PERL
Power Programming

a language by Larry Wall

Practical Extraction and Report Language
or
Pathologically Eclectic Rubbish Lister

Tom Christiansen
<tchrist@mox.perl.com>

There's more than one way to do it!

What is Perl?

- **Perl is a general-purpose programming language that makes it remarkably easy to program easy and medium-hard things, yet still possible to program seriously non-easy things.**
- **Optimized for text, Perl can also handle binary data, thus significantly extending problem domain. It is not just about text processing, nor is it just about the Web!**
- **Perl's process, file, and text manipulation facilities make Perl particularly well-suited for tasks involving quick prototyping, system utilities, software tools, system management tasks, data base access, graphical programming, and world wide web programming.**

What Does Perl Look Like?

- **Despite allegations that it's just C code viewed over a modem with line-noise, it's actually more like interpreted C with the Unix scripting languages *sh*, *sed*, and *awk* thrown in. Perl will be easier if you already know a little bit about those — the more, the better.**

- **Perl also derives very small elements from a wide range of other tools and languages, including Ada, Algol, APL, BASIC-PLUS, cpp, C++, Cobol, csh, emacs, Fortran, Lisp, sort, python, test, tr, troff, and vi. Don't worry: you don't need to know these, but if you do, you'll occasionally be reminded of them.**

When Not to Use Perl?

- **When your boss forbids it — but consider replacing them :-).**
- **When you already have an existing application written in another language that's all done (and done well), or you have an application language specifically designed for a certain task (e.g. Prolog).**
- **Perl's efforts at backwards compatibility can sometimes get in the way of clean programming. Some of this is due to its not being a purely reference-based language like Scheme or Python.**
- **For various reasons, Perl is probably not well-suited for applications like device drivers, large-scale simulations, or other time- or space-critical code. While the new native-code compiler may change this somewhat, Perl remains a dynamically typed language, not a statically typed one.**

Features

- Easy to get started on due to similarity to familiar UNIX tools.
- Faster program development.
- Faster execution than shell script equivalents.
- Translators for old *find*, *sed*, and *awk* scripts.
- Comes with a symbolic debugger, written in Perl.
- Portable across many different architectures.
- No arbitrary limits (like string length).
- Can dynamically link C library code into running Perl.
- Built-in security features (tainting, safeperl)
- Good for writing reusable library code (including object-oriented).
- Fits nicely into UNIX tool-and-filter philosophy.
- It's free!

Compiled or Interpreted?

- The Perl language definition doesn't specify whether it's compiled to machine code (like C or Fortran), strictly interpreted (like Tcl or the command shells), or byte-compiled (like Java). Some languages manifest all three implementations, like BASIC or Pascal.
- When you run a Perl program, the compiler builds a parse tree and hands this off to the interpreter to execute, so Perl's essentially byte-compiled.
- A multimoded compiler backend in late alpha is available from CPAN (described below).

```
$ perl -MO=B     # byte-compiled
$ perl -MO=C     # non-optimized C code
$ perl -MO=CC    # optimized C code
```

The Compiler Myth

- **Look in `/CPAN/authors/Malcolm_Beattie/` for the latest version of the compiler.**
- **Compiling to C does not make Perl code not make your code smaller, faster, more portable, nor more secure:**
  - **It's still just as big, at least if you don't have a `libperl.so` lying about.**
  - **While sometimes it's faster, usually this is minimal. You won't make CGI programs faster this way.**
  - **Binaries aren't portable.**
  - **Security through obscurity isn't.**
- **Legitimate advantages are that you save a bit on compile time, and it's easier to distribute in compiled form without fear of what's where.**

Getting Sources and Info from CPAN

- **The best source of Perl information is the CPAN archives. See `http://www.perl.com/perl/` or `ftp://ftp.perl.com/perl/` for a list of these.**
- **The CPAN multiplexer dynamically redirects you to a fast site based on incoming domain name:**

`http://www.perl.com/CPAN` *[selection]*
`http://www.perl.com/CPAN/` *[dispatch]*
`http://www.perl.com/CPAN/README.html`
`http://www.perl.com/CPAN/modules/`
`http://www.perl.com/CPAN/doc/`
`http://www.perl.com/CPAN/ports/`
`http://www.perl.com/CPAN/src/latest.tar.gz`

