A Brief Introduction to Perl

Now, everybody’s solution to a system administration problem is not to write a new scripting language, but that’s exactly what Larry Wall did back in 1987 when he found the news reader rn (which he also wrote) to be insufficient for the task. In the years since then, Perl, the ”Practical Extraction and Report Language,” or ”Pathologically Eclectic Rubbish Lister,” depending on who you talk to, has grown from a one-person hack to a community supported language with ports for almost every modern operating system available. In many cases Perl has become THE tool of choice for system administration, because it can actually do just about anything.

With its powerful and well-developed module system, including tie-ins to many useful C libraries, the possibilities with perl and endless. Nevertheless, most of these uses are beyond the scope of this tutorial. The focus of this tutorial is rather on the Perl language itself, and a few of its main strengths. After working through this tutorial or attending the Reflections Projections workshop, you should have a basic grasp of the fundamental aspects of the Perl language, and enough knowledge to get something out of the references listed at the end of this tutorial.

1.1 Starting Perl

The simplest way to start Perl is just by running the Perl program with no arguments. In this tutorial, every line that begins with a ‘%’ should be interpreted as a shell command. So type ‘perl’ on the command line and press enter:

% perl
You have now invoked the Perl interpreter! Patienly the interpreter sits there, waiting for you to type in some Perl code for it to compile. Let’s give it some Perl to play with:

```
% perl
print "I like Perl\n";
```

When you’re done, just type Ctrl-D to tell Perl you have finished writing the program. Now, the interpreter can go to work executing your Perl code:

```
% perl
print "I like Perl\n";
I like Perl
```

As you can see, once Perl finishes executing your code, it returns you back to the command prompt so you can do whatever other work you might want to accomplish. However, this is not the only way to use Perl, for by the Perl motto: ”There’s more than one way to do it” (often abbreviated TMTOWTDI). You could also have created a file called, say “script.pl”, with some Perl code in it and then run it through the Perl interpreter with the command:

```
% perl script.pl
```

Alternatively, you could use the `-e` switch to have Perl run a small bit of code. This method is called a Perl one-liner.

```
% perl -e 'print("I like Perl");'
```

Finally (for our purposes), you could make the a file containing Perl code (a script) executable by either setting its viewer to be the Perl program in Windows or putting the following text on the first line of a script file in Unix:

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

Under Unix, you would also have to make the program executable before you could run it by setting the execute file permission with:

```
% chmod a+x script.pl
```

Then, the perl program could be run with the command:

```
% ./script.pl
```
2 The Essence of Perl

Perl takes inspiration from both scripting languages and compiled languages like C. Generally, whitespace is ignored and lines must be terminated with a semicolon (;). However, comments are formatted as in shell scripts. Every character on a line after a associative array (#) is a comment and ignored by the compiler. Perl has no way to comment out blocks of code like C’s /* ... */ sequence. In summary:

```perl
#!/usr/bin/perl -w
# this line is a comment
print "this is a statement, terminated by a semicolon\n"; # comments here too
print "proper formatting",
" is not",
" necessary",
" but it is",
# comment in the middle of a statement
" preffered\n";
#
# print "this statement will not be executed\n";
```

2.1 Variables

Perl variables fall into just three different types: scalars, arrays, and associative arrays. The simple type of a variable is indicated by the character which begins the variable name (either The variable name, or "identifier" may contain letters, numbers, and the underscore ‘_’). Identifiers, like all of perl are case sensitive, so ‘$var’ is a different variable than ‘$VAR’. The variables are summarized below and will be explored more completely in the following sections.

**scalar**
- identifier begins with a dollar sign ($)
- may contain an integer, floating-point number, or string

**array**
- identifier begins with the at sign (@)
- contains an array of scalars referenced by integers.
- use @array[<integer>] to access an array element
associative array

- identifier begins with the percent sign (%)
- also called an “associative array”
- contains a collection of scalars referenced by scalars
- use `%associative_array<scalar{}}` to access an associative array element

2.2 Scalars and Literals

Unlike some other languages, you don’t have to tell Perl what kind of type a scalar will be — just assign it a value using the equals (=) sign:

```perl
$exclamation = "woohoo!";
$answer = 42;
$pi = 3.1415927;
$exclamation = 132; # perfectly legal
```

For the most part, numerical literals are fairly straightforward in Perl. Like in C, you can enter an integer in hex or octal by prepending ‘0x’ or ‘0’ to the beginning of the number. Strings literals are nevertheless more interesting animals.

2.2.1 Strings

Strings can be specified in actually a myriad of different ways (remember TM-TOWTDI), but we’re only going to concentrate on the three most common.

