Table of Contents

Perl Mongers News 3

Perl Foundation News 3

Book Reviews 4

Down Translating XML 6

Module::Release and Beyond 12

Perl News 14

New Modules 15

Functional Perl Programming 16

Faking Stored Procedures 20

How (not) to be a Perl advocate 27

The Perl Review

Volume 1, Issue 1

Winter 2004

Editor/Publisher: brian d foy

Tech Editors: Adam Turoff, Kurt Starsinic

Design: Eric Maki

Columnists: David H. Adler, Allison Randal

Copy Editors: Jim Brandt, Beth Linker, Dave Lundberg

www.theperlreview.com
letters@theperlreview.com

The Perl Review
5301 N. Kenmore Ave #2
Chicago, IL, 60640.

The Perl Review (ISSN 1553-667X) is published quarterly by brian d foy.

Publication Office:

5301 N. Kenmore Ave #2, Chicago, IL 60640. US subscriber rate is $16 per year (4 issues). Foreign subscriber rate is $30 US per year (4 issues). Single copies are $5 in the US, $8 elsewhere.

Postmaster: Send address changes to The Perl Review, 5301 N. Kenmore Ave #2, Chicago, IL, 60640.

Subscribers: Subscribe online or send a check or money order drawn in US funds to the above address.

Advertisers: Send email to ads@theperlreview.com for rates and information.

Publisher's Page

The second issue of the print version of The Perl Review is a big jump over the first: I'm not doing the design (thank god for me *and* for you), the cover looks more like a "real magazine", and thanks to Eric Maki, who used a perl POD-to-RTF converter to get things into the final layout, things went very smoothly.

A lot of things have happened behind-the-scenes on the business underpinnings. We now do our own credit card processing using Business::OnlinePayment::AuthorizeNet, and accept MasterCard, Visa, and American Express. You can still pay through PayPal and lots of other options if you really want to. We're not rolling in money, but we are paying the bills. The next big step is a European edition that will cut the subscription rates to be comparable to those in the US. We're looking at that for the next issue.

We also got out of the blogging business. We wanted to add those to our website (http://www.theperlreview.com), but we're here to publish a magazine, not administer systems and software. So, we set up some discussion forums on LiveJournal (another heavy Perl user) and have some RSS feeds: our website has all the details. We don't want to reinvent the wheel and we want to focus on producing great content, so look for us to use more existing services to pull things together.

We're also getting more organized with the content. We rushed to get the first issue printed in time for the Open Source Convention, and we've had time to breathe since then. Dave Adler, one of the charter members of Perl Mongers (he was there when I came up with the idea) starts a column on PM happenings. Allison Randal of The Perl Foundation starts a column about what's happening there. We're still looking for other column ideas. Want to write for **The Perl Review**? Let us know!

It's almost too bad that this issue is print. When we published PDF-only versions, we could have as many pages as we liked, and we could change that anytime we liked. Frank Antonsen's article on functional programming in Perl started as an almost definitive work on the subject, and it was really good stuff: we just didn't have the space to print it all. The articles by Alberto Manuel Brandão Simões (*Down translating XML*) and Zach Thompson (*Faking Stored Procedures*) also had a lot of great stuff that we had to cut for space. Next time we'll plan for more pages.

And finally, happy birthday to Perl, DBI, and Randal Schwartz. See our Perl Calendar section on page 14 for details.

Perl Mongers News

by David H. Adler, dha@panix.com

To start things off, we have two brand new groups of Mongers: Aarau.pm in Switzerland and Cascais.pm in Portugal. Also, ThousandOaks.pm is resuming regular monthly meetings as of December and Jacksonville.pm has requested that I tell you that they're still there.

A number of Perl Monger groups are getting Perl workshops up and running in their area. London and Melbourne are having theirs in December while Israel, Holland and Germany each have one scheduled for February. Thomas Klausner is keeping up with these—or at least trying to, as more keep popping up all the time—at http://www.yapceurope.org/-workshops.html

Perl Seminar New York (aka perlsemny) will work on testing `Text::Template`, joining Chicago.pm, SouthFlorida.pm and London.pm in working on the Phalanx testing project. Details on the Phalanx Kwiki at http://phalanx.kwiki.org/

On the international relations front, Paris.pm seems to be having a large number of foreign visitors, SaoPaulo.pm is entertaining a member of Rio.pm for the bulk of November and NY.pm may be getting a brief visit from London.

Finally, Mark Jason Dominus' long-awaited book, *Higher Order Perl*, is almost in physical form, and Philadelphia.pm is going to be helping him put together a book signing. No details yet, but when they arrive, phl.pm will surely announce them.

Happy Mongering.

Dave Adler is a Perl Mongers Charter Member and is the "Ruthless Godless Dictator" of NY.pm (http://ny.pm.org). He is perhaps the nation's leading authority on Monty Python. You can find out more about Perl Mongers and Perl user groups at http://www.pm.org/ .

Perl Foundation News

By Allison Randal, allison@perl.org

The Perl Foundation would like some feedback from companies that use Perl every day to find out what you need from Perl 5, Perl 6, and the migration process between the two.

We would also like some funding to speed up key areas of Perl 5, Perl 6, and Parrot development. We value input from everyone who uses Perl, but for this exercise we want to spend some intensely focused time with a small group of companies. So how do we choose? Which companies have something valuable to say?

It's impossible to tell. But if a company depends on Perl enough that they'll invest in it as an investment in their own future, it's a pretty safe bet that we need to hear what they have to say.

So, we're forming an advisory board of sponsors. The first meeting will be in January. Sponsors over a certain threshold (probably no more than 15 companies) will join in open group discussions centering around our current plans for the next few years and how we can meet your needs better.

Sponsors over a higher threshold (probably around 4–5 companies) will each get a block of time to tell the group how they use Perl now, how they plan to use it in the future, and how we can help them do it.

What could you do with an hour of Larry's undivided attention?

Drop me an email if you're interested. A project plan on how we intend to use the funding is available. And, if you're not the right person respond to this, feel free to pass it along.

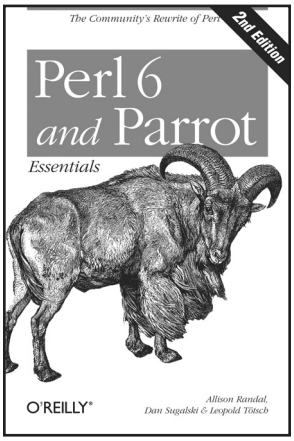
Allison Randal is the president of The Perl Foundation. Founded in 2000, The Perl Foundation (TPF) is a non-profit 501(c)(3) corporation established to advance the use and development of the Perl programming language to get more information go to http://www.perlfoundation.org

Book Review

Perl 6 and Parrot Essentials 2nd Edition

Reviewed by brian d foy

by Allison Randal, Dan Sugalski, Leopold Tötsch
O'Reilly Media
ISBN 0-596-00737-X



This edition updates last year's *Perl 6 Essentials* and promotes Parrot to the title. You could read all of Larry Wall's Apocalypses and Damian Conway's Exegeses (http://dev.perl.org/perl6/) to get the plan for Perl 6, but you could also go crazy doing it. The authors, all Perl 6 developers, digest most of the those, along with other sources, to give a short, 300 page summary of it all.

The first half of the book covers most of the new language features of Perl 6 and the new syntax. Even in this summary, there is a lot of stuff to cover, and the authors do a great job of putting everything into place. The second half of the book covers the Parrot engine that runs Perl 6. I'm not an internals person, but I still found it interesting. Just don't ask me to remember the details.

If you want to learn to program in Perl 6, this probably isn't the book for you (and there isn't a Perl 6 update to *Learning Perl* yet), but if you want to read the blueprints and plans, this is the best place to get those.

Online: http://www.oreilly.com/catalog/059600737X/

Briefly Noted

short reviews by brian d foy

Joel On Software

by Joel Spolsky
Apress LP, ISBN 1-59059-389-8

This book has nothing to do with Perl, and as little to do with open source. It is all about common sense though. Every essay on the practice of writing software as a vocation has several points that will make you reconsider some aspect of how you do things (or don't do things). I couldn't put the book down before I finished it.

SpamAssassin

by Alan Schwartz
O'Reilly Media, ISBN 0-596-00707-8

Just what I needed: a paper guide for SpamAssassin, a great spam filtering tool written in Perl. I still hate reading a computer screen, and this thin book (200 pages not counting the index) is the perfect size to keep near my desk. It covers SpamAssassin 3.0.

Beginning Perl

by James Lee, with Simon Cozens and Peter Wainwright
Apress LP, ISBN 1-59059-391-X

Although this edition is a lot thinner than the first, it's still pretty thick and suffers from too much non-Perl detail in a lot of places.

It's also hard to look at since the code sections aren't clearly separated from the text. Maybe the next edition will start from scratch.

Correction: Last issue we used **O'Reilly Media**'s old name, O'Reilly & Associates. We apologize for the confusion.

