A Lightning Tour of Perl

October 13, 2005

Contents

What's Perl good for? A short answer is system programming and text processing. What makes it good? Its concise syntax, regular expression engine, extensive function and module library, and excellent documentation. For system programming it's a complete programming language so it can do much more than shell scripts can. For text processing it simplifies data extraction and output generation. This tutorial (http://fuslpcjs.culham.ukaea.org.uk/tutorials/perltour/html/index.html) explores some of Perl's key features and advantages. It's also available in pdf.

The tutorial is built around three small programs showing Perl at work. The code and associated files are available through links in the html version of this document, so you can download and run them yourself. Perl is big, and this tutorial is necessarily limited, so there's lots more to find in the extensive resources listed at the end. Among important topics not covered here are subroutines and modular programming, essential elements of larger Perl programs.

1 First Example

Let's dive in with a small program which changes patterns in text files. This has lots of applications, for example changing variable names across code files. It also provides a template for other file processing programs.

```
#! /usr/bin/perl -w

use strict;
```

```
{
   my ($file, $line, $contents);

   # process files
   foreach $file (@ARGV) {
      if (-f $file) {
         print("processing $file\n");

         # read file contents
         open(FILE, $file) or die("can't open $file for read");
         $contents = '';
         while ($line = <FILE>) {
            $contents .= $line;
         }
         close(FILE) or die("can't close $file after read");

         # change patterns throughout
         $contents =~ s/pattern1/pattern2/g;
         $contents =~ s/pattern3/pattern4/g;

         # write out edited file contents
         open(FILE, ">$file") or die("can't open $file for write");
         print(FILE $contents);
         close(FILE) or die("can't open $file for write");
      }
   }
}
```

This covers quite a bit of Perl ground:

```
#! /usr/bin/perl -w
```

First the #! line identifying the interpreter the shell will call to run the program. If this doesn't resolve to a valid interpreter the shell will complain. The program must have execute permission. The -w flag enables warnings and should always be used.

```
use strict;
```

This invokes the strict pragmatic module, which among other things insists that all variables be declared.

```
   my ($file, $line, $contents);
```

A 'my' variable declaration, local to enclosing braces. Perl variable names start with special characters, discussed below.

```
   # process files
   foreach $file (@ARGV) {
```

One of Perl's looping statements. @ARGV is a system array which contains the program command line arguments; the string variable $file is assigned each argument in turn and the loop body is executed. Perl comments start with # and run to the end of the line.

```
    if (-f $file) {
```

Tests if the file is editable. If it is a directory the test will fail. Perl provides many different file tests.

```
        print("processing $file\n");
```

A C-style call of a Perl function; both shell and programming language syntaxes can be used. The double-quoted string features variable and backslash interpolation, one of Perl's syntactic strengths.

```
# read file contents
        open(FILE, $file) or die("can't open $file for read");
        $contents = '';
        while ($line = <FILE>) {
           $contents .= $line;
        }
        close(FILE)  or die("can't close $file after read");
```

The file is opened and the contents are read into a single string, which could be megabytes in size if required.

```
        # change patterns throughout
        $contents =~ s/pattern1/pattern2/g;
        $contents =~ s/pattern3/pattern4/g;
```

Perl's regular expression magic at work, changing all instances of one pattern to another in the string. You could add more lines like this, to change many patterns.

```
        # write out edited file contents
        open(FILE, ">$file") or die("can't open $file for write");
        print(FILE $contents);
        close(FILE) or die("can't open $file for write");
```

Finally the file is re-opened, this time for writing, and the modified contents are written out.

2 Data Types

2.1 Scalars

Scalar variables have single values and names prefixed with $.

```
my ($a, $b, $c, $d, $e);  # declares null scalars $a .. $e
$a = 5;           # decimal integer
$b = 0x5a;        # hex integer
$c = 2.7e-3;      # float
$d = 'string\n';   # single-quoted string (non-interpolating)
$e = "string\n";  # double-quoted string (interpolating)
```

The type of a scalar variable is weakly constrained. In principle a single variable, at different points in a program, could be assigned an integer, float, string or reference (pointer) value, though it's not generally a good idea to play fast and loose.

All arithmetic is done with double precision:

```
my ($a, $b, $c);
$a = 2;
$b = 3;
$c = $a/$b;  # 0.6666666666667
```

Numeric string values are converted to numbers if used in arithmetic expressions:

```
my ($a, $b, $c);
$a = '2';
$b = "3";
$c = $a/$b;  # 0.6666666666667
```

This makes it easier to extract numeric data from an ascii file.

