p' nor 'o' or both 'p' and 'o'. I first discovered the former method as a variant of Jotto that would make coming up with guesses more challenging. I soon learned that this method finds secret words in fewer guesses, and I adopted it as my main method for playing Jotto.

2 A first program

Jotto is a game that is easily played over email. A little more than a year ago, I started playing Jotto over email with people. At the height, I had games going on with at least six people. Obsession kicked in. I needed more Jotto. I decided to write a computer opponent using Perl. I started working through an introductory book on Perl, but then I just dived right in. I had programmed before, just not in Perl. I figured that I would learn best by working on something I was interested in instead of exercises in a book.

When I played this in high school, I started making a list in my notebook of as many five-letter words I could think of. I'm pretty sure I had delusions of making a computer program to play Jotto. I would just sit down and write five-letter words in my list. Then I would go around and ask people to give me a five-letter word. I was pretty good about remembering which ones I already had. I went through the list after I had collected a bunch, and I only found a few duplicate words. My list was somewhere around 1300 words.

For the computer game, I started with a list from a friend that was a pared-down version of `/usr/dict/words`, gone through manually with words taken out that he thought were too uncommon or jargony. This list is 3252 words. It is currently what I use for the easy dictionary in my Perl Jotto game. I then added in more words from various sources, and I've got a dictionary that is 5886 words, which I use for my hard dictionary in my Perl Jotto game. I found a comprehensive Scrabble word list, and I'm going to start using that at some point. I want to hand-check to make sure the words I have that it doesn't are actually words, though, since I know that some of the words I have aren't good (like `ascii`). The Scrabble word list is 12024 words.

Sometime later, I have integrated the Scrabble word list. My `words.5`¹ is 12386 words, and my `words.5.easy`² is 4014 words. The easy list represents some of my own work, but it is also based on the work of others.

The first script I wrote to pick a secret word and score my guesses. In doing that, I learned how arrays work, and I made good use of `split`, `sort`, and `shift`.

I wanted to share my creation with the world. After pondering several options, I decided to make it available via `ssh`. I told a few people about it and received some feedback that I used to improve the program—I made the output easier to read, and I added in an option to give up and reveal the secret word.

This was not enough, since I had no way to access `ssh` from work. So I decided to write an email interface for it. I learned enough about `procmail` to figure out how to have it redirect email to a program, more about regular expressions so I could parse incoming email, about reading and writing files to save and load the game state, and how to send mail from within a Perl program.

2.1 The next step

The next step was to write a program that could guess my secret word. The easiest method to implement was just a brute-force method. It starts by picking a random word from its dictionary. When I tell it the score for that guess, it scores the word it picked against every other word in its dictionary and eliminates those words that get a different score. It then randomly picks a word from the pared-down list. Eventually, it either picks the correct word or determines that the secret word is not in its dictionary.

I implemented this method, but I was immediately curious about whether a better algorithm existed. A friend of mine had suggested an algorithm that might be better, so I attempted to implement that. I made one test script and two different functions for picking a word. I learned how to pass references to lists and

¹<http://jmac.org/~kevin/words.5>

²<http://jmac.org/~kevin/words.5.easy>

hashes in order to send the current dictionary and the current state of the game to the functions. I separated the scoring function into a file by itself so that I could share it between programs—I did not really know about writing my own modules at this point.

I ran 1000 games with each method. I wrote a program to calculate the average score and the standard deviation of the scores. I learned how to use `$—` so that I could get real-time results instead of having to wait until the programs finished running. In the end, it turned out that the smart function performed worse than the dumb function. After some analysis, I realized that I had misinterpreted my friend’s algorithm. I was ready to move onto other Jotto programming, though, so the algorithm is still in its dumb state.

2.2 Enter the module

I had my Perl mentor look at some Jotto code to see if he could find any way to speed it up. One of the things he told me was, “You really should put this into a module.” So I learned how to write my own module, and I put the Jotto scoring function and a function to verify legality of a Jotto guess into `jottolib.pl`, in code listing 3. Having the module, I was able to write some other programs very easily.

One of these was a program to come up with Jotto puzzles. A Jotto puzzle is a series of Jotto guesses and scores with a couple of conditions. First, there can only be one secret word that is possible based on the guesses and scores. Second, no guess can have a score greater than 2. The second condition is there to make the puzzles more challenging. This is a jotto puzzle:

```
barry 0
doted 1
pomps 1
lives 1
duffs 2
could 2
quoif 2
```

3 What next?

I have a few more Jotto ideas to pursue, but I will be ready to start on some other games, soon. I want to implement Reiner Knizia’s *Battle Line*. I want to implement a text version of Kory Heath’s *Zendo*. I look forward to learning as much from those projects as I did from working on Jotto.

4 References

Jotto rules - <http://www.centralconnector.com/GAMES/jotto.html>

Jotto on the Web - <http://www.aq.org/~kevin/jotto>

Jotto by ssh - ssh `jotto@jmac.org`, password: `playjotto`

Jotto by email - send a blank message to `kevin@jmac.org`

Looney Labs - <http://www.wunderland.com>

Zendo - <http://www.wunderland.com/WTS/Kory/Games/Zendo>

Battle Line - http://www.gmtgames.com/nabl/battleline_main.html

5 Acknowledgements

I would have had nothing to write an article about if it were not for many people helping me feed my Jotto obsession. I acknowledge the efforts of two people in particular. Jason McIntosh provided server space for my Jotto programs, looked at my messy code and made suggestions, and served as my Perl mentor. Matt Ryan provided much inspiration by playing many games of Jotto with me and by sharing his ideas for an improved Jotto algorithm.

6 About the Author

Kevin Jackson-Mead has edited mathematics textbooks professionally and has done volunteer editing for New Genre (<http://www.ngenre.com>) and for a role playing game system (<http://www.100percentgaming.com>), but what he really wants to do is get on the professional Boggle tournament circuit.

Code Listing 1: playjotto.pl

```
1  #!/usr/bin/perl -Tw
2
3  # Version 0 - initial version playable from the jotto account
4  # Version 0.1 - added info
5  #           added quit
6  #           indented score to line up with word
7  # Version 0.2 - added choice of dictionary
8  # Version 0.3 - added status
9  # Version 0.4 - moved score sub to lib file
10
11  require "./jottolib.pl";
12
13  srand;
14
15  print <<"HERE";
16  Kevin's Jotto, Version 0.4
17
18  Type \"info\" for the rules of the game.
19
20  Type \"status\" to see your guesses and their scores.
21
22  You may type \"quit\" at any time to give up and reveal the word.
23  HERE
24
25  $i = 0;
26  $si = 0;
27  $again = "y";
28
```

```
29 $words = "jottowords";
30 $dict = "jottowords.easy";
31
32 print <<"HERE";
33 What kind of game would you like?
34 1: Easy (sane dictionary)
35 2: Hard (insane, full dictionary, with words like rubra)
36 (Just hit enter for an easy game.)
37 HERE
38
39 $type = <STDIN>;
40 chop($type);
41
42 if ( $type && $type == 2 ) {
43     $dict = $words;
44 }
45
46 # open the dictionary for checking valid words
47 open(WORDS, $words) or die "Could not open $words: $!";
48 while ($word = <WORDS>) {
49     chop($word);
50     $words[$i++] = $word;
51 }
52 close(WORDS);
53
54
55 # open the dictionary to pick the secret word from
56 open(SWORDS, $dict);
57 while ($sword = <SWORDS>) {
58     chop($sword);
59     $swords[$si++] = $sword;
60 }
61 close(SWORDS);
62
63 while ($again eq "y") {
64
65     $secretword = $swords[rand(@swords)];
66     $tries = 1;
67     $jots = 0;
68     $ti = 0; # index for the array that stores the guesses
69     @twords = ();
70
71     print "I have a secret 5-letter word.\n";
72
73     while ($jots != 6) {
74         print "Guess: ";
75         $guess = <STDIN>;
76         chop($guess);
77
78         # let the person give up
79         if ($guess eq 'quit') {
80             print "Giving up is okay. This is a hard game.\n";
81             print "The word was $secretword.\n";
82             last;
83         }