Forthcoming Titles:

Randal Schwartz's Perls of Wisdom
Randal Schwartz
Apress LP, ISBN 1-59059-323-5, 375 pp.

Perl 6 Now: The Core Ideas Illustrated with Perl 5
Scott Walters
Apress LP, ISBN 1-59059-395-2, 400 pp.

Pro Perl: From Professional to Expert
Peter Wainwright
Apress LP, ISBN 1-59059-438-X, 1100 pp.

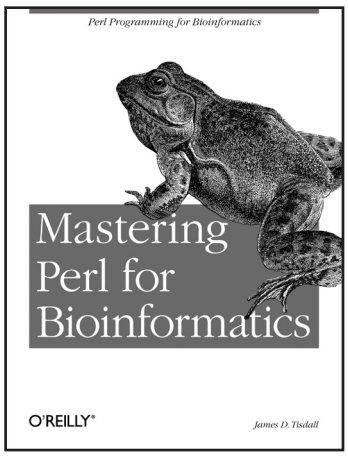
Pro Perl Debugging
Richard Foley, Joe McMahon
Apress LP, ISBN 1-59059-454-1, 300 pp.

Book Review

Mastering Perl for Bioinformatics

Reviewed by Eli Bendersky

James D. Tisdall
O'Reilly Media
ISBN 0-596-00307-2
396 pages



Mastering Perl for Bioinformatics is written well, but beware, it might not be what you are looking for. It is not a book for the general Perl folk eager to learn bioinformatics. It is a book about Perl, with some detours into other topics of computer science. A book "for programmers who want to learn more about bioinformatics" (as the back cover says) should include theoretical background, descriptions of the problems bioinformaticians are trying to solve, etc. None of this is present.

The treatment of Perl object-oriented programming is clear and detailed, and dives into the advanced topics like attribute lists, inheritance, and **AUTOLOAD**. The discussion of relational databases in the 6th chapter is also very nice and gives a good start for Perl programmers who want to use of the DBI modules. I utterly enjoyed the explanation of the Edit Distance algorithm with a reference to Dynamic Programming in the 2nd chapter.

If you're a beginning Perl bioinformatics programmer, interested in OOP, databases, web, graphics and the Bioperl library, you may find this book useful. If you want to learn bioinformatics, you'll have to find something else.

Online: http://www.oreilly.com/catalog/mperlbio/

Eli is a programmer in IBM's Haifa Research Lab, using Perl extensively both for work and for personal pet projects.

Short Note

Extracting Images from Word Files

by brian d foy

For a while I had to deal with a group of old Microsoft Word documents. I could load them into my word processor, but I couldn't see the images. After a little research, I discovered that the images were stored inline as Portable Network Graphics (PNG) files. Not only that, the PNG parts has a definite and constant header and footer.

I wrote this little script to read in a list of Word documents from the command line, extract all of the PNG portions, and save them as external files named after the original document name and the order in which it appeared in the file.

```
#!/usr/bin/perl
my $HEADER = "\211PNG";
my $FOOTER = "IEND\xAEB`\x82";

foreach my $file ( @ARGV )
   {
   print "Extracting $file\n";
   (my $image_base = $file)
      =~ s/(.*)\..*/$1/;

   my $data = do {
      local $/;
      open my( $fh ), $file;
      <$fh> };

   my $count = 0;

   while($data =~ m/($HEADER.*?$FOOTER)/sg)
      {
      my $image      = $1;
      $count++;
      my $image_name =
         "$image_base.$count.png";
      open my $fh, "> $image_name"
         or warn "$image_name: $!", next;
      print "Writing $image_name: " .
         length($image), " bytes\n";
      print $fh $image;
      close $fh;
      }
   }
```

Down Translating XML

Alberto Manuel Brandao Simões

ambs@cpan.org

Down translation is the process of converting XML (eXtensible Markup Language) data streams into another format, such as HTML or plain text. The Comprehensive Perl Archive Network (CPAN) contains lot of XML processing tools, and a new user can get lost in all of them. The **XML::DT** module, originally written by José João Almeida and currently maintained by José and I, is a handy tool for this transformation, even if it isn't the best tool for every job.

History

The **XML::DT** module stands for XML Down Translation, and was first developed by José João Almeida at the University of Minho, Portugal in 1999. He stole some features from other XML and SGML tools like OmniMark and Balise. It ran using **XML::Parser** as base XML parsing engine. Later, as one of José's students, I modified it to support a subset of XPATH and ported it to **XML::LibXML**, and so I became a co-developer of the module.

We ported it to **XML::LibXML** because **XML::Parser** development was stalled, libxml2 is faster, and libxml2 also provides ways of parsing HTML (non-XHTML even) files. When I install **XML::DT**, I can choose the expat (**XML::Parser**) or the libxml2 (**XML::LibXML**) backend.

A Simple Down Translation

Suppose I have my address book in XML, and I have the name, city, country, and email address for each person. The root element **book** comprises **address** elements which describe each person.

```
<?xml version="1.0"?>
<book>
  <address>
    <name>Alberto Simoes</name>
    <city>Braga</city>
    <country>Portugal</country>
    <email>ambs@cpan</email>
  </address>
  <address>
    <name>Jose Joao Almeida</name>
    <city>Braga</city>
    <country>Portugal</country>
    <email>jjoao@cpan</email>
  </address>
</book>
```

Now, I want to convert this to plain text. I start with **XML::DT**'s **mkdtskel** (make down translation skeleton) script which creates an initial script for me. I supply the name of an XML file, and based on the structure it finds, it dumps a skeleton of a perl script that uses **XML::DT** to process it. For my address book it outputs this short script.

```
#!/usr/bin/perl
use XML::DT;
my $filename = shift;

%handler=(
   # '-outputenc' => 'ISO-8859-1',
   # '-default'   => sub{"<$q>$c</$q>"},

   'address' =>   sub{ "$q:$c" },
                     # occurred 2 times
   'book'    =>   sub{ "$q:$c" },
                     # occurred 1 times
   'city'    =>   sub{ "$q:$c" },
                     # occurred 2 times
   'country' =>   sub{ "$q:$c" },
                     # occurred 2 times
   'email'   =>   sub{ "$q:$c" },
                     # occurred 2 times
   'name'    =>   sub{ "$q:$c" },
                     # occurred 2 times
);
print dt($filename,%handler);
```

This script has two main parts: the hash where I specify the desired actions for each element I process, and the **XML::DT** function that uses the hash and an XML file to do the down translation.

The hash has three types of keys: **XML::DT** specific configuration keys, names of tags in the XML document, and optional XPATH selectors which I discuss later.

The **%handler** hash contains two main configuration keys, both initially commented out (**-outputenc** and **-default**), and all the XML tag names that occur in the XML document.

For each tag name, the hash associates an anonymous subroutine where I specify the actions the script takes to process each tag. It uses three package variables: **$q** is the name of the current tag, **$c** is the contents of the tag, and **%v** is a hash of the attributes and values of the tag. For instance, in the tag

```
<name id="37">Alberto Simoes</name>
```

$q is **name**, **$c** is "Alberto Simoes", and **%v** is **(id = 1)**.

When I run this program on my skeleton program (without changes), I get my plain text version. The **dt** function simply made one format look like another.

```
book:
   address:
      name:Alberto Simoes
      city:Braga
      country:Portugal
      email:ambs@cpan

   address:
      name:Jose Joao Almeida
      city:Braga
      country:Portugal
      email:jjoao@cpan
```

Down Translating To Html

Now I want to down translate my address book to an HTML table with four different columns: name, city, country and email. For that, I simply change the **%handle** hash to output the right thing for each element. Remember, **$c** is just the content of the tag.

```
%handle = (
   name    => sub{ "<td>$c</td>" },
   city    => sub{ "<td>$c</td>" },
   country => sub{ "<td>$c</td>" },
   email   => sub{ "<td>$c</td>" },
   address => sub{ "<tr>$c</tr>" },
   book    => sub{ "<table>$c</table>" },
);
```

When I process the address and book tags, **$c** already includes the result of the other processing functions. The **book** tag contains the address tags, so its version of **$c** contains whatever the address processing returns. Since the address tag contains the name, city, country, and email tags, the value of **$c** for address is the concatenated values of those tags.

From the inside out (looking at it the other way), the name, city, country, and email processors create table cells. The address processor gets the concatenated result of all of those and wraps table row tags around it. The book processor gets the concatenated result of all the table rows and wraps a table tag around it.