There are different types of string literals. Double-quoted strings are subject to variable and backslash interpolation, while single-quoted strings are left alone:

```
my ($a, $b, $c);
$a = 'spherical tokamak';
$b = "MAST is a $a\n";  # 'MAST is a spherical tokamak' followed by a newline
$b = 'MAST is a $a\n';  # 'MAST is a $a\n'
```

'Here' document literal string syntax is the same as provided by Unix shells.

```
 my ($a, $b);
$a = 3;
$b = <<EOT;
Line 1
Line 2
Line $a
EOT
print($b);  prints 'line 1
                   line 2
                   line 3'
```

The label (EOT here) is arbitrary. You can quote it with single or double quotes to control interpolation. Default behaviour is to interpolate.

2.2 Arrays

Array variables have list values and names prefixed with @. In contrast, array elements are scalars and have names prefixed with $. Again, typing is weak, and a single array can contain mixed element types if appropriate.

```
my (@a);  # declares empty array @a
@a = (1, 2.1, 'four');
print($a[0]);  # 1
print($a[1]);  # 2.1
print($a[2]);  # four
$a[0] = 'one';  # $a is now ('one', 2.1, 'four')
```

push() and pop() treat an array as a stack:

```
my (@a);
push(@a, 1, 2.1, 'four');
```

This appends the list to the end of the array.

Note that scalars, arrays and hashes each live in a different namespace. $a[0] is the first element of array @a, but $a (without square brackets) is an unrelated variable. Normally you should avoid this in variable naming, but it can occasionally be useful:

```
my ($a, @a);  # declares null scalar $a and empty array @a
@a = ('one', 'two', 'three');
$a = join(', ', @a);  # 'one, two, three'
```

2.3 Hashes

The hash is the jewel in Perl's data type crown. A hash is an associative array, indexed by a string instead of a numeric index as with normal arrays. As the name suggests, Perl implements them using hash tables for fast lookup. Hash names have a % prefix. Again, individual hash values are scalars, with names prefixed with $. Hash indices use curly braces to distinguish them from ordinary array indices.

```
my ($b, $k, %a);  # declares null scalars $b, $k and empty hash %a
$a{'this'} = 1;
$a{'that'} = 'two';
$b = 'this';
print($a{$b});  # 1;
$b = 'that';
print($a{$b});  # two;
$b = 'them';
$a{$b} = 7.93e-5;
$k = join(', ', keys(%a));  # them, that, this
```

Note that hashes are unordered. If you walk them you see the key/value pairs in apparently random order. If this matters there are workarounds.

You can write hash literals in other ways. A hash is really just an ordinary array of key/value pairs with a fast lookup mechanism, so you can do:

```
my ($b, %a);
%a = ('this', 1, 'that', 'two', 'them', 7.93e-5);
$b = join(', ', values(%a));
print($b);  # 7.93e-5, two, 1
```

But you might prefer this:

```
%a = (this => 1,
      that => 'two',
      them => 7.93e-5);
```

Note that quotes around hash keys are optional.

2.4 References

Perl references play the role that pointers do in C. They make compound data structures possible because they are scalars, and so arrays and hashes can have them as values.

```
my ($ar, @a, @b);
@a = (1, 2, 3, 4);
$ar = \@a;  # $ar is now a reference to @a
@b = @$ar;  # copies @a to @b
print($$ar[0]);  # 1
```

I have to admit this syntax takes some getting used to, though in lots of cases it's transparent because of other syntactic wizardry.

You can also create a reference to an anonymous array using square brackets, or to an anonymous hash using curlies.

```
my ($ar, $br);
$ar = [1, 2, 3, 4];
print($$ar[0]);  # 1
$br = {this => 1, that => 'two', them => 7.93e-5};
print($$br{that});  # 'two'
```

2.5 Data Structures

How can you implement a C struct in Perl?

```
struct s {
   int a;
   float b;
   char *c;
};
struct s t = {1, 2.5, "string"};
```

With a hash:

```
my (%t);
$t{a} = 1;
$t{b} = 2.5;
$t{c} = 'string';
```

What about an array of C structs?

```
struct s t[100];
t[0].a = 2;
```

With an array of hashes:

```
my (@t);
$t[0]{a} = 2;
```

How about a hash of arrays? Later you'll see a dictionary (hash of words) where the values are the positions in a text where the word is found. As each word may appear more than once, and array of positions is required:

```
my (%d);
$d{dog}[0] = 34;
```

Alternatively you can operate explicitly on one of the sub-arrays:

```
push(@{$d{dog}}, 34);
```

Compound data structures like these, and more complicated ones, use references to achieve their effect. Perl dereferencing syntax can be a little daunting, but it's worth fighting through. There are more examples in code which follows.

