Paper 188-26

A Perl Primer for SAS® Programmers

David L. Cassell, OAO Corp.

ABSTRACT

The Perl programming language should be viewed as not a competitor of SAS®, but rather as a colleague. There are frequent places in web-based programming, as well as in the data validation and system administration work surrounding web programming, where Perl can work hand-in-hand with SAS® software. However, Perl is usually regarded as an arcane language which looks more like line noise than code. This paper is designed to serve as a quick introduction to Perl, with examples which show that Perl can be used in logical ways which are then easy to integrate into webpages.

IN THE BEGINNING...

Let's start with some relatively simple Perl code. We'll open up a file called many.bad.strings and count how many lines have the character '<' but do not have '</' .

```
#!/usr/bin/perl -w
use strict;
my $count = 0;
open FILE, 'many.bad.strings'
    or die "Can't look at strings: $!";
while (<FILE>) {
    if index($_, '<')  > 0 { $count++ ;}
    if index($_, '</') > 0 { $count-- ;}
}
close FILE or die "File is hung: $!";
print "The count is $count.\n";
my $extra = $count + $count*$count -
            $count**3 ;
print "The secret code number is $extra.\n";
```

Now first of all, this could easily be done in a DATA step within SAS®. You didn't need to learn any Perl to do this. And for those of you familiar with HTML, the presence of a '<' character and absence of '</' does not guarantee that you have found the beginning of an HTML element either! But this is a primer, so let's take this step by step.

The first two lines of the program can be considered boilerplate. The '#!' [or "shebang" as it is called in the unix world] and the path to the Perl program tell a unix operating system that this is a Perl program, and where to find the Perl executable to use in order to run the program. The '-w' turns on warnings, so that you get error-checking, as in the log of a SAS® program. The 'use strict' pragma tells Perl to be extremely careful with things like variable names.

In a DATA step, you could use a RETAIN statement to create a variable, initialize it, and make sure that its value is retained through iterations of the DATA step. Here, line 3 does the same. The keyword 'my' means that the variable is local instead of global, much as the %local statement in the SAS® macro language. Perl variables all have non-alphanumeric beginnings [scalars start with $, arrays start with @, associative arrays start with %, and references start with a \], but assignment is done as in SAS®, and the statement even ends with the familiar semicolon.

In a SAS® DATA step,you would use the INFILE statement to tell which file to open - but if the open failed you would have little recourse. Perl uses the open function in line 4, but provides extensive error-handling options. Here the program merely dies after printing out an error message which includes the 'special' variable $! [which holds the explanation of the error as the operating system has reported it to Perl]. All of Perl's special variables look like a dollar sign followed by a single non-alphanumeric character. The error-handling could be considerably more sophisticated than shown here, but that isn't our goal in this paper. Also note that there are no parentheses for the open function. That could have been written as:

```
open (FILE, 'many.bad.strings')
    or die ("Can't look at strings: $!");
```

but in Perl the parentheses are not needed if the parser can figure the code out without them.

In a SAS® DATA step, you usually use an implicit loop to process the file line by line. But in Perl you specify the loop. One of several ways is the while loop, which continues until a false condition is met - just like the 'do while' construct in the DATA step. But Perl has many shortcuts. Here is one of the classics. The angle operator <> is the file-reading operator. It reads a filehandle a line at a time, in this case the filehandle FILE we created for our input file. But while in this case does a little something extra. It actually tests whether the newly-read line is the end of the file, and automatically continues reading until the file ends.

The next two lines should look a little like SAS® code, except for those pesky dollars signs, and the curly brackets instead of the SAS® if-then statement. Perl uses + and - just like SAS®, so this could have been written as

```
$count = $count+1;
```

The only unusual part is the special variable $_ , which Perl uses as the 'default' variable anytime it is convenient to do so. If you ever see a piece of Perl code which seems to be missing the expected variable, expect that the code is using whatever has been most recently assigned to the special variable $_ .

The program then explicitly closes the filehandle. You could leave that off. Just as SAS® automatically closes the input file at the end of the DATA step, Perl will close

the file for you if you choose. But Perl will also give you the option of handling something bad or unexpected, or using the filehandle in unusual ways. Here the program only alerts the user if the operating system refuses to release the filehandle.