> ***"toxml() creates the enclosing tags for me"***

I often create other HTML or XML tags inside these processing functions, so **XML::DT** includes an utility function named **toxml** that create the enclosing tags for me. The **toxml** takes three arguments: a tag name, an anonymous hash of attributes and values, and the content of the tags.

```
toxml( "td", {}, $c );
     #  <td>$c</td>
toxml( "foo", {a => "b" }, "bar" );
     #  <foo a="b">bar</foo>
```

To simplify things even more, if I call **toxml** without parameters it simply uses **$q**, **%v** and **$c**: So, if I call it with no arguments,

```
toxml()
```

I am doing the same thing as supplying these arguments

```
toxml( $q, \%v, $c )
```

I can change my down translator configuration hash to something much shorter. I simply want to change the tag names, so I set **$q** to the HTML tag name that I want, then call **toxml()** without arguments.

```
%handler = (
   name      =>   sub{ $q="td"; toxml },
   city      =>   sub{ $q="td"; toxml },
   country   =>   sub{ $q="td"; toxml },
   email     =>   sub{ $q="td"; toxml },
   address   =>   sub{ $q="tr"; toxml },
   book      =>   sub{ $q="table"; toxml },
);
```

The result is a bare-bones HTML table.

```
<table>
   <tr>
      <td>Alberto Simoes</td>
      <td>Braga</td>
      <td>Portugal</td>
      <td>ambs@cpan</td>
   </tr>
   <tr>
      <td>Jose Joao Almeida</td>
      <td>Braga</td>
      <td>Portugal</td>
      <td>jjoao@cpan</td>
   </tr>
</table>
```

Using Defaults

The **-default** directive defines a function to call in case there is no key with the name for the specified element. Instead of using the same anonymous subroutine to process the name, city, country, and email, I don't specify explicit processors for those. When **dt** encounters a tag name I haven't specified, it uses the processor in the -default key. In this case, I use the -default key to make table cells.

```
%handler = (
   address   => sub{ $q="tr";    toxml },
   book      => sub{ $q="table"; toxml },
   -default  => sub{ $q="td";    toxml },
);
```

I get the same output I did earlier, but with a lot less code.

Coverting XML to XML

Now, suppose I just want to change the XML format because I want the email tag to contain the address as an attribute value, like

```
<email uri="ambs@cpan"/>
```

This is a simple task. I don't want to change anything else, so the **-default** key handles everything except the **email** tag which has its own, explicit entry. In the email processor, I set the **uri** value in the **%v** hash, clear the value from **$c**, and call **toxml** without arguments.

```
%handler = (
   -default => sub{ toxml },
   email    => sub{  $v{uri} = $c;
                     $c=""; toxml },
);
```

Using XPATH

After using the module for a while, I wrote some functions that could do two different actions depending on context, attributes or some other condition that depended on the current position in the XML tree. Most of these situations involved selections very similar to XPath ones.

I added the (**<pathdt>**) function to take a subset of XPATH as keys of the configuration hash, and then rewrite them as a traditional configuration hash with some extra code to perform the selection.

The following table shows some of the subset of XPath supported by **XML::DT**. Check the **XML::DT** documentation for the full supported subset.

- To select the root element of a document I use simply its name. In fact, this is not correct, as I can have more elements on the document with that same name. With

  ```
  /book
  ```

  I select the root element of the document, if its name if **<book>**

- To select elements at a special depth, or with special parents, I can use a full path (using asterisks as placeholders):

  ```
  /*/address/*
  ```

this selects all elements child of **<address>** which, in turn, is child of the root element.

- If I want to select only the elements **<book>** with the attribute **<city>** defined, somewhere on the document tree

```
//book[@city]
```

and, if I want to force the contents of **<country>** to **<Portugal>**:

```
//book[@country='Portugal']
```

Defining Types

So far, my examples showed XML processing based on the creation of a new string constructed by concatenation and interpolation. This is just **XML::DT**'s default functionality. The definition of XML processing based on types can change this and give powerful capabilities to my XML processors.

I can to associate types to XML elements. For example, the simple **<ul>** or **<ol>** elements of XHTML contain sequences. On the earlier example of the address book, the **<address>** tag contains a map, from the child element names to their contents.

To show this, I want to convert RSS (Rich Site Summary) to HTML. The RSS DTD is simple: it has a header named **<channel>** with information about the RSS feed. The body is a sequence of news items named **<item>** with information like the news title, URL and description. I can get examples of RSS feeds in most news sites like slashdot or use.perl.org. This is an example from *The Perl Review's* feed of free articles.

```
<?xml version="1.0"?>

<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/↲
22-rdf-syntax-ns#"
 xmlns="http://my.netscape.com/rdf/simple/↲
0.9/">

  <channel>
    <title>The Perl Review</title>
    <link>http://www.theperlreview.com/
    </link>
  </channel>

  <item>
    <title>Perl One-liners, by Jeff
    Bay</title>
    <link>http://www.theperlreview.com/↲
Articles/v0i1/one-liners.pdf</link>
  </item>

  <item>
    <title>Cooking Perl with Flex, by
    Alberto Manuel Simoes</title>
    <link>http://www.theperlreview.com/↲
Articles/v0i3/flex.pdf</link>
  </item>

</rdf:RDF>
```

The header is a map containing all the meta-information. The same can be said about each **<item>**. We could, also, define a type for the sequence of **<item>** elements, but I choose not to do that for simplicity.

Then, our handle will start with:

```
%hdl=(
    -default => sub {$c},

    -type => {
              channel   => 'MAP',
              item      => 'MAP'
             },
     );
```

The default action for tags is to return its content. This is the best way to define a default action when using types. So, in the processing directives for **<channel>** and **<item>**, **<$c>** will contain a hash reference, where keys are the child elements names for that tags, and values are their processing contents (in this case, the literal contents, as the default action is to return it unmodified).

To process the RSS header (the **<channel>** element), I can define a **channel** processor:

```
'channel' => sub {
  h1(a(   { -href=>$c->{link}},
          img( -src    => $c->{image},
          -border => 0,
          -alt    => $c->{title}} ),
          $c->{description}
      );
  },
```

I use functions from **CGI** module to simplify the HTML construction. **<$c>** is an hash reference with keys for all the meta-information contained

in the **<channel>** header. In this case, I want just to use the **<link>**, **<image>**, **<title>** and **<description>** child elements to construct a header for the HTML page.

For each item, I have a similar action, using the **<link>**, **<title>** and **<description>** child elements:

```
'item' => sub {
  h3(a({-href=>$c->{link}},$c->{title})).
    blockquote($c->{description});
  },
```

Also, the RSS DTD can use an **<image>** tag both inside and outside the **<channel>** element. I want to ignore the one outside it. To do that, I can write the following XPATH processing directive:

```
'/*/image' => sub{""},
```

Given that all XML files contains exactly one root element, I can substitute the **<*>** for the root element name (**<rdf:RDF>**).

Finally, the RSS examples I looked at were not consistent regarding the URL of the images. Some used the **<rdf:resource>** attribute while some others used the **<rdf:about>**, so I defined a rule that used either one of them.

```
'image' => sub{ $v{'rdf:resource'} ||
                $v{'rdf:about'} },
```

I put it all together to create my down translator.

```
#!/usr/bin/perl
use CGI qw/:standard/;
use XML::DT;

%hdl = (
  -default => sub {$c},
  -type => {
    channel   => 'MAP',
    item      => 'MAP' },
  '/*/image' => sub{ "" },
  'channel' => sub{
    h1(a({ -href=>$c->{link}},
      img{ -src=>$c->{image},
           -border=>0,
           -alt=>$c->{title}}),
           $c->{description}); },
  'item' => sub {
      h3(a({-href=>$c->{link}},$c->{title})).
          blockquote($c->{description}); },
    'image' => sub{ $v{'rdf:resource'} ||
                    $v{'rdf:about'} },
);

print pathdt(shift(), %hdl);
```

When I run my RSS file through my down translator, I get the HTML version.

Other types

Here is a list of some types currently supported by **XML::DT,** together with a simple explanation of their use.

STR

This is the default type for all elements. The child element strings are concatenated and returned.

THE_CHILD

This returns the result of processing the only child of the element. It can be useful in cases like:

```
<foo>
  <bar>
    zbr
  </bar>
</foo>
```

On this example, **<foo>** is just an wrapper for **<bar>**. If I define **<foo>** type as **<THE_CHILD>** the result of processing the **<bar>** element will be returned.

SEQ

This defines the element as an array of sub-elements. It returns the reference to an array with all the child element contents. It ignores child element attributes.

SEQH

This is also a sequence, but I get more information about the child elements. Each array element is a hash reference where the key **<-q>** is the name of the child, the key **<-c>** is the child contents, and remaining keys are child attributes.

MAP

As seen in the example, this returns a reference to an hash where child element names are the keys, and contents used as the values. It ignores child element attributes.

MMAPON(element1,element2)

This is the same as **<MAP>**, but for some specific elements (for some keys), the value is an array reference. This is useful when there are a few elements with repetitions.

ZERO

The simpler type: simply return the empty string.

Tips and Tricks

The **XML::DT** module does a lot more than I have shown, and can process data from many other sources.

HTML processing

When installing **<XML::DT>**, it prompts me to use **<XML::Parser>** or **<XML::LibXML>** as my XML engine. If I choose the later one, I can process html documents as if they were XML documents.