3 Regular Expressions

Perl has an excellent regular expression engine which sets the standard by which others are measured. Regular expressions (regexps) are instances of a very powerful pattern matching language. They let you do amazing things with very little code, particularly given Perl's compact pattern matching syntax.

Here's an example of regexps at work. The job is to extract name/value pairs from a fortran namelist which contains many lines like this:

```
!----    JETtransp Update Mon Jun  7 09:33:24 BST 2004    --------------------
!HOST = pppl
FTIME = 0.343
KMDSPLUS = 1
MDS_PATH = '\top.INPUTS'
NLEBAL     =.true.     ! .true. to perform e- energy balance calculation  [ - ]
EOIN       = 20.0, 3.0, 0.55, 20.0, 3.0, 0.55 !                           [eV ]
 NLCO(1) = .T
 TBONA(1) =  8.899999E-02
```

Note the variable syntax for names and boolean values, variable spacing, mixture of single values and comma-separated lists, single-quoted strings, and comments. Each name/value pair occupies one line, so line-oriented processing is appropriate.

This short program does the job:

```
 #! /usr/bin/perl -w

use strict;

{
   my ($file, $line, $name, $key, $value, $boolean, $integer, $float,
       $string, $n, %nameValues);

   $name = '[A-Z][A-Z0-9_]+(?:\(\d\))?';
   $boolean = '\.true\.|\.TRUE\.|\.false\.|\.FALSE\.|\.T|\.F';
   $integer = '-?\d+';
```

```
   $float = '-?\d+\.\d*E?-?\+?\d*|-?\d*\.\d+E?-?\+?\d*';
   $string = '\'[^\']+\'';

   $file = 'namelist.txt';
   open(FILE, $file) or die("can't open $file");
   while ($line = <FILE>) {
      if (($line =~ m/^\s*($name)\s*=\s*($boolean)/) ||
          ($line =~ m/^\s*($name)\s*=\s*((?:$float\s*,?\s*)+)/) ||
          ($line =~ m/^\s*($name)\s*=\s*((?:$integer\s*,?\s*)+)/) ||
          ($line =~ m/^\s*($name)\s*=\s*($string)/)) {
         $nameValues{$1} = $2;
      }
   }
   close(FILE) or die("can't close $file");

   foreach $n (sort(keys(%nameValues))) {
      print("$n = $nameValues{$n}\n");
   }
}
```

After the declarations, which include hash %nameValues to store the results, match patterns are defined for names and values. Note the necessary use of single quotes to inhibit character and variable substitution.

```
   $name = '[A-Z][A-Z0-9_]+(?:\(\d\))?';
```

A name starts with a capital letter, followed by one or more capitals, digits or underscores. This is optionally followed by a digit in parentheses. (?: ...) defines a group which the final ? makes optional. Parentheses are special characters in regexps, so literals must be escaped with '\'. \d indicating a decimal digit is another special character.

```
   $boolean = '\.true\.|\.TRUE\.|\.false\.|\.FALSE\.|\.T|\.F';
```

A boolean value is either .true. or .TRUE. or ..... Here we have | for alternation, and \. for a literal '.'. Without the escape, '.' normally means any character except a newline.

```
   $float = '-?\d+\.\d*E?-?\+?\d*|-?\d*\.\d+E?-?\+?\d*';
```

Float syntax is a bit messy. The only required feature is a digit before or after a decimal point. This pattern is sloppy because it says that exponent digits are optional, but it works with sensible data.

```
   $integer = '-?\d+';
```

An integer is one or more digits with an optional minus sign. Because floats can start with an integer, you have to test the line for a float match before testing it for an integer.

```
   $string = '\'[^\']+\'';
```

A string is any sequence of non-single-quote characters in single quotes. Literal single quotes must be escaped here. ^ as the first character in square brackets means 'not the following characters'.

