Introduction to Perl

Patrick M. Ryan
`patrick.m.ryan@gsfc.nasa.gov`
Hughes STX Corporation

Revised November 16, 1993

1 What is Perl?

Perl has become the new language of choice for many system management tasks. Combining elements of C, awk, sed, grep, and the Bourne shell, Perl is an excellent tool for text and file processing. Although Perl is often described as a "system management language", it is useful for many tasks that would otherwise be done with shell scripts.

"Perl" is an acronym for *Practical Extraction and Report Language*. It was developed by (and is still maintained by) Larry Wall of Netlabs. It is freely available software and compiles on nearly all major architectures and operating systems. These include all the major UNIX[1] variants as well as VMS and even DOS.

Perl contains features of the Bourne shell (`/bin/sh`), `awk`, `sed`, and `grep` as well as access to systems calls and C library routines. It is said that Perl fills the niche between shell scripts and C programs.

Perl is not a compiled language but it is faster than most interpreted languages. Before executing a Perl script, the `perl` program reads the entire script into memory and "compiles" it into a fast internal format. In nearly all cases, a Perl script is faster than its Bourne shell analogue. Note that by convention, one refers to the *P*erl language in upper case and the *p*erl program in lower case.

This document is intended to be an overview of the major features of Perl and does not describe every facet of the language. Much more extensive reference materials are available. These references, as well as pointers to example scripts, are detailed at the end of this document.

[1] "UNIX" is a trademark of AT&T. No, make that Unix Systems Laboratories. Or maybe Novell, Inc ...

2 Basic Syntax

Perl is a free-form language like C. Perl's control flow structures are very much like C's. There are no FORTRAN-like line constraints.

Perl programs, by convention, sometimes end in `.pl`. This is not a requirement, however, and most Perl scripts simply invoke the interpreter through the use of the `#!` construct. The first line of a Perl script (at least in the UNIX world) usually looks like this:

```
#!/usr/local/bin/perl
```

In Perl, every statement must end with a semicolon (`;`). Text starting with a pound sign (`#`) is treated as a comment and is ignored.

Blocks of Perl code, such as those following conditional statements or in loops are always enclosed in curly brackets ({...}).

3 Data Types

Perl has three basic data types:

- scalars
- arrays of scalars
- associative arrays of scalars (also known as hash tables)

3.1 Scalars

The scalar is the basic data type in Perl. A scalar can be an integer, a floating point number, or a string. Perl figures out what kind of variable you want based on context. Scalars variables *always* have a dollar sign (`$`) prefix. Therefore, a string assignment looks like this:

```
$str = "hello world!";
```

not:

```
str = "hello world!";
```

In Perl, an alphanumeric string with no prefix is (generally) assumed to be a string literal. Thus, the second statement above attempts to assign string literal "hello world!" to string literal "str".

Perl's string quoting mechanisms are similar to those of the Bourne shell. Strings surrounded in double quotes ("...") are subject to variable substitution. Thus, anything that looks like a scalar variable is evaluated (and possible interpolated) into a string. Strings inside single quotes ('...') are passed through basically untouched.

Perl variables do not have to be declared ahead of time. The are allocated dynamically. It is even possible to refer to non-existent Perl variables. The default value for a variable in a numeric context is 0 and an empty string in a string context. Perl has a facility for determining whether a variable is undeclared or if it is really is a zeroish value.

Perl variables are also typed and evaluated based on context. String variables which happen to contain numeric characters are interpolated to actual numeric values if used in a numeric context. Consider this code fragment:

```
$x = 4;        # an integer
$y = "11";     # a string
$z = $x+$y;
print $z,"\n";
```

After this code is executed, `$z` will have a value of `15`.

This interpolation can also happen the other way around. Numeric values are formatted into strings if used in a string context. Numeric values do not have to be manually formatted as in C or FORTRAN. This type of interpolation takes place often when writing standard output. For instance:

```
$answer = 42;
print "the answer is $answer";
```

The output from this fragment would be "the answer is 42".