```

```

84         elsif ($guess eq 'info') {
85             print <<"HERE";
86             I have a 5-letter word. It is pulled from
87             my dictionary of many thousands of words.
88             Your guess must be a 5-letter word that is
89             in my dictionary. If your guess is not my
90             secret word, then I will give you a number
91             from 0 to 5, indicating the number of letters
92             your word has in common with my word. Letters
93             are counted on a 1-to-1 basis, so, a double
94             letter in my word matching up with only one
95             of those letters in your word would count for
96             1. A double letter in your word mathching up
97             with only one of those letters in my word would
98             count for 1. If both words had that double
99             letter, it would count for two.
100            Examples:
101            My word: llama Your guess: eagle Score: 2
102            My word: blast Your guess: bully Score: 2
103            My word: extra Your guess: grate Score: 4
104            HERE
105
106                next;
107            }
108            elsif ($guess eq 'status') {
109                if ($#twords == -1) {
110                    print "You haven't made any guesses yet.\n";
111                }
112                else {
113                    foreach $tword (@twords) {
114                        print $tword, "\n";
115                    }
116                }
117
118                next;
119            }
120            elsif ($guess =~ /^[^a-z]/ || length($guess) != 5) {
121                print "$guess not a legal guess\n";
122            }
123            else {
124                $i = 0;
125                while ($words[$i] && ($guess gt $words[$i])) {
126                    $i++;
127                }
128                if ($guess ne $words[$i]) {
129                    print "$guess is not a valid word\n";
130                }
131                elsif (($jots = score($guess, $secretword)) == 6) {
132                    print "Correct.\nIt took you $tries tries.\n";
133                }
134                else {
135                    # indent the number to line up with the word
136                    print "        $jots\n";
137                    $tries++;
138                }

```

```

139             # this is a convenient place to store the guess
140             $twords[$ti++] = $guess . ' - ' . $jots;
141         }
142     }
143 }
144
145 print "Would you like to play again (y/n)? ";
146 $again = <STDIN>;
147 chop($again);
148 }
149
150 print "Goodbye.\n";
151

```

Code Listing 2: processjottomail.pl

```

1  #!/usr/bin/perl -w
2
3  require "./jottolib.pl";
4
5  $JOTTODIR = "jottogames/";
6  $admin    = 'your_email@example.com';
7  $JOTTOLOG = $JOTTODIR.'LOG';
8  $DICTIONARY = "words.5.easy";
9
10 # Extract the From: address from the header.
11 # Just eat the rest of the header (up until a blank line).
12 while (<>) {
13     if ($_ eq "\n") {
14         last;
15     }
16     if (/^From: /) {
17         s/.*<.*>.*\n/$1/;
18         $from = $_;
19         $from = "\L$from";
20     }
21 }
22
23 # Let's send them an email
24
25 open(MAIL, "|/usr/sbin/exim $from");
26 print MAIL "From: Jotto <$admin>\n";
27 print MAIL "To: $from\n";
28 print MAIL "Subject: Your Jotto Game\n";
29 print MAIL "\n";
30
31
32 if (!-e "$JOTTODIR$from") {
33
34 # They don't have a game in progress, so start a new one
35
36     open(WORDS, $DICTIONARY);
37     $i = 0;
38     while ($word = <WORDS>) {

```

```

39         chop($word);
40         $words[$i++] = $word;
41     }
42     close(WORDS);
43
44     srand;
45
46     $word = $words[rand(@words)];
47
48     open(GAME, ">$JOTTODIR$from");
49     print GAME "$word\n";
50     close(GAME);
51     open(JOTTO, ">>$JOTTOLOG");
52     print JOTTO 'date'."$from started with $word\n";
53     close(JOTTO);
54     print MAIL "Starting a new game up for you.\n";
55     print MAIL "Word chosen for you.\n";
56     print MAIL "Send your guess in the format:\n";
57     print MAIL "guess <guess>\n";
58 }
59
60 # Now let's process the body.
61 # (That sounds pretty disgusting.)
62 else {
63     while (<>) {
64         $_ = "\L$_";
65         if (/^guess /) {
66             # we've got a guess
67             s/^guess (.*)\n/$1/;
68             $guess = $_;
69
70             # find out what the secret word for the game is
71             open(GAME, "$JOTTODIR$from");
72             $secretword = <GAME>;
73             chop($secretword);
74
75             # find out how many guesses they's made already
76             $tries = 0;
77             while(<GAME>) {
78                 $tries++;
79             }
80             close(GAME);
81
82             # check_guess returns 0 or an error message
83             if ($errmsg = &check_guess($guess, $secretword)) {
84                 print MAIL $errmsg;
85
86                 # record in the game log
87                 open(JOTTO, ">>$JOTTOLOG");
88                 print JOTTO 'date'."$from guessed $guess and got: $errmsg";
89                 close(JOTTO);
90             }
91             elsif (($jots = &score($guess, $secretword)) == 6) {
92                 $tries++;
93                 print MAIL "Correct.\nIt took you $tries tries.\n";

```

```
94         print MAIL "\n\nJust send another guess to start another game.\n";
95
96         # record the guess in the game log
97         open(JOTTO, ">>$JOTTOLOG");
98         print JOTTO 'date'."$from correctly guessed $guess\n";
99         close(JOTTO);
100
101         # remove the game file so that they can start a new game
102         system("rm $JOTTODIR$from");
103         last;
104     }
105     else {
106         print MAIL "$guess scores $jots.\n";
107
108         # record the guess
109         open(GAME, ">>$JOTTODIR$from");
110         print GAME "$guess $jots\n";
111         close(GAME);
112         open(JOTTO, ">>$JOTTOLOG");
113         print JOTTO 'date'."$from guessed $guess and scored $jots\n";
114         close(JOTTO);
115     }
116     print MAIL "\n\nGuesses:\n";
117     open(GAME, "$JOTTODIR$from");
118     # discard the first one, since that's their secret word
119     <GAME>;
120     while (<GAME>) {
121         print MAIL $_;
122     }
123     close(GAME);
124     last;
125 }
126 else {
127     # didn't get a guess
128     print MAIL "There didn't appear to be a guess in your last email.\n";
129     print MAIL "To send a guess, put the following on a line by itself:\n";
130     print MAIL "guess <guess>\n";
131
132     # let's record this, too, just for error checking
133     open(JOTTO, ">>$JOTTODIR");
134     print JOTTO 'date'."$from sent a mail with no guess";
135     close(JOTTO);
136 }
137 }
138 }
139
140 print MAIL "\nIf you have any questions, please contact $admin.\n";
141 close(MAIL);
```

Code Listing 3: jottolib.pl

```

1  #!/usr/bin/perl -w
2
3  # Jotto functions
4
5  sub check_guess {
6      local($guess) = @_;
7
8
9      # open the dictionary for checking valid words
10     open(WORDS, $words);
11     while ($word = <WORDS>) {
12         chop($word);
13         $words[$i++] = $word;
14     }
15     close(WORDS);
16
17
18     if ($guess =~ /^[^a-z]/ || length($guess) != 5) {
19         "$guess not a legal guess\n";
20     }
21     else {
22         $i = 0;
23         while ($words[$i] && ($guess gt $words[$i])) {
24             $i++;
25         }
26         if ($guess ne $words[$i]) {
27             "$guess is not a valid word\n";
28         }
29         else {
30             0;
31         }
32     }
33 }
34
35
36 sub score {
37     local($theguess, $theword) = @_;
38
39     if ($theguess eq $theword) {
40         # this is the word, so return a special value
41         length($theguess)+1;
42     }
43     else {
44         my @gchars = sort(split(//, $theguess));
45         my @wchars = sort(split(//, $theword));
46         my $wcount = 0;
47         my $jcount = 0;
48         my $found;
49         my @wthing;
50
51         while ($#gchars+1) {
52             $found = 0;
53             while ($#gchars+1 && $wchars[$wcount] && $found == 0) {
54                 if ($gchars[0] eq $wchars[$wcount]) {

```

```
55             $jcount++;
56             shift(@gchars);
57             $found = 1;
58             for ($i = 0; $i <= $wcount; $i++) {
59                 $wthing[$i] = shift(@wchars);
60             }
61             for ($i = ($wcount-1); $i >= 0; $i--) {
62                 unshift(@wchars, $wthing[$i]);
63             }
64         }
65         else {
66             $wcount++;
67             $found = 0;
68         }
69     }
70     $wcount = 0;
71     if ($found == 0) {
72         shift(@gchars);
73     }
74 }
75 $jcount;
76 }
77 }
78
79 1;
80
```

Processing RSS Files with XSLT

Dr. A J Trickett, atrickett@cpan.org

Abstract

Rich Site Summary format XML files are traditionally converted to HTML for display in a browser by a brute force approach. The XML transformation language XSLT can be used instead, while Perl deals with the logistics.