Moving on, the file is opened and read a line at a time. Each line is tested against a compound pattern of the form name = value, where the patterns discussed above are interpolated. This gets a bit complicated when the value is a list. Here is the simple case:

```
    if (($line =~ m/^\s*($name)\s*=\s*($boolean)/) ||
```

=~ is the match operator. This tests the line to see if it matches the pattern within the outer / markers. \s* is optional white space. The parentheses in the pattern capture the value matched by the sub-expression they contain, in string variables called $1, $2 etc. If a test succeeds, the captured name/value pair is written to the nameValues hash:

```
        $nameValues{$1} = $2;
```

Finally, the name/value pairs are printed in sorted order to prove the pudding.

4 Final Example

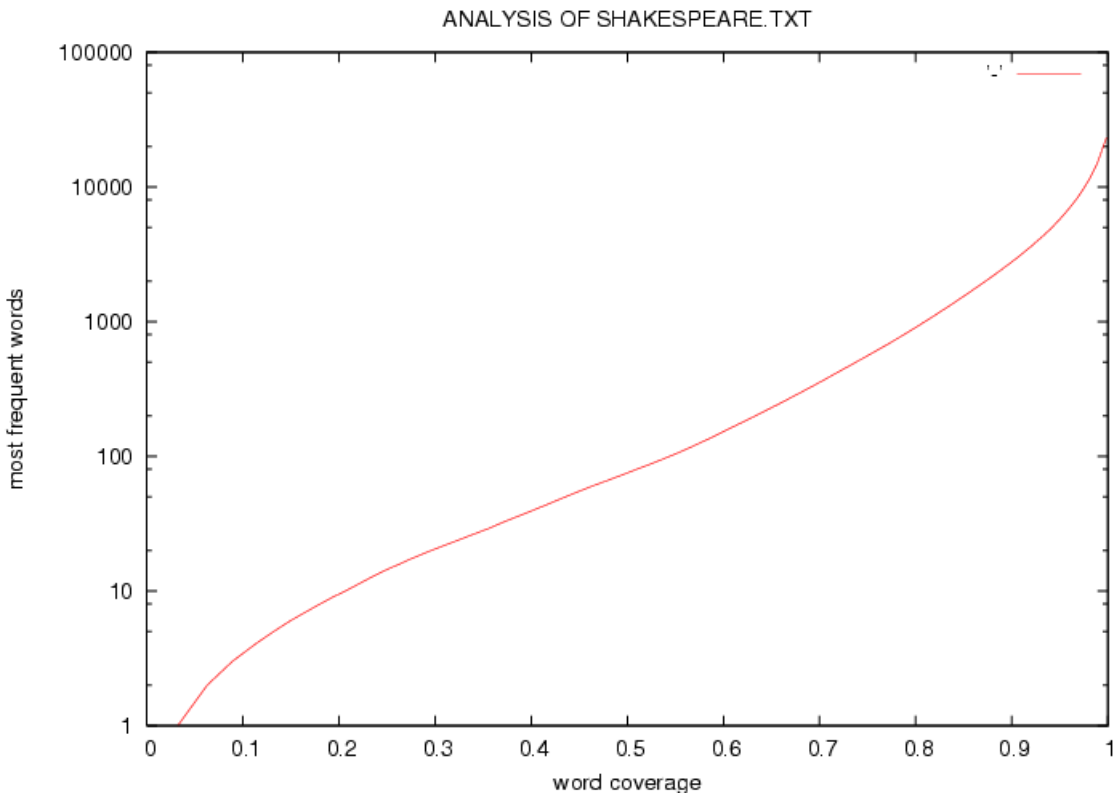
Tying all this together, here is a computational linguistics program which analyses word usage in a text file, applied to the Project Gutenberg (http://www.gutenberg.org) version of the complete works of Shakespeare:

```
 #! /usr/bin/perl -w

use strict;
use Text::Wrap;

{
   my ($sourceFile, $wordFile, $plotFile, $line, $contents, $word, $wordCount,
       $totalCount, $vocabularyCount, $incrementalCount, $coverage, $frequency,
       $i, $sortedWords, @words, @sortedWords, %index, %wordsByFrequency);

   # open source file and store contents in a string
   $sourceFile = 'shakespeare.txt';
   open(SOURCE, "$sourceFile") or die("can't open $sourceFile");
   while ($line = <SOURCE>) {
      $contents .= $line;
   }
   close(SOURCE) or die "can't close $sourceFile";

   # open analysis file and write header
   $wordFile = 'words.txt';
   open(WORDS, ">$wordFile") or die("can't open $wordFile");
   print(WORDS "ANALYSIS OF \U$sourceFile\E\n\n");

   # open plot file and write header
   $plotFile = 'commands.gnuplot';
   open(PLOT, ">$plotFile") or die("can't open $plotFile");
   print(PLOT <<EOF);
set log y
set title "ANALYSIS OF \U$sourceFile\E"
set xlabel "word coverage"
set ylabel "most frequent words"
set term postscript color
set out "plot.ps"
```

```
plot '-' with lines
EOF

   # compact file and count words
   @words = ();
   $contents =~ s/-/ /g;  # replace hyphens with spaces
   while ($contents =~ m/([^\s]+)/g) {  # extract space-separated words
      $word = $1;  # must copy $1!
      if (($word !~ m/[A-Z][A-Z]+/g) &&  # don't match eg 'LEAR'
          ($word !~ m/^[A-Z]+\.$/)) {    # don't match eg 'Lear.'
         $word =~ s/^\W+//;  # strip leading 0
         $word =~ s/\W+$//;  # strip trailing punctuation
         $word =~ s/'.*$//;  # strip everything after an apostrophe
         push(@words, $word);
      }
   }
   $totalCount = scalar(@words);
   print(WORDS "Total word count = $totalCount\n\n");

   # make word index
   %index = ();
   for ($i = 0; $i < $totalCount; $i++) {
      $words[$i] = "\l$words[$i]";  # lower case first character
      if (($words[$i] !~ m/_/) &&   # no underscores
          ($words[$i] !~ m/\d/)) {   # no digits
         push(@{$index{$words[$i]}}, $i);
      }
   }
   $vocabularyCount = scalar(keys(%index));
   print(WORDS "Vocabulary count = $vocabularyCount\n\n");

   # make inverted word index
   %wordsByFrequency = ();
   foreach $word (keys %index) {
      $frequency = scalar(@{$index{$word}});
      push(@{$wordsByFrequency{$frequency}}, $word);
   }

   # record word usage
   $coverage = 0;
   $incrementalCount = 0;
   print(WORDS "Key: [incremental count] (word frequency) {word count}\n\n");
   foreach $frequency (sort reverseNumeric keys(%wordsByFrequency)) {
      @sortedWords = sort(@{$wordsByFrequency{$frequency}});
      $sortedWords = wrap('', '', join(' ', @sortedWords));
      $wordCount = scalar(@sortedWords);
      $incrementalCount += $wordCount;
      $coverage += $wordCount*$frequency/$totalCount;
      print(WORDS "[$incrementalCount] ($frequency) {$wordCount}:\n");
      print(WORDS "$sortedWords\n\n");
```

```
      print(PLOT "$coverage $incrementalCount\n");
   }
   print(PLOT "e\n");
   close(WORDS) or die("can't close $wordFile");
   close(PLOT) or die("can't close $plotFile");
   system("gnuplot $plotFile") or die("can't execute gnuplot");
}


# sub numeric:
# For numeric sorting. Normal sorting is ascii. Usage 'sort numeric @list;'.

sub numeric ()
{
   $a <=> $b;
}


# sub reverseNumeric:
# For reverse numeric sorting (highest number first). Normal sorting is ascii.
# Usage 'sort reverseNumeric @list;'.

sub reverseNumeric ()
{
   $b <=> $a;
}
```

