

The Perl Journal

Implementing Symbolic Algebra & Calculus in Perl

Jonathan Worthington • 3

Graphing Perl Data with *GraphViz::Data::Structure*

Joe McMahon • 7

Keeping Up with the World, Part II

Simon Cozens • 10

Private CPAN Distributions

brian d foy • 13

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Jack J. Woehr:

Perl Core Language, Second Edition • 15

LETTER FROM THE EDITOR

Don't Make 'Em Like They Used To

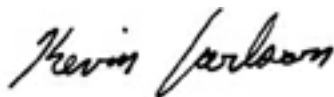
I heard a report on the radio the other day about studies that show that American consumers are becoming increasingly indifferent to traditional notions of product quality. It used to be (50 years ago, at any rate) that salespeople put the quality of a product's construction at the top of a sales pitch. If you could convince Mr. and Mrs. America that their blender was going to last 50 years, you had them.

Not so these days. Quality no longer means durability and solid construction. It has come to mean "packed with features." We're so much more affluent now that, with a few exceptions, we actually expect to discard most things we buy within a few short years. So we want to be sure our new digital camera has more bells and whistles than the last camera we bought (and maybe more bells and whistles than the one our neighbor has), but we're really not all that concerned about whether it will still be functioning in 10 years. Salespeople just can't get anywhere selling digital cameras that are "built to last."

One of the major engines behind this change in consumer attitudes is the rapid pace of innovation in digital electronics. We expect a product to be obsolete even before its components wear out, so longevity doesn't matter. And if cheaply built products are more affordable, so much the better. This new definition of quality—a high features-to-price ratio instead of durability—has spread to other industries, helped along by commoditization and the price pressures brought about by cheap offshore labor.

Software is probably connected to this trend, but it's hard to say for sure because software is so fundamentally different from other products. What does "quality" mean in software? It has never meant "durability," since binary code doesn't wear out. If your operating environment never changes, your code will continue to operate in exactly the same way for all eternity. Of course, no one's operating environment stays the same. We constantly upgrade our OSes and install other software that alters the computing environment. So old software does break, but only indirectly. This enforced obsolescence has led us to see software as being just as ephemeral and fragile as the cheap physical goods that we buy. Software has been dragged along with the PC in a downward spiral—we throw away our obsolete PC after three years of use, and often our perfectly functioning software must go with it.

It's impossible to make software that's "built to last." You can't know what changes in environment will break your code in the future. The best you can do is make it function perfectly now. Make it do as much as possible for the user, without error and without confusion. And make your code as easy to alter as possible, so that when new code inevitably must be written, the path is clear.



Kevin Carlson
Executive Editor
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to editors@tpj.com, faxed to (650) 513-4618, or mailed to *The Perl Journal*, 2800 Campus Drive, San Mateo, CA 94403.

THE PERL JOURNAL is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: \$18.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2005 by CMP Media LLC, unless otherwise noted. All rights reserved.



The Perl Journal

EXECUTIVE EDITOR

Kevin Carlson

MANAGING EDITOR

Della Wyser

ART DIRECTOR

Margaret A. Anderson

NEWS EDITOR

Shannon Cochran

EDITORIAL DIRECTOR

Jonathan Erickson

COLUMNISTS

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

CONTRIBUTING EDITORS

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

INTERNET OPERATIONS

DIRECTOR

Michael Calderon

SENIOR WEB DEVELOPER

Steve Goyette

WEBMASTERS

Sean Coady, Joe Lucca

MARKETING / ADVERTISING

PUBLISHER

Michael Goodman

MARKETING DIRECTOR

Jessica Hamilton

GRAPHIC DESIGNER

Carey Perez

THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403
650-513-4300. <http://www.tpj.com/>

CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND MARKETING Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT/GROUP DIRECTOR INTERNET BUSINESS Mike Azgara

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

VICE PRESIDENT, COMMUNICATIONS Alexandra Raine

PRESIDENT, CHANNEL GROUP Robert Faletra

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED TECHNOLOGIES Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS Paul Miller

VICE PRESIDENT, GROUP PUBLISHER NETWORK COMPUTING

ENTERPRISE ARCHITECTURE GROUP Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE DEVELOPMENT MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING SOLUTIONS Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES Marie Myers

Perl News

The Perl Foundation Calls for Donations

The Perl Foundation is looking for money that it can give away. Specifically, the Foundation would like to provide funding in 2005 and 2006 to Larry Wall, Patrick Michaud, Leopold Totsch, and an unspecified “second Parrot developer.” According to Allison Randal, the timing is urgent: “Parrot and Perl 6 are both close enough to completion that a few full-time developers could polish them off in a very small number of ‘Christmases.’ Another (more urgent) reason is that we’ve just learned that we have about 6 weeks left of Leo’s time before he’s forced to take a sabbatical from Parrot to pursue the noble task of ‘putting food on the table.’ This would set the project back by six months or more. So here we stand, on the edge of acceleration or a severe setback.”

The German Perl Workshop was recently able to raise enough money to fund 20 days of development; each day of development time costs \$200. The Perl 6 & Parrot Proposal (<http://www.perlfoundation.org/gc/grants/2005-p6-proposal.html>) shows the cost of achieving each milestone in development. Donations to The Perl Foundation can be made online at <https://donate.perlfoundation.org/index.pl?node=Contribution%20Info>.

Pugs Interpreter Hits Milestone

Autrijus Tang launched a new project to create an interpreter for Perl 6: Pugs. Written in Haskell, Pugs “aims to implement the full Perl6 specification, as detailed in the Synopses.” Work on Pugs has been proceeding at a breakneck rate: Autrijus has attracted 11 other developers to the project, and after only a month of development, they’ve succeeded in getting the Perl 6 module Test.pm to run correctly. Pugs is up to version 6.0.8 and is available from CPAN. It’s licensed under both GPL Version 2 and Artistic License Version 2.0b5.

According to the FAQ at <http://pugscode.org/>, “Similar to perl5, Pugs first compiles Perl 6 program to an AST, then executes it using the built-in evaluator. However, in the future Pugs may also provide a compiler interface that supports different compiler back ends. If implemented, the first compiler backend will likely be generating Perl 6 code, similar to the *B::Deparse* module. The next one may generate Haskell code, which can then be compiled to C by GHC. At that point, it may make sense to target the Parrot virtual machine. Other back ends (such as perl5 bytecode) may be added if people are willing to work on them.”

CPAN::Forum Opens

Gabor Szabo has launched a new web site, <http://www.cpanforum.com/>, for discussion of CPAN modules. He notes: “One of the objectives is to let people easily monitor discussions on several modules of their interest without subscribing to many mailing lists. It will also help lots of module authors for modules that do not have a mailing list

(I guess about 95 percent of all the modules on CPAN) to provide support.” Users can opt to be notified by e-mail when new messages are posted to the forums that interest them, and an RSS feed is also available.

Phalanx Marches On

Andy Lester has reorganized his Phalanx project, an effort to add tests to CPAN modules, find hidden bugs, and improve documentation. When his initial group of 12 testers, each working alone, failed to produce any additional tests, Andy decided to make the project more community-oriented. A web site has been put up at <http://qa.perl.org/phalanx/>; a wiki has been organized at <http://phalanx.kwiki.org/>; and a perl-qa list is hosted at lists.perl.org.

“When Ponie was announced in July 2003, we knew that it would require a regression test suite larger than Perl had ever had before. Fortunately, with the CPAN, we have a huge selection to choose from,” Andy explains. “We’re hoping that the 100 distributions we’ve selected, mostly on rough statistical analysis of usage, will cover a huge part of the CPAN that’s used by the Perl community...Bugs will be found. They’re out there.”

PerlIO Security Issues Raised

Kevin Finisterre has discovered two means by which a malicious local user can take control of a machine using `setuid perl`. Both depend on manipulating the “`PERLIO_DEBUG`” environment variable `PerlIO`. Kevin comments: “In the July 18, 2002 highlights for Perl 5.8.0 there was a ‘New IO Implementation’ added called `PerlIO`. The new `PerlIO` implementation was described as both a portable stdio implementation (at the source code level) and a flexible new framework for richer I/O behaviours. As an attacker, I would definitely say that `PerlIO` has some rich behavior.”