Note that integer constants may be specified in octal or hexadecimal as well as in decimal.

String constants may be specified by way of "here documents" in the manner of the shell. Here documents start with a unique string and continue until that string is seen again. For example:

```
$msg = <<_EOM_;
```

```
   The system is going down.
   Log off now.
_EOM_
```

3.2 Arrays of Scalars

Perl scripts can have arrays (or "lists") consisting of numeric values or strings. Entire array variables are prefixed with an "at" sign (@). It is also possible to assign to the array elements by name. Here are some examples of valid Perl array assignments:

```
@numbers = (3,1,4,1,5,9);
@letters = ("this","is","a","test");
($word,$another_word) = ("one","two");
```

Of course, Perl array elements can also be referenced by index. By default, Perl arrays start at 0. Perl array references look like this:

```
$blah[2] = 2.718281828;
$message[12] = "core dumped\n";
```

Note that since an array element is a scalar, it is prefixed by a $.

The `$#` construct is used to find out the last valid *index* of an array rather than its size. The `$[` variable indicates the base index of arrays in a Perl script. By default, this value is 0. Here is a code fragment which tells you the number of elements in an array:

```
# assume that @a is an array with a bunch of interesting elements
$n = $#a - $[ + 1;
print "array a has $n elements\n";
```

`$[` can be reset to use a different base index for arrays. To have FORTRAN-style array indexing, set `$[` to 1.

Arrays are expanded dynamically. You need only assign to new array elements as you need them. You can pre-allocate array memory by assigning to its `$#` value. For instance:

```
$#months = 11;     # array @months has elements 0..11
```

Perl has operators and functions to do just about anything one would need to do to an array. There are facilities for pushing, popping, appending, slicing, and concatenating arrays.

Perl can only do one-dimensional arrays but there are ways to fake multi-dimensional arrays.

3.3 Associative Arrays of Scalars

Associative arrays are Perl's implementation of hash tables. Associative arrays are arguably the most unique and useful feature of Perl. Common applications of associative arrays include creating tables of users keyed on login name and creating tables of file names. The prefix for associative arrays is the percent sign (`%`).

Associative arrays are keyed on strings (numeric keys are interpolated into strings). An associative array can be explicitly declared as a list of key-value pairs. For example:

```
%quota = ("root",100000,
          "pat",256,
          "fiona",4000);
```

Associative arrays elements are referenced in the following way:

```
$quota{dave} = 3000;
```

In this case, `"dave"` is the key and `3000` is the value. Note that the reference above is to a scalar and is thus prefixed by a $.

Here is another example. In Perl scripts, there is a predefined associative array called `%ENV` which contains all of the environment variables of the calling environment. Here is a bit of Perl code to see if you are running X Windows:

```
if ($ENV{DISPLAY})
{
      print "X is (probably) running\n";
}
```

There are routines for traversing the contents of associative arrays and for deleting elements. The relevant Perl routines are `each`, `keys`, `values`, and `delete`.

Note that the namespace for Perl variables is exclusive. One can refer to scalars, arrays, associative arrays, subroutines, and packages with the same name without fear of conflict.

4 Operators and Comparators

Perl's set of operators and comparators comprise nearly all of C's operators and comparators. All of the usual arithmetic expressions and precedence are the same in Perl as they are in C. Listed below are expressions which are valid in Perl but not in C. These descriptions are paraphrased from the Perl man page.

`**` The exponentiation operator.

`**=` The exponentiation assignment operator.

`()` The null list, used to initialize an array to null.

`.` Concatenation of two strings.

`.=` The concatenation assignment operator.

`eq` String equality (== is numeric equality). Other FORTRAN-style comparators are also available. These are only used on strings.

`=~` Certain operations search or modify the string `$_` by default. This operator makes that kind of operation work on some other string. The right argument is a search pattern, substitution, or translation. The left argument is what is supposed to be searched, substituted, or translated instead of the default `$_`.

`x` The repetition operator.

`..` The range operator

`-f, -x, -l, ...` Unary file test operator. Perl has the ability to test various file permission settings in the same way as the UNIX `test` command. Consult the Perl manual page for a full listing of Perl file test operators.