The program generates an analysis of word usage and the following word coverage plot. It takes 7 seconds to process the 5Mb source file on my 64-bit Linux system at home.



The plot shows that the 10 most frequently used words 'cover' 20% of the complete works, 100 words 55%, and 1000 words (~5% of the vocabulary) 80%. It gets progressively harder to improve your understanding of the text as your grasp of the vocabulary increases! Note that there are quite a few typos, and some proper names remain after filtering, so this analysis is only approximate.

5 Resources

For more information visit Perl.org (http://www.perl.org) and CPAN (http://www.cpan.org). There are lots of books about Perl. Standard works include Learning Perl, (http://www.oreilly.com/-catalog/lperl3/index.html)Programming Perl (http://www.oreilly.com/catalog/pperl3/index.html) (the bible), Perl Cookbook (http://www.oreilly.com/catalog/perlckbk2/index.html), and Advanced Perl Programming. (http://www.oreilly.com/catalog/advperl2/index.html) For everything about regular expressions read Mastering Regular Expressions. (http://www.oreilly.com/catalog/regex2/-index.html)

Finally, I've written lots of Perl code over the past few years (around 40,000 lines at the last count), for system programming, XML processing, and PCS configuration. For example, this document was written in XML and processed by my Perl XML toolchain. All this code is available for reference, and for use where it fits.

John Storrs