One method of attack uses `PERLIO_DEBUG` to cause a buffer overflow in the function responsible for logging the `PerlIO` data. The second exploits `PERLIO_DEBUG` message output with `setuid` to create root-owned files with world-writable permissions. “At this point, the game is pretty much over,” Kevin explains. “Since the file is world-writable the attacker can add any content he or she desires to the file that was created. Charles Stevenson suggested that a file could be written to `/etc/crontab.d/xxxx`, which would allow an attacker to control the machine with the next run of `cron`. I considered a few alternatives like writing to `root’s crontab`, making an `sshd root authorized_key` file, as well as a few others. In my example exploit, I took a more immediate and risky route by writing to `/etc/ld.so.preload` and providing a trojan.”

Rafael Garcia-Suarez fixed the security holes in `bleadperl`, and Mandrake provided a patch, available at <http://cvs.mandrakesoft.com/cgi-bin/cvsweb.cgi/~checkout~/SPECS/perl/perl-5.8.6-bug33990.patch?rev=1.1&content-type=text/plain>.

Implementing Symbolic Algebra & Calculus in Perl

Computers are good at working with numbers. With the ability to perform millions of calculations in a fraction of a second, complex problems can be solved numerically. Sometimes, however, it's useful to find an analytical solution to a problem, or at least part of a problem, that is initially expressed analytically.

One advantage is that a solution is obtained without any loss of accuracy. A solution computed from a numerical method may suffer from this, even if the method is mathematically supposed to give an exact answer due to the limits of floating-point arithmetic, for example.

Another advantage is that the more general solution provided by an analytical method may make further computation cheaper. Suppose, for example, that an analytical solution is found for the derivative of an expression. If the derivative needs to be evaluated at many points, evaluating the derivative at each point simply means substituting values for the appropriate variable.

This article will look at a way of representing mathematical expressions so they can easily be manipulated and evaluated. In part, it will reference three modules I have worked on recently, which may or may not be a good starting point for working with mathematical expressions analytically in Perl. These modules are, namely:

- **Math::Calculus::Expression**, a module for representing an expression.
- **Math::Calculus::Differentiate**, a module that computes the (analytical) derivative of an expression with respect to a single variable.
- **Math::Calculus::NewtonRaphson**, which demonstrates a conjunction of numerical and analytical methods, and provides a way of solving an equation, currently in just a single variable.

These modules are all available on CPAN, and can be installed in the usual way:

```
perl -MCPAN -e 'install Math::Calculus::Expression'
perl -MCPAN -e 'install Math::Calculus::Differentiate'
perl -MCPAN -e 'install Math::Calculus::NewtonRaphson'
```

Jonathan is currently working on an undergraduate degree in Computer Science at the University of Cambridge. He can be contacted by e-mail at jonathan@jwcs.net.

Representing Expressions

An expression in human-readable form, such as $\sin(2*x)$, is not easy to manipulate in a computer program. Regular expressions are not suitable because the language of valid mathematical expressions is not regular. Perl 5's regexes are more powerful than real regular expressions and a recursively defined regex could help here, but this probably would not have led to an easy way of implementing programs to manipulate expressions. Therefore, having a separate internal representation seemed to be the way forward.

The requirements for the representation were simple: It needed to be able to represent a mathematical expression, be relatively easy to translate to and from a human-readable form, and be simple to write programs to operate on. After some pondering, the possibility of using a binary tree-based structure came to mind. Under this scheme, a node can contain any of the following:

- An operator, such as $+$, $-$, $*$, $/$ (divide), and $^$ (raise to a power). These would always have two branches—one for each operand.
- A function, such as \sin , \cos , \tan , \ln , \exp , and the like. These would always have one branch, as these functions only take one parameter.
- A variable or constant, such as x , k , 2.4 , and the like. These would have no branches.

Any branch in the tree can be assumed to have brackets around it without any loss of expression. This is because an expression such as $5*x+2$ is equal to the expression $(5*x)+(2)$, which is also equal to $((5)*(x))+(2)$. This suggests that this representation is capable of representing all mathematical expressions, for they can all be translated into an “everything bracketed” form.

Out of this analysis comes a simple method of converting an expression in human-readable form to a binary tree. Taking the expression $5*x+2$ as an example, it is clear that the tree built from it should look like Figure 1.

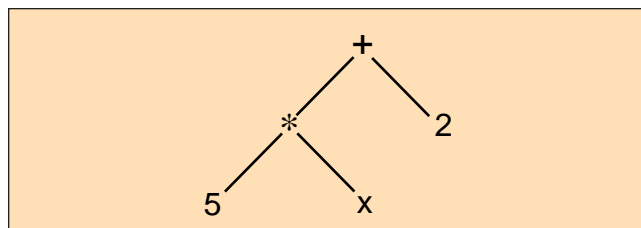


Figure 1: Tree built from the expression $5*x+2$.

Notice that the left subtree is the same tree that would be found if the expression $5*x$ was to be turned into a tree, and that the right node is the same output that would be expected if the constant 2 was put into the tree builder. This is a strong hint that recursion will be useful here. In fact, the entire process of building the tree can be summarized in three steps. First, find the operator to use for this node. Second, recursively build the left subtree, passing everything to the left of our chosen operator. Third, recursively build the right subtree, passing everything to the right of our chosen operator.

The second and third steps are simple, but the first requires a little thought. It is important that the correct operator is chosen at each step, and for this the tree builder needs to understand precedence. To help keep the tree building function itself neat, another function can be defined that takes two operators and returns True if the precedence of the first operator is lower than that of the second. This means that finding the operator to choose to split on is a case of scanning through the expression character by character looking for operators that are not nested inside any brackets, then when one is found, checking if it has lower precedence than the operator that is currently regarded as the best one to choose. If it does have lower precedence, it is marked as the current best choice operator, and the scan continues to the end of the expression.

This isn't quite the whole story; there are two special cases to be aware of. These should be checked for before the previously described routine is entered, for if either are true, it would fail.

The first of these special cases involves the expression to build a tree from a constant or variable. If this is the case, it will either be a number (which may or may not be an integer) or a single letter. In both cases, a $-$ sign may precede it. This case can be identified using a simple pattern match; `/^-?(\\d+(\\.d|[A-Za-z])$)/` will work.

The second special case is where the entire expression is a function of a subexpression. In this case, the expression will start with the name of a recognized function, such as *sin*, followed by an opening bracket. Again, a $-$ sign may precede it. The name of the function is stored where the operator would otherwise be located, and the left branch of the tree is generated by recursively calling the tree builder on the function's parameter. There is no right branch.

Converting the tree representation back into a human-readable form can also be achieved through a (much simpler) recursive process. A pretty printing function can be defined as follows:

- If the current node is a constant or variable, this can simply be returned.
- If the current node is a function, recursively call the pretty print function on the left branch of the tree, then return something of the form “<function name>(<left branch>).”
- If the current node is an operator, recursively call the pretty print function on the left branch and the right branch, then return something of the form “(<left branch>) <operator> (<right branch>).”

Some extra tracking of operators can be done to work out whether brackets are needed or if they can be omitted because precedence rules make them redundant.

So far, two of the three requirements for the representation are satisfied; it can represent mathematical functions and can easily and fairly cheaply be converted to and from a human-readable representation. The third condition was that it should be easy to manipulate mathematical expressions using this representation. The binary tree-based structure has some useful properties. It is now trivial to identify and extract subexpressions; every subtree is a valid expression. Precedence is completely determined by depth in the tree, and there is no need to do any more work to check operator precedence when manipulating the expression. If this representation has been formed, it also means that the expression is valid, removing much of the responsibility for error

checking from code manipulating the expression. The general benefits of separating parsing from evaluation are gained, too: Now there is just one place to fix parsing bugs.

Math::Calculus::Expression

The *Math::Calculus::Expression* module implements a class that represents a mathematical expression using the binary tree method described earlier. Remember that an expression is not an equation; there must be no “=” symbols or other statements about mathematical equivalence. It understands the operators $+$, $-$, $*$, $/$ (divide) and $^$ (raise to power) and their precedence, as well as the functions *sin*, *cos*, *tan*, *sec*, *cosec*, *cot*, their inverses (add an *a* to the start of those names—for example, *asin* is inverse *sin*) and hyperbolic equivalents (add an *h* to the end—for example, *sech* is hyperbolic *sec*), *ln* (natural logarithm), and *exp* (base of natural logarithms to the power of the parameter). Use of brackets to control precedence is supported. All subexpressions must be separated by an operator, so $5*x$ and $(x+2)*(x-3)$ are valid but $5x$ and $(x+2)(x-3)$ are not. Spacing does not matter, and most certainly does not control precedence. $2+x * 5$ means exactly the same as $2 + x*5$.