**single quotes (’)** single quoted strings such as ‘neato, a string’ are interpreted literally. This literally means that you can put anything between single quotes, and Perl will include it in the string. To put an apostrophe (’ ) in the string you need to escape it with a backslash (\').

```perl
$password = 'sor$h&lm';
# dollar sign is not interpolated
$passphrase = 'I\'ll need $10 for that';
# need to escape the apostrophe
```
**double quotes ("** double quoted strings like "a string with a $string in it" have variable interpretation performed on them. This means that wherever a variable is included in the string, that variable will be replaced by its value in the final text.

```perl
$price = 20;
print "why does this fish cost $price dollars?\n";
# will print out "why does this fish cost 20 dollars?"
```

**backticks (’** the backticks operator performs variable interpretation on the string, but then the resulting string is executed as a shell command, and the output of the command is returned.

```perl
# on unix kernels, the uname command returns selected
# information about the system its running on. the -sr
# switch prints out the system type and version
$system = 'uname -sr';
print "$system\n";
# will print "SunOS 5.8" on a machine running
# Solaris version 5.8
```

Next, we note the operators usable on strings in Table 1. Note that they are listed in order of precedence.

<table>
<thead>
<tr>
<th>operator</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>string concatenation</td>
</tr>
<tr>
<td>.=</td>
<td>string concatenation with assignment</td>
</tr>
</tbody>
</table>
2.2.2 Numbers

As mentioned earlier, numbers are fairly straightforward in Perl. The literals are the same as in C, and the operators are fairly equivalent. For completeness, I’ve included table of arithmetic operators for completeness. Use Table 2 for reference. Operators are ordered by precedence.

<table>
<thead>
<tr>
<th>operator</th>
<th>description</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>unary negation</td>
<td>-$a</td>
</tr>
<tr>
<td>--</td>
<td>increment</td>
<td>$a--</td>
</tr>
<tr>
<td>++</td>
<td>decrement</td>
<td>$a++</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
<td>$a ** $b</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>$a * $b</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>$a / $b</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td>$a % $b</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>$a + $b</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>$a - $b</td>
</tr>
</tbody>
</table>

2.3 Arrays

Arrays in Perl are a bit like arrays in C, but only a bit. For one thing, arrays can contain any random assortment of scalars. For another, you don’t actually have to fill up every element of an array, nor do you need to declare the size of an array. All those technical details that made C programming tedious are taken care of by Perl. As in C, array indexes start at 0. Here are some examples of arrays:

```perl
# just put some values into an array
@a[0] = 12; $a[3] = "hi"; $a[0] = 'cool';
#
# note that the @ sign is replaced by a $ sign when
# assigning to an element of an array. just think
# about a[0] as a scalar, not an array
#
#assigning arrays to arrays
@@b = @a;
#
# using lists to initialize an arrays
@@a = ("Pete", 23, 'Perl Monger');
```
2 THE ESSENCE OF PERL

@b = (555 - 2043);
@c = (@a, @b);
# @c now contains the elements in @a followed by the
# elements in @b
#
# and many more...

As you can see, arrays in Perl are pretty easy to use and flexible. Note
the use of lists above. Lists are used many times in Perl, but the syntax is
so simple that the reader should be able to pick it up in the course of these
two sections.

2.4 Associative Arrays

The Perl associative array or "hash" data type is probably one of Perl’s most
useful features. An associative array is a special kind of an array that is
indexed by strings. That’s a very simple definition for an extremely powerful
tool. Working with perl, you’ll probably find yourself using associative arrays
in many unique and creative ways simply because they are so simple. Here’s
just a few examples:

# use the dollar sign just like for arrays
$days{’December’} = 31;
#
# other scalar indices are implicitly translated into strings
$height{0} = 4;
$height{’0’} = 4;
# the above two assignments mean exactly the same thing
#
# a bunch of elements can be added to an associative array by
# assigning to it a list of key, value pairs.
%jobs = (’Andy’, ’Cook’, ’Denny’, ’Teacher’, ’Patrick’, ’Student’);
#
print "$jobs{’Andy’}\n"
# prints ’Cook’
#
# or, with a little syntactic sugar
%authors = (’Kernighnam and Richie’ => ’The C Programming Language’,
 ’Dubois’ => ’MySQL’,
 ’Stewart’ => ’Calculus: Early Transcendentals’);
2.5 Conditionals

Comparisons in Perl can be performed between any two scalars, provided they are either both strings or both numbers. The comparison operators are nevertheless different for strings and numbers.

<table>
<thead>
<tr>
<th>strings</th>
<th>numbers</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>==</td>
<td>equality</td>
</tr>
<tr>
<td>ne</td>
<td>!=</td>
<td>inequality</td>
</tr>
<tr>
<td>gt</td>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>ge</td>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>lt</td>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>le</td>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>cmp</td>
<td>&lt;=&gt;</td>
<td>comparison, returns -1,0,1</td>
</tr>
</tbody>
</table>

You can also use the boolean operators && (AND) and || (OR) to build up more complicated conditionals.