1 Introduction

A number of years ago I lived in Southern California. At the time the best place to get news was via National Public Radio, Public Broadcasting Service or the BBC World Service. Unfortunately, none of them had particularly good reception where I lived, and so I was forced to turn to the web for news of home. In those boom-time days, it seemed every web site was a portal, whether it made business sense or not. This helped me build a custom news page with British and European news, along with world business and technology news.

Returning to Europe I abandoned my academic training and rushed headlong into enterprise web content management working for one of the pioneering XML companies. We were so pioneering that we never made a profit and as the boom turned to bust the company imploded to nothing. My thoughts again turned to syndicated news and portals—How was it done? Was it easy? Could we use it to build a federated web site that was easy for a small company to build and maintain?

2 RSS Basics

Rich Site Summary (RSS) files allow people to syndicate a web site. RSS is an initialization of one of several possible phrases: Rich Site Summaries, Really Simple Syndication, or Resource Description Framework Site Summaries. As RSS evolved, the meaning of RSS has shifted to match it's evolving abilities.

An RSS file uses the Extensible Mark-up Language (XML) to give a summary of the content on that site. The XML specification is a descendent mark-up language of Standard General Mark-up Language (SGML) developed in the 1970s, but unlike SGML, which is complex to use, the designers wanted XML to be simple like HTML, which had proved popular as the basis of web sites. Typically, a site's content management system constructs RSS as stories and articles show up on the web site. Most sites place their RSS files on their web sites, and so the files are easy to download. Mirroring tools that only download the RSS files if they have changed are ideal for the task.

3 Extensible Style Language Transformations (XSLT)

The XML Style Sheet Language - Transformation (XSLT) is a World Wide Web Consortium (W3C) standard for converting XML documents to another format. A style sheet written in XML consists of a number of rules which the XSLT engine uses to convert the source XML to another format. A full introduction is beyond the scope of this paper, and I list many references at the end.

At the simplest level XSLT takes one XML document as a Document Object Model (DOM) tree, and converts it to another format. The XSLT file is an list of transformation templates that apply to specific parts of the input DOM tree. Each individual template may operate in isolation of other rules, to give a result tree.

Code listing 1 shows a simple XML document with a `<statement>` tag and a `<footer>` tag. I want to apply a style sheet to it to produce the output in code listing 3.

In code listing 2, the first rule of my style sheet tells the engine to start at the root of the DOM tree `/`. It then outputs a `<div>` tag, and then the second rule tells the engine to look in the tree for a path than matches `root/statement` from the current context. If it finds a match the second template outputs a `<p>` tag, followed by the content of the current input tree node value `Hello World!`, then a `</p>`. Flow returns to the calling template, which outputs a `</div>`.

Code Listing 1: An example XML document

```
1 <?xml version="1.0"?>
2 <root>
3 <statement>Hello World!</statement>
4 <footer>Foo</footer>
5 </root>
```

Code Listing 2: An example style sheet

```
1 <xsl:template match="/">
2   <div>
3     <xsl:apply-templates select="root/statement"/>
4   </div>
5 </xsl:template>
6
7 <xsl:template match="statement">
8   <p>
9     <xsl:value-of select="."/>
10  </p>
11 </xsl:template>
```

Code Listing 3: Result of transformation

```
1 <div><p>Hello World!</p></div>
```

As with Perl, XSLT has more than one way to do it, which can intimidate new users. Like Perl, XSL-T is a very flexible language, so it is easy to write this style sheet in a totally different manner and get exactly the same result. I often find other people's XSL style-sheets very confusing, just as I did with other people's Perl scripts, but with time they do start to make sense.

3.1 Using Perl

Code listing 4 uses the LWP::Simple module to retrieve files, and the GNOME libxslt-based XML::LibXSLT to transform them. The first command line argument to the script specifies the file to fetch, and the second specifies the XSLT template to use. Once LWP::Simple fetches the XML file, XML::LibXML and XML::LibXSLT convert it to HTML via XSLT. Perl provides the framework for the download and conversion, and the XSL stylesheet provides the rules of the conversion—which separates code and appearance.

Code Listing 4: Fetch and transform XML file

```
1  #!/usr/bin/perl
2
3  use strict;
4  use LWP::Simple;
5  use XML::LibXML;
6  use XML::LibXSLT;
7
8  my $site = shift;
9  my $xslt = shift;
10
11 my $rss = get($site);
12
13 my $xslt = XML::LibXSLT->new;
14 my $parser = XML::LibXML->new;
15
16 my $source_xml = $parser->parse_string($rss);
17 my $style_xsl = $parser->parse_file($xslt);
18
19 my $stylesheet = $xslt->parse_stylesheet($style_xsl);
20 my $transformed = $stylesheet->transform($source_xml);
21
22 print $stylesheet->output_string($transformed);
```

4 Problems with RSS

There are two RSS families, and they are different so that I cannot use the same XSL style sheet on all of them. In theory, I should be able to convert one RSS file into another one, but it is not that simple.

Netscape Communications developed the original RSS format, version 0.9, and UserLand later simplified it to create version 0.91. Independently, the RSS-DEV Working Group developed version 1.0, a new and incompatible format based on the W3C Resource Description Format (RDF) core. UserLand, unhappy with the RDF-based RSS, continued to extend and expand RSS up to its current version, 2.0.

The RDF-based RSS format uses XML namespaces, which has its advantages, but makes the document much more verbose and more difficult to transform with XSLT. A number of XSLT proprietary extensions make this much easier, although they are not universally supported.

As many RSS files are automatically generated from badly written HTML by content managements systems, the resultant RSS content is often poorly formed or invalid. Some site editors correct their RSS feeds, but all too often there is nothing to be done but to accept that the incoming feed will be wrong.

The W3C requires an XML parser to abort processing if it that encounters a badly formed or invalid document. The major Perl XML parsers comply and will die in those cases. If a document format is invalid, the parser cannot convert it to a DOM tree, so Perl cannot transform the document. This is deliberate feature of XML to prevent potential ambiguity of on-the-fly second-guessing that HTML parsers perform. Most web browsers will read and display almost any form of HTML no matter how badly formatted it is.

4.1 The XML::RSS module

The XML::RSS module converts between Perl structures and RSS formats. I can use it to programmatically convert one RSS version into another one; however, the module has a number of problems and limitations, plus one fatal flaw as of version 0.97—it does not output properly escaped XML, so any '&' is incorrectly outputted as '&', a special character in XML since it signals the start of an entity encoding. The module should encode any literal '&' as an '&'.

As a result of an email I sent to brian d foy regarding his recent article in this journal using the XML::RSS, he took it upon himself to fix the module, and another project on SourceForge was born. The project developers have not released a version that fixes this problem, although they have been working on it.

4.2 XML::RSS::Tools

The XML::RSS::Tools module attempts to circumvent source and XML::RSS escaping problems, while using XML::RSS to normalise the source. It incorporates HTTP tools and the GNOME-based XSLT engine to provide giving a complete a tool-kit. Code listing 5 uses the module to download the file, then transforms and outputs the result in one step. It has the same command line arguments as the earlier example—an RSS file location and an XSL template.

I create an XML::RSS::Tools object by initialising the module to its default configuration. Inside an eval, I use the object to load the source file and the xsl file, transform the source, and output the result as a string. I use an eval block in case an invalid RSS file causes the XML parser to die.

Code Listing 5: Using XML::RSS::Tools

```
1  #!/usr/bin/perl
2  use strict;
3  use XML::RSS::Tools;
4
5  my $rss = XML::RSS::Tools->new;
6  eval {
7      print $rss->rss_file(shift)->xsl_file(shift)->transform->as_string;
8  };
9
10 print $rss->as_string('error') if ($@);
```

This XSL style sheet in code listing 6 converts a single RSS feed into a XHTML fragment. It starts with the standard XML and XSLT header details. I told the process to turn off the XML declaration to make the fragment easier to directly incorporate in a XHTML document. I selected XML output and turned on indents to give a neater document.

The first template rule selects the XML root of the document, outputs a literal `<div>` tag, then applies the `rss/channel` rule, and outputs a `</div>` tag.

The `rss/channel` rule is where I process the details of the channel. I start by creating a number of variables, and populating them with the details to create an image link and the heading. I use an `xsl:if` to check if there is a image to link to, and if so populate an `` tag with it. I create `<h3>` and `<a>` tags which link to the originating site. I output an `<hr/>` tag to separate the title, and then create an un-ordered list to put the individual story titles in. Inside the `` tags I place an `xsl:apply-templates` command which inserts the contents of each item. One of the many nice things about the XSL language is that I do not need to know how many items there are in a given story—the simple rule will find them all.