To do this, simply add the key **<-html>** to the handler, with value 1.

```
%handler = (
             -html => 1,
             ...
           );
```

String processing

There are function variants to **<dt>** and **<pathdt>** to process strings. For that, I can use **<dtstring>** and **<pathdtstring>** which receive a string instead of a filename.

Remote document processing

I can grab a remote file, using **<dturl>** and **<pathdturl>** and passing an URL instead of a filename. This feature requires **LWP::Simple.**

Checking context

Sometimes it is useful to know the name of the parent tag. To do that, **<inctxt>** receives a pattern and return a true value if the current element path matches the provided pattern. This function is meant to be used in the element functions in order to achieve context dependent processing. □

References

XML::DT

http://search.cpan.org/dist/XML-DT

Omnimark Technologies

http://www.omnimark.com

Balise, AIS Software

http://www.balise.com

"XML::DT - a Perl Down Translation Module"

http://www.infoloom.com/gcaconfs/WEB/granada99/dia.HTM#DIA-002

Module::Release and Beyond

brian d foy
comdog@panix.com

In the middle of 2002, I was straining under the load of maintaining a lot of modules. Open source and sharing is great, but a responsible open source contributor also maintains the stuff he makes public. I needed a way to conserve my time and automate anything that I could.

I looked at a lot of the mistakes that I was making, and a good deal of them had nothing to do with coding logic or syntax. I wasn't consistently creating complete distributions, I was letting old versions of files sneak into my uploads, or I was forgetting to include files in MANIFEST.

I needed to improve my process, which really means that I needed to have a process. I had to figure out what I wanted to do before I allowed myself to release a module. I listed several things I needed to do, and some are specific to me (more later).

- run make test to ensure tests pass
- test that I list all prerequisites in Makefile.PL
- test the documentation for formatting errors
- run make disttest to ensure all necessary files show up in the distribution
- ensure everything into source control
- update the Changes file
- package the distribution
- upload to PAUSE and Sourceforge
- ensure that uploads worked

I should go through all of these steps every time, but I shouldn't have to do it manually. Anything that I do myself is something that I have a chance to screw up or forget. I need to automate this so it happens every time, and in the same way each time.

I started working on a script to do this, and it was a script in the true sense of the word: a series of steps executed in sequence, just like two actors reading lines. I lifted this code from the first public version of what I started calling "release", which was simply comprised the bits of code I had developed to implement my process. It's the first step in my process.

```
TEST: {
   print "Checking make test... ";
   unless( -e 'Makefile.PL' ) {
      print " no Makefile.PL---skipping\n";
      last TEST;
   }
   my $tests = `make test 2>&1`;

   die "\nERROR: Tests failed!\n" .
      "$tests\n\nAborting release\n"
      unless $tests =~ /All tests successful/;

   print "all tests pass\n";
}
```

The code really just pretends to be me. It calls external programs just like I would do them when I did it by hand. It stops the process when a step fails, and it prints progress messages so I know something is happening.

To release a module, I type the command and sit back while it does its magic.

```
% release
```

It goes through all the steps, doing everything for me and doing it the same way each time. Not only do I have a process, but it's consistent and repeatable. Everything happens the same way each time and I don't forget to do anything. If something isn't right (the most common being an out-of-date CVS repository), it stops and makes me fix it before it goes on. Many of my distributions problems never showed up again.

The first public version, release-0.10 (which is still in BackPAN and on Sourceforge) came out in December 2002. In January 2003, I suddenly disappeared from

life for a bit while I was in the Army, but I arranged for Andy Lester to touch and maintain any of my projects that needed attention.

Through that, Chris Nandor (of use.perl and OSDN) made some changes to the initial script so he could use it. I had programmed it for me, so he took the brian parts out of it and added some of his own. Then, Max Maischein cleaned up a lot of the system specific bits so the script could pull information from environment variables and Config.pm

To make that even easier, Ken Williams turned release into Module::Release. Users could now write their own process with whichever steps they liked (and without the steps they didn't need).

Now that it was a module, other people could add to or extend it. Nik Clayton added bits to use subversion instead of CVS, which I use because Sourceforge uses it. He also added extensions to make it work with Module::Build.

After about 15 months, I was out of the Army and back in the civilian world, and Module::Release was on my mind. I had seen some of the progress, although I hadn't looked at the code or used it. I was glad that it was useful to other people, and wanted to work on it some more.

Up to that point, the entire process revolved around a conventional Perl module distribution: that's even in the name Module::Release. What I really need is a more general approach, so I started on plans for Module::Release 2.0.

The first goal is extendability. Anyone who wants to use the new version should be able to easily add steps that I haven't thought about, never expected, or don't want to use. It shouldn't depend on how I do things. Ken Williams did a lot of this work already, so I just need to compartmentalize things in their own namespaces: one for CVS, one for Makefile.PL functions, one for Module::Build functions, and so on. People should be able to add their own, too.

The second goal is to forget about modules. It's really just a workflow, so it doesn't need to revolve around Perl module distributions. Although I doubt developers of other languages will use the new Module::Build, they should be able to. I could also use it to publish a website, for instance, since that has many of the same steps as a code distribution: test, store in source control, and publish.

I've been fortunate enough to present this stuff at a couple of Perl Mongers meetings and collected a lot of feedback: cheers for the good ideas and raspberries for the bad ones.

From all that, I'm announcing Starburst, the next generation of Module::Release. I have it partially working on my machine, but I'm not eating all of my own dog food just yet. It's a base module that loads plugins at run time so your script doesn't need a long list of modules at the top. A single line pulls it all in.

```
use Starburst;
```

From there, each plugin can export functions that I can string together to create my process. It's nothing fancy, and uses plain old Perl whenever possible.

```
cvs_up_to_date();
my $file = make_dist();
ftp_upload( $site, $file );
```

Each plugin can export functions, and behind the scenes Starburst.pm does a little delegation magic to handle it all without any design goofiness in the plugins. I want the newest module writer to be able to hack up a plug-in without having to know too much Perl voodoo.

Starburst will come with a lot of plugins, and that's what I'm working on right now. I'm partitioning Module::Release into several plug-ins, and creating a few new ones, such as a version of Chris Nandor's function to announce new distros to use.perl.

I expect to have something on CPAN by the end of the year, and I'd be more than happy to show up at Perl Mongers meetings to demonstrate it. It's saved me a lot of time and hassle, and I think it can do the same for you. Until then, Module::Build and release work as well as they always have. □

*brian d foy is the original author of release, and the publisher of **The Perl Review**.*

Perl Calendar

Oct. 12, 2004

DBI's 10th Birthday
http://dbi.perl.org

Oct. 17, 2004

Perl 5's 10th Birthday
Perl was first released in 1987

Nov. 22, 2004

Randal Schwartz's Birthday
Randal was first released in 1961

Nov. 26, 2004

Perl 5.8.6 available

Dec. 1–3, 2004

Open Source Developer's Conference
(formerly YAPC::AU::2004)
Monash Caulfield
http://www.osdc.com.au/

Dec. 11, 2004

London Perl Workshop
Imperial College Student Union
http://london.pm.org/lpw/

Feb. 9–11, 2005

German Perl Workshop
Dresden, Germany
http://www.perl-workshop.de/2005/

Feb. 25, 2005

Dutch Perl Workshop
http://workshop.perlpromo.nl/

March 14–17, 2005

O'Reilly Emerging Technology Conference
San Diego, CA
http://conferences.oreillynet.com/etech/

March 26–27, 2005

YAPC::Tapei—Perl in the Enterprise
http://taipei.pm.org/

Aug 31 – Sept. 2, 2005

YAPC::EU::2005—Perl Everywhere
University of Minho, Braga, Portugal
http://braga.yapceurope.org/

Perl News

White Camel Awards

http://www.perl.org/advocacy/white_camel/2004.html

At the Open Source Convention, The Perl Foundation awarded the 2004 White Camel awards for contributions to the Perl community. They recognized Jon Orwant (first publisher of The Perl Journal), Dave Cross (Perl Mongers and London.pm), and brian d foy (founder of Perl Mongers and publisher of this magazine).

Grand Master Larry Wall

http://www.opensource.org/osa/awards.php

At the Open Source Convention, the Open Source Initiative awarded Larry Wall its Grand Master medal, but for writing the patch program, not perl.

BBEdit 8.0 released

http://www.barebones.com/products/bbedit/

Bare Bones released a major upgrade to their popular Mac text editor. It integrates with Affrus, a new Mac Perl debugger, and adds Perforce integration.

Komodo 3.0 released

http://www.activestate.com/Products/Komodo/

ActiveState released another version of their integrated development environment. It supports many languages besides Perl and runs on Linux, Solaris, and Windows.

DBI Call for Funding

http://dbi.perl.org/donate/

Support the continued development of the DBI project with a monetary donation. The Perl Foundation set up a way for people to donate to the further development of DBI.