5 A Word about Default Arguments

Many functions and syntactic structures in Perl have default arguments. In most cases, this default argument is the variable `$_`. While this is a handy feature for experienced Perl programmers, it can make their code incomprehensible to those just learning the language. For novices, it can be a nuisance when one does not understand how the value of `$_` is determined.

I recommend that when you are first learning Perl, put in all arguments explicitly. In many cases, Perl figures out what you are trying to do based on context and assigns values to `$_` according to its own rules. The value of `$_` can change subtly (or even drastically) depending on context.

Once you have a few lines of Perl under your belt and understand the ways of `$_`, feel free to use the default arguments. It is a nifty feature which allows you to write slick, fast, (and cryptic) Perl code.

6 Regular Expressions

Where once you had to execute a `grep` or `expr` every time you wanted to compare a string to a regular expression ("regexp"), you can now stick regexps right in your code. Perl's regexp handling capabilities are another reason that you'll never want to write another Bourne shell script.

6.1 Matching Regular Expressions

Perl regular expressions look very much like those in `vi`.

`.` Match any one character except a newline.

c`*` Match zero or more instances of character *c*.

c`+` Match one or more instances of character *c*.

c`?` Match zero or one instance of character *c*.

`[`*class*`]` Match any of the characters in character class *class*.

`\w` Match an alphanumeric character (including "_")

`\W` Match an non-alphanumeric character (including "_")

`\b` Match a word boundary

`\B` Match non-boundaries

`\s` Match a whitespace character

`\S` Match a non-whitespace character

`\d` Match a numeric character

\D Match a non-numeric character

^ Match the beginning of a line

$ Match the end of a line

Also, \n, \r, \f, \t and \NNN have their usual C-style interpretations.

The actual syntax for the pattern matching command is m/*pattern*/gio. The modifiers are g for "global" match, i for "ignore case", and o for "only compile this regexp once". With the m command, you can use any pair of non-alphanumeric characters to bound the expression. This especially useful when matching filenames that contain the "/" character. For example:

```
if (m!^/tmp_mnt!)
{ print "$_ is an automounted file system\n"; }
```

If the delimiter you choose is "/", then the leading m is optional.

Perl even has the ability to do multi-line pattern matching. Refer to the documentation on the $* variable for complete information.

6.2 Extracting Matched String from Regexps

As in vi, grep, and sed, Perl can return substrings which are matched in a regular expression. For instance, here is some Perl code to (sort of) emulate the UNIX basename command:

```
$file = "/auto/home/pat/c/utmpdmp.c";
($base) = ($file =~ m|.*/([^/]+)$|);
```

The result of this code fragment is that $base has the value "utmpdmp.c". The parens in the regexp indicate the substring we want to extract.

The return value of a regular expression match depends on context. In an array context, the expression returns an array of strings which are the matched substrings. In a scalar context, typically in a test to see whether or not a string matches a regexp, the expression returns a 0 or 1.

Here is an example of a scalar context. The <STDIN> construct, discussed in detail later, reads in one line of standard input.

```
$response = <STDIN>;
if ($response =~ /^\s*y/i)
{ print "you said yes\n"; }
```

Note that the distinction between an "array context" and a "scalar context" is important in Perl. Many routines and syntactic structures return different types of values depending on context. We will say more about array contexts later.

7 Flow Control

Perl has all of the flow control structures one normally expects in a procedural language as well as a few extras.

7.1 If-Then-Else

The Perl `if` statement has the same structure as in C. Perl uses the same conjunctions and boolean operators as C: `&&` for "and", `||` for "or", and `!` for "not". One important note is that the C-style one-statement `if` construct cannot be used. All of the code following a Perl conditional (`if`, `unless`, `while`, `foreach`) must be enclosed in curly brackets. For instance, this C fragment:

```
if (error < 0)
      fprintf(stderr,"error code %d received\n",error);
```

becomes this Perl fragment

```
if ($error < 0)
{ print STDERR "error code $error received\n"; }
```

The Perl analogue to C's `else if` construct is `elsif` and the `else` keyword works as expected.