Perl Publications

- *Learning Perl* 2ed ['Llama']
- *Learning Perl for Win32* ['Gecko']
- *Cross Platform Perl*
- *Programming Perl* 2ed ['Camel']
- *Advanced Perl Programming* ['Panther']
- *Web Client Programming with Perl* ['Pelican']
- *Mastering Regular Expressions* ['Owls']
- *The Perl Journal* Magazine

Other Published Resources

- **Web-related publications:**
  - *HTML: The Definitive Guide*
  - *CGI Programming in C and Perl*
  - *Web Client Programming with Perl* ['Pelican']
  - *How to Setup and Maintain a Web Site*
  - *WebTechniques* Magazine
- **Online manpages are essential: e.g. perl, perldata, perltoot, and perlfaq. If 'man CGI' isn't found, try 'perldoc CGI'.**
- **On the web, start with** *http://www.perl.com/perl/*

Other Sources

- **The Frequently Asked Questions list from CPAN can also be found at** *http:/www.perl.com/perl/faq/* **from your web browser.**
- **Man pages are at** *http:/www.perl.com/perl/manual/*
- **Expert docs are at** *http:/www.perl.com/perl/all_about/*
- **USENET newsgroups *comp.lang.perl.announce*, *comp.lang.perl.modules*, *comp.lang.perl.misc*, and *comp.lang.perl.tk* for discussion, questions, comments, and examples. *comp.infosystems.www.authoring.cgi* also carries a lot of Perl traffic.**

Perl instead of Shell Tools

```
# tail -r, no limit
perl -e 'print reverse <>'
# save on execs
find /tmp -mtime +3 -print | perl -nle unlink
# post-date files
perl -e '$n=7*24*60*60+time(); utime($n,$n,@ARGV)' /tmp/*.c
# add first and penultimate columns
perl -lane 'print $F[0] + $F[-2]'
# in-place edit of *.c files changing all foo to bar
perl -p -i.bak -e 's/\bfoo\b/bar/g' *.c
# find palindromes
perl -lne 'print if $_ eq reverse' /usr/dict/words
```

Perl instead of Shell

- **Consider the following Perl code; what general observations can you make about Perl coding from these statements?**

```
#!/usr/bin/perl
$kernel = `cat /vmunix`;   # wow!

print "
  The woods are lovely, dark and deep,
  But I have promises to keep,
";

chdir "/tmp" || exit 1;
```


Perl instead of C

- **Now, consider the following Perl code; what general observations can you make about Perl coding from *these* statements?**

```
open(KERNEL, "< /vmunix");

while (read(KERNEL, $block, 4096) > 0) {
    $blockno++;
    if (index($block, "xyzzy") >= 0) {
        printf("cookie in 4k block %d\n",
                $blockno);
    }
}
```

Perl for Systems Programming

```
use IO::Socket;
use Net::hostent;

die "usage: $0 port\n" unless @ARGV == 1;

my $port = shift;
my $server = IO::Socket::INET->new(
                    Proto     => 'tcp',
                    LocalPort => $port,
                    Listen    => SOMAXCONN,
                    Reuse     => 1);
die "can't setup server" unless $server;

print "[Server $0 accepting clients on port $port]\n";
while (my $client = $server->accept()) {
    my $hostinfo = gethostbyaddr($client->peeraddr,AF_INET);
    printf "Connect on port %s from %s at %s\n", $port,
        $hostinfo->name || $client->peerhost,
        scalar localtime;
    $client->close();
}
```


Perl instead of Fortran

- It's easy to deal with multidimensional arrays in modern versions of Perl:

```
for $x (1 .. 10) {
    for $y (1 .. 10) {
        for $z (1 .. 10) {
            $a[$x][$y][$z] =
                $x ** $y + $z;
        }
    }
}
```