Apress LP Sneak Peeks

http://www.apress.com/betabooks/

Can't wait for a book to come out? Apress LP now gives you sneak peeks of upcoming titles in its "Alpha and Beta Books" program. You can view HTML or PDF versions of developing parts of books as well as post comments on the previews.

Perl Software Design mailing list

http://www.metaperl.com/sw-design/

Terrance Brannon has started a mailing list for "the discussion of software design in general, and software design with Perl in particular".

Parrot 0.1.1 "Piocephalus"

http://www.parrotcode.org/

Wonder what Piocephalus is? We did too, but it's just another kind of parrot, and it's the name of the latest version of Parrot, which is the engine that will run Perl 6 and PONIE.

Perl 5.8.6 released

http://www.cpan.org/src/stable.tar.gz

The latest stable version of Perl is now 5.8.6. See the perl586delta.pod for details, which includes this curious statement "Most C source files now have comments at the top explaining their purpose, which should help anyone wishing to get an overview of the implementation." If you've been scared off internals development because you couldn't tell which is what, you have to find a new excuse.

TAP: Test Anything Protocol

http://use.perl.org/~petdance/journal/22057

Andy Lester choose a name for the way that Test::Harness does things: watching tests by printing "ok \d" or "not ok \d" is now the "Test Anything Protocol", or TAP, thanks to suggestion from Joe McMahon. Any thing, in any language, can follow the protocol and Test::Harness can analyze the results. For instance, PHP users can use Test::Harness to test their stuff (which Geoffrey Young and Chris Shiflett are working on with Apache-Test).

The Perl Review now accepts credit cards

http://www.theperlreview.com/subscribe.html

We were stuck with PayPal before, but now we've worked out everything with our bank and we can do credit card processing directly. You can subscribe using MasterCard, Visa, or America Express.

Correction: In some versions of the last issue, we inadvertently called Denis Kosykh "David". We apologize for the error.

New Modules

http://www.cpan.org

CPAN::Mini turns Randal Schwartz's minicpan script into a module. Decrease your local CPAN footprint by a factor of schwartz.

Google now allows POP access to its mail service GMail, and **GMail::Checker** uses it to see if you have mail. You may also want to see **WWW::GMail**, **Mail::Webmail::Gmail** and **WWW::Scraper::Gmail**.

Ever want to use Dijkstra's guarded commands in Perl? Now you can with **Commands::Guarded**. Why you would do this is up to you.

Feed::Find can look at a web resource and figure out if there is a syndication feed associated with it. It looks at the content type or information in HTML to make its guess.

Geo::Postcode accepts full or partial UK postcodes, validate them, separate them into their significant parts, translate them into map references and calculate distances between them. Older modules do similar things:in the US, **Geo::Coder::US** can estimate longitude and latitude for US addresses. For those living a completely digital life, **Geo::IP** can translate an IP to a country.

Telephony::CountryDialingCodes can internationalize your war dialer by looking up dialing codes by country or the other way around.

Test::MockModule can override real module functions so you can test them.

Chart::Scientific can make 2D plots from Perl Data Language (PDL) structures

Functional Perl Programming

Frank Antonsen
frankantonsen@netscape.net

Functional programming (FP) is based on the definition and evaluation of functions. You do everything by evaluating functions, and you rarely use variables or typical iteration constructions such as **`for`** or **`while`** loops. This style of programming minimizes side effects, is easy to optimize and refactor, and allows the you to build up complex functions out of simpler ones.

Perl, however, is a multiparadigm programming language, meaning it does not force you to program according to a specific paradigm. It has elements of functional, procedural, object-oriented, and other sorts of designs in it. You can program any way that you like, and you can use the good parts of each style to their greatest advantage. Perl isn't a FP language since it isn't based on the definition and evaluation of functions, but you can still use some of the ideas of FP.

A common feature of FP is list processing. In fact, Lisp, the very first FP language, was invented in 1959 for precisely that purpose. It's name stands for "list processing". In academia, the most widely used functional languages are ML (for Meta Language), and Haskell.

Functional languages can in general achieve a higher level of optimisation than non-pure FP languages, and this is one of the major reasons for their importance in academic circles. The syntax of these languages is easier to master than that of Lisp. One could say, that Lisp has a pure functional *syntax*, whereas ML and Haskell borrow the notation from the Pascal/Ada/Eiffel branch of languages.

FP is characterised by a number of features, but I only have room to cover some here.

- Lack of side effects
- Iteration instead of recursion
- Higher order functions
- Composition of functions

No side effects

A side-effect is something causing a change in the program's environment, such as writing to a file or the screen or modifying certain global variables. Many statements in Perl, and most other languages, have side-effects. Perl is full of side effects since so many functions set variables. For instance, the match operator can set several variables, including **`$``**, **`$&`**, **`$'`**, and the memory match variables **`$1`**, **`$2`**, and so on. A global match in scalar context also sets the value returned by **`pos()`**. An assignment to a variable is a side effect.

Pure functional languages are a subset of FP and have no *side-effects*. Since side-effects have been eliminated everywhere else, the compiler can make a large number of optimisations. What makes optimisation of non-FP programs difficult is the fact that variables can have non-trivial scopes.

In this way, many side effects have been avoided—you can be assured that any variable you may declare anywhere in your program also has the same meaning or value anywhere else. This makes FP-programs "safer" than procedural ones, and is one of the main reasons why it easier to *prove* the correctness of an FP-program than a procedural one. This also accounts for the greater interest among academics for FP languages.

Recursion, not iteration

FP languages use recursion instead of iteration. Functions call themselves to repeatedly do something. Most Perl idioms use iteration instead of recursion, but you can turn an iterative solution into a recursive one.

All of Perl's iteration commands can be built up from a **`while`** loop.

The standard **for** loop

```
for( INIT; CONDITION; INCREMENT ) {
     ... # some code
}
```

is just another way you write this **while** loop.

```
INIT;
while( CONDITION ) {
     ... # some code
     INCREMENT;
}
```

To see how you can begin to turn this into a program using recursion instead of iteration, you can rewrite it once more as

```
INIT;
LOOP: {
     ... # some code
     INCREMENT;
     redo LOOP if CONDITION;
}
```

Now, this almost has the form of a recursive subroutine, so you make the final alterations to turn it into an FP-style "loop by recursion":

```
sub loop {
     my $i = shift;
     ... # some code
     return if CONDITION;
     loop( $i+1 );
}
```

Another feature of pure FP is the lack of conditionals. You can easily get rid of the **if** with a short circuit operation. You replace

```
return if CONDITION;  # conditional return
```

with the condition as the first part of the short circuit and the return as the second part. The **return** happens only if the condition is true.

```
CONDITION && return;
```

My **loop** function becomes:

```
sub loop {
     my $i = shift;
     ... # some code
     CONDITION && return;
     loop( $i+1 );
}
```

This is essentially how the loop-construct above would be written in Prolog or Miranda (except, of course, that the notation is different).

But, in fact, you can do much more. Suppose the body of the loop is written as a single function (which you can always arrange).

```
foreach $i ( LIST ) {
     body($i);
}
```

You can rewrite that using the **foreach** as a statement modifier.

```
body( $i ) foreach $i ( LIST );
```

But it could also be written even more "functionally" by using **map** to apply the function to each item in the list. This is a purely functional way of doing loops.

```
map body( $_ ), LIST;
```

Furthermore, if you think of a program as a list of statements, then you can even run the entire program by a call to **map** combined with **eval**:

```
map eval( $_ ), @program;
```

Just like in Lisp, you can consider a program as just another list, and you can run it by applying a special function to it. Indeed, that is what FP is all about.

Higher order functions

A higher order function takes another function as an argument. Perl comes with some higher order functions built-in: **map**, **grep**, **sort**, and **eval** take code blocks as arguments. These go by different names in different FP languages.

There is one more important higher order function found in most FP languages but absent in Perl (interestingly enough, it has been added to Python). This function is often known as **fold** (ML and Haskell has two versions, **foldl** and **foldr**, for "folding" from the left and right respectively), elsewhere in the FP literature, it may also be known as **reduce**, which is the name Python uses.

To fold a list, you take two arguments and apply a function to them. You take the result of that operation and another element from the list and do it again, and go on until you are done.

Using the **add** routine from earlier, you can add the list of numbers **(1,2,3,4,5)**. You can write this out as a series of chained function calls.

```
my $sum = add( 1, 2 );
   $sum = add( $sum, 3 );
   $sum = add( $sum, 4 );
   $sum = add( $sum, 5 );
```

You can write this without the temporary variable **$sum**.

```
add( add( add( add(1,2), 3 ), 4), 5 )
```

and so on for more than five elements in the list. That's too much typing though.

