

1 Document Info

Copyright (c) 2002 Scott Little <little@eng.utah.edu>

Version: 1.30

Created: 03/23/2002

2 General Intro, Purpose, etc.

This tutorial is not really ever finished. It is a work in progress, but is in a state now that should be useful to many people.

*****Start Warning*****

This tutorial is based on my own experiences and knowledge. Everything in it is not guaranteed to be 100% correct or even tested, but I did try my best (If by some freak accident you do find inaccuracies or errors please let me know.). I don't think that anything will cause your computer to self destruct but it may. I can't take responsibility for that. Sorry!!! If you find anything in the tutorial that is incorrect or misleading then let me know and I will try to correct it. Thanks!

*****End Warning*****

Hello, and welcome to my very own perl tutorial. This tutorial was motivated by a desire to help some of my friends learn perl so that they can write scripts to simplify their lives when living in a *nix world (For this reason the tutorial just covers how perl works in a *nix environment. A lot of things will apply to Windows users, but some will not. Sorry, but I don't use Windows (Well, okay I play games under Windows, but that is about it!) so I just don't know how it works. The other drawback to this is that the scripts may not be portable. Again, sorry about that, but I just don't know or care to learn about Perl in a Windows environment.). I taught myself Perl just for the fun of it and have since found it to be very useful. I use it to do many different things from data munging to shell scripting.

This tutorial assumes that you understand general programming constructs, but not a whole ton about Perl. It is written to give you a quick idea about how to do certain things that I find common and useful. For a full treatment of the language I recommend the camel book (Programming Perl) by Larry Wall and published by O'Reilly. The camel book isn't really for beginners, but hopefully this tutorial can get you to the point where the camel book would be useful. Okay, enough general rambling. Lets get going!

3 The Anatomy of a Perl Script

Okay, hang on for the ride. Perl scripts are similar to shell scripts. We need to tell them where to find the interpreter. This is done by using the very first line in the file. Consequently the very first line in every perl script looks something like this:

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

I have found that /usr/local/bin/perl is the most common location to find Perl. I run RedHat Linux at home and it is in /usr/bin/perl on my system. I just create a symbolic link to /usr/local/bin/perl. It is easier. This is done with the following command that will require root:)

```
ln -s /usr/bin/perl /usr/local/bin/perl
```

There is another way to solve this problem. I have now modified my default Perl header so that the first two lines are:

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

This method should cover both of the common cases. If you run into a case where Perl is located in a different location you adjust accordingly.

Well, that is about all that is really consistent from one script to another. I usually add my own custom header via emacs. I think it is really nice for consistency, but you can do what you like. I also add cvs hooks into my scripts because most of them are in a cvs repository somewhere. The Linux section of my web page contains a copy of my .emacs as well as howtos if you want to see how I add a consistent header and setup cvs.

4 Variables in Perl

Oh boy, variables in perl are a treat!!! Variables in perl don't need to be declared. When you use them they spring into existence (If you don't like these feature of the language or find that you shoot yourself in the foot often because spelling errors create new variables instead of using the old one you wanted then I would recommend reading more about the strict pragma). You may have to spend a bit of time getting accustomed to the new method, but it works well. Since variables aren't declared, how do I know what type they are? Good question. Lets talk about that.

Perl does have three variable "types". They are scalars, arrays, and associative arrays or hashes. A scalar variable is represented with a leading \$. Scalars hold single data items. Items can be anything. Strings, ints, floats, transistors, etc. Arrays are represented with a leading @, and as you would expect hold groups of data items. Associative arrays are represented with a leading % and allow one data item to be looked up by its associated data item. Lets look at a few examples of how to use our new variables!

These are examples of how to declare a scalar. The declaration is very free and things generally work how you would expect them to work.

```
$foo = 1234;
$bar = "pmos";
$big_num = 1.234e4567;
$real_num = 1.678;
```

Okay, below is an example of an array declaration. It is really quite straight forward. Scalars could also be added into the list and Perl would extract the value of the scalar and add the value to the array. That is called variable interpolation and it is the rule not the exception in perl. Very, very handy mechanism if you ask me.

```
@who = ("me", "you", "him", "her");
@who = ("me", $you, "him", $her);
```

Well, I know the question that everyone is asking now is how to do I figure out the length of the array? Well, in perl that can be done a couple of different ways (like most everything else of course). Two common ways are shown below. I have used both methods. Sometimes one works better than the other, but usually they both work equally well. In the example the \$var variable will hold the length of the array when the calculation is complete.

```
$var = length($array);  
$var = $#array;
```

If you are now asking how to work with multidimensional arrays then that will have to wait for another day and time. In perl, one dimensional arrays are easy, but using multidimensional arrays requires understanding of a bit more complex perl constructs. The camel book definitely gets into things like this.