Where Perl uses the print function, the SAS® programmer would use a 'put' statement [or perhaps a '%put' statement]. And, just as one can put the lines to a file with an extra line of code, you can do so in Perl too. And, just as you can use double quotes in SAS® so that you can include a macro variable in your quote, so the double quotes in Perl permit you to include any variable for "interpolation" [as the Perlites say]. But in Perl you have to add your own line ending: the \n at the end of the quoted string. Perl automatically converts the \n into whatever is the correct line ending for your operating system. The variable $extra is created as a function of $count .

SOME PERL TRAPS

Note that Perl uses the same operators as SAS® does - for the most part. Here we use addition, subtraction, and exponentiation. There are several key differences you should know about when looking at Perl operators.

operator	SAS®	Perl
||	string concatenation	or
%	macro keywords	modulus - the mod() function
|	logical or	bitwise or
&	logical and	bitwise and
^	logical not	bitwise xor
~	logical not	binding operator for pattern matching
.	macro resolution	string concatenation

Another key difference is in testing for equality. In typical SAS® code you would compare two quantities like this:

```
if count = 10 then . . .
```

but in Perl you use a double equal-sign to test for equality, like this:

```
if $count == 1 { . . .
```

There are other differences and features of Perl operators [for example, without the 'strict' pragma we used above, Perl will let you accumulate the count starting with an undefined value of $count and treat the starting point as zero for you] but these are enough for now.

PERL CODE - AND HOW TO FIX IT

Next let's look at some classic [translation: bad] bits of Perl code that have spread throughout the Web. Here's a very common one, which in one form or another has even made its appearance in presentations at past SUGIs.

```
sub PH {
  print "<!doctype html public \"-//w3c//dtd
    html 4.0 transitional//en\">";
  print "<html>";  print "<head>";
  print "<meta http-equiv=\"Content-Type\"
    content=\"text/html; charset=iso-8859-1
    \">";
  print "<meta name=\"Author\"
    content=\"$username\">";
  print "<meta name=\"GENERATOR\"
    content=\"Mozilla/5.01 [en] (WinNT; U)
    [Netscape]\">";
  print "<title>Lost Angeles Vacation
    </title>";  print "</head>";  print
    "<body>";
}
```

Now this is not very attractive. It is not particularly readable either. Sticking multiple lines of code on one line works in Perl just as in SAS®, but it is just as difficult to read. And I find the backwhacked double quotes fairly unattractive, lending to the general poor readability.

But in Perl there is a saying: "There's More Than One Way To Do It". In fact this is so common that it is often abbreviated to TMTOWTDI [which is pronounced 'tim-toady'].

Perl provides alternate quoting operators. qq() is equivalent to double quotes - although almost any non-alphanumeric character can be used in place of the parentheses. And using qq// will let one avoid having to put backslashes before all those internal double-quotes. Let's see how the code looks using qq{} instead of regular double quotes, and using decent rules for lining up text. We'll also make the subroutine name a little better:

```
sub PrtHead {
  print qq{<!doctype html public "-
    //w3c//dtd html 4.0 transitional//en">};
  print qq{<html>};
  print qq{<head>};
  print qq{<meta http-equiv="Content-Type"
    content="text/html; charset=
    iso-8859-1">};
  print qq{<meta name="Author" content=
   "$username">};
  print qq{<meta name="GENERATOR"
   content="Mozilla/5.01 [en] (WinNT; U)
   [Netscape]">};
  print qq{<title>Lost Angeles Vacation
    </title>};
  print qq{</head>};
  print qq{<body>};
}
```

Well, that is a little better. But it can be made a lot more readable and maintainable, just by learning one more Perl trick. Perl also permits the "here-document" structure that is available in Unix shell programming.