A new expression is instantiated using the *new* method of the Expression class:

```
use Math::Calculus::Expression;
my $expr = Math::Calculus::Expression->new();
```

Various methods can then be called on the *expression* object. The *setExpression* method takes an expression in human-readable form as its only parameter and attempts to convert it into the internal representation. If this fails, which will happen if the expression is invalid, then a *False* value will be returned. A traceback that may contain useful information is also available by calling the *getTraceback* method, and a short error message can be obtained by calling the *getError* message:

```
unless ($expr->setExpression(
    '2*sin(x^2 + 4) + cosh(x)')) {
    print "Error: " . $expr->getError . "\n";
    print "Traceback: \n" . $expr->getTraceback;
}
```

Expressions stored in an *expression* object can be retrieved in a human-readable form using the *getExpression* method:

```
# Prints 2*sin(x^2 + 4) + cosh(x)
print $expr->getExpression;
```

Note that *getExpression* may not return exactly the same string that was provided when *setExpression* was called. The internal representation retains the meaning of the expression, but not spacing and extraneous brackets.

It is often useful to tell the module which letters correspond to variables; any that don't will be assumed to be constants. The *addVariable* method is used to declare a letter as a variable.

```
$expr->addVariable('x');
```

The *simplify* method attempts to make the expression simpler. Its bag of tricks include constant evaluation ($2*3*x$ becomes $6*x$), redundant-term elimination ($0*\sin(x)$ and $2*x-2*x$ will become 0, and if part of a more complex expression, will disappear), and null operator elimination (x^1 and $1*x$ and $x+0$ will all become x). The method is simply invoked on an expression object and modifies the expression that it stores.

```
$expr->simplify;
```

The *evaluate* method takes a hash, mapping any variables and named constants (collectively letters) in the expression to numerical values, and attempts to evaluate the expression and return a numerical value. It fails if it finds letters that have no mapping or if an error such as division by zero occurs. The currently stored expression is not modified, so this method can be called to evaluate the expression at many points.

```
$value = $expr->evaluate(x => 0.5);

# Use defined as 0 is false but valid
unless (defined($value)) {
    print "Error: " . $expr->getError . "\n";
}
```

The *sameRepresentation* method takes another expression object as its parameter and returns True if that expression has the same internal representation as the expression the method is invoked on; under the current implementation, that means identical trees. Be careful—while it can be said that if two expressions have the same representation, they are equal; it would be false to say that if they have different representations, they are not equal. It is clear that $x+2$ and $2+x$ are equal, but their internal representation may well differ. As the internal representation and its semantics may change over time, the only thing that is safe to say about this method is that if it returns *True*, the expressions are equal. If it returns *False*, nothing can be said about equality.

```
print "Same representation" if $expr->
    sameRepresentation($expr2);
```

Finally, the *clone* method returns a deep copy of the expression object (“deep copy” meaning that if the original is modified, the copy will not be affected and vice versa):

```
$exprCopy = $expr->clone;
```

Operating on an Expression

The *Math::Calculus::Expression* class uses a hash for each node. The hash has keys *operator* (storing the operator or function name), *operand1* (for the left branch), and *operand2* (for the right branch). The values corresponding to the branch keys (*operand1* and *operand2*) may be undefined, contain a number or letter, or contain a reference to another hash. Using a hash-based structure helps with program readability and allows others modules that need to do more sophisticated manipulation to augment the tree with their own data by simply putting extra entries into the hash.

Generally, tree operations are implemented using recursion, as a depth-first exploration of the tree is usually required. A breadth-first exploration may also be useful in some cases, and this could also be implemented. This article will take a look at the depth-first approach. Operations can be subdivided into three categories: those that modify the tree, those that use this tree to build a new tree, and those that use the tree to generate something else (such as a single value). Knowing when to use the third of these is fairly easy, but it’s not always obvious when choosing between the first and the second. A good rule of thumb is to consider whether the operation changes the values or the existing structure of the tree. If the structure is changing, then it will usually be simpler to build a new tree.

As a simple example, consider writing a module that can substitute an expression or a constant for a particular letter. A good starting point is to inherit from *Math::Calculus::Expression*, as this will provide a constructor and other useful methods:

```
package Math::Calculus::Substitute;
use strict;
```

```
use Math::Calculus::Expression;
our @ISA = qw/Math::Calculus::Expression/;
```

The structure of the expression tree may change if a complex expression is substituted for a letter, but note that this is simply an extension and not a change to the structure that is already in place—basically, a single value is being changed, but it may be changed to be a subtree rather than another value. Therefore, this problem is simple enough that it’s best to modify the existing tree. A procedure can be written that will recurse down the tree looking for a branch that is equal to the letter that is being sought and replace it with whatever replacement has been provided, which it will assume is ready to be substituted. This is OK because it will be a private method rather than the public interface to the module.

```
sub recSubstitute {
    # Grab the parameters.
    my ($self, $tree, $letter, $sub) = @_;

    # Look at each branch.
    my $branch;
    foreach $branch ('operand1', 'operand2') {
        # If it's a reference to a subtree, recurse.
        if (ref $tree->{$branch}) {
            $self->recSubstitute($tree->{$branch},
                                $letter, $sub);
        }
        # Otherwise, if it's the letter...
        elsif ($tree->{$branch} eq $letter) {
            # Substitute.
            $tree->{$branch} = $sub;
        }
    }
}
```

Finally, the public method needs to be implemented. It needs to use a couple of the private methods of the *Expression* class that exist for the usage of subclasses, namely *getExpressionTree* to get a copy of the raw expression tree and *setError* to set or clear the current error message. As the substitution could be a letter/number or an expression object (but not a raw tree—remember that the tree representation is internal), this method needs to ensure that what is passed to *recSubstitute* is something that can be substituted straight into the tree.

```
sub substitute {
    # Grab parameters.
    my ($self, $letter, $sub) = @_;

    # Check letter really is a letter.
    if ($letter !~ /^[A-Za-z]/) {
        $self->setError('A valid letter must
                        be passed.');
```

```
        return undef;
    }

    # If the sub is a reference, then it
    # should be pointing to an Expression
    # object. Extract the raw expression.
    my $realSub;
    if (ref($sub)) {
        $realSub = $sub->getExpressionTree;
    } else {
        $realSub = $sub;
    }

    # Get a reference to this object's
```

```

# expression tree.
my $tree = $self->getExpressionTree;

# Make the recursive call.
$self->recSubstitute($tree, $letter, $realSub);

# Clear the error and return success.
$self->setError('');
return 1;
}

```

The next two sections will take a quick look at a couple of other modules that inherit from the *Expression* class and do some more complex manipulations.

Math::Calculus::Differentiate

Differentiation is one part of the calculus. It involves taking a function and finding its gradient function—another function that defines how steep the function is at any given point. In many cases, the function to differentiate can be written in the form $y=f(x)$. As $f(x)$ is an expression and finding the derivative is a clearly defined set of manipulations on the expression, the tools described so far provide all that is needed to implement a module to do differentiation analytically.

As many of the manipulations involve significant structural changes to the existing tree, the *Differentiate* module builds a new tree as it walks the existing one. This is hidden from the end user of the module as the public *differentiate* method takes the new tree and sets the current expression object to point to it. This is a good thing to do in general, as it leaves the user with only two kinds of operations to worry about: those that don't change the tree and return a result, and those that do modify the tree and return success or failure.

Usage of the module is simple: Call the *differentiate* method and pass in the variable to differentiate with respect to. Note that all differentiation done by this module is partial; if there are multiple variables, then others are taken to be constants, as is the norm in partial differentiation. The following example shows the module at work:

```

use Math::Calculus::Differentiate;

# Create new expression object.
my $expr = Math::Calculus::Differentiate->new;

# Set expression and variable.
$expr->setExpression('x^2 + sin(x)');
$expr->addVariable('x');

# Differentiate with respect to x and simplify.
$expr->differentiate('x');
$expr->simplify;

# Output result.
print $expr->getExpression; # Prints 2*x + cos(x)

```

Note that error checking has been omitted from this example to keep it simple, and should of course be included in any real program that uses this module.