Perl instead of Lisp

```
#!/usr/bin/perl -n
permut([split], []);
sub permut {
    my @car = @{ $_[0] };
    my @cdr = @{ $_[1] };
    unless (@car) {
        print "@cdr0;
    } else {
        my(@newcar,@newcdr,$i);
        foreach $i (0 .. $#car) {
            @newcar = @car;
            @newcdr = @cdr;
            unshift(@newcdr, splice(@newcar, $i, 1));
            permut([@newcar], [@newcdr]);
        }
    }
}
```


Perl instead of Scheme

```
$f1 = mkcounter(10);
$f2 = mkcounter(20);
print $f1->(), $f2->(), $f1->();
```

11 21 12

```
sub mkcounter {
    my $start = shift;
    return sub { return ++$start };
}
```

More Perl instead of Lisp/Scheme

```
# sort hostnames by domainname

  map  { print $_->[0] }

  sort { $a->[1] cmp $b->[1] }

  map  {
        @f = split /\./;
        if ( /^\s*\d+(?:\.\d+(?:\.\d+(?:\.\d+)?)?)?\s*$/ ) {
            splice(@f, $#f, 0, (0) x (4-@f));
            @f = map { sprintf "%03d", $_ } reverse @f;
        }
        [ $_, sprintf "%20s " x @f, reverse @f ];
     } <ARGV>;
```


Perl instead of Tcl/Tk

```
$frame1 = Frame $topwin -relief => "sunken", -bd => 2;
$lister = Listbox $frame1 qw{
            -selectmode browse
            -font fixed
            -bd 2
          };
$art_button     = Button $frame2
    -text             => "article",
    -relief           => "raised",
    -background       => yellow1,
    -activebackground => DarkOrchid1,
    -command          => [
        sub {
            get_article();
            $hdr_color = ($hdr_color + 1) % @hdr_colors;
            configure $hdrs
              -text       => $hdrtext,
              -background => $hdr_colors[$hdr_color];
        }
    ];
```

Perl instead of JavaScript

```
<%@ LANGUAGE = PerlScript%>
<HTML><HEAD>
<TITLE>Testing loops in PerlScript</TITLE>
</HEAD>
<BODY> <BODY BGCOLOR=#FFFFFF>
<!-- ActiveWare PerlScript sample
    PerlScript:  The coolest way to web solutions.  -->
</TD></TR></TABLE>
<H3> Repeating a Loop While a Condition Is True</H3><P>
Use the while keyword ....
<% #First example of the While keyword
  $MyNum = 20;
  $Counter = 0;
  while ($MyNum >10) { $MyNum--; $Counter++; };
%>
The first loop made <%= $Counter %> repetitions.<P>
<% $Counter = 0; $MyNum = 9;
 $Counter++; $MyNum--;
 while ($MyNum > 10) { $MyNum--; $Counter++; };
%>
The second loop made <%= $Counter %> repetitions.<P>
```


Perl instead of C

```
%TV = (
  flintstones => {
    series   => "flintstones",
    nights   => [ qw(monday thursday friday) ],
    members  => [
      { name => "fred",    role => "lead", age  => 36, },
      { name => "wilma",   role => "wife", age  => 31, },
      { name => "pebbles", role => "kid",  age  =>  4, },
    ],
  },
);
$TV{flintstones}{members}{pebbles}{age}++;
$rec = {
    TEXT      => $string,
    SEQUENCE  => [ @old_values ],
    LOOKUP    => { %some_table },
    THATCODE  => \&some_function,
    THISCODE  => sub { $_[0] ** $_[1] },
    HANDLE    => \*STDOUT,
};
```

Building Data Structures

- We don't usually bother with classic data structures like records, linked lists, trees in Perl because the built-in arrays and hashes take care of most of those needs.
- Nonetheless, building them is very easy. For nearly all of them, the key is to use an anonymous hash as the fundamental unit in your data structure. For example:

```
$record = {
    NAME   => "Jason",
    EMPNO  => 132,
    TITLE  => "deputy peon",
    AGE    => 23,
    SALARY => 37_000,
    PALS   => [ "Norbert", "Rhys", "Phineas"],
};
```