Perl has an `unless` statement which reverses the sense of the conditional. For instance:

```
unless ($#ARGV > 0)     # are there any command line arguments?
{ print "error; no arguments specified\n"; exit 1; }
```

Perl's ideas about truth are similar to C. In a numeric context, a zero value is considered "false" and anything non-zero is "true". An empty string is "false" and a string with a length of 1 or more is true. Scalar arrays and associative arrays are considered "true" if they have at least 1 member and "false" if empty. Non-existent variables, since they are always 0, are "false".

Note that Perl does not have a `case` statement because there are numerous ways to emulate it.

7.2 The `while` statement

Perl's `while` statement is very versatile. Since Perl's notion of truth is very flexible, the `while` condition can be one of several things. As in C, Perl conditional statements can be actions or functions.

For instance, the `<STDIN>` statement with no argument assigns a line of standard input to the `$_` variable. To loop until the standard input ends, this syntax is used:

```
while (<STDIN>)
{
      print "you typed ",$_;
}
```

In keeping with the recommended beginner practice of including all default arguments, that code would look like this:

```
while ($_ = <STDIN>)
{
      print "you typed ",$_;
}
```

As stated before, an array is "true" if it has any elements left. For instance:

```
@users = ("nigel","david","derek","viv");
while (@users)
{
     $user = shift @users;
     print "$user has an account on the system\n";
}
```

This `while` loop will continue as long as `@users` has at least one element. The `shift` routine pops the first element off the named array and returns it.

Perl has two keywords used for shortcutting loop operations. The `next` keyword is like C's `continue` statement. It will immediately jump to the next iteration of innermost loop. The `last` keyword will break out of the current conditional statement. It is analogous to C's `break` statement.

7.3 The `for` and `foreach` statements

The `for` and `foreach` statements in Perl are actually identical. They can be used interchangeably in any context. Depending on what job is being performed, however, one usually make more sense than the other.

Just to make things more confusing, there are two ways that the `for/foreach` statement can be used. One way is exactly like C's 3-argument `for` statement. For instance:

```
@disks = ("/data1","/data2","/usr","/home");
for ($i=0; $i <= $#disks; ++$i)
{ print $disks[$i],"\n"; }
```

However, once you understand Perl's built-in ways of iterating over an array's elements, you will almost *never* need to use the 3-argument `for` statement.

Perl's one-argument `foreach` statement is similar to the `foreach` statement in the C Shell. Given an array argument, the `foreach` statement will iteratively return that array's elements. This contrasts with the destructive traversal demonstrated before with the `while` and `shift` statements.

The code fragment we just saw can be rewritten as:

```
@disks = ("/data1","/data2","/usr","/home");
foreach(@disks)
{ print $_,"\n"; }
```

This construct is much more elegant and does not (necessarily) destroy the contents of `@disks`.

An subtle but important point to note is that, in the fragment above, `$_` is really a pointer into the array, not simply a copy of a value. If the code in the loop modifies the `$_`, the array is changed.

7.4 Goto

Yes, Perl even has a `goto` statement. `goto` *label* will send control of the program to the named label. The usual caveats against GOTOs apply in Perl as elsewhere. Don't use GOTOs unless you really need them!

8 Built-In Routines, C Library Routines, and System Calls

Perl has a rich set of built-in routines and access to most of the interesting functions in the C library. The manual pages for Perl go into exhaustive detail about all of these routines so I will simply discuss a few of the more commonly used ones. Most of these descriptions are paraphrased from the man pages.[2] The default argument for most of these routines is `$_`. Note that parentheses around function parameters are usually optional.

8.1 Built-In Routines

`chop` *expr* Chop off the last character of a string and return the character. This might not seem like a very interesting thing to do until you understand Perl file I/O. Upon reading a line of input into a variable, Perl preserves the newline (`\n`). Usually, you don't need the newline so you probably want to `chop` it off.