You can write your own **fold** routine in Perl, although you need a few utility functions first. You want to process a list, so you need a way to break that list into the first item, or "head", and the remaining items, or tail. You can't modify the list, so you have to work with copies of it. My **head** function returns the first item in `@_`, and the tail function returns the rest of the items, if any. Neither of them affect any variable or have a side effect. You add a **len** subroutine to return the length of a list.

```
sub head { shift }
sub tail { shift; return @_ }
sub len  { scalar @_ }

sub fold {
     my( $f, @a ) = @_;
     return head( @a )
     if len( tail( @a )  ) < 1;

     my ($a, $b) =  ( head( @a ),
                      head( tail( @a ) ) );

     fold( $f, ( $f->($a,$b),
             tail( tail( @a ) ) ) );
}
```

You can write this even more tersely by replacing the occurrences of the dummy variables **$a, $b** by their definitions **(head @a, head tail @a)** in the recursive step. This would make the program harder to read, though, and, in any case, the use of dummy variables in this way is entirely consistent with FP practice.

To calculate the factorial using **fold** you can now define a function **mult** and use this as its first argument.

```
sub mult { $_[0] * $_[1] };
print "Result is ",
       fold( \&mult, 0 .. 10 ), "\n";
```

Since this is really just a utility function you are not likely to need elsewhere in the program, it is better to take advantage of another FP-construction also present in Perl: the *anonymous functions*. Whereas ordinary functions are globally defined, and hence "stick around" for the entire lifetime of the program, anonymous functions are "use once and throw away" functions.

Most FP languages introduces a special keyword for defining anonymous functions (typically called **lambda**), but Perl uses a much more simple, and natural approach: you simply omit the function name in the definition. Ironically, in this sense, Perl treats functions much more on an equal footing with other data structures than do most FP-languages.

Hence, you can rewrite the factorial computer with an anonymous subroutine.

```
my $n = fold( sub{ $_[0] * $_[1] }, 1..10 );
```

You can get even better by hacking on the definition of **fold** so you can make it work like **grep** which takes a block of code. The **fold** functionality stays the same, but you add a prototype saying that the first argument is a code reference. The perlsub man page explains prototypes.

```
sub fold(&@) { ... }
```

Once you have the prototype in place, you can make the statement even shorter. It even looks more like Perl now. Be careful that you don't have a comma after the code block. Perl will warn you about this for **grep** or **map**, but not for your own defined functions.

```
my $n = fold { $_[0] * $_[1] } 1 .. 10;
```

Composition and currying

Probably one of the easiest way of generating new functions from old ones is through *composition*. Composition turns two or more functions into a single call.

You can combine the functions **foo** and **bar** into a single function **baz**. In this example, the result of **bar** is the argument list for **foo**, but that is hidden behind **baz**.

```
sub foo { ... }
sub bar { ... }
sub baz { foo( &bar ) }
```

You could repeatedly have **foo(&bar)** statements in your code, but when you decide to change it, you have to change all of them. When you hide all that behind another function, not only is your code shorter but you have a single point of maintenance.

You can do this with anonymous functions too. The **f** function returns the square of its arguments, and the **g** function returns the value of its argument increased by one.

```
my $f = sub { $_[0] ** 2 };
my $g = sub { $_[0] + 1  };
```

If you want to square the argument plus one (given 5, for instance, square 6), you can compose the functions into one as before. The function **h** does it all in one step.

```
my $h = sub { $f->( &$g ) };
```

It is this possibility of combining functions (similar to how Unix combines processes through pipes) which can make FP very powerful. In most practical programming problems, you divide the problem into smaller sub-problems solved in functions. You build up more complex behavior by combining functions.

Another common way of creating new functions from old ones is through *partial instantiation*. A function of two variables may be turned into a function that takes one argument by fixing one of the arguments. This is also often referred to as "currying" of functions, after mathematician Haskell Curry, who has really maximised his impact on FP terminology: his first name is used for an programming language and his last name for a common technique!

Suppose you have a function to add two numbers.

```
sub add { $_[0] + $_[1] }
```

Now you want a function to just add one to a number, like the earlier anonymous function **$g**. You already have a function to add numbers, and you can reuse it by currying it. You fix the second argument at **1** by wrapping it in another

```
my $g = sub { add( 1, $_[0] ) }
```

A common perl idiom for this is a closure, which is just an anonymous subroutine that references a lexical variable that has gone out of scope. For this example, you create the anonymous sub in a scope defined by the **do** block and decide the value for partial instantiation at run time.

```
my $g = add_curry(1);
my $h = add_curry(5);
my $plus1 = $g->(3);  # 3 + 1 = 4
my $plus5 = $h->(4);  # 4 + 5 = 9

sub add_curry {
     my $offset = $_[0];
     sub { add( $offset, $_[0] ) };
}
```

Conclusion

Although Perl is not a functional programming language, it is flexible enough to let you fake it, at least partially. I couldn't cover everything, but functional programming depends on functions without side effects and uses them to build up more complex behavior. You don't have to program like this, but you can do what Perl does: steal what you find useful. □

Faking Stored Procedures

Zach Thompson

hideo@lastamericanempire.com

Perl's DBI module has become the tool of choice for those squaring off with databases with Perl, and, in my current post, I find myself doing exactly that. I process a lot of data, and to get it down quickly and efficiently, I create dynamic DBI "stored procedures" in my scripts. The stored procedure is a just a series of database instructions that I run as a group. I implement these directly in my scripts as Perl closures.

We have legions of people downloading and creating data, filling directories full of hundreds, or sometimes thousands, of files that must be parsed and crammed into an Oracle database. We have several types of files that we must handle slightly differently, although they still belong to the same set of data and we should process them in one shot by the same program. I carve up these unwieldy chunks of data into hundreds of thousands of bite-sized morsels for our customers to consume.

With thousands of files to process, I have to consider efficiency and resource concerns, especially if my program is run from cron with hundreds of other programs waiting for their 15 seconds of fame. The hacking brethren before me resorted to the Oracle "sqldr" tool, figuring Perl was simply not up to the task of handling large volumes. I've never liked backticking the tools that ship with the various databases when there is a Perl solution that can do it.

Of course, I don't want to process several thousand files into memory only to have the first INSERT statement fail. Even with DBI's "connect-and-prepare once, execute many" model, such a strategy could prove to be a miserable waste of time, and depending on the size of the files, my system or its administrator may not appreciate my gross memory transgressions. Such errors would be a little less devastating by committing a file's worth of data at time.

```
my $dbh = DBI->connect($dsn,
     {
        RaiseError => 1,
        AutoCommit => 0,
        PrintError => 0,
     });

my $insert_stmt = $dbh->prepare(
   q{ insert into table (
        col1,
        col2,
        col3,
        col4,
        col5
     ) values ( ?, ?, ?, ?, ? )
   } );

for my $file ( @files ){
   my $data = parse_file( $file );
   $insert_stmt->execute( @$_ ) for @$data;
   archive_file($file);
}
$dbh->disconnect;
```

For simple programs, this approach works. However, when I add statement handles to look up values, multiple **INSERT** statements, transactions, or even more complex SQL statements, the DBI code overwhelms the main flow of my program in a hurry. I have had to modify programs with the main logic embedded in a mess of DBI, and I can say that it is not the most maintainable structure.

It would be much cleaner if my main processing loops could stand on their own, despite the complexity of the DBI code. I can put the DBI stuff somewhere else.

```
for my $file ( @files ) {
   my $data = parse_file( $file );
   insert_data( $data );
   archive( $file );
}

sub insert_data { ...DBI code as before... }
```

I exiled the DBI code to the **insert_date()** subroutine. The flow of the program is cleaner, and the database code is now in a single spot, which makes it easier to maintain in the future.

However, even if I connect to the database earlier in the program, I now have the problem of preparing for every file whatever statement handles I need in **insert_data()**. We need a way to save our prepared statements between calls to **insert_date()** without putting them in the main flow of my program.

What is a Closure, Anyway?

Not long ago, in a company far, far away I mucked around on Microsoft SQL servers creating stored procedures at will to handle the raw data my programs had coaxed from the ether. A stored procedure is simply a set of SQL statements that I group together to do a particular task. The stored procedure treats a bunch of SQL statements as one operation. I save these in the database server, and they act as shorthand for their complex task: functional calls for databases, if you will.

Stored procedures were an effective way to greatly simplify the interaction between my programs and the database. In fact, after connecting to the database with Win32::ODBC, the whole job usually boils down to a single **if** statement.

```
if( $con->Sql(
   'exec stored_procedure list of data'
   ) )
{ ..handle error.. }
```

As long as the list of data I passed in was in the right order, I would never need to know what monstrous collection of Transact-SQL statements **stored_procedure** hid.

If I cannot create stored procedures, either by policy or missing features, I can still play database developer and fake it in my code by using closures with DBI. A closure is just an anonymous subroutines that references a lexical variable that has gone out of scope.

```
my $incrementer = print_increment();
for my $word ( qw/what will happen?/ )
{
   $incrementer->($word)
}

sub print_increment{
   # initialization #
   my $call_count;
   # closure #
   return sub {
      print ++$call_count, ': ', $_[0], "\n";
   };
}

# OUTPUT:
# 1: what
# 2: will
# 3: happen
```

The call to the anonymous subroutine referenced by **$incrementer** prints the value of **$call_count** as well as the string passed in as a parameter. Normally, lexical variables such as **$call_count** would be swept into the abyss at the end of their scope: in this case the end of **print_increment()**. However, because the returned subroutine reference has access to **$call_count** is value is available until the closure, **$incrementer**, goes out of scope.