The `item` rule creates a pair of variables for the link, outputs an `` tag, constructs an `<a>` tag, and close with a literal `` since this is XML.

Combining the style sheet with a RSS format will give the XHTML fragment in code listing 7.

```

_____ Code Listing 6: Style sheet to transform RSS to XHTML _____
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet version="1.0"
3      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4      exclude-result-prefixes="xsl">
5  <xsl:output method="xml" omit-xml-declaration="yes" indent="yes"/>
6
7  <xsl:template match="/">
8      <div>
9          <xsl:apply-templates select="rss/channel"/>
10         </div>
11 </xsl:template>
12
13 <xsl:template match="rss/channel">
14     <xsl:variable name="link" select="link"/>
15     <xsl:variable name="description" select="description"/>
16     <xsl:variable name="image" select="image/url"/>
17     <xsl:if test="$image">
18         
19     </xsl:if>
20     <h3>
21         <a href="{ $link}" title="{ $description}"><xsl:value-of select="title" /></a>
22     </h3>
23     <hr/>
24     <ul><xsl:apply-templates select="item"/></ul>
25 </xsl:template>
26
27 <xsl:template match="item">
28     <xsl:variable name="item_link" select="link"/>
29     <xsl:variable name="item_title" select="description"/>
30     <li>
31         <a href="{ $item_link}" title="{ $item_title}"><xsl:value-of select="title" /></a>
32     </li>
33 </xsl:template>
34
35 </xsl:stylesheet>

```

Code Listing 7: XHTML result of RSS transformation

```
1     <div>
2         
5         <h3>
6             <a href="http://use.perl.org/" title="All the Perl that's
7     Practical to Extract and Report">use Perl</a>
8         </h3>
9         <hr />
10        <ul>
11            <li><a
12 href="http://use.perl.org/article.pl?sid=02/12/20/220252" title="">Parrot
13 v0.0.9 "Nazgul" released</a></li>
14            <li><a
15 href="http://use.perl.org/article.pl?sid=02/12/18/2350216" title="">Happy
16 Birthday Perl</a></li>
17            <li><a
18 href="http://use.perl.org/article.pl?sid=02/12/16/1648246" title="">POE
19 0.24 Released</a></li>
20        </ul>
21    </div>
```

5 Conclusion

Perl is a powerful language for collecting, downloading and manipulating data. The XML::RSS::Tools module works around some problems in XML::RSS and incorporates the XSLT processing. This way, the code is separate from the presentation details.

6 References

XML In A Nutshell, 2nd edition by Harold & Means, O'Reilly and Associates.

XSLT Quickly by Bob Ducharme, Manning Publications.

XSLT by Doug Tidwell, O'Reilly and Associates.

Beginning XSLT by Jeni Tennison, Wrox Press Ltd.

XSLT Programmer's Reference 2nd Edition by Michael Kay, Wrox Press Ltd.

XSLT Cookbook by Sal Mangano, O'Rielly and Associates.

Content Syndication with RSS by Ben Hammersley, O'Reilly and Associates.

"What is RSS?" by Mark Pilgrim, <http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html>

Separating code, presentation, and configuration

brian d foy, comdog@panix.com

Abstract

I take a program from a previous article and separate the code, presentation, and configuration into separate parts to make the program more flexible and easier to maintain.

1 Introduction

In the last issue, I presented a program I use to pull and display Rich Site Summaries (RSS) from other web sites¹. I used literal values in the code to specify which files to download and how to present the data, and I promised I would fix that in this issue.

Code listing 1 shows the same program I presented in the previous article. The @files array holds the files I want to download, \$base is the directory where my output is stored, and several print statements create HTML with simple variable interpolation (rather than CGI.pm's HTML functions, for example). This code is inflexible and a maintenance hassle. When I want to change the list of sites or the output, I risk breaking the program if I type the wrong thing or make another mistake.

Code Listing 1: RSS downloader with hard-coded values

```
1  #!/usr/bin/perl -w
2  use strict;
3
4  use LWP::Simple;
5  use XML::RSS;
6
7  my @files = qw(
8  http://use.perl.org/useperl.rss
9  http://search.cpan.org/rss/search.rss
10 http://jobs.perl.org/rss/standard.rss
11 http://www.perl.com/pace/perlnews.rdf
12 http://www.perlfoundation.org/perl-foundation.rdf
13 http://www.stonehenge.com/merlyn/UnixReview/ur.rss
14 http://www.stonehenge.com/merlyn/WebTechniques/wt.rss
15 http://www.stonehenge.com/merlyn/LinuxMag/lm.rss
16 );
17
18 my $base = '/usr/home/comdog/TPR/rss-html';
19
20 foreach my $url ( @files )
21 {
22     my $file = $url;
```

¹“Simple RSS with Perl” by brian d foy, *The Perl Review* v0 i5, November 2002, <http://www.ThePerlReview.com>

```

23
24     $file =~ s|.*|/|;
25
26     my $result = open my $fh, "> $base/$file.html";
27
28     unless( $result )
29     {
30         warn "Could not open [$file] for writing! $!";
31         next;
32     }
33
34     select $fh;
35
36     my $rss = XML::RSS->new();
37     my $data = get( $url );
38     $rss->parse( $data );
39
40     my $channel = $rss->{channel};
41     my $image   = $rss->{image};
42
43     print <<"HTML";
44     <table cellpadding=1><tr><td bgcolor="#000000">
45     <table cellpadding=5>
46         <tr><td bgcolor="#aaaaaa" align="center">
47 HTML
48
49         if( $image->{url} )
50         {
51             my $img = qq||;
52             print qq|<a href="$channel{link}">$img</a><br>\n|;
53         }
54         else
55         {
56             print qq|<a href="$channel{link}">$channel{title}</a><br>\n|;
57         }
58
59     print <<"HTML";
60         <font size="-1">$channel{description}</font>
61     </td></tr>
62     <tr><td bgcolor="#bbbbff" width=200><font size="-1">
63 HTML
64
65     foreach my $item ( @{ $rss->{items} } )
66     {
67         print qq|<b>&gt;</b><a href="$item{link}">$item{title}</a><br><br>\n|;
68     }
69
70     print <<"HTML";
71         </font></td></tr>
72     </td></tr></table>
73     </td></tr></table>
74 HTML
75
76     close $fh;
77 }

```

2 Separating presentation

A good design does not tie itself to a particular presentation of the data. My program should fetch the data and make it available to something that presents it—that I am working with RSS should not matter. I might want to produce HTML, TeX, plain text, or even some format that I cannot anticipate.

Everyone seems to write their own templating system, but I like Mark-Jason Dominus's `Text::Template`. It does almost everything I need, does not require extra programs to do its work, and is pure Perl. It has a simple interface and I do not have to learn a templating language because the templates use Perl.

Code listing 2 is the same program as code listing 1, but uses `Text::Template` instead of embedded HTML. In line 5 I import the `fill_in_file()` method. In line 13 I specify the template I will use. All of the HTML in the program is now in the template file in code listing 3.

The `Text::Template` module can accept data as a hash. The keys of the hash become variable names in the template, and the value becomes the template variable value, but also determines the variable type. If the hash value is an a simple scalar, the template variable is a scalar. If the hash value is an anonymous array, the template variable is an array, and so on.

The object created by `XML::RSS` is an anonymous hash. The module has an abstract interface for creation, but not for access. This is just the sort of thing I need to pass to my template. In the template, `$rss->channel`, which has a anonymous hash value, becomes `%channel` in the template, and `$rss->items`, which has an anonymous array value, becomes `@items` in the template.

Code Listing 2: Use a template

```
1  #!/usr/bin/perl -w
2  use strict;
3
4  use LWP::Simple;
5  use Text::Template qw(fill_in_file);
6  use XML::RSS;
7
8  my @files = qw(
9  http://use.perl.org/useperl.rss
10 );
11
12 my $base    = '.';
13 my $template = 'rss-html.tpl';
14
15 foreach my $url ( @files )
16 {
17     my $file = $url;
18
19     $file =~ s|.|/||;
20
21     my $result = open my $fh, "> $base/$file.html";
22
23     unless( $result )
24     {
25         warn "Could not open [$file] for writing! $!";
26         next;
27     }
28
```

```
29     my $rss = XML::RSS->new();
30     my $data = get( $url );
31     $rss->parse( $data );
32
33     print fill_in_file( $template, HASH => $rss );
34     close $fh;
35 }
```

Inside the template, Text::Template runs blocks of code it finds between curly braces. It replaces the block of code with the last evaluated expression. The variable names are the keys of the hash reference I passed as an argument to `fill_in_file()` in code listing 2.