Okay, now you wverbatimant to know how to use the hashes in perl I suppose. These are really nifty little guys. You can do an efficient lookup of a value based on the key. A couple of ways to add values to a hash are shown below. There are some other ways, but these seem to be the most intuitive to me. The first method is the one that I generally use to create static hashes. The second method can be used to dynamically create hashes (remember about variable interpolation). The method works very well.

```
%possessive = (  
    "me" => "mine",  
    "you" => "yours",  
    "he" => "his",  
    "she" => "hers",  
);  
  
$possessive{"it"} = $its;
```

Now all the hashes have been filled so what if I want to use the data. It is very similar to adding the data. The example below should give a reasonable explanation.

```
$stuff_from_hash = $possessive{$hash_key};
```

You may have noticed that often times when referring to hashes or arrays the variable name is prefixed by a \$ not the @ or %. This is because we are referring to the variable in its scalar context. At first, it can be confusing and hard to remember when to switch context, but eventually it will make sense.

So that is how variables are declared. One important thing to remember is that scoping of variables in perl is global. Yep, you heard me right. All variables are global (Well, unless you would like to force them to be lexically scoped of course! You can read more about scoping under the subroutines section if you are interested or hate global variables!). Just remember this little fact to help avoid you from getting bitten too hard.

As if having a nice, global set of your own variable wasn't enough perl also provides a number of built in variables that you can use if you would like. You should be aware of a couple of these built-in variables. \$_ is the default input and pattern-search space. If the program receives an input

and isn't told where to put it then it will be found in `$_`. This is also true with pattern matching. The `@_` variable is where subroutine arguments are stored by the calling routine to be used by the called routine. If you want to read more about this default variable then the subroutines section is the place to find the information. The last one that I will mention here is the `@ARGV` variable. This variable contains the command line arguments passed to the program. This variable will be discussed in much more depth in the I/O section of the tutorial.

5 Control Structures

Hmm, well, now we know the basics about Perl's variables it is now time to learn about Perl's control structures. They are really quite similar to the control structures found in C or Java with a few hand additions.

Shown below is the `if-elsif-else` structure of perl. It works just as you would expect. Just don't forget that it is `elsif`.

```
if(condition) {
    code block
}
elsif(condition) {
    code block
}
else{
    code block
}
```

`Unless` is just a fancy way to do a simple `if` block when negating the condition may seem a bit confusing. It is handy, but probably doesn't get used that often. I have found occasion to use it though. It is just nice to have the code read a bit more like English.

```
unless(condition) {
    code block
}
```

This is just a standard while loop as you would expect. Nothing fancy or different.

```
while(condition) {
    code block
}
```

`Until` is similar to the `unless` statement. It is a way to do a while loop when negating the condition would seem confusing.

```
until(condition) {
    code block
}
```

Perl uses standard for loops. They behave just as you would expect. Nothing is unusual or different.

```
for($i=0;$i<MAX;$i++) {  
    code block  
}
```

Now the foreach statement is a very convenient little construct. It is used to step through each value of a list and perform operations on the value. The construct and some examples are shown below.

```
foreach VAR (LIST) {  
    code block using VAR  
}
```

An example would be to step through a list of files that is contained in the @files array. This would be constructed as follows:

```
foreach $file (@files) {  
    do stuff to the files  
}
```

Another common example is to perform operations on each element of a hash. A nice way to do this in an organized way is:

```
foreach $key (sort keys %the_hash) {  
    code to operate on the hash keys  
}
```

Perl has two different types of comparison operators. There are operators that work on strings and operators that work on numbers. It is not hard to compare a number to a string in Perl. Just be careful which comparison operator you use. The comparison operators are shown in Table 1.

Perl also uses the bitwise and (&), or (|), and xor (^) operators. Perl uses the C-style logical short-circuit operators. They are &&, ||, and !. Perl also adds the more readable and, or, and not operators. Be careful when substituting the symbolic operator for the alphanumeric operator because the alphanumeric operators have a much lower precedence.

6 I/O and Filehandles

One important part of I/O is how the arguments given to a Perl script are accessed internally. This is done using the @ARGV array. The program name is contained in the \$0 variable, and the remaining arguments are split up into the @ARGV array using space as a delimiter. The arguments can then be accessed by indexing @ARGV. An example of this is shown below:

Numeric	String	Meaning
>	gt	greater than
>=	ge	greater than or equal to
<	lt	less than
<=	le	less than or equal to
==	eq	equal to
!=	ne	not equal to
<=>	cmp	comparison - returns -1 if the left operand is less than the right operand, 0 if they are equal and +1 if the left operand is greater than the right

Table 1: Perl Comparison Operators

```
if($ARGV[0] eq "--help" || $ARGV[0] eq "") {
    print "There is no help for this script.\n";
    exit;
}
```

This code snippet checks to see if the first argument contains `-help` or is blank. If this is the case then it prints out an error message to `STDOUT` and exits the program. Not very complicated, but it is a useful little code chunk for many scripts.