And now for the truths and the lies...

```perl
# values that evaluate to true
1;   # duh!
("a","b"); # the list has elements
" ";   # whitespace is true
"false"; # so are strings
"00";  # a string

# values that evaluate to false
0;      # by convention
();     # list is empty
"";     # string is zero-length
"0";    # interpreted as zero
```

3 Controlling Perl

Perl has if statements to control execution and do/while as well as for and foreach loops to repeat series of commands. Also, for reusing code, Perl provides subroutines. The syntax for these constructs is much like C, but not quite, so pay attention. Those fine points are the worst.
3.1 Conditional Statements

The `if`, `else`, `elsif` statements in Perl conditionally execute a piece of code based on the value of a conditional expression. The basic form of the `if` and `elsif` statements is `if/elsif (<conditional>) {<code>}`. The curly braces are required. Perl will give you an error if you forget to put them in. In general, a conditional statement is an `if` statement followed by zero or more `elsif` statements and an optional `else` statement. The code in the first `if` or `elsif` associated with a true conditional will be executed. The code in the `else` statement will be executed if none of the conditionals are true. Following is an example:

```perl
# remember, only the first if/else with a true
# conditional gets executed
#
if (1) {
   print("this message will always print");
}
elsif (1) {
   print("this message will never print");
}
#
# regular expressions can be used in conditionals too
#
$text = "words or words or words";
if ($text =~ /letters/) {
   print ("\"$text\" includes the word \"letters\"\n");
}
elsif ($text =~ /words/) {
   print ("\"$text\" includes the word \"words\"\n");
else {
   print ("\"$text\" is most uninteresting\n");
}
```

3.2 Loops

Sometimes you want to execute a piece of code many times in succession. Instead of forcing you to retype the code, Perl has loops that allow for controlled repetition of a piece of code. The most simple form of a loop is the `while` loop that executes a piece of code as long as a conditional expression is true. As before, the curly braces are not optional.
while (<conditional>) {
    <code>
}

Also, Perl has a do...while statement that is like a while loop, but it is executed at least once.

do {
    <code>
} while (<conditional>)

In addition, Perl has the more complicated for loop. The for loop is useful for doing something a specific number of times or possibly executing some code for each member of an array.

for (<init>; <conditional>; <step>) {
    <code>
}

and an example...

# print "Have a nice day" five times
#
for ($n = 0; $n < 5; $n++) {
    print "Have a nice day\n";
}
#
# print the string "123456789"
#
for ($num = 1; $num < 10; $num++) {
    print $num;
}
#

Finally, perl has a very special loop for iterating over the contents of an array. the foreach loop assigns in turn each member of an array to the given scalar and then executes the block of code.

foreach <scalar> (<array>) {
    <code>
}
To look at each key/value pair in associative array, use the `keys` function like as follows.

```perl
%aug_birthdays = ('Robert' => 10,
                  'Florence' => 15,
                  'David' => 25);
foreach $person (keys %aug_birthdays) {
    print "$person’s birthday is Aug $aug_birthdays{$person}";
}
```

### 3.2.1 Controlling Loops

Perl has two statements for controlling how a loop is executed. They are summarized below.

- **next** skip the rest of the code in this loop and go on to the next iteration.
- **last** exit out of this loop.

### 3.3 Subroutines

Subroutines provide a convenient mechanism for packaging up code that you want to execute in many different places. They can be given arguments and may return values. In this case, subroutines can be used as functions. The most simple subroutine takes no arguments and returns no values.

```perl
sub my_print {
    print "hello\n";
}
```

To call a subroutine, either place an ampersand in front of the subroutine name, or follow the name with a pair of parentheses. A bare word followed by a semicolon may also be executed as a subroutine in some circumstances (but avoid doing that).

```perl
# all of the following do the same thing
&my_print;
my_print();
my_print;
```
To call a subroutine with arguments, put a list of arguments in the parentheses after the subroutine name.

my_print("hello");

To get the arguments passed to a function, you must use the perl special variable `@_`. As indicated by the prefix, this variable is an array of all the arguments passed to the subroutine. To make the above code work, change the my_print subroutine thus:

```perl
sub my_print {
    print "@_
";
}
```

The last thing you need to know is how to return a value from a function. This is actually the most silly of them all. The programmer doesn’t have to specifically tell Perl what to return from a subroutine. Perl just returns the result of the last expression in the subroutine. For example, to write a recursive factorial subroutine for perl, you could say:

```perl
sub factorial {
    if ($_[0] == 0 {
        1;
    }
    else {
        $[0] * factorial($[0]-1);
    }
}
```