Math::Calculus::NewtonRaphson

Solving equations analytically is sometimes impossible for people, let alone computers. This is a case where a numerical method is almost always the right thing to use. The Newton Raphson method of solving an equation requires that it is rearranged into the form $f(x)=0$. Again, $f(x)$ is an expression and it is therefore suitable to use the system described in this article to manipulate it.

The Newton Raphson method is iterative; that is, it takes an initial guess and produces a solution. That solution can be fed back in to produce a better solution. This process can be repeated until a solution has been found to the desired accuracy. Being able to use the Newton Raphson method involves knowing the derivative of $f(x)$, $f'(x)$ itself, and having a good initial guess. The first of these can be calculated with the *Math::Calculus::Differentiate* module, and the second two are things that the user of the module supplies.

This is an example of a module that produces a result without modifying the existing expression—at least, it does from the user's point of view. Under the hood, it will need to make a copy of the expression and differentiate it, then form a new expression involving the original one and the derivative. This can then be passed to the *evaluate* method inherited from the *Expression* class. A simple example of the module at work follows:

```

use Math::Calculus::NewtonRaphson;

# Create new expression object.
my $expr = Math::Calculus::NewtonRaphson;

# Set expression and variable.
$expr->setExpression('x^2 - 4');
$expr->addVariable('x');

# Solve with Newton Raphson - initial guess of 3.
my $result = $expr->newtonRaphson(3);

# Display result, which will
# be 2 as 2^2 - 4 = 4 - 4 = 0
print $result; # Prints 2

```

It is possible that a solution will not be found; in which case, the *newtonRaphson* method will return *undef*. Any real-world program should check for this case.

Where From Here

There is still plenty of scope for improving the simplifier and writing more modules. One big problem with what I have described in this article is that there is currently no way to test for equality—whether two expressions have the same representation in the current tree format is too tight a condition. One solution is to try to find a canonical representation—one in which two things that are equal will always get the same representation. This is what happens in binary decision diagrams, and it enables us to cheaply determine equality by comparing representations. One problem with this is that work may need to be done to ensure that the expression stays in its canonical representation, and enabling easy tree manipulation may not be possible. Another option is to build an equality checker that makes a copy of each representation in some canonical form and then compares their representations. Decomposing the expressions into their Taylor series and comparing the series could be an option. This doesn't make equality checking as cheap as may be desirable, however.

I hope this article has offered some insight into a way of working with mathematical expressions analytically. Whether it is a good way is debatable, though even if you think this entire approach is a mistake, it's another person's mistake to learn from. While you may not need to deal with mathematical expressions regularly (or ever), some of the principles illustrated here (such as the idea of using a different internal data representation that is unseen by the end user) are more widely applicable.

Graphing Perl Data with *GraphViz::Data::Structure*

It used to be that if you wanted to draw a diagram of a data structure, or any other picture that consisted of nodes and edges, you had to do it by hand (or with a computer drawing program), carefully working out the layout of what node went where and how the edges were routed to get the clearest possible picture. This could be a big problem if you completed a diagram and then found out there were a couple more nodes to add; sometimes, you'd have to completely discard the old diagram and start over from scratch. This was a big waste of time when you had diagrams that were constantly changing.

The folks at AT&T Bell Labs created the graphviz package to deal with this problem. It parses a language, called “dot,” for the description of graphs—the nodes and their names, and the node connections, like this:

```
digraph G {  
    a -> b;    a -> c;  
    b -> d;    b -> e;  
}
```

The “digraph” keyword denotes to dot that this is a directed graph (edges have a start and an end, with an arrowhead showing the direction in which the edge “flows”). The “arrow” specifications show that *a* is connected to *b* and *c*, and that *b* is connected to *d* and *e*. When this is rendered by graphviz, you get the picture in Figure 1. (*dotty* is part of the graphviz package—among other things, it can display dot graphs.) In a traditional drawing program, if you wanted to add more nodes, you'd have to figure out how to lay out the graph all over again. With dot, you simply add the new definitions:

```
digraph G {  
    a -> b;    a -> c;  
    b -> d;    b -> e;  
    e -> c;    f -> a;    b-> f;  
}
```

And it takes care of the layout; see Figure 2.

Joe is a test automation architect at Yahoo! who has contributed to core Perl test support, supplied the debugger's internal documentation, and supports a number of CPAN modules. He can be contacted at mcmahon@ibiblio.org.

The possibility of simply describing a drawing rather than painstakingly rendering it by hand opens up many more possibilities for programmers. Instead of having to output textual descriptions of data and relationships, it's now possible to describe those relationships and see a picture of them. The aforementioned examples illustrate this well; even with the description of the connections in the dot file, it's still very difficult to envision exactly how the nodes are connected without the picture.

When graphviz was first released, it was still necessary to put together the dot input files by hand. This could be time consuming—less so than hand-diagramming, admittedly—but very often, a graph creator would go through many iterations of edit/graph/view to try to get just the right graph with all the proper connections.

Enter Perl, and Data Structures

Leon Brocard created the *GraphViz* Perl module to address this problem, making it far easier to create graphs. Now, instead of having to figure out connections and write the necessary dot code to display a graph, one could simply write Perl code to generate the graph, and have *GraphViz* render the graph for you via dot. For instance, *GraphViz* has several interesting demo classes included with the package, including one that visualizes XML and one that renders Perl data structures as graphs (*GraphViz::Data::Grapher*).

This is where *GraphViz::Data::Structure* got its start. I wanted to use the XML visualizer, but it has a minor problem in that it doesn't preserve strict order of sub trees in the XML. I was going to render these graphs for others who weren't particularly XML-savvy at the time, so accurate order was important. Also, since this was partially a propaganda tool as well (XML was perceived as “too complicated” at the time), the graphs also had to look attractive.

Dot Records

Readers who are familiar with texts such as Donald Knuth's *Art of Computer Programming* will no doubt remember the beautiful graphs that were used to describe data structure algorithms: nice rectangular nodes with pointers linking them together. I wanted to achieve a similar effect in my XML visualizations. It so happens that the dot language supports what it calls a record: a rectangular node that can be arbitrarily subdivided into rectangular subnodes. These subnodes can then be linked together via ports that describe the source and target sub nodes. This seemed an ideal way to handle the data structure visualizations: It was supported by dot and it would be familiar to my readers.

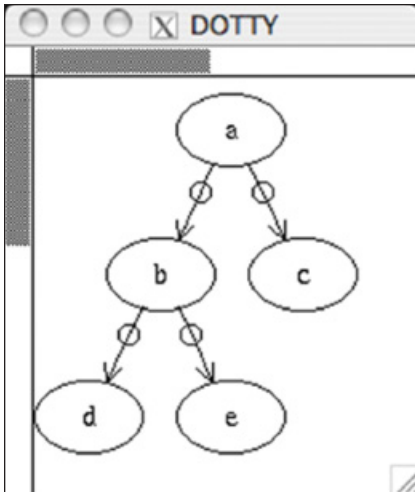


Figure 1: Simple directed graph rendered by graphviz.

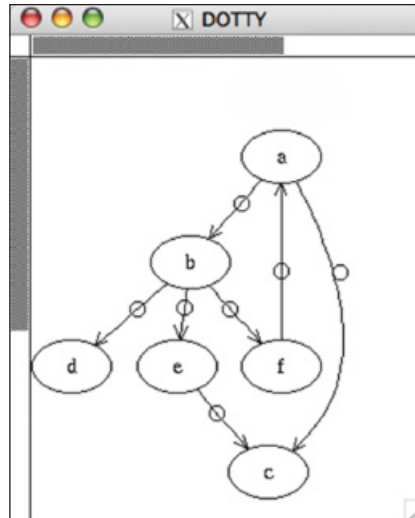


Figure 2: Expanded directed graph.

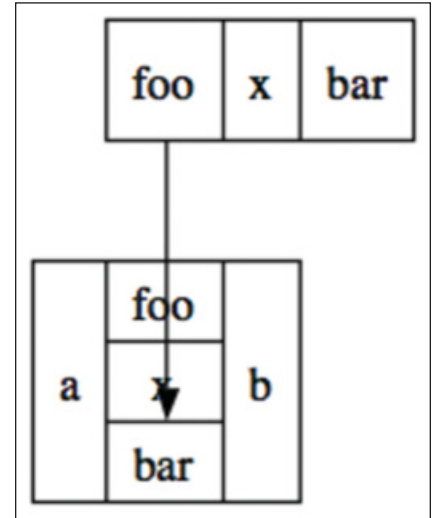


Figure 3: Data structure graph.