And Perl lets you use the here-doc as an argument to a function, in this case to the print function. We'll even make the subroutine name a bit more mnemonic, since [as in SAS® Version 8] one can use more than eight characters for the name:

```
sub PrintHeader {
 print HTML <<EOF;
<!doctype html public "-//w3c//dtd html 4.0
transitional//en">
<html>
<head>
  <meta http-equiv="Content-Type"
  content="text/html; charset=iso-8859-1">
  <meta name="Author" content="$username">
  <meta name="GENERATOR" content="Mozilla/5.01
  [en] (WinNT; U) [Netscape]">
  <title>Lost Angeles Vacation</title>
</head>
<body>
EOF
}
```

Note that the closing 'EOF' has to match exactly what is on the line where the print function sits, except for the closing semicolon. The closing EOF has to be at the start of the line, with no spacing before and no characters afterward. Due to a peculiarity of Windows files, one cannot even have that closing EOF as the last line of the program: add on a blank line if that is what you have.

There. That's much better. The purpose of the subroutine is obvious. The layout of the HTML is clear. The Perl variable interpolated in the middle of the HTML can be found by the casual eye. Maintenance of the code is now possible by anyone who knows a little HTML. Look back at the original and decide which you prefer.

ANOTHER EXAMPLE

Next, let us look at a short bit of incorrect Perl which was unfortunately common on websites up until January of last year (for reasons that you should be able to guess at once).

```
  ($sec,$min,$hour,$mday,$mon,$year,$wday,
    $yday,$isDST) = localtime(time);
  print "19$year\n";
  if    $mon == 0 {print "Month: Jan\n"}
  elsif $mon == 1 {print "Month: Feb\n"}
  elsif $mon == 2 {print "Month: Mar\n"}
     . . . .
  else            {print "Month: Dec\n"}
```

Now then. Not only is this fairly ugly, but it is seriously wrong. Perl does the unusual but Y2K-compliant [and C-like] thing. It returns a "year" value which is the actual year minus 1900. So code like that above will today print the obviously wrong value 19101 . Oops.

But it is messier for more reasons than that. The localtime function automatically uses the time function, so that part of the first line is unneeded. In fact Perl does not even require the parenetheses around time . The coder is only using the fifth and sixth numbers in the array that localtime builds, but doesn't know how to make do with less. And the coder ends up with a sequence of twelve if-elsif-else clauses to get the right month.

Note that Perl does a few things differently from SAS® here. Where SAS® uses an 'if-else if-else' form, Perl uses a special keyword elsif . Note that Perl uses a block defined by curly brackets instead of a then clause. Note that Perl doesn't require a semicolon for a single statement in the brackets - semicolons are statement separators in Perl, not statement closers. And note that Perl automatically converted the year [a number] to a character string in the print function without complaining - Perl does that because it assumes the coder knows what he or she is doing.

Now there are several better ways to print out the year and month. One simple way is the use of **context**. Perl maintains an important distinction between single [or 'scalar'] values and multiple [or 'list'] values. In fact, many Perl functions will return different results depending whether they are used in scalar or list context. This feature leads to many confused programmers, because sometimes Perl can be smarter about the context than the beginning programmer!

Perl's localtime is one of the functions which acts this way. As above, localtime in list context gives a list of date and time variables. But localtime in scalar context gives a scalar: a single string with the date and time in it. Fortunately Perl lets you force scalar context using the scalar function. So if you had tried the following this past Sunday at 1:30 in the afternoon:

```
  print scalar localtime;
```

you would have seen this output:

```
  Sun Apr 22 13:30:00 2001
```

which would suffice for many date-time requests.

Or one could use a Perl array to store the months rather than the if-elsif-else mess above.

```
my ($mon,$year) = (localtime)[4,5];
my @months = qw(Jan Feb Mar Apr May Jun Jul
  Aug Sep Oct Nov Dec);
print "Year: ", $year+1900, "\nMonth: ",
  $months[$mon],"\n";
```

Here the first line uses the default input for localtime, so the time function is implicit. The parentheses around localtime automatically give list context, so the output is a list rather than a scalar. The brackets after localtime are an 'array slice', selecting a set of elements out of the whole list instead of forcing one to work with the entire list. And then the two elements [the fifth and sixth of the array since all arrays in Perl start counting at zero] are assigned to $mon and $year .

The second line uses the qw// function, which quotes 'barewords' automatically and separates them into the elements of an array, so you do not have to write the line as

```
my @months = ('Jan', 'Feb', 'Mar', 'Apr',
   'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct',
   'Nov', 'Dec');
```

Now the array @months has the twelve months of the year, accessible much as in SAS® arrays in the DATA step.