`defined` *expr* Determine whether or not the named variable really exists or not. This function will return true if the named variable has a value and is not simply undefined.

`die` *expr* Utter a final message and pass away. This function will print out a string argument and then cause the script to terminate. It is used most often when some kind of fatal error occurs.

`each` *array* Return the key-value pairs of an associative array in an iterative manner.

`join` *expr*,*array* Joins the separate strings of *array* into a single string with fields separated by the value of *expr*, and returns the string.

`pop` *array* Pop off the top element off the named array and shorten the array by one.

`print` *expr* Print out the arguments. More on the `print` function later.

`push` *array*,*list* Treat *array* as a stack and push the values of *list* on to the stack. Has the effect of lengthening the array.

[2] And a few are shamelessly copied word for word

`shift` Shifts the first value of the array off and returns it, shortening the array by 1 and moving everything down. Shift() and unshift() do the same thing to the left end of an array that push() and pop() do to the right end.

`split(`*/pattern/*,*expr*,*limit*) Splits a string into an array of strings and returns it. The pattern is treated as a delimiter separating the fields. A common use of this function is to split up lines of the UNIX `/etc/passwd` file into its component fields. This is similar `awk`'s functionality only more versatile.

`substr` *expr*,*offset*,*len* Extract a substring out of *expr* and returns it.

8.2 UNIX-type Utility Routines

`chmod` Change the permission bits of the named files.

`chown` Change the owner and group of the named files.

`mkdir` Make directories.

`unlink` Remove a file.

`rename` Rename a file.

`rmdir` Remove a directory.

8.3 C Library Routines

Many C library routines can be accessed in Perl. This is a sampling of them.

`getpw, getgr,` ... Perl has access to all of the C routines which access passwd, group, and hostname information.

`bind, connect, socket,` ... Interprocess communication facilities are available in Perl.

`stat` Access file information via the UNIX `stat(2)` library routine.

`exit` Exit the program with the specified exit status.

9 Operating System Interaction

Perl can execute system commands in several ways.

Perl can run shell commands via the `system` routine. This acts essentially like C's `system(3)` call. A string is passed to the shell for execution. The output from the command is sent to standard output. The exit status is put into the `$?` variable.[3]

Perl also evaluates backquotes (also known as "backticks" or "grave accents") in way similar to the shell. This is useful when you want to run a shell command and capture the output. Here is an example in which a script gets the name of the host:

```
$host = `hostname`;
chop($host);
```

Again, the exit status of this command will be put into `$?`. Note that we need to `chop` off the newline from the output.

10 File Handling

Perl's has I/O routines for reading and writing text files as well as "unformatted" files.

10.1 Text I/O

Perl reads and writes text files by way of filehandles. By convention, filehandles are usually in upper case.

Files are opened by way of the `open` command. This command is given two arguments: a filehandle and a file name (the file name may be prefixed with some modifiers). Lines of input are read by evaluating a filehandle inside angled brackets (`<...>`). Here is an example which reads through a file:

```
open(F,"data.txt");
while($line = <F>)
{
    # do something interesting with the input
}
close F;
```

[3] Actually, the entire status word is put into `$?`. Read the man page for details.

The file name argument can have one of several prefixes. If the file name is prefixed with <, the file is opened for reading. (This is the default action.) If the file name is prefixed with > then it is opened for writing. If the file exists, it is truncated and opened. Finally, a prefix of >> opens the file for appending. Here are a few examples:

```
# peruse the passwd file
open(PASSWD,"</etc/passwd");
while ($p = <PASSWD>)
{
     chop $p;
     @fields= split(/:/,$p);
     print "$fields[0]'s home directory is $fields[5]\n";
}
close PASSWD;

# enter some information into a log file
open(LOG,">>user.log");
print LOG "user $user logged in as root\n";

# read a line of input from the user
$response = <STDIN>;
```

There are 3 predefined filehandles which have obvious meanings: `STDIN`, `STDOUT`, and `STDERR`. Trying to redefine these filehandles with `open` statements will cause strange things to happen. `STDOUT` is the default filehandle for `print`.