Record specs look something like this:

```
digraph G {
  node [shape=record];
  a [label = "<f0> foo | x | <f1> bar"];
  b [label = "a | { <f0> foo | x | <f1> bar } | b"];
  a:f0 -> b:f1
}
```

The resulting graph is shown in Figure 3.

As you can see, the syntax is a bit arcane, but we can do what we want to do: Create a familiar-looking structure and link the subparts to other things. This allows us to use one of the XML parsing classes to create a tree of objects and then display that tree, all properly linked together and in order. I chose to use `XML::Parser's style=>objects` to do this, mostly because I understood it best and secondarily because it creates relatively uncomplicated objects.

I was able to cobble together a very basic module that could handle the `XML::Parser` object tree pretty quickly; it only required a very basic tree-walk and simple array-like layouts. I showed it around, and Leon commented that the output was really neat, but he wondered: Could I generalize it?

One Thing Leads to Another

It turned out that the basic code I had was enough to do the simple job of dumping a very specific set of hashes and arrays that pointed to each other, but was woefully incomplete when it came to doing arbitrary structures. I decided that the module should be able to dump any Perl data structure, no matter what it was, including self-referential ones. This ruled out using `Data::Dumper` to do the structural analysis because `Data::Dumper` can't handle these structures easily (it can dump them fine, but I'd have had to parse the resulting Perl to patch up the self-references, which looked like even more work). This meant I needed a tree-crawler that could handle any of Perl's data structures: scalars, arrays, hashes, globs, and all of the different kinds of references, plus undef.

The basic algorithm is rather simple. At each level, recursively call the tree-walk subroutine to render all of the nodes under this one, and then handle the current node. Arrays and hashes (and globs) have to be iterated over to handle each of their elements.

This was all pretty simple—perhaps the most difficult problem in the tree walk was accurately determining the type of thing being pointed to by a reference. Perl 5.6 and Perl 5.8 differ significantly in what `ref()` returns for scalar references, which led to the code be-

ing broken for a considerable amount of time on 5.8 until I could compare 5.6 and 5.8 runs head-to-head to spot the problem.

Brace Yourself

This was no problem at all compared to the research and trial-and-error needed to figure out how records really work. It's possible, using curly braces, to tell `GraphViz` to switch cell layouts within a record from vertical to horizontal. This was necessary for objects (and globs) because I wanted them have the name and type on top, and the contents on the bottom. This feature was not precisely documented, and it took a lot of work to figure out exactly how the braces should be nested to get the desired results. Once this was worked out via test-driven development (we'll get to that in a minute), it was pretty clear sailing. Globs were handled as a special case of blessed hashes, for instance.

A Good Set of References

Code references were a bit more of a problem. I could see that the debugger was able to track down where code references came from, but it wasn't obvious how this was done. A lot of inspection of the debugger's `dumpvar.pl` code was needed to figure out how to look up the actual location where a code reference was defined, and I stole a small amount of code from the `dumpvar.pl` library routine to handle this.

Testing

A particularly interesting question is, how can you automate the testing of something that is fundamentally visual? The Perl code couldn't know if the graphs looked right; it could only judge whether or not the output was as expected. On the other hand, not automating the tests at all seemed really wrong. I was able to solve this problem in the classic computer science manner: When presented with two exclusive alternatives, do both—selectively.

I needed to look at the actual graphical output once to make sure it was right. After that, I could tell the `GraphViz` object embedded inside my object to render the graph `as_canon`, which is canonical dot source code. This source code could be cleaned up a bit to make sure that it was easily comparable, and then automated testing could be used to make sure that the canonical graph was still the same each time after I succeeded in getting the graph right visually. This was extremely helpful when trying to work out the compatibility problem between 5.6 and 5.8.

For test-driven development, once I had figured out how the records worked, I could hand-compose one drawing and save it

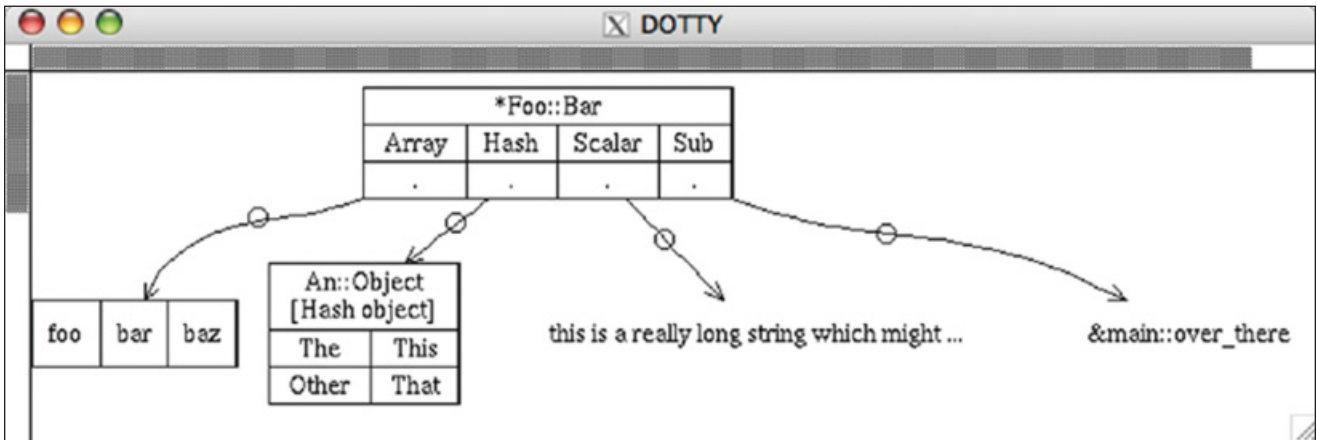


Figure 4: Graph demonstrating the capabilities of `GraphViz::Data::Structure`.

as “expected” output, then develop code that output that dot code. Once the basic optical test and the comparison tests both passed, I had a basis from which I could develop more tests.

It became evident early on that developing the tests by hand, even with *Test::More* helping out, was extremely painful. So I developed a helper program that let me specify the tests I wanted to run, run them to do a visual check, and then collect the output for later automated testing. This sped development up considerably because I no longer had to hand-code dot output for every test case once the output started looking sensible.

Current Events

At the moment, *GraphViz::Data::Structure* has the following features:

- It can dump arbitrary complex data structures.
- It can terminate the dump at a given level (nice if you only want to see the top *N* layers of a data structure).
- It can trim strings output to dot to a specified minimum length (it was discovered during testing of the XML dump code that dot will just segfault if text strings in nodes are “too long”).
- It uses plain text wherever possible to minimize graph size (constants, empty hashes, and arrays).

It’s quite useful as an adjunct when trying to get a handle on particularly messy data structures. You can dynamically spin off a graph at any point in your program by piping the output of *GraphViz::Data::Structure* to a compatible program. This is particularly useful in the debugger.

GraphViz::Data::Structure In Action

Here’s a quick example that shows just about everything *GraphViz::Data::Structure* can do. We create a data structure, graph it, and then launch dotty to display it (shown in Figure 4):

```
#!/usr/bin/perl -w
use strict;
use GraphViz::Data::Structure;

my ($a,$b,$c,$d);

sub over_there { 'sample sub' }

$a=*Foo::Bar;
*Foo::Bar=&over_there;

*Foo::Bar=$b;
$b="this is a really long string which might cause
```

dot to segfault, so it will be truncated";

```
*Foo::Bar = \@c;
@c=qw(foo bar baz);
```

```
*Foo::Bar = bless \%d, "An::Object";
%d = (This=>'That', The=>'Other');
```

```
my $dot_source =
  GraphViz::Data::Structure->new(
    \%a, title=>"a little\nof everything"
  )->graph->as_canon;
open DOTTY_IN, ">dotty_input"
  or die "Can't create dotty input file: $!\n";
print DOTTY_IN $z;
close DOTTY_IN;
system "dotty dotty_input &";
```

You can see that we have the **Foo::Bar* glob pointing to each of the things we assigned to it; in particular, *GraphViz::Data::Structure* has trimmed the long string and has labeled the hash we blessed with its class name. Note also that the array’s elements are in order, and that the hash’s key-value pairs (and the glob’s entries) are all lined up nicely, and that we know the actual name of the subroutine associated with the code reference.