In Perl filehandles are used to input and output data from the Perl script. The concept of a file handle is really quite simple. A few common filehandles are predefined. These are the standard Unix filehandles: `STDIN`, `STDOUT`, and `STDERR`. These are opened and available for use unless you change them. Filehandles can also be created by explicitly opening them. The Perl convention is to name the filehandle in all capital letters.

An example of how to open a filehandle is shown below:

```
open INFILE, "\"$ARGV[0]\" or die "Can't open file \"$ARGV[0]\" for
reading.\n";
```

The Perl `open` function is used to open the filehandle. Filehandles can be opened in many different ways. The `open` function can take a number of arguments. The syntax is:

```
open FILEHANDLE, MODE, LIST
```

Some examples are:

```
open LOG, "> logfile";
open INFO, "< datafile";
```

The different modes along with the accompanying properties of the mode are shown in Table 2.

Mode	Read Access	Write Access	Append Only	Create Nonexisting	Clobber Existing
< PATH	Y	N	N	N	N
> PATH	N	Y	N	Y	Y
>> PATH	N	Y	Y	Y	N
+ < PATH	Y	Y	N	N	N
+ > PATH	Y	Y	N	Y	Y
+ >> PATH	Y	Y	Y	Y	N
COMMAND	N	Y	n/a	n/a	n/a
COMMAND	Y	N	n/a	n/a	n/a

Table 2: Perl Filehandle Modes

Now that we understand how to open a filehandle. What can we do with them?? How do we read from them? How do we write to them? Well, Perl does provide a few convenient ways to read from a filehandle. The first method can be used when waiting for user input for instance. This method is shown below:

```
$user_input = <STDIN>;
```

This line would grab all the information from the STDIN filehandle until a new line is reached (new line is the delimiter for `|` (the angle operator)) and assign it to the `$user_input` scalar variable. The important thing to notice here is that the name of the filehandle is surrounded by `|`. The angle operator tells Perl to read from the filehandle. Another nifty thing that Perl provides is a very easy way to read data in from a file and operate on the data as long as the file contains new data. A sample code snippet of this is shown below:

```
while(<INFILE>) {
    $new_line = $_;
    #do stuff
}
```

The while loop will continue executing the body of the loop until it receives an undef which in Perl will be the same as EOF. The `|INFILE|` puts the corresponding line into the Perl's default variable which is `$_`. The while loop could also be written

```
while($new_line = <INFILE>).
```

The final good and safe thing to do when finished doing file operations is to close the filehandle. This is a very simple operation that uses the Perl close function. The syntax is simply: `close FILEHANDLE`.

There are a few other neat little tricks that can be performed using Perl filehandles. One that I have found to be rather useful is a little trick that allows the output of a system call to be piped into a perl FILEHANDLE. This is very convenient when it becomes useful to know the current state of the system while a program is executing or get the output from an external program. An example of this code is shown below:

```
open(INFO, "ps -aux | grep foo |");
```

This one liner will execute the ps command and pipe the results into grep and then pipe the results into the INFO filehandle. The INFO filehandle can then be examined just like any other filehandle. The key item is the final — in the quoted command. This pipes the data from the command into the Perl filehandle. Nifty, eh!?!

****Start Another Way****

There is another really quick and easy way to do this using the backticks (the unshifted version of on most keyboards). Just surround the desired command with backticks and assign the result to a variable. For example:

```
$info = `ps -aux | grep foo`
```

does the same thing as the above command.

End Another Way

@ARGV can be a tricky little thing. Here are a couple of notes about how @ARGV works differently than you would naturally expect. There are a number of ways to get the length of an array. I have used them all with varying degrees of success in different situations. To get the length of the @ARGV array the following line of code should be used:

```
$size = scalar(@ARGV);
```

-Check the camel book for this info -how does @ARGV work with <>?

7 Commonly Used Perl Functions

Perl contains many functions that are useful for text and data munging. I will attempt to discuss a few of the functions that I use frequently. I won't be able to describe all of them, but this should be a good start.

The chop and chomp functions do almost the same thing. When you read a line in from a filehandle often the file will still contain that pesky new line character on the end of the line. The job of chop and chomp is to remove this annoying little guy. chomp is a safer function because it will only remove the new line character if it exists. The chop function will blindly remove the last character from the line. Not always what you would like, but it can still be very useful.