Code Listing 3: An HTML template

```
1 <table cellpadding=1><tr><td bgcolor="#000000">
2 <table cellpadding=5>
3     <tr>
4         <td bgcolor="#aaaaaa" align="center">
5             <a href="{ $channel{link} }">{
6
7                 $image ? qq|| : $channel{title}
8
9             }</a><br>
10
11         { $channel{description} }
12     </td>
13 </tr>
14
15 <tr>
16     <td bgcolor="#bbbbff" width=200><font size="-1">
17 {
18     my $str;
19
20     foreach my $item ( @items )
21     {
22         $str .= qq|<b>&gt;</b><a href="$item{link}">$item{title}</a><br><br>\n|;
23     }
24
25     $str;
26     } </font></td>
27 </tr>
28 </td></tr></table>
29 </td></tr></table>
```

Once I have the templating system in place, I can change the presentation without affecting the logic of the code. If I decide to change the way I present the data, I only change the template. If I want plain text instead of HTML, I simply modify the template for the new format that I want, as I do in code listing 4.

Code Listing 4: A plain text template

```
1 { $channel{title} }
2
3 { $channel{description} }
4
5 {
6 my $str;
7
8 foreach my $item ( @items )
9     {
10        $str .= qq|* $$item{title}\n|;
11    }
12
13 $str;
14 }
```

3 Separating configuration

A good design also permits the script to adapt to a different environment. In code listing 1, I hard-coded the value of the output directory into the script which makes it fragile—if my home directory changes, my script breaks. Also, in code listing 2, I hard-coded the template file name, even though I can change the presentation by changing the template. I should be able to give each template a descriptive name rather than use the same name no matter the content.

Many freely-available scripts that I find on the internet require that the user edit a top portion of the script, or an included library that contains only configuration data. This approach requires the end user to know the basics of the programming language and to edit code—a mistake breaks the script. Bad configuration data can give unexpected results, but they should not break the program.

I can specify run-time configuration data in several ways, and I only show one of them. The Comprehensive Perl Archive Network (CPAN)² has several modules to parse different configuration file formats or command line arguments. Designers should choose an approach that fits their needs.

When I first started to routinely separate configuration data from my scripts, I tried several of the modules on CPAN, and settled on `ConfigReader::Simple` which uses a simple key-value line-oriented format. I used it often enough that I started to send in my changes to Bek Oberin, the original author, then completely took over maintenance of the module.

Code listing 5 adapts code listing 2 to use `ConfigReader::Simple`. I create a new configuration object, then read values from the object. The module turns the names of the configuration keys into method names for easy access (although exotic key names that I cannot coerce into a Perl identifier require the `get()` method to access their value). Code listing 6 shows the configuration file.

Code Listing 5: Using `ConfigReader::Simple`

```
1 #!/usr/bin/perl -w
2 use strict;
3
4 use ConfigReader::Simple;
```

²<http://search.cpan.org>

```
5 use LWP::Simple;
6 use Text::Template qw(fill_in_file);
7 use XML::RSS;
8
9 my $config = ConfigReader::Simple->new( './rss.config' );
10
11 my $base = $config->base;
12 my $template = $config->template;
13 my $extension = $config->extension;
14
15 my @files = split /\s+/, $config->files;
16
17 foreach my $url ( @files )
18 {
19     my $file = $url;
20
21     $file =~ s|\.*/||;
22
23     my $result = open my $fh, "> $base/$file.$extension";
24
25     unless( $result )
26     {
27         warn "Could not open [$file] for writing! $!";
28         next;
29     }
30
31     my $rss = XML::RSS->new();
32     my $data = get( $url );
33     $rss->parse( $data );
34
35     print $fh fill_in_file( $template, HASH => $rss );
36     close $fh;
37 }
```

Code Listing 6: Configuration file

```
1 base .
2 template rss-html.tmpl
3 files http://use.perl.org/useperl.rss
4 extension html
```

4 Conclusion

I can reduce the size of my programs by separating the code from the presentation logic and the configuration information. This separation makes the program more flexible and easier to adapt to new environments. Templates allow the output to show up in many forms without changes to the code, and configuration files allow me to change the way the program operates without affecting the code. The `Text::Template` and `ConfigReader::Simple` make this about as simple as it can be.

5 References

All modules mentioned in this article are in the Comprehensive Perl Archive Network (CPAN)
– <http://search.cpan.org>

6 About the author

brian d foy is the publisher of *The Perl Review*.

Paying Homage to Perl (PHP)

Ed Summers, ehs@pobox.com

Abstract

Perl and PHP are two extremely popular tools for generating dynamic web content. Both are shining examples of successful open source development and are similar enough syntactically that distinguishing one from the other can sometimes be difficult. I provide an overview of the similarities and differences of syntax to serve as a guide for the Perl hacker who needs to do a bit of PHP, and the PHP hacker who needs to do a bit of Perl.

1 Introduction

It is no secret: there is more than one way to generate dynamic HTML. In some ways today's widespread use of Perl can be attributed to being in the right place at the right time when the Web took the world by storm in the mid 1990s. Of course it did not hurt that Perl was especially good at working with textual data.

Today the scenery has changed somewhat, and there are a wealth of options available to the web publisher who needs to produce interactive web pages, including Java, Python, ColdFusion, Active Server Pages, and PHP. Since many of these languages share C as a common ancestor, they share a lot of syntax as well. For example, PHP and Perl share so much syntax that it is possible to look at a block of code and not know if it is Perl or PHP, as in code listing 1.

Code Listing 1: Perl or PHP?

```
1 $x[0] = 'The';
2 $x[1] = 'Perl';
3 $x[2] = 'Review';
4 for ($i=0; $i<3; $i++) {
5     if ( $i == 1 ) {
6         print 'PHP';
7     } else {
8         print $x[$i];
9     }
10
11     print ' ';
12 }
```

PHP and Perl share quite a bit more than syntax however. Both are two of the big success stories of the open source development model. Furthermore, when PHP was first released in 1995 as Personal Home Page / Form Interpreter (PHP/FI) it was actually a set of Perl programs. Today, PHP v4.0 has its own interpreter written in C, but it still holds to its Perlsh roots. Its syntax draws heavily on the language features that

made Perl such a popular tool for working with the Web and both can be embedded in Apache, the most popular web server¹.

Given their pervasive use, programmers may find themselves involved in a project where they need to build or maintain an application using Perl or PHP. Some may end up in a situation where they are encouraged to answer “which is better Perl or PHP?”. A lot of programmers have strong opinions about this already. However, as Mark-Jason Dominus points out in his article “Why I Hate Advocacy”², “apparently it [is] inconceivable that there might be *two* right ways of doing something.” The motivation to decide which solution is better sometimes clouds the issue from the start. Dominus examines a situation where the designer of the Perl Archive³ received complaints because part of the website was implemented with PHP:

One big problem with thinking and talking like this is that it means we can't learn anything new. Suppose that PHP has some advantage over Perl that would lead Jasmine to use it in place of Perl on her website. If that's true, wouldn't it be cool if Perl could copy that advantage in the next version?

It would be cool—this is precisely what has made Perl such an enjoyable language for me in the first place. Dominus's comments remind me of one of a talk Larry Wall gave⁴ at Linux World 1999 in which he described Perl's postmodern design philosophy:

When I started designing Perl, I explicitly set out to deconstruct all the computer languages I knew and recombine or reconstruct them in a different way, because there were many things I liked about other languages, and many things I disliked. I lovingly reused features from many languages. . . I've [borrowed] over the course of the years from C, sh, csh, grep, sed, awk, Fortran, COBOL, PL/I, BASIC-PLUS, SNOBOL, Lisp, Ada, C++ and Python. To name a few.

This leads to another thing that Perl and PHP have in common: just as Perl borrowed liberally from other languages, so PHP borrowed from Perl.

2 The Basics

Like Perl, PHP is an interpreted language: the code that I write is read by an interpreter, which parses my instructions and then performs them. Since Perl predates the Web and is a general purpose language, the interpreter is generally known to run from the command line. On the other hand, Rasmus Lerdorf designed PHP to quickly build HTML content, so its interpreter is normally compiled directly into an Apache webserver. However, the Perl and PHP interpreters can run both ways: the I can run the PHP interpreter from the command line in much the same way as Perl; and `mod_perl` allows me to embed the Perl interpreter directly into Apache for increased performance. There are differences in how the embedded interpreters behave: most significant is that out of the box `mod_php` does not cache compiled code⁵ while `mod_perl` does. Caching compiled code has a significant impact on performance and memory usage.