I can use this technique, albeit a little more complex, to effectively turn your DBI code into a stored procedure for the duration of your program. The general structure of **print_increment()** comprises two sections: initialization and declaration. The initialization section houses all of the variables that the closure needs, and the declaration creates the closure.

Using Closures with DBI

Suppose I have files containing stock exchange information. Each file contains the data for a specific date with one line per company. From each line in the file, I extract 10 values: a company name, date, day high, day low, open price, closing price, volume, 52-week high, 52-week low, and the percentage change for the day. In other words, from the for loop I showed earlier, **parse_files()** will return an array reference where each element itself is an array reference of the form with those ten items.

I insert each array reference as a single row into the table "**stock_prices**". However, I also need to look up the appropriate ticker symbol for each company in the table "**ticker_map**". I need to do

several things for each row, but I can bundle those into my Perl stored procedure.

The bare bones structure is very similar to the previous closure example. The initialization section will contain our prepared statement handles, and the closure loops through the rows in **`$data`** while providing transaction handling by using **`eval`**:

(See Listing 1, below.)

Initialization

The first half of **`init_stock_inserter`** is a typical setup, except for the fact that I pass in the database handle. This is really the only loose end preventing me from quarantining all of my database code in **`init_stock_inserter()`**. So, why did I create the connection elsewhere and passed it in? There are really two reasons.

First, if I wanted to create multiple inserters by

Listing 1:

```
 1: sub init_stock_inserter {
 2:    my ( $con ) = @_;    # Pass in connection
 3:
 4:    # Declare and initialize lexical variables that we want our
 5:    # anon sub to access
 6:
 7:    my $get_ticker = $con->prepare(
 8:       q{ select ticker from ticker_map where company = ? }
 9:    );
10:
11:    my $insert_stock_prices = $con->prepare(
12:       q{ insert into stock_prices (
13:             ticker, company, date, day_high, day_low,
14:             open, close, volume, year_high, year_low, change
15:          ) values (
16:             ?,    ?,    TO_DATE(?, 'YYYY/MM/DD'),     ?,    ?,
17:             ?,    ?,    ?,    ?,    ?,    ?
18:          )
19:       } );
20:
21:    return sub {
22:       my ( $data ) = @_;
23:       eval {
24:          for my $row ( @$data ) {
25:             $get_ticker->execute( $row->[0] );
26:             my $ticker = ( $get_ticker->fetchall_array() )[0];
27:             $ticker or die join( ' ', 'No ticker for', $row->[0], 'found' );
28:             $insert_stock_prices->execute( $ticker, @$row );
29:          }
30:          $con->commit;
31:       };
32:
33:       if( $@ ) {
34:          $con->rollback;
35:          $con->disconnect;
36:          notify( 'Insert error: ' . $@ );
37:       }
38:    };
39: }
```

calling a similar inserter subroutine several times, I would need to create a connection each time. For instance, if I had to process two different types of files, say, NASDAQ and NYSE, the data from each type might be inserted into its own table with its own set of statement handles, though still using the same database connection. So, passing in the database handle saves me multiple connections.

Second, I really have no good way to explicitly disconnect when I'm finished with the handle. I could have a second parameter to the subroutine reference, which would indicate that the **$data** coming in is from the last file in the queue, but I would also need to keep track of the number of files outside of **init_stock_inserter()** in addition to making sure I don't re-create or delete the connection when calling the inserter the second time. And that's just plain ugly.

I can easily hide the connection code away with the other subroutines, so it's not a big deal.

```
sub make_DB_connection {
   return DBI->connect( ...,
      {
         RaiseError => 1,
         AutoCommit => 0,
         PrintError => 0,
      }
   );
}
```

Next, I prepare two statement handles with placeholders for the data One handle I use to obtain the ticker symbol and the other to insert the new found ticker and associated data. Nothing special here.

The Closure

The **return()** statement is where things get interesting. The only parameter my sub needs is the data itself, which I pass in for each file. Since I use references to arrays for each row, I know the order element order maps to the right placeholders. This works as long as **parse_file()** returns data in the same order as the placeholders.

I implement transactions in DBI using an eval block. Everything the closure does is enclosed within that block. For eval to manage the transaction, when I create the DBI object, I set **RaiseError** to 1 and **AutoCommit** to 0.

RaiseError causes any database error to die with the specific database error message. The eval block exits and sets **$@** with the error message.

When **AutoCommit** is false, I can roll back any rows I may have already been inserted. The scope of my transaction is be at the level of an entire file: I either insert all of the data from that file or none at all. This prevents me from having to determine whether a particular company's data has already been inserted for a specific date if my program chokes midway through a file.

The for loop simply iterates through the rows of data. Both **$get_ticker** and **$insert_stock_prices** are lexical variables (as was **$call_count**) and are available through the variable to which the returned sub reference is assigned.

For each row of data, I fetch the associated ticker using the company value and ensure that it's not undefined or empty. The final statement in the loop simply calls my insert handle with the ticker and the array to which **$row** refers. My call to **$insert_stock_prices** effectively becomes

```
$insert_stock_prices->execute(
   $ticker,  company,
   date,     day high,
   day low,  open price,
   closing price,
   volume,
   52-week high,  52-week low,
   change
);
```

Once I finish the for loop, I am certain that all of the rows passed in were inserted successfully, and I can commit my transaction to the database. **$con**, too, is just a lexical vari- able and is preserved between invocations of the subroutine reference. All of the database handle methods, such as **commit()** and **rollback()**, are available too.

The final component in my inserter provides the error handling for a failed transaction. I need to rollback any rows I already inserted, close the database connection, and then call a general error handling routine. After closing the database connection, the script should die also since the closure no longer has a database connection.

The Closure in Action

Using `init_stock_inserter()`, I arrange the main flow of my program in a way that will make my grandchildren giddy when it comes time to maintain it.

```
1: my $con = make_DB_connection(); # Connect
2: my $stock_inserter =
     init_stock_inserter( $con );
3:
4: for my $file (@files){
5:   my $data = parse_file($file); # Parse
6:   $stock_inserter->( $data );   # Insert
7:   archive( $file );             # Archive
8: }
9: $con->disconnect;        # Disconnect
```

I reduced the main flow to less than ten lines. My simulated stored procedure boils down to two lines of code. On line 2, I call `init_stock_inserter()` to create my closure, which I store in `$stock_inserter`. I execute the closure on line 6 by passing in `$data` that I freshly extracted from one of the files with `parse_file()`. Once `$stock_inserter` gets a reference to some data, all of its primed statement handles are brought to bear on conducting the data to its new home.

Further Fine Tuning

Although the initialization section in `init_stock_inserter()` was fairly simple, it doesn't necessarily have to be. I can add all the database related code (other than the connection itself).

For example, if the stock exchange files I process always contained the same 10 companies in each file, I can look up their ticker symbols during initialization and store them in a hash. I use the company field for each row to retrieve the ticker from the hash without having to talk to the database.

```
sub init_stock_inserter{
   my ($con) = @_;
   my $tickers = get_tickers($con);
   ...
}
```

`get_tickers()` contains the code to perform the lookups from the `ticker_map` table. When I execute `$insert_stock_prices` in the closure, I retrieve the ticker from the hash.

```
$insert_stock_prices->execute(
   $tickers->{ $row->[0] }, @$row   );
```

Listing 2:

```
 1: sub init_stock_inserter{
 2:  my ($con) = @_;
 3:    my %tickers;     # hash to hold the tickers we've already retrieved
 4:   ...statements as before
 5:
 6:   return sub{
 7:    my ( $data ) = @_;
 8:    eval{
 9:       for my $row ( @data ) {
10:          my $company = $row->[0];
11:          unless( exists $tickers{$company} ) {  # check if we've already looked it up
12:             $get_ticker->execute( $company );
13:             my $ticker = ( $get_ticker->fetchall_array() )[0];
14:             $ticker or die join(' ', 'No ticker for', $company, 'found.' );
15:             $tickers{$company} = $ticker;
16:          }
17:          $insert_stock_prices->execute($tickers{$company}, @$row);
18:       }
19:    };
20:  };
21:  ...error checking as before
22: }
```

Even in my original implementation of **init_stock_inserter()**, I could dramatically reduce the number of lookups from the **ticker_map()** table by storing the ticker symbols as we go.

(See Listing 2.)

Now the lookup code, lines 12–15, is only executed once for each company, and the ticker is stored in **%tickers** for future reference. As with our statement handles, the closure will preserve the state of **%tickers** between invocations. If I were processing a year's worth of files, this might save me a few hundred lookups for each company, depending on how often data appears for that company in the files.