Perl's file input facility acts very differently if it is called in an array context. If input is being read in to an array, the *entire* file is read in as an array of lines. For instance:

```
$file = "some.file";
open(F,$file);
@lines = <F>;   # suck in the whole file.  yum, yum,...
close F;
```

This capability, though useful, should be used with great care. Ingesting whole files into memory can be a risky thing to do if you do not necessarily know what size files you are dealing with. Perl already does a certain amount of input buffering so reading in a file at once does not necessarily yield an increase in I/O performance.

10.2 Pipes

Perl can use the open routine to run shell commands and read or write to them in the manner of C's `popen(3S)` call.

If a file name argument *starts* with the pipe character (|), the file name is treated as a command. The command is then executed and the program can be sent input via the `print` command.

If the file name argument *ends* with a pipe, the command is executed and that command's output can be read using the `<...>` facility. Here are two examples:

```
open(MAIL,"| Mail root");   # send mail to root
print MAIL "user \"pat\" is up to no good\n";
close MAIL;                 # mail is now sent

open(WHO,"who|");   # see who's on the system
while ($who = <WHO>)
{
    chop $who;
    ($user,$tty,$junk) = split(/\s+/,$who,3);
    print "$user is logged on to terminal $tty\n";
}
close(WHO);
```

10.3 Unformatted File Access

Perl can do direct reading and writing of bytes via the `sysread` and `syswrite` calls.

Given the narrow scope of this introduction to Perl, I will not discuss these functions. The references at the end provide complete information.

10.4 Use of the `print` command

We have already seen several examples of the use of the `print` command in Perl. Now perhaps is a good time to see a more exact description of what it does.

The print command is very flexible and, in most cases, can do the same thing several different ways. In general, `print` takes a series of strings separated by commas, does any necessary variable interpolation, and then prints out the result. The string concatenation operator (.) is often used with `print`. All of the following lines yield the same output.

```
print "But these go to 11.\n";
$level = 11;
print "But these go to ",$level,".\n";
print "But these go to $level.\n";
printf "But these go to %d.\n",$level;
print "But these " . "go to ". $level.".\n";
print join(' ',("But","these","go","to","$level.\n"));
```

As you can see, the C-style printf command is available. However, because of Perl's ability to automatically interpolate numeric values to strings, printf is rarely needed.

There are, in fact, subtle performance issues that can be addressed with each of the methods in the example above. Wall and Schwartz's book, listed in the references, talks about these issues.

As seen in several of the previous examples, the print command also takes an optional filehandle argument.

11 Some Notes about Perl Array Contexts

In C, every expression yields some kind of value. That value can be used as input to another routine without having to store it in a temporary variable. Thus, you can do things like chdir(getenv("HOME")).

In Perl, many routines and contexts yield arrays. These resultant arrays can be passed to other routines, iterated over, and subscripted. This eliminates the need for many temporary variables.

Here are a few examples. In this first case, we use the sort routine. This routine takes an array as a parameter and passes back a sorted version of the same array.

```
@names = ("bill","hillary","chelsea","socks");
@sorted = sort @names;
foreach $name (@sorted)
{ print $name,"\n"; }
```

In fact, one can iterate directly over the output from sort.

```
foreach $name (sort @names)
{ print $name,"\n"; }
```

This example shows that we can even put a subscript on an array context.

```
$name = (getpwuid($<))[6];
print "my name is ",$name,"\n";
```

The function `getpwuid` returns an array. We want the "real name" (or GECOS) field from the passwd entry so we put a subscript of `[6]` on the array context and put the result in `$name`.

12 Subroutines and Packages

Perl has the ability to do modular programming by way of subroutines and libraries.

12.1 Subroutines

Perl scripts can contain functions (usually called "subs") which have parameters and can return values. Listed below is a skeleton for a Perl sub called `sub1`.

```
sub sub1
{
      local($param1,$param2) = @_;
# do something interesting
      $value;
}
```

This sub can then be called in this way:

```
$return_val = do sub1("this is","a test");
```

The `do` can be replaced by `&`. This is actually the preferred method:

```
$return_val = &sub1("this is","a test");
```

There are a few thing to keep in mind when writing subroutines. Parameters are put into the array `@_` inside the routine. Since all variables are global by default, we use the `local()` function to copy the values into local variables.