Future Plans and Ideas

GraphViz now supports HTML records, which use table-like markup to define the internal contents of a record node. It’s possible that the data layout code could be simplified considerably using this method of defining nodes. The tree-walking code, which does all the hard work, should be refactored to separate the graph generation from the visit code; that way, other programs could use the visit code to safely explore arbitrary data structures. Last, some self-referential structures generate dot code that exposes bugs in dot—the graphs will not render. It’s currently unclear as to whether this is caused by bugs in the record layout code or if it’s a deeper problem; trying HTML layout may possibly solve this problem.

Conclusion

So now there’s a tool to draw pretty pictures of your complex Perl data structures, and it’s even invocable on the fly. Use it to debug those weird data structure bugs. Draw pretty pictures of your data relationships. And as Larry says, “have the appropriate amount of fun.”

TPJ



Keeping Up with The World, Part II

Simon Cozens

In last month's column, we saw how to put together a Curses-based RSS feed reader. We began by looking around on CPAN and assembling a collection of modules that did most of the job for us, and spent the rest of the article merrily plugging them together. When it comes to the second part of the problem—making the RSS reader talk to existing databases of feed information—then things are a little different. We're on our own.

Where We Are Now

At the moment, our copy of press has a hard-coded list of RSS feeds that we want it to read. Each time we restart press, it forgets what we've already read; if you remember the original point of writing press, it was to enable me to take home the data from my normal RSS reader, NetNewsWire, to use its caching and record of which articles had been read, and then return to my usual network source and have NetNewsWire carry on as if I'd been reading all the time.

To do this, we'll need to both read and write the format used by NetNewsWire, and then alter our press program to use the NetNewsWire files as a data source instead of our hard-coded list. To make this a generic change, we'll split off the data-store parsing code into a module, *Press::NNWI*; this means that if I later move to a new version of NNW, or indeed any other news reader, I can simply create a module to read and write its data stores, and slot that in with no changes to press.

NNW uses two sources of information about what we're reading: The first is an XML "plist" file of the feeds we're subscribed to, and the second is a SQLite database of which articles we have seen.

A plist is an XML data serialization format invented by Apple, and is capable of storing more complex kinds of data than our simple list of URLs and titles. In fact, NNW stores a hierarchy of feeds, allowing us to group feeds into categories and subcategories. So part of my preferences file looks like this:

```
<dict>
  <key>isContainer</key>
```

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

```
<string>1</string>
<key>name</key>
<string>Tech News</string>
<array>
  <dict>
    <key>home</key>
    <string>
      http://www.google.com/googleblog/
    </string>
    <key>name</key>
    <string>Google Blog</string>
    <key>rss</key>
    <string>
      http://www.google.com/googleblog/atom.xml
    </string>
    ...
  </dict>
  <dict>
    <key>home</key>
    <string>
      http://www.groklaw.net
    </string>
    ...
  </dict>
</array>
</dict>
```

Plist *dict* elements, or dictionaries, are a bit like hashes (and *array* elements are, well, arrays) so here we have a hash that is a container, called "Tech News," which contains some feeds, themselves expressed as hashes. We can use the *Mac::PropertyList* module to turn this into a Perl data structure:

```
use constant PREFS =>
  "Library/Preferences/
  com.ranchero.NetNewsWire.plist";
my $file ||= (getpwuid $<)[7]."/".PREFS;
my $stuff =
  Mac::PropertyList::parse_plist_file($file);
```

To get the effect of supporting groups of feeds, we'll actually turn this into a flat list of titles, some of which will only be used for display purposes. That is, something like:

```
@titles = (
    "Torgo-X zhoornal",
    "[ Tech News ]",
    "Slashdot",
    "Planet Perl",
    "Groklaw",
    "[ Friends ]",
    "No-sword",
    ...
);
```

Notice that those entries in brackets are not real RSS feeds—they’re just titles for the relevant group. Also note that we’ve indented the feeds inside each group by a space.

The second piece of information we have from NetNewsWire is a file in Library/Application Support/NetNewsWire called “History.db.” Looking at this file, it contains a piece of useful information:

```
% head -1 ~/Library/Application\ Support/
NetNewsWire/History.db
** This file contains an SQLite 2.1 database **
```

Looking at it with the ever-so-handly *DBI::Shell*, we find that it is very simply constructed: one *history* table, which contains the columns *link*, *flRead* and *flFollowed* (flags to signify if the link was read or followed), and *dateMarked*. The link is, obviously, the link of each item in the feed, although it is munged a little from being merely a URL: It has a number in front of the URL and one at the end. We can guess that these are related to the ability to detect when the content of an article has changed, and indeed, with a little playing around, we find that the first number is the length of the headline and the second number is the length of the content. We’re going to have to both read and write these rows in the *history* table to ensure that the history file is kept updated when we’re using press. Now that we’ve seen what we want, how do we get there?

Easy is in the Eye of the Beholder

I used to have long talks with people about the usefulness (or lack thereof) of “design patterns” and what patterns there are for Perl programming. It took me a while to realize that I used many design patterns in my Perl programming, but I did so unconsciously and instinctively. So for me, a list of Perl design patterns wouldn’t make sense because I automatically recognize the pattern I need when the situation calls for it.

The situation we have with turning the nested XML structure into a flat array is one such case. When I was confronted with this, I was already halfway through writing a recursive closure, and suddenly the problem was solved. Here’s how I did it. We’ll start with the outline of the code:

```
use constant PREFS => "Library/Preferences/
com.ranchero.NetNewsWire.plist";
use constant DBPATH =>
    "Library/Application Support/NetNewsWire/";

sub get_feedlist {
    my ($self, $file, $db) = @_;
    my $file ||= (getpwuid $<)[7]."/".PREFS;
    my $db    ||=
        (getpwuid $<)[7]."/".DBPATH."History.db";
    my (%labels, @values);

    my $stuff =
        Mac::PropertyList::parse_plist_file($file);
    # Do some magic with $stuff here
```

```
    return (@values, %labels);
}
```

As with last time, *@values* will be the list of blog names and group names, and *%labels* is the mapping between names and feed objects. For the moment, we’ll only think about filling *@values* by extracting the names from the plist. Since the data structure is intrinsically recursive, we should already be thinking of a recursive walk over the plist tree. Something like this:

```
sub walk_plist {
    my ($data, $depth) = @_;
    my @rv;
    for my $feed (@$data) {
        if ($feed->{isContainer}) {
            # It’s a group, not a feed,
            # so note it and recurse.
            push @rv, (" " x $depth).
                "[".$feed->{name}->value."]",
                walk_plist($feed->{childrenArray},
                    $depth + 1);
        } else {
            # It’s an ordinary feed
            push @rv, (" " x $depth).
                $feed->{name}->value;
        }
    }
    return @rv;
}
```

This is great, except for two things: First, we have to deal with two variables (*@values* and *%labels*); and second, we’re passing around a lot of values as we build up the various lists. Isn’t there an easier way to do it? Well, “easier” is rather subjective, but by turning this subroutine into a closure, we can avoid passing around the variables, and keep everything tidily in one place.

```
sub get_feedlist {
    my ($self, $file, $db) = @_;
    my $file ||= (getpwuid $<)[7]."/".PREFS;
    my $db    ||=
        (getpwuid $<)[7]."/".DBPATH."History.db";
    my (%labels, @values, $walker);

    my $stuff =
        Mac::PropertyList::parse_plist_file($file);

    $walker = sub {
        my ($stuff, $depth) = @_;
        for my $feed (@$stuff) {
            if ($feed->{isContainer}) {
                push @values, (" " x $depth).
                    "[".$feed->{name}->value."]",
                    $walker->($feed->{childrenArray},
                        $depth + 1);
            } else {
                # Make an appropriate XML::RSS object
                $rss = ...;
                push @values, $rss;
                $labels{$rss} = (" " x $depth).$name;
            }
        }
        $walker->($stuff, 0);
        return (@values, %labels);
    }
```


Since we're using a closure, we've got access to the `@values` and `%labels` hashes in the enclosing scope—we don't need to pass anything around because it's all there. And as it's a closure, the recursion works just fine, too. Of course, we skipped over a little bit—making the `XML::RSS` objects. This is where the history database comes in.