The third line prints out the results much like the SAS® 'put' statement in a DATA step, stringing quoted text and variables to be evaluated. The print function requires these to be separated by commas, but does not insert spacing between these in the output. Note that 1900 is added to the year in the middle of the print statement, and the correct element of the array @months is selected using $months[$mon] - Perl requires square brackets for looking up its array elements. Also note that there is a '$' in front of the array here, instead of the '@' sign arrays use. The Perl rule is to use the symbol for what you want, not the symbol for what you already have - and we want a scalar value out of that array.

AND YET ANOTHER...

Now here is the sort of intimidating Perl code which shows up in cut-and-paste code on webpages. This is supposed to decode HTML entities.

```
for (@$array) {
  s/(&\#(\d+);?)/$2 < 256 ? chr($2) : $1/eg;
  s/(&\#[xX]([0-9a-fA-F]+);?)/
    $c=hex($2); $c < 256 ? chr($c) : $1 /eg;
  s/(&(\w+);?)/$entity2char{$2} || $1/eg;
}
```

Does it do what it is supposed to? Is it possible for a beginner to tell? This shows a lot of Perl which we have not talked about, and which is not for the Perl beginner. A Perl 'newbie' would have to trust that the code was correct, and that it was in fact decoding the parts of strings the user wanted. That's a lot to take on faith, given that this might have been pulled out of a total stranger's Perl code.

This code is written for compaction, not clarity. It contains, among other arcane components: a reference to an array; three string substitutions; multiple cute regular expression features; the use of the associative array [known in Perl as a 'hash'] %entity2char; and also substitution operators using string interpolation, and regular-expression variables, as well as more than one statement in the second part of the operator.

The Perl hash is a data type which functions as a table of key-value pairs with hashing for extremely fast lookup of the values in the table. Paul Dorfman has exposited about implementing associative arrays, as well as hashing, so this can be mimicked in SAS®. The hash %entitychar is being accessed above using the quantity in the special variable $2 as its key. As with Perl array lookup, the '$' is in front of the hash since we want to get a scalar, and you use the symbol for what you want, not the symbol for what you have.

In Perl (as in SAS®) the ability to match intricate parts of strings requires the complications of regular expressions. Perl's regular expressions (as above) do not look anything like those of SAS®, but rather look like the regular expressions commonly seen in Unix® utilities and programs. In fact, the simple forms look rather like the wildcards in seen MS-Windows® .Perl's regular expressions have some extremely powerful features, and are about as fast as you could hope for. The standard form of the Perl substitution operator is

```
$variable =~ s/pattern/substitutes/options;
```

But, as before, when the pattern is in the default variable $_ , Perl handles this automatically for you, like this

```
s/pattern/substitutes/options;
```

This is the form we see in the example.

Still, the details of these are topics for more advanced tutorials, not a quick intro. Particularly when we don't need the constructs. Clearly the following code is easier to use and to follow.

```
use HTML::Entities;
decode_entities( $x );
```

This uses the aspect of Perl known as modules. A Perl module can be called via the use function, and extra functions and features can be imported by that call. Here the HTML::Entities module is called in the first line, and a function decode_entities() from the module is used in the second line. This replaces all the previous code, plus some important housekeeping code as well. In many ways, Perl modules can be thought of as the equivalent of SAS® PROCs or macro libraries.

In conclusion, you cannot summarize Perl in one quick introduction, any more than you could do the same with SAS®. Perl is a large language, with:

- more data types including multi-dimensional data constructs;
- subroutines with sophisticated prototyping;
- many more built-in functions, like those in the DATA step;
- modules, which could be considered analogous to SAS® PROCs and libraries of macro functions;
- methods for building screens, which would be analogous to SAS/AF®;
- object-oriented programming, for those who want it;
- and a whole lot more.

But now you have seen some of the basics, and you are a little more prepared for that time when someone drops a Perl program on your desk and says, "Hey, can you convert this to SAS®, and, umm, by the way, I need it yesterday..."

Acknowledgements

Contact Information

The author may be contacted by mail at

David L. Cassell
OAO Corp., c/o U.S. EPA
200 SW 35th St.
Corvallis, OR 97333

or by e-mail at

Cassell.David@epa.gov