¹When this article was written 59.91% of the web servers (Netcraft Web Server Survey, September, 2002. <http://www.netcraft.com/survey/>) on the Internet ran Apache, and of those 40.03% ran `mod_php`, and 30.79% ran `mod_perl` (“Apache Module Report”. Security Space, September 1st, 2002. https://secure1.securityspace.com/s_survey/data/man.200208/apachemods.html)

²<http://www.perl.com/lpt/a/2000/12/advocacy.html>

³<http://www.perlarchive.com>

⁴“Perl, the first postmodern computer language” by Larry Wall. <http://www.wall.org/~larry/pm.html>

⁵Add-on packages allow it to cache results. “Caching PHP Programs with PEAR” by Sebastian Bergmann. <http://www.onlamp.com/pub/a/php/2001/10/11/pearcache.html>

Perl is a general purpose programming language whereas PHP is a specialized language for creating WWW applications. Both have their benefits: Perl can parse the human genome in addition to creating HTML. PHP's syntax leans towards generating HTML, and is not as overwhelming for new programmers. It is different from Perl since it uses an inside-out approach (although programmers are not required to use it this way). All PHP code is within special tags in an HTML file. When a PHP-enabled Apache encounters such a file it will generate the verbatim HTML, and hand off PHP code to the PHP interpreter.

In code listing 2, the `<?php ... ?>` sections are XML style tags for flagging an area of the document for processing by PHP, although PHP supports other tagging schemes: SGML style, ASP style, and script style. The XML style is the most popular since it is compliant with the XHTML standard. This approach is allows designers to easily focus on the HTML presentation while the programmers focus on the PHP portion. This approach is also a good way to introduce designers to the terminology and techniques of programming. In fact, the popularity of inside-out design has produced the EmbPerl Apache module which allows me to embed Perl code in HTML documents in much the same way.

Code Listing 2: PHP embedded in HTML

```
1 <html>
2 <body>
3
4 <?php $time = time(); ?>
5
6 <p>If you are using a Mac it has been <?php print $time; ?> seconds since
7 January 1 1904.</p>
8     <p>Otherwise it has been <?php print $time; ?> seconds since January 1,
9 1970.</p>
10
11 </body>
12 </html>
```

It also possible to write PHP in a more Perl like fashion, as in code listing 3. This example illustrates some general similarities between Perl and PHP: semicolons to end statements; whitespace; comments (although PHP natively supports C and C++ style comments as well); and the use of sigils `$` to identify variables. PHP's function names are not case-sensitive, but variable names are. While it is not shown here, PHP and Perl both use C style curly braces to denote blocks of code.

Code Listing 3: HTML embedded in PHP

```
1 <?php
2 $time = time();
3
4 print "<html><body>";
5 print "<p>If you are using a Mac it has been $time seconds since ".
6 "January 1 1904.</p>";
7 print "<p>Otherwise it has been $time seconds since January 1, 1970.</p>";
8 PRINT "</body></html>";
9 ?>
```

2.1 Code Reuse and Documentation

One of the most distinguishing features of Perl is the Comprehensive Perl Archive Network (CPAN)⁶ which is a virtual library of Perl code that allows programmers to stand on the shoulders of giants who have come before. PHP has a similar resource called the PHP Extension and Application Repository (PEAR)⁷ which also provides access to reusable code libraries. Since Perl has such a head start on PHP, PEAR is a mere shadow of CPAN, but PEAR is likely to change as PHP developers recognize the benefits of code reuse.

Perl and PHP documentation are both available online⁸. I can embed my own documentation in both Perl and PHP programs. Perl uses the Plain Old Documentation (pod) format⁹, and PHP uses phpdoc¹⁰, inspired by javadoc, and is still in beta. Code listing 4 shows a pod example and code listing 5 shows the PHP equivalent.

Code Listing 4: Pod documentation

```
1 =head1 friend()
2
3 To get a string saying someone is your friend, pass in your friend's name
4 as a string and you'll get it back.
```

Code listing 5 shows the PHP equivalent to code listing 4. Although phpdoc is not integrated fully into PHP as pod is in Perl, like javadoc it has the benefit of explicitly defining the types of parameters that I can pass to a function, as well as its return values. Pod benefits from being a general purpose formatting language—I originally wrote this article in pod.

Code Listing 5: PHPdoc documentation

```
1 /**
2  * To get a string saying someone is your friend, pass in your friend's
3  * name and you'll get it back.
4  *
5  * @param   string
6  * @return  string
7  */
8 function friend ( $name ) {
9   return( "$name is my friend" );
10 }
```

3 Scaling Operations

PHP and Perl allow me to store integer, floating point, and string data in the same type of variable: the scalar. Scalar variables use the dollar sign, \$, to identify the variable name, and are the basic building block of other variable types. Having weakly-typed scalar variables is a great boon to web developers since it

⁶<http://www.cpan.org>

⁷<http://pear.php.net>

⁸<http://php.net> and Perl <http://www.perldoc.com>

⁹See the perlpod manual page

¹⁰<http://www.phpdoc.de>

makes it easy to work with numeric and textual data interchangeably without having to worry about buffer overflows and the casting of variables to different types. Scalars values in Perl and PHP are limited by the C library type sizes (for numbers) and the amount of available memory (for strings)

Unlike Perl, PHP has native support for boolean values which allow me to assign the values `true` and `false` to a scalar variable. Perl supports booleans by evaluating a scalar as false if it contains `0`, `undef`, or the empty string, and true otherwise. Being able to explicitly assign a variable to `true` or `false` encourages a certain amount of clarity; however PHP also supports Perl’s DWIMery—the only difference being that like Java, PHP uses the keyword `null` instead of Perl’s `undef`.

The standard comparison operators `<`, `>`, `<=`, `>=` and `==` are available to compare scalars. In general, the two languages do similar type casting behind the scenes to allow me to compare integer, float and string values in a meaningful way. However, the tests for equality which look similar but are actually quite different. Perl uses `==` for numeric and `eq` for string comparisons, both of which will do implicit type casting. PHP uses the `==` for both numeric and string tests of equality (with implicit type casting), and `===` as a test of equality without implicit type casting. Table 8 shows the different behavior of `==` in particular can be the source of subtle errors for the unwary.

	Perl	PHP
<code>'2e' == '2c'</code>	true	false
<code>'TRS80' == 'trs80'</code>	true	false

Table 1: Comparison results

In addition, PHP has the `<>` operator which is an alias to `!=`. Neither language currently supports chained comparison operators (`1 < $x < 10`) but Larry has it slated for addition in Perl 6¹¹.

The usual cast of mathematical operators are present in both languages (`+`, `-`, `*`, `/`, `%`) as well as the postfix/prefix and bitwise operators (`++`, `--`, `!`, `&`, `~`, `^`, `<<`, `>>`). Conspicuously absent from this list is the exponentiation operator, which is `**` in Perl. PHP does not have an equivalent operator, but it does have the `pow()` function. The common mathematical functions in both languages include `cos()`, `sin()`, `log()`, `sqrt()`.

The power and flexibility of Perl’s string operations distinguished it early on as a tool for working with the Web, so it is hardly surprising that PHP has many of the same functions, albeit some have slightly different names. The most reknown of these string manipulation tools are regular expressions, which PHP supports via the `preg_match()`, `preg_replace()`, `preg_split()` functions. PHP also supports POSIX style regular expressions which “are less powerful, and sometimes slower, than the Perl-compatible functions”¹². Rather than itemizing all the string manipulation functions in PHP and Perl, Table 2 illustrates how to achieve similar ends with both languages.

3.1 A Tale of Two Arrays

One of the real joys of programming in Perl are the array and hash data structures with their associated functions and syntactic sugar. Arrays and hashes automatically grow in size as needed and can be multi-dimensional. PHP also has built in support for arrays and hashes although they are not quite the same under the covers, and they are named differently. Perl distinguishes between scalar, array and hashes using the `$`, `@` and `%` sigils, whereas PHP only uses the `$`. In some ways this can be seen as taking weakly-typed

¹¹ “Apocalypse 3” by Larry Wall. <http://dev.perl.org/perl6/apocalypse/3#rfc%20025:%20operators:%20multiway20comparisons>

¹² Programming PHP. Rasmus Lerdof and Kevin Tratoe. Oreilly. Sebastopol, CA : O’Reilly & Associates, 2002, p. 95.