History Repeating

Just as we needed a special subclass of `XML::Feed` to handle feeds while controlling when articles get marked as read, we'll need another subclass of that to control reading and writing the marked-read status to the NetNewsWire database.

```
package XML::Feed::NNW1;
use base 'XML::Feed::Manual';
```

First, we'll create an accessor for the history database file, so that we can tell each feed where to look to find its database:

```
sub history {
    my $self = shift @_;
    $self->{history} = shift if @_;
    $self->{history};
}
```

We need this because the constructor calls an accessor for each of the options passed to it. Next, we need to hook into the headline-loading process, to mark the items that the database says are read. `_load_cached_headlines` is a sensible place to do this:

```
sub _load_cached_headlines {
    my $self = shift;
    $self->SUPER::_load_cached_headlines;
    # Now load the history
    my $dbh = DBI->connect(
        "dbi:SQLite2:$self->{history}"
    ) or return;
}
```

At this stage, we have a list of headlines. We check the database for each one to see if they've been seen before:

```
my $sth = $dbh->prepare(
    "SELECT * from history where link = ?"
);
for my $head ($self->headlines) {
    $sth->execute($self->encode_headline($head));
    $self->SUPER::mark_seen($head)
        if @{$sth->fetchall_arrayref};
}
}
```

The `encode_headline` method will be used to transform the article's URL into the encoded form used by NetNewsWire in its storage. We call the superclass' `mark_seen`, because our own `mark_seen` is going to look like this:

```
use Time::Piece;
sub mark_seen {
    my ($self, $head) = @_;
    $self->SUPER::mark_seen($head);
    my $dbh = DBI->connect(
        "dbi:SQLite2:$self->{history}"
    ) or return;
    $dbh->do(
        "insert into history (link, flRead,
            flFollowed, dateMarked)
            values (?, 1, 0, ?)"
    );
    $self->encode_headline($head);
}
```

```
localtime->strftime("%Y%m%d"));
}
```

`Time::Piece` is a handy little module that allows us to parse and format the current time in various ways, and we can use that to spit out the date in the format used by the history database. Again, we need to encode the headline. Once we've done that, we put a row into the history file so that it's marked when we go back to using NetNewsWire.

Finally, we need to do two things: The first is to implement `encode_headline`. We need to prepare the headline and description by trimming leading and trailing space from them:

```
sub encode_headline {
    my ($self, $head) = @_;
    my $d = $head->description;
    my $h = $head->headline;
    chomp ($d, $h);
    $d =~ s/^\[\n\s\]//; $d =~ s/\s+$//;
    $h =~ s/^\s+//; $h =~ s/\s+$//;
```

Then we return the URL bracketed with the length of the description and then length of the headline:

```
return $d.$head->url.$h;
}
```

Finally, we can create these `XML::Feed::NNW1` objects in our recursive closure:

```
my $url = $feed->{rss}->value;
my $name = $feed->{name}->value;
my $rss = XML::RSS::Feed::NNW1->new(
    url => $url,
    history => $db
);
$labels{$rss} = (" " x $depth).$name;
```

And we have to tell press not to use our static list of feeds, but instead to use the `Press::NNW1` module to get its list:

```
use Press::NNW1;
my ($values, $labels) =
    Press::NNW1->get_feedlist();
$feedbox->values($values);
$feedbox->labels($labels);
bolden_news();
```

And now press can read and write NetNewsWire 1 histories!

Moving On

Let's take a step back and look at what we have: As we mentioned earlier, we now have a main program (press) and a separate library that handles everything about the subscription and history information. When NetNewsWire 2 hits the big time, all we need to do is create a `Press::NNW2` library, which reads the plist files and formats for that application. Someone is working on having a `Press::Bloglines` library to interface to that as well; by making press modular, anyone can create back ends to whatever RSS syndication service they like.

Using the same modular system, it would be easy to create a program that read-in syndication data from any source and reproduced it in another format, to aid migration between RSS applications. But that's not really what I had in mind for press—I just wanted something to keep me up-to-date with the world.

TPJ



Private CPAN Distributions

brian d foy

In the January 2004 issue, I wrote about “CPAN in the Desert.” I was using a “minicpan” (a minimal, local copy of the Comprehensive Perl Archive Network) to install modules. Since then, the “private” minicpan idea has gone even further and I can now add my own modules to my local copy of CPAN, so I can install them with `CPAN.pm` or `CPANPLUS`. I can even burn my modified minicpan to a CD and give it to other people.

When I wrote the first article, I had no connection to any sort of network. I certainly didn’t have space on my hard drive for all 3 GB of the full CPAN archive, so I was a bit stuck if I wanted modules I hadn’t already installed. Randal Schwartz’s minicpan program came to the rescue. His program downloads only the latest versions of the modules on CPAN, and it also ignores large files such as Perl distributions. Instead of 3 GB, I end up with about 400 MB: Certainly small enough to fit on a CD.

Since then, Ricardo Signes turned Randal’s original minicpan script into a module, `CPAN::Mini`. That really doesn’t mean much to most people because Ricardo’s distribution comes with an updated minicpan script that does the same thing as the original, although it is more configurable. You can even install `CPAN::Mini` and simply use the minicpan script without ever looking at the code.

Later, Shawn Sorichetti came along with another module, `CPAN::Mini::Inject`, that can take one of my private modules and shoehorn it into my local CPAN mirror (and it works for both the full and mini versions because they have the same structure). I find this especially handy for clients who hire Stonehenge to write private modules. Like any good Perl programmer, we reuse code from CPAN, so our work has dependencies. Our clients need to be able to install these dependencies even if they are behind a firewall or restricted from installing things from an external CPAN mirror. By giving them a CD with everything on it, and setting things up so they can install everything with `CPAN.pm`, we can avoid various acrobatics to get around local policy or restrictions.

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

Indeed, for the most restrictive cases, we can even make a micro-`cpan` by removing everything except the stuff that we need.

To inject a module into my local repository, I need a module to start with. I’ll use a fictional module, `Inject::Test`, which is magically ready to distribute as `Inject-Test-1.01.tgz`. I don’t need to write a script to use `CPAN::Mini::Inject` because it comes with one already, named “`mcpani`.”

Before I can use `mcpani`, which I can also use as a replacement for minicpan, I need to configure it. As of version 0.36, `mcpani` looks in one of four places for a configuration file, in this order:

- A file pointed to by the environment variable `MCPANI_CONFIG`
- `$HOME/.mcpani/config`
- `/usr/local/etc/mcpani`
- `/etc/mcpani`

On my PowerBook, where I am the only user, I choose `$HOME/.mcpani/config`. My configuration file is simple; see Example 1.

The “local” directive tells `mcpani` where the minicpan will be. This is the smaller version of the canonical CPAN. The “remote” directive tells `mcpani` where to fetch files for the minicpan. I want to use passive FTP because outside servers cannot create a connection to my laptop, and I want the directory permissions to be 0755. The “repository” directive tells `mcpani` where to put the modules I want to inject into the minicpan. The repository keeps my private stuff separate so the minicpan script doesn’t lose that data when it mirrors the real CPAN and overwrites the files `CPAN::Mini::Inject` needs to modify.

Before I can start anything, I need a minicpan, which I’m going to put in `/MCPANI`. Before I can inject private modules into the local mirror, I need a local mirror. I tell `mcpani` to update the local mirror (which also creates it the first time), and I add the `-v`

```
local: /MCPANI
remote: ftp://ftp.cpan.org/pub/CPAN ftp://ftp.kernel.org/pub/CPAN
repository: /myMCPANI
passive: yes
dirmode: 0755
```

Example 1: `mcpani` config file.

switch so it operates in verbose mode. This is going to take a while, so I want to see what mcpani is doing.

```
% mcpani --mirror -v
```

When this completes, I'm at the same point I was at with Randal's original minicpan script. I have a minimal version of CPAN that includes only the latest versions of the modules.

*Once I go through all of my
private modules and inject them
into my local CPAN mirror, I can
burn that mirror to CD*

Now I want to add private modules to my mirror so I can install them as if they were on CPAN. I could do that on my own by editing a lot of files and copying archives to the right places. Although mcpani is going to do this for me, I like to know how things work.

CPAN automatically maintains several files so that tools like CPAN.pm can find the distributions. Starting from the minicpan root directory, the file `~/modules/02packages.details.txt` contains entries that give the module name, the latest version, and the file location in `~/authors/id`. This is how CPAN.pm finds the file it needs to install, and this is where I need to add information about modules that I want to inject into my minicpan. Example 2(a) shows the line from `02packages.details.txt` for my *Business::ISBN* module.