Perl has a `return` statement which can be used to explicitly return values. This is usually unnecessary, however, because the return value from a Perl subroutine is simply the value of the last statement executed. Thus, if you want a subroutine to return a zero, the last line of the routine can be `0;`.

12.2 Packages

Perl has a library of useful routines which you can include in your scripts. The Perl analogue of C's `#include` statement is `require`.

For instance, Perl has a library to do command-line parsing similar to C's `getopt(3)` function.

```
require 'getopts.pl';
&Getopts('vhi:');
if ($opt_v) { print "verbose mode is turned on\n"; }
```

It is also possible to write your own libraries and include them in other scripts.

13 Predefined Variables

Perl has a sizable set of predefined variables. These are all documented in detail in the man pages so I will only describe a few of the common ones.

`$_` Default argument for many routines and syntactic structures.

`$?` Status word returned from last system call. The lower bytes contain the signal upon which the program died (if any) and the upper bytes contain the exit code.

`$$` Process ID of script.

`$<` Real user ID of user running script.

`@ARGV` Command-line arguments of script. Note that `$ARGV[0]` is the first actual argument, not the name of the program (as in C). The name of the script is in the variable `$0`.

`%ENV` Associative array containing the environment variables of the calling environment.

14 Command Line Options

Perl has a set of command-line switches. Here are a few of the most useful ones.

`-c` Check the syntax of a Perl script but do not execute it.

`-e` Specify Perl code on the command line.

`-w` Warn the programmer about any questionable uses of variables. These include variable used only once and variables which are referenced before being assigned. New Perl programmers are advised to check their scripts with `perl -c -w` *script.pl*.

15 References

15.1 Manuals and Books

There are several very good references to Perl available. The first and foremost is the Perl manual page. It is about 90 pages long and describes all aspects of Perl, albeit in a terse manner. For me, it is the reference of first resort since I can scan through it in an Emacs buffer.

The book Programming Perl by Perl author Larry Wall and Randal Schwartz is the definitive compendium of all things Perl. It is known colloquially as "the Camel book" due to O'Reilly and Associates' habit of putting animals on the covers of their books, in the case a camel. It should be noted, however, that it is not the best book to buy for learning Perl from scratch simply because it is so big. It is a better book to read once you know the basics.

There is a book due out sometime this fall by Randal Schwartz called Learning Perl. It is being written presumably in response to the difficulty of learning Perl from the Programming Perl book.

There is a Usenet newsgroup devoted to the Perl language called `comp.lang.perl`. This is a forum for discussing nuances of Perl and asking questions about the language. New Perl programmers are encouraged to read the manual pages and the Perl FAQ (mentioned below) and to consult experienced local Perl programmers before posting to the group.

15.2 How to get Perl

The current version of Perl is 4.036. Although version 5 is in alpha test right now, version 4 is the stable version. Version 4 can be found via the `archie` protocol at hundreds of ftp sites.

I have put the current version of Perl in the anonymous ftp account on my machine. The code can be found at:

```
jaameri.gsfc.nasa.gov:/pub/perl/perl-4.036.tar.gz
```

There are several useful things in that directory. They are:

- `perl-mode.el` An Emacs major mode for editing Perl code.
- `perl.faq` The list of Frequently-Asked Questions about Perl.
- `refguide.ps` PostScript format reference guide for perl 4.036.
- `examples/` A directory of example Perl scripts

The directory of example scripts is a good place to start hacking with real Perl code. Even though I refer to these as "example" scripts, they are all real Perl scripts that I wrote to solve real problems.

I welcome comments, bug fixes, fan mail, et cetera about anything in the ftp directory or about Perl in general. I can be reached at `patrick.m.ryan@gsfc.nasa.gov`.