Perl	PHP
<code>\$a . \$b</code>	<code>\$a . \$b</code>
<code>lc(\$x)</code>	<code>strtolower(\$x)</code>
<code>uc(\$x)</code>	<code>strtoupper(\$x)</code>
<code>ucfirst(\$x)</code>	<code>ucfirst(\$x)</code>
<code>\$a <=> \$b</code>	<code>strcmp(\$a,\$b)</code>
<code>substr(\$x,0,1)</code>	<code>substr(\$x,0,1)</code>
<code>substr(\$x,0,3) = 'abc'</code>	<code>substr_replace(\$x,'abc',0,3)</code>
<code>index(\$x,\$y)</code>	<code>strpos(\$x,\$y)</code>
<code>length(\$x)</code>	<code>strlen(\$x)</code>
<code>\$x = tr/aeiou/z/</code>	<code>\$x = strtr(\$x,'aeiou','zzzzz')</code>
<code>\$x = /perl/</code>	<code>preg_match('/perl/', \$x)</code>
<code>\$x = /[aeiou]/</code>	<code>preg_match('/[aeiou]/', \$x)</code>
<code>\$x = /perl/i</code>	<code>preg_match('/perl/i', \$x)</code>
<code>\$x = s/php/perl/</code>	<code>\$x = preg_replace('/php/', 'perl', \$x, 1)</code>
<code>\$x = s/perl/php/g</code>	<code>\$x = preg_replace('/perl/', 'php', \$x)</code>
<code>split/=/, \$x</code>	<code>preg_split('/=/', \$x)</code>
<code>split/(=)/, \$x</code>	<code>preg_split('/(=)/', \$x, -1, PREG_SPLIT_DELIM_CAPTURE)</code>

Table 2: String manipulations

languages to a whole new level. Those who find Perl's sigils to be a bit odd in the first place probably will not miss them much, but Perl programmers who have come to depend on them may feel a bit lost since in some ways all variables are scalars in PHP—they just scale to arrays and hashes as well as numbers and strings.

PHP refers to Perl's arrays and hashes as indexed arrays and associative arrays. However, PHP has only associative arrays since indexed arrays are just associative arrays where the keys are numeric. The keys in PHP's associative arrays are ordered so that it can return elements in order. One way to think of PHP's indexed arrays is as a shorthand for creating associative arrays with ascending numeric keys. Table 3 compares Perl and PHP array and hash operations.

Perl	PHP
<code>@x = (1,2,3);</code> <code>\$x[3] = 4;</code>	<code>\$x = Array(1,2,3);</code> <code>\$x[3] = 4;</code>
<code>%x = ('a' => 1, 'b' => 2);</code> <code>\$x{'c'} = 3;</code>	<code>\$x = Array('a' => 1, 'b' => 2);</code> <code>\$x['c'] = 3;</code>
<code>push(@y,\$x);</code> <code>\$x = pop(@y);</code> <code>\$x = shift(@y);</code> <code>unshift(@y,\$x);</code>	<code>array_push(\$y,\$x);</code> <code>\$x = array_pop(\$y);</code> <code>\$x = array_shift(\$y);</code> <code>array_unshift(\$y,\$x);</code>
<code>@x = split('/:/', \$y);</code> <code>\$y = join(':', @x);</code>	<code>\$x = split('/:/', \$y);</code> <code>\$y = join(@x, ':');</code>
<code>@x = @y[2,3,4];</code> <code>@x[2,3,4] = @y;</code>	<code>\$x = array_slice(\$y,2,3);</code> <code>array_splice(\$x,2,3,\$y)</code>

Table 3: Array operations

Perl has two very different data structures for working with indexed and associative arrays. The data structures are differentiated with the `@` and `%`, and accessed using the `[]` and `{}` notation rather than just the `[]`. Perl uses the `$` rather than the `@` or `%` when indexing into an array or hash is that the result is a scalar since arrays and hashes contain only scalars. Perl programmers are familiar with the `=>` notation that PHP uses to distinguish associative arrays from indexed arrays; however, they are probably not familiar with the `Array()` construct, which is almost a type of constructor for both indexed and associative arrays.

Since PHP's arrays have iterator functions: `current()`, `reset()`, `next()`, `each()`, `key()`, `prev()`, and `end()`. While Perl has some implicit iterator functions for arrays and hashes they tend to work behind the scenes, and it is handy to have functions that work directly with the iterators. Perl 6 will boast arrays and hashes that are first class objects which can have methods such as PHP's iterating functions.

3.2 Control Structures

Perl and PHP have the usual control structures: `if()`, `for()`, `while()`, `return()`, and `exit()`. Both languages have a `foreach()` statement to iterate over arrays; however, their syntax and behavior are quite different given the different nature of arrays in each language. Table 4 shows the differences in syntax.

Perl	PHP
<code>foreach \$x (@y) { }</code>	<code>foreach (\$y as \$x) { }</code>
<code>foreach \$x (keys(%y)) { }</code>	<code>foreach (\$y as \$k => \$v) { }</code>
<code>foreach \$x (@y) { \$x++; }</code>	<code>foreach (\$y as \$x) { \$x++; }</code>

Table 4: Syntax of `foreach()`

PHP uses the `as` keyword to indicate what sort of variable aliasing to use while iterating through the array. The two variants correspond to the types of arrays that PHP supports (indexed and associative). Perl's `foreach()` allows me to modify the aliased variable inside the body of my loop and have it modify the array or data structure that I am iterating over. The third example in Table 4 illustrates alias modification to increment each element in an array—this does not work under PHP.

PHP has a `switch()` statement. The absence of a `switch()` in Perl has been noted, and rationalized away since it is possible to use existing Perl syntax to do roughly the same thing¹³. High on the list of new stuff for Perl6 is a super-charged `switch()` statement—which is so super and charged it isn't even called a `switch()`¹⁴.

Loop control is slightly different in PHP and Perl since PHP does not allow you to label blocks of code, and Perl's `next()` is equivalent to PHP's `continue()`.

3.3 Functionally Speaking

Perl and PHP allow me to extend the core language with user defined functions. PHP uses the `function` keyword and Perl uses `sub` to identify them. The function name is case sensitive in Perl, but not in PHP.

PHP allows me to declare parameters as part of the function definition. Supporting parameter definitions allows function parameters to take default values; and warnings are automatically thrown when I call a

¹³Programming Perl by Larry Wall, Tom Christiansen and Jon Orwant. Sebastopol, CA : O'Reilly & Associates, 2002. p. 124.

¹⁴“Exegesis 4” by Damian Conway. <http://www.perl.com/pub/a/2002/04/01/exegesis4.html>

Perl	PHP
<pre> LOOP1: while (\$x) { LOOP2: while (\$y) { next LOOP1 if \$x == \$y; } } </pre>	<pre> while (\$x) { while (\$y) { if (\$x == \$y) continue 2; } } </pre>

Table 5: Comparison results

function with the incorrect number of parameters. Perl's functions give you access to the special array variable `@_` which contains the function parameters, which I can copy or use directly. I can use function prototypes to validate the number and types of parameters that are passed to a Perl function; however, they will not work with methods in object-oriented Perl, which ignores prototypes. In Perl I have to explicitly validate the parameters and assign default values. Table 6 shows an `area()` function whose second parameter has the default value of one. In PHP the default value is part of the prototype, whereas in Perl I have to set it myself by checking the value of the second parameter explicitly.

Perl	PHP
<pre> sub area { my (\$x,\$y) = @_; \$y=1 if !\$y; \$area = \$x * \$y; return(\$area); } </pre>	<pre> function area (\$x, \$y=1) { \$area = \$x * \$y; return(\$area); } </pre>

Table 6: Default function arguments

The `area()` function also illustrates the variable scoping rules in PHP and Perl. All variables in PHP functions are automatically lexically scoped, whereas variables in Perl's functions are automatically global. Like Perl, PHP functions can contain global variables—however I must declare them as global or they are undefined. On the other hand, PHP programmers who expect their Perl function variables to not interfere with global variables will be suprised when they do.