Additionally, in my author directory, `~/authors/id/B/BD/BDFOY`, CPAN maintains a `CHECKSUMS` file that has signature information about the distributions, including sizes and MD5 checksums. CPAN.pm checks these to ensure it has the right file. Before I can inject modules, I need to update this file so it contains the information for my private module. Here's the entry for my *Business::ISBN* module:

```
'Business-ISBN-1.79.tar.gz' => {  
  'mtime' => '2004-12-14',  
  'md5-ungz' => '82a4626a451ab68ab96c143d414ced96',  
  'md5' => '318f1851ccb236136c1d1f679539e08d',  
  'size' => 350871  
},
```

Now it's time to actually inject a distribution. I have my distribution file, `Inject-Test-1.01.tar.gz`. First, I have to add it to my repository. That's the private staging area.

```
% mcpani --add --module Inject::Test --authorid BDFOY  
--modversion 1.01 --file Inject-Test-1.01.tar.gz
```

When I look in `/myMCPANI`, I see a new directory, `authors`, and a file called `"modulelist"`. The `authors` directory has the same structure as the `authors` directory in CPAN: There is a subdirectory named `"id"` that contains directories for each letter of the au-

thor ID, which in turn has directories for the first two letters, and then the full ID (CPAN used to have a flatter directory structure, but now that it has about 4000 author IDs, it needs to separate them into directories). This is still just my repository though.

I haven't injected *Inject::Test* into my minicpan yet. I still need to move the file into my minicpan author directory and modify `02packages.details.txt` and `CHECKSUMS`. The `--inject` switch to mcpani takes modules from my repository and puts them into my local minicpan.

```
% mcpani --inject -v  
Injecting modules from /myMCPANI  
/MCPANI/authors/id/B/BD/BDFOY/  
Inject-Test-1.01.tar.gz ... injected
```

When I look in my author directory, I find that `Inject-Test-1.01.tar.gz` is there. Inside my `CHECKSUMS` file, I now have an entry for *Inject::Test*:

```
'Inject-Test-1.01.tar.gz' => {  
  'mtime' => '2005-01-28',  
  'md5-ungz' => '61aca20c21b0227233f6b2754a3efbe4...',  
  'md5' => '0cbb2e70f7c71afa675d0c7c1b52faf0...',  
  'size' => '25486'  
},
```

In the `02packages.details.txt`, I find a line for my private module, which allows CPAN.pm and other tools to install this module as if it really was on CPAN; see Example 2(b).

Once I go through all of my private modules and inject them into my local CPAN mirror, I can burn that mirror to CD (or something else) and give that to clients. On the CD, they should have everything they need to install the module and its dependencies. Since a minicpan is only about 400 MB, I have plenty of room on a standard CD to include external libraries such as `expat`, `gd`, or various other things the client may need. I don't have to worry about their network situation, firewalls, company policy, or the myriad other things that keep people from installing the things they need.

When I update, I just update the mirror, which wipes out my changes to `02packages.details.txt` and `CHECKSUMS`. Since my private modules are in a separate place, I simply reinject everything. To automate the process, I can create my own mcpani script using *CPAN::Mini::Inject*.

Life with CPAN just gets easier and easier.

References

CPAN: <http://www.cpan.org>

minicpan: <http://www.stonehenge.com/merlyn/LinuxMag/col42.html>

CPAN::Mini: <http://search.cpan.org/dist/CPAN-Mini>

CPAN::Mini::Inject: <http://search.cpan.org/dist/CPAN-Mini-Inject>

TPJ

```
(a)  
# line from ~/modules/02packages.details.txt  
Business::ISBN 1.79 B/BD/BDFOY/Business-ISBN-1.79.tar.gz  
  
(b)  
Inject::Test 1.01 B/BD/BDFOY/Inject-Test-1.01.tar.gz
```

Example 2: (a) Line that allows CPAN.pm to find the *Business::ISBN* distribution; (b) line that allows the same for *Inject::Test*.



Perl Core Language

Jack J. Woehr

Perl Core Language is part of Paraglyph's "Little Black Book" series, billed as "concise problem solvers." The content of *Perl Core Language* consists primarily of chapters in two parts. The first part of each chapter is the In Brief section (it might simply have been called "Brief"), a summary of the theme that unifies the chapter. This is followed by Immediate Solutions, topics (often occupying less than a full page), such as "Removing Trailing and Leading Spaces," "Using Array References as Hash References," or covering one of the scores of Perl function calls described by the book.

The book takes you from the basic language up through object Perl, Modules, CGI, XML, and Perl Debugging. The industry ritual of two "on beyond zebra" chapters at the end goes unchallenged: Here the extras are Chapter 20 on counters, guestbooks, e-mailers, and secure scripts; and Chapter 21 on multiuser chat, cookies, and games.

Though it starts with a Chapter on "Essential Perl" covering the boilerplate topics of downloading and installing Perl, making sure the shell interpreter can execute your scripts on any conceivable platform, language basics, and so on, *Perl Core Language* is not a book for learning Perl (nor for learning XML—you should be well grounded before you tackle Chapter 19, "Perl and XML"). It's a book for those who have learned Perl, and probably umpteen other languages, which explains why they need such a book open on their desk as they dive into a Perl project or merely into a day's coding. The book is the digested manual, often paired with clarifying examples composed by a skilled exigete for application on the fly. Hundreds of useful (albeit shallow) headings fit into a book under the application of such a formula, but the concision comes at a price. Holzner gets into the rhythm of his formula to the extent that, occasionally, he appears not to have parsed what he was writing for sense, as in the *sprintf* article:

- %e - A floating-point number, in scientific notation
- %E - Like %e, but using an uppercase E.
- %f - A floating-point number, in fixed decimal notation.
- %g - A floating-point number, in %e or %f notation
- %G - Like %g, but with an uppercase G

In case you missed the error, read the %G line again. %G most definitely does not produce an "uppercase G" in its output.

Perl Core Language is another one of these books whose blurbs reference Perl 6.0 ("[C]overs the current version...as well as highlighting critical features of the upcoming 6.0 version"). Anyone following the developers' list for Perl 6 (perl6-internals@perl.org or in entertaining the weekly summary at <http://www.perl.com/pub/a/2005/01/p6pdigest/20050118.html>) knows that if you want to know about Perl 6, you start at <http://dev.perl.org/perl6/>. If you don't understand what you read there, you don't really want to

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

Perl Core Language, Second Edition

Steven Holzner

Paraglyph Press, 2004

528 pp., \$29.99

ISBN 1932111921

know right now, anyway. In any case, for the target reader of *Perl Core Language*, the contours of Perl 6 can be most usefully summed up by the Magic 8-Ball: "Ask again later."

Certain other literary aspects of the work rub this reviewer the wrong way. First of all, prolific author Steven Holzner is an old hand at tech writing. Amazon offers 96 (!) of his books. Yet, oddly enough, none of these previous authorships are listed in this book under the "About the Author" heading. Second, we're sure Holzner wouldn't program a new object class merely to encapsulate one-to-one the capabilities of an extant class. Then why are the "Acknowledgments" solely aimed at the book's production team? Shouldn't the roles played by these individuals simply appear de rigueur inserted into the book by the publisher as "Credits" without reference to the author's wishes? Normative these days in tech book "Acknowledgments" are frank comments like, "I could never have done it without my wife Hroswitha, who entered therapy so she could handle my locking myself in my basement office for two years to write this book for a \$7000 advance against nonexistent royalties." Now, if Holzner had offered praise to these individuals for doing their jobs skillfully despite their being woefully underpaid and abused by seven layers of lackluster management or something of the sort, there would be the kind of value added above and beyond simple credits that would justify calling it an "Acknowledgment."

The web site <http://www.paraglyphpress.com/> turns out to be a tumble-down shed since the new barn got raised: <http://paraglyph.oreilly.com/>. There, you'll find the link to *Perl Core Language* pointing to <http://www.oreilly.com/catalog/1932111921/>. Hardly an intuitive URL, but consolidation marches onward in tech book publishing as in all other industries. Anyway, once arrived at the cited URL, you will find none of the usual amenities: no table of contents, no sample chapter, no downloadable example code. Everything about this effort bespeaks itself of hurry, which is hardly unusual hereabouts, but rush reaches an apogee in the present case wherein the same author publishes several other books in the course of the same 12 months. Steven Holzner clearly understands Perl, has impressive authorship credits, and is approaching rough parity with Herbert Schildt, another workhorse of this industry. If only Holzner, at the peak of his form, had time to stop and smell the roses!

TPJ