Creating Multiple Inserters

Creating multiple inserters is easy enough: I just call my init subroutine for each inserter I wish to create. With a little modification to our main loop and the initialization section of **init_stock_inserter()**, I could have my program handle several file types with different database requirements. I might iterate through each file type as follows in our main loop.

```
my $con = make_DB_connection();

for my $type (qw/NYSE NASDAQ/){
   my $stock_inserter =
      init_stock_inserter($con, $type);
   for my $file (@files){
      my $data = parse_file($file);
      $stock_inserter->($data)
      archive_file($file);
   }
}
```

Now I loop through both New York Stock Exchange (NYSE) and NASDAQ type files. These may differ in file structure, lookup tables referenced, and the tables that ultimately stores the data. Nevertheless, the steps in processing the data remain the same. I now pass the type of file being processed to the init routine in addition to the database connection. This allows my init routine to distinguish which type of file it is expected to handle and to initialize the closure appropriately.

```
sub init_stock_inserter{
   # Note: SQL statements truncated
   my ($con, $type) = @_;
   my ( %tickers, $insert_stmt,
        $ticker_query);
   if($type eq 'NYSE'){
      $insert_stmt =
         'insert into NYSE_prices...';
      $ticker_query =
         'select ticker from NYSE_ticker_map...';

   }  elsif($type eq 'NASDAQ') {
      $insert_stmt = 'insert into NASDAQ_prices';
      $ticker_query =
         'select ticker from NASDAQ_ticker_map';

   } else {
      notify( 'Invalid file type specified: '
              . $type )
   }

   my $get_ticker =
      $con->prepare( $ticker_query );
   my $insert_stock_prices =
      $con->prepare( $insert_stmt );
   ....as before
}
```

The major difference in this new version is that I use a chain of conditionals to determine the specific SQL statements to use according to the file type. Once I assign the SQL statements, I prepare the same two statement handles I did earlier. My closure won't even mind if the data passed in differs in the order or number of columns in each row. As long as the dereferenced **$row** maps to the placeholders in **$insert_stmt**, the data will be inserted. I could even further simplify **init_stock_inserter()** by delegating the SQL generation to another subroutine entirely.

```
sub init_stock_inserter{
   my ( $con, $type ) = @_;
   my ( %tickers, $get_ticker,
        $insert_stock_prices );

   my ( $insert_stmt, $ticker_query ) =
        get_sql($type);
   $get_ticker =
        $con->prepare( $ticker_query );
   $insert_stock_prices =
        $con->prepare( $insert_stmt );
   ...create and return closure
}
```

The initialization section is now compact and clean as a whistle. Our new subroutine `get_sql()` doesn't even need to contain the SQL itself. It could suck the statements in from a prescribed file, thus eliminating the SQL from our program altogether. Having declared the two SQL statement variables, `$insert_stmt` and `$ticker_query`, inside a superficial block, I ensure that they return to obscurity as quickly as they were generated. After all, I only need them to prepare my statement handles, and I don't need my closure dragging around extra weight. However, the statement handle variables must now be declared outside of the block so that our closure can still access them.

Final Musings

Using closures with DBI is an effective way to introduce some order into complex database code while still taking advantage of the efficiency that DBI provides with prepared statement handles. DBI closures also allow me to simulate calls to stored procedures native to the database. I can rope off database specific code, and the SQL statements in particular, to prevent them from dominating the flow of my programs. In Perl, SQL should be second-class citizen and shouldn't be allowed to run the show. Give it a spin the next time the mailman drops 10,000 files on your doorstep! □

Perl Directory

Advocacy

The Perl Directory
http://www.perl.org

Perl.com
http://www.perl.com

The Perl Foundation
http://www.perlfoundation.org

The Perl Timeline
http://history.perl.org

News

use.perl
http://use.perl.org

Software

Perl 6
http://dev.perl.org/perl6/

Comprehensive Perl Archive Network (CPAN)
http://www.cpan.org

Perl Authors Upload Server (PAUSE)
http://pause.perl.org

CPAN Search
http://search.cpan.org

Kobes' Search
http://kobesearch.cpan.org

Database Interface (DBI)
http://dbi.perl.org

Documentation

Perldoc.com
http://www.perldoc.com

perlfaq
http://faq.perl.org

Help

Learn Perl!
http://learn.perl.org

PerlMonks
http://www.perlmonks.com

How (not) to be a Perl advocate

brian d foy
comdog@panix.com

Perl advocacy as a hot topic comes and goes. Mostly it deals with countering some myth or story making the rounds, or answering some new feature in another language. The latest crisis seems to be the release of PHP 5. A lot of advocacy that I see is defensive and reactive.

On the other side, there is a lot of offensive advocacy too. Isn't it wonderful how that word can mean "taking the initiative" as well as "causing anger and annoyance"? Often, this is the advocate who tells you that you should be using Perl before he even knows what you are doing.

I'd like to be a force for the good sort of advocacy, whatever that may be, and I want to publish more of it here. First, though, we need some ground rules so we don't get into fist fights.

What is our goal?

Why do we want people to use Perl? There are some good reasons to use Perl, such as

- it has a rapid development cycle
- the development team already knows it
- the system already has a Perl API
- Perl is free and available

And there are some bad reasons to use Perl

- it annoys Eric Raymond
- everything is a CGI script or a web page
- we don't know any other language
- we want more Perl jobs

People advocate Perl for different reasons, and that's a good thing. People *use* Perl for different reasons, and there are a variety of reasons to use Perl.

How much of our advocacy is about what we know and think, and about what actually works for other people? How much is about what we know how to do and how we can answer the question with what we know?

"Why I Hate Advocacy"

A long time ago, Mark Jason Dominus wrote this article for Perl.com. He talked about the larger problem of Right and Wrong. He didn't necessarily choose sides, but realized that people think in terms of Friends and Enemies. Another language can't be good without that reflecting poorly on Perl. As he points out, Perl's creation depended on a lot of things that were already good. To paraphrase one of his statements, Perl is just a programming language and you can't hurt its feelings.

We don't have to choose sides. Perl doesn't have to be the best at everything to be good. Another language doesn't have to suck for Perl to be good. PHP doesn't have to lose for Perl to win.

"Be an advocate, not an asshole"

Nat Torkington gave this talk at YAPC 19100. He hit on the idea of *selfish* advocacy, where we talk about promoting Perl because it benefits us, and *selfless* advocacy that promotes Perl because it makes life better for everyone. Selfish: I need a job and I know Perl, so let's use Perl. Selfless: Our company can save money and time by using mod_perl .

Nat tells us to focus on the needs of the other side and appeal to those needs. Different people and different roles care about different things. If we need to convince the money guys, talk about the monetary savings. If we need to convince a manager, talk about the savings in time. Find the thing that motivates that audience and use it.

He lists what doesn't help: dumping on other languages, flamewars, hyperbole (Perl does not cure cancer), personal attacks, and bickering. To convince people, you can't be, well, an asshole.

"Hello World" doesn't matter

At some point, every programmer wants to compare languages, whether to bolster their good opinion of their own language or to explore what else might be out there. In the small, this is a red herring.

Look around the net and you'll run into some language comparisons. Some are done as fairly as possible, and some are really just excuses to put down everything except the authors favorite. Perl has its share of these too, mind you.

The problem is that the metric, often a small program like "Hello World", adding some number, or some such trivial thing, is an awful way to compare languages. We can write a perl version of "Hello World" on the command line, but does that really mean its any better than Java for "enterprise" development?

So what works?

Nobody really knows what works, which is why there is a constant crisis mode when Perl advocates feel threatened. The Right Way depends on who we are talking to and what we want to accomplish.

We can take a lesson from the linux advocate Jon "maddog" Hall who contributed to the Linux Advocacy Guidelines. That guide has not only the usual HOWTO portions of linux documentation, but also a code of conduct. That document emphasizes a few things for everyone to remember.

- Play nicely
- Respect different opinions
- Offer alternatives

In short, be reasonable. If Perl is the right answer, then it can stand on its own merits. We just have to let people know about it and let them make up their own minds.

Another language doesn't have to suck for Perl to be good

The best thing, I think, is to simply show people what worked for you and how Perl solved our problem, saved our job, and made the universe a better place. We can do that without shoving it down people's throats. The O'Reilly Perl Success Stories does a wonderful job of that.

Do you have a good Perl story?

Can you tell your Perl story in a couple pages? Send it to us at letters@theperlreview.com. It doesn't have to be long or detailed. Explain what you did, how it worked out, and why you are happy with it.

Further Reading

"Why I Hate Advocacy" by Mark Jason Dominus
http://www.perl.com/cs/user/print/a/495?x-t=blog_this.view

"Be an advocate, not an asshole" by Nat Torkington
http://prometheus.frii.com/~gnat/yapc/2000-advocacy/

Linux Advocacy Guidelines
http://www.linuxgazette.com/issue14/advocate.html

O'Reilly Media's Perl Success stories
http://perl.oreilly.com/news/success_stories.html