# and to use it...
$result = factorial(6);
print "$result
";
#
# prints "720"

### 3.3.1 Localization and Declaration

Up to this point we’ve neglected to talk about variable declaration or scope inside Perl. In Perl variables do not need to be declared, but they don’t
have to be. (remember TMTOWTDI). The benefits of declaring variables are increased code readability and also a little more control over scope. You see, in Perl variables are global by default. Whether the variable is initialized in a subroutine, loop, or elsewhere, it is visible to the entire rest of the script. Sometimes this is useful, but often it may not be. Perl provides two mechanisms for declaring/scoping variables, local, and my (we’ll pretend for now that our doesn’t exist).

local performs dynamic scoping on a variable by giving a global variable a temporary value. In most cases, this is not what you want and therefore you should avoid local declarations.

my performs lexical scoping on a variable. This is almost always what you want, so use it!

4 Monkey Read, Monkey Write

5 Search and Destroy, etc.

One of the most useful parts of perl is probably its facility for using regular expressions. Regular expressions are little bits of code that describe a piece of text. Perl is not alone in its use of regular expressions. In fact they are so useful that most computer languages either have them build in or have regular expression libraries build for them. What is presented here is only a miniscule subset of what can be done with regular expressions, but whole books could be written on them (in fact one has – see the references). You will definitely want to learn more about regular expressions as you start to use Perl.

5.1 Regular Expressions

Regular expressions describe a bit of text. In perl, we can perform three basic operations with regular expressions: matching, capturing, and substitution. Let’s see what Perl’s regular expressions can do with the string:


which is a line from the access_log of an Apache HTTP server.
5.1.1 Matching

With matching, we can see if the text conforms to a certain description. In the most simple case, we can check if the text contains a certain sequence of characters. In the following example, we check if the string $log\_line$ contains the word “apache\_pb”, which it does. The ‘=’ operator binds a string to a regular expression and returns ‘true’ if the string matches the expression.

# matching a string to a string
$log\_line =" /apache\_pb\/; # returns true

Regular expressions also provide something called a “character class” which can match a certain set of characters at the point it appears in the regular expression. Regular expressions also include facilities for specifying how many characters in the text being evaluated should match each character in the regular expression. Thirdly, a regular expression may specify where a match should occur. The exact syntax for these facilities is beyond the scope of this tutorial, but you can peruse the following examples for a taste of regular expressions.

# a simple example
#
# or, a much more convoluted example
$ipnum = qr/([0-1]?\d?|2\d\d|2\d\d\d\d)/;
$log\_line =" /$ipnum\.$ipnum\.$ip\_num\.$ip\_num\s/;
#
# and finally, a decent compromise
$log\_line =" /^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\s/;

All these regular expressions will match the IP address at the beginning of $log\_line$, but they aren’t all quite the same. The first will match any sequence of four integers separated by periods (208.190.213.12 and 1.45.3.999 and 314156.2171828.1618.1412). The second will match only valid ip addresses and 0.0.0.0 – not a valid ip address, but close. The third one falls somewhere in the middle, and should be sufficient for our purposes.

5.1.2 Capturing

In perl, you can slurp up any text matched in a regular expression by putting parentheses around it. The parenthesized parts of a successful regular expression match will be put in the variables $1, $2, $3, ... in order according
to where the parenthesis are in the regular expression. For example, the following regular expression will put the four parts of the ip address in $1, $2, $3, and $4.

```
$log_line = '/^\d+.(\d+).(\d+).(\d+)/s;'
```

alternatively, if we wanted to grab the whole ip address in one operation, we could put the parentheses around the whole ip address expression.

```
$log_line = '/^\d+\.\d+\.\d+\d+/s;'
```

### 5.1.3 Substitution

Finally, we take a brief look at the substitution capabilities of perl. Substitution is most simply described in the analogy to search and replace operation in many text editors. The substitution operator (`s/<find>/<replace>/`) has two parts. The first part specifies a regular expression to find, and the second part specifies some text to replace it with. See if you can follow the following example.

```
$text = "I built my house on the sand.";
$text = $text/s/sand/rock/;
print "$text\n";
# prints "I built my house on the rock."
```

Basically, the regular expression stops when it matches the word "sand" and then replaces that match with the word "rock". If you wanted to, say, change all the o’s to 0’s in a string, then you’d have to use the ‘g’ modifier to tell Perl to keep going after it finds one match.

```
$text = "more tomes roam rome";
$text = $text/s/o/0/g;
print "$text\n";
# prints "m0re t0mes r0am r0me"
```

### 5.2 Summary

The preceding discussion on regular expressions is really much too brief, but it would really require a complete new tutorial to fully describe regular expressions in Perl (or any other language, for that matter). Nevertheless, there is much more information online about regular expressions, or you can
check out some of the references to see what they have to say about them (much more than I’ve said).

Nevertheless, this section should have given you at least a glimpse of what is possible with regular expressions.

References