Using Structures

- Once you create the record, it can be stored in hashes or arrays, or both:

```
# use %ByName, a hash of records
$ByName{ $record->{NAME} } = $record;

# use @Employees, an array of records
$Employees[ $record->{EMPNO} ] = $record;

# use @ByAge, an array of arrays of records
push @{ $ByAge[ $record->{AGE} }, $record;
```

- Now use those structures normally:

```
$ByName{"Jason"}{SALARY} *= 1.035;

@peons = grep { $_->{TITLE} =~ /peon/i } @Employees;

foreach $rec ( @{ $ByAge[23] } ) {
    print $rec->{NAME}, "\n";
}
```

More Perl instead of C (binary tree)

```
sub insert { my($tree, $value) = @_;
  unless ($tree) {
    ($_[0] = {})->{VALUE} = $value;
    return;
  }
  if    ($tree->{VALUE} > $value) { insert($tree->{LEFT},  $value) }
  elsif ($tree->{VALUE} < $value) { insert($tree->{RIGHT}, $value) }
  else                            { warn "dup insert of $value\n"  }
}
sub in_order { my($tree) = @_;
  return unless $tree;
  in_order($tree->{LEFT});
  print $tree->{VALUE}, " ";
  in_order($tree->{RIGHT});
}
sub search {
  my($tree, $value) = @_;
  return unless $tree;
  if ($tree->{VALUE} == $value) { return $tree }
  search($tree->{
        ($value < $tree->{VALUE}) ? "LEFT" : "RIGHT"}, $value)
}
```


Perl instead of C++ (BinTree #1)

```
package BinTree;
use strict;
# CONSTRUCTORS
sub new($);    # $t = BinTree->new();
sub copy($);   # $t2 = $t1->copy();
# DESTRUCTOR unneeded: Perl does its own GC
# INSTANCE METHODS
sub insert($); # $t->insert($node);
sub remove($); # $t->remove($node);
sub search($); # $n = $t->search($value)
sub inorder($);    # $t->inorder(\&func);
sub postorder($);  # $t->postorder(sub { });
sub preorder($);   # $t->preorder(\&whatever);
```

More Perl instead of C++ (BinTree #2)

```
sub new($) {
    my ($class, $value) = @_;
    $class = ref $class if ref $class;
    my $tree  = {
        VALUE   => $value,
        LEFT    => undef,
        RIGHT   => undef,
    };
    bless($tree, $class);
    return $tree;
}
sub left()  { $_[0]->{LEFT} }
sub right() { $_[0]->{RIGHT} }
sub value() { $_[0]->{VALUE} }
```


More Perl instead of C++ (BinTree #3)

```
sub inorder($) {
    my($tree, $coderef) = @_;
    $tree->left->inorder($coderef)   if $tree->left;
    $coderef->($tree->value);
    $tree->right->inorder($coderef)  if $tree->right;
}
sub search($) {
    my($tree, $value) = @_;
    return unless $tree;
    if ($tree->value == $value) { return $tree }
    my $branch = ($value < $tree->value)
                    ? "left"
                    : "right";
    return  $tree->$branch
          && $tree->$branch()->search($value);
}
```

More Perl instead of C++ (Stack #1)

```
package Stack;

sub new($) {
    my $class = shift;
    my $new_stack  = [];          # allocate empty array
    bless($new_stack, $class);    # make it an object
    return $new_stack;
}
sub copy($) { my $self  = shift;
    my $clone = $self->new();
    @$clone = @$self;
    return $clone;
}
sub push($@) { my $self = shift;
    push(@$self, @_);
}
sub pop($) { my $self = shift;
    confess("Stack empty!") unless @$self > 0;
    return pop(@$self);
}
```


More Perl instead of C++ (Stack #2)

```
use overload ('==' => \&is_equal);
sub is_equal($$$) {
    my ($s1, $s2) = @_;
    if ($s1->size() != $s2->size()) {
        return 0;
    }
    for (my $i = 0; $i < $s1->size(); $i++) {
        if ($s1->[$i] != $s2->[$i]) {
            return 0;
        }
    }
    return 1;
}

use overload ( '""'  => \&stringify );
sub stringify($) {
    my $self = shift;
    return '[' . join(",", @$self) . "]";
}
```