Perl	PHP
<pre> \$x = 1; function incrementX { \$x++; } </pre>	<pre> \$x = 1; function incrementX() { global \$x; \$x++; } </pre>

Table 7: Comparison results

Unlike Perl, the lexical scoping of PHP variables is limited to function calls and as a result is often referred to as *function-level scope*. Perl allows me to define lexically-scoped variables in any block of code, including if-else statements and loop constructs. PHP also has the `static` variable type which allow variables to maintain their value between successive calls to a function, while making it invisible to outside code. In some ways PHP's `static` variables allow you to do things that you might do with closures in Perl.

3.4 References and Objects

Perl and PHP allow you to store references to other variables and functions in scalar variables, although they support references in very different ways. PHP uses the `&` to create a reference from a variable, while Perl uses the `\` character. Perl requires you to dereference a reference using the appropriate sigil (`$@%&`) or the `->` before you can use it; whereas PHP does not.

Perl	PHP
<code>@x = (1,2,3,4,5);</code>	<code>\$x = Array(1,2,3,4,5);</code>
<code>\$y = @x;</code>	<code>\$y = \$x;</code>
<code>print \$x->[2];</code>	<code>print \$y[2];</code>
<code>if (ref(\$x) eq 'SCALAR') {}</code>	<code>if (is_scalar(\$x)) {}</code>
<code>if (ref(\$x) eq 'ARRAY') {}</code>	<code>if (is_array(\$x)) {}</code>
<code>if (ref(\$x) eq 'HASH') {}</code>	<code>if (is_array(\$x)) {}</code>
<code>if (ref(\$x) eq 'CODE') {}</code>	<code>if (function_exists(\$x)) {}</code>

Table 8: Comparison results

Perl and PHP diverge quite a bit when it comes to defining and using object oriented features. In general PHP's syntax for declaring object classes is more explicit than Perl's syntax for declaring packages and modules which is far looser and reveals more of the internal details. Table 9 shows a simple class for representing a circle, which has a method for calculating the area. Table 10 shows code that uses the circle class.

Perl	PHP
<code>package Circle;</code>	<code>class Circle {</code>
<code>my \$PI = 3.14159;</code>	<code>var \$radius;</code>
<code>sub new {</code>	<code>var \$PI = 3.14159;</code>
<code> my (\$class,\$r) = @.;</code>	<code>function Circle (\$r=0) {</code>
<code> my \$self = {</code>	<code> \$this->radius = \$r;</code>
<code> radius => \$r</code>	<code>}</code>
<code> };</code>	<code>function getArea() {</code>
<code> bless \$self, \$class;</code>	<code> \$area = \$this->PI *</code>
<code> return(\$self);</code>	<code> pow(\$this->radius,2);</code>
<code>}</code>	<code> return(\$area);</code>
<code>sub getArea {</code>	<code>}</code>
<code> my \$self = shift;</code>	<code>}</code>
<code> my \$r = \$self->{radius};</code>	
<code> my \$area = \$PI * \$r ** 2;</code>	
<code> return(\$area);</code>	
<code>}</code>	

Table 9: Circle class

PHP's constructors (similar to Java) are the function with the same name as the class; PHP allows you to

Perl	PHP
use Circle;	require("Circle.php");
\$circle = Circle->new(5);	\$circle = new Circle(5);
print \$circle->getArea();	print \$circle->getArea();

Table 10: Using the Circle class

define the object attributes which are carried around automatically with objects of that type; PHP object attributes can be accessed using the special `$this` variable that refers to the object on which the method was called. Perl is extremely loose and allows any method to be a constructor, which must bless a variable into the appropriate package. Perl method calls have to remember that the first parameter passed to the method will be the object. The PHP class implementation is much more succinct and clear, and much of this is due to the ability to define object attributes and the use of the special `$this` variable. So much of Perl's object oriented features are learned conventions, and in some ways it can seem like a dark art for the beginning programmer. Perl's object-oriented features are much more flexible and extensible than PHP's. PHP does not support multiple inheritance, and the naming of modules in Perl allows you to build complex hierarchies of modules which are impossible in PHP. Perhaps this is why the CPAN has done so well, while extending PHP has been limited mainly to compiling components into the interpreter when you build it.

4 Perl and PHP together

Perl and PHP can be used together effectively in an organization. Consider a situation where a group of web designers carefully layout HTML and create graphics and flash animations on a website; while a group of programmers work to connect up this static content with a database to provide dynamic content. Although Perl modules like `HTML::Mason`¹⁵ and `HTML::Template`¹⁶ and Apache's `embperl`¹⁷ can provide solutions that would work as well, it is not difficult to imagine PHP being used to cater to both the designers and the programmers. Also imagine that the programmers would like to have something like `CGI.pm`'s arsenal of exportable HTML functions that guarantee that HTML is well formed. Why not write a Perl program that writes the PHP library? No sooner said than done, or at least that's what Andy Lester did (who this code has been copied from).

The Perl program in code listing 6 generates a library of PHP functions (`HTML.php`) that I can use in other PHP applications that I want to have handy functions for generating HTML. I can automatically add tags by altering the `@exports` array. Not only is Perl a language that has borrowed from liberally from many languages, and had other languages borrow from it, but it turns out Perl is also a good PHP programmer.

5 About the author

Ed Summers is currently working as a programmer at Follett Library Resources in McHenry, Illinois.

Code Listing 6: Create `HTML.php` with Perl

¹⁵<http://www.masonhq.com>

¹⁶<http://html-template.sourceforge.net>

¹⁷<http://perl.apache.org/embperl>

```

1  #!/usr/bin/perl
2
3  use strict;
4
5  my @exports = qw(
6  html head body title h1 h2 h3 h4 h5 h6 p br hr a small big font
7  b i em strong pre kbd tt div span center nobr blockquote ul ol li
8  tag_dl dt dd table tr td th form input select option textarea img
9  );
10
11 my @empty_tags = qw( br hr input img );
12 my @lf_after = qw( table select tr td th p );
13 my @lf_between = qw( table select tr );
14 my @funcs;
15
16 for my $func ( sort @exports ) {
17     my $after = in_array( $func, @lf_after ) ? '\n' : '';
18     my $tag = uc $func;
19     my $body;
20     if ( in_array( $func, @empty_tags ) ) {
21
22         $body =
23
24         <<"PHP_CODE";
25         function $func( \%parms = null ) {
26             if ( isset(\%parms) ) {
27                 return "<$tag" . _build_parm_string( \%parms ) . ">" $after; \
28             } else {
29                 return "<$tag />" $after; \
30             }
31         }
32
33         PHP_CODE
34
35     } else {
36         my $sep = in_array( $func, @lf_between ) ? '\n' : ' ';
37         $body =
38
39         <<"PHP_CODE";
40         function $func() {
41             \%args = func_get_args();
42             return _nonempty_tag( '$tag', $sep, \%args ) $after;
43         }
44
45         PHP_CODE
46
47     }
48
49     push( @funcs, $body );
50 }
51
52 # Create HTML.php
53 open( OUT, ">HTML.php" ) or die "Can't create";
54 print OUT "<?php\n";
55 print OUT "\n### DO NOT EDIT THIS FILE.  IT IS CREATED BY HTML.pl.\n\n";

```

```
56 print OUT <DATA>; # Constant stuff
57 print OUT join( "", @funcs );
58 print OUT "?>\n";
59 close OUT;
60
61 sub in_array {
62     my $needle = shift;
63     my @haystack = @_ ;
64
65     for ( @haystack ) {
66         return 1 if $_ eq $needle;
67     }
68     return 0;
69 }
70
71 ## PHP CODE FOLLOWS
72
73 __DATA__
74 function _nonempty_tag( $tagname, $sep, &$args ) {
75     $return = '<' . $tagname;
76
77     // If there's an Array(), it's our parms
78     if ( $args && is_array($args[0]) ) {
79         $return .= _build_parm_string( array_shift($args) );
80     }
81     $return .= '>';
82
83     // At this point, everything else is text
84     $first = true;
85     foreach ( $args as $val ) {
86         if ( is_array( $val ) ) {
87             $val = implode( $sep, $val );
88         }
89         if ( $first ) {
90             $first = false;
91         } else {
92             $return .= $sep;
93         }
94         $return .= $val;
95     } // while
96
97     $return .= "</$tagname>";
98
99     return $return;
100 }
101
102 function _build_parm_string( &$parms ) {
103     reset( $parms );
104     $str = "";
105     foreach ( $parms as $key => $val ) {
106         $str .= ' ';
107         $str .= $key;
108         if ( isset($val) ) {
109             $val = htmlspecialchars($val);
110             $str .= "=" . $val . " ";
111         }
112     }
113 }
```

```
111     }
112 } // foreach
113
114 return $str;
115 }
116
```