The split function is a very handy little guy. It can be used to split up the contents of a line into individual array elements based upon a user specified delimiter. For instance, maybe you are reading in a file that has the data you need separated by commas. You can use the split function to isolate each data item into an array location. The code snippet to perform this operation is shown below:

```
@split_data = split /,/, $data_from_file;
```

The arguments to the split function are a regular expression for the delimiter contained within a set of slashes. Then a comma to indicate that the next argument is coming up. Finally, the scalar variable that contains the data to be split is given. That is about it for the split function. It is a handy little guy.

Another set of useful functions are the pack and unpack functions. These functions are complex and versatile. I do not pretend to understand exactly how these functions work, but I will give a simple example of how they can be used to do some data extraction. Another common way to receive data in the input file is by a specified field length. This may seem like it could be annoying to parse, but here is the code to parse a line which contains three fields where the fields are 10, 8, and 3 characters long respectively.

```
@parsed_data = unpack("A10, A8, A3", $data_to_parse);
```

Not too bad! The unpack function is used to unpack ASCII (hence, the A) chunks of data that are 10, 8, and 3 characters long respectively. The scalar variable to be parsed is give last as was the case with the split function. Both pack and unpack are equally useful. These functions can be used to convert data from binary to hex to ASCII, etc. Very nice functions.

8 System Calls

I generally use Perl as a replacement for a shell scripting language. I am not sure whether this is a good choice or not, but it is the choice that I have made because I haven't bothered to learn much about any specific shell. Because I use Perl in place of a scripting language, being able to make calls to executables on the system is critical. This section will attempt to describe the basics of making these calls.

There are more ways to make a system call than I am going to describe here. I use these methods because I receive feedback about the success or failure of the call. I find this useful for error checking purposes.

The two functions that I use are system and exec. The syntax for both functions is the same, but I will use system in most examples because I tend to use it more often. There is a difference between the two functions. The difference is that after the shell command is executed a system call will return control to the script. An exec call will exit the script after successfully executing the shell command. An example of how to use exec and system is shown below:

```
system "/usr/local/bin/shell_command"
```

A nonzero return value indicates that an error was returned by the shell command. A zero return values indicates that everything went well. Due to this little fact most of my system calls follow the format shown below:

```
if(system "/usr/local/bin/shell_command") {  
    print "An error occurred.\n";  
    exit;  
}
```

Okay, so the error message should be more useful and specific. Also, exiting the program on an error might not be what you want to do, but it is surely an option.

To make Perl a bit more portable among different platforms a few of the common file operations have been translated to perl functions. It is best to use these functions when you perform file type operations. Also, when you execute a script and try to perform file operations those operations are performed in the directory where you executed the script unless you explicitly change the directory inside the perl script. Just try to remember that because it will bite you somewhere along the line if you do much file manipulation without thinking about that little fact. Below is a list of each of the Perl specific file operations and how they are called.

Perl has functions for `chdir`, `chmod`, `chown`, `mkdir`, `rmdir`, `symlink`, and `unlink`. The syntax and a short explanation for these functions follows. There are also many other Perl functions to do many other system functions. The camel book has a huge list, and I just don't have room or the time to list them all.

```
chdir - changes the current working directory
chdir DIR
```

```
chmod - changes the permissions for a list of files
chmod MODE LIST
```

```
chown - changes the owner and group for a list of files
chown NUMERIC_UID NUMERIC_GID LIST
note: on most systems -1 will leave the value unchanged
```

```
mkdir - creates a directory with the name and mask specified - if no
mask is specified then 0777 is assumed.
mkdir DIRNAME, MASK
```

```
rmdir - removes the specified directory if it is empty - the function
will return 1 on success
rmdir DIRNAME
```

```
symlink - creates a new symbolic link
symlink ORIGINALFILE, SYMLINK
```

```
unlink - removes a list of files
unlink LIST
```

9 Scoping & Subroutines

Here seems like a good place to talk about the strict pragma. The default for Perl is to be a very free-form language with few rules. This is both good and bad. I enjoy not having to worry about declaring variables, but sometimes it can be painful because spelling errors or stupid mistakes create very unexpected and hard to find errors. The strict pragma is designed to give a bit more structure to Perl. The strict pragma enforced lexical scoping among other things. Lexical scoping means that you must declare variables, and these variables are only valid withing the lexical scope (if you don't know exactly what lexical scope means you can just equate it to the scoping rules in C++).

To start using the strict pragma just put the line `use strict;` at the beginning of your script. Then to declare a variable you must type `my VAR`. The `my` indicates that you are using lexical scoping. This isn't the best description, but it should serve to get you started. I really recommend using the strict pragma because it enforces a number of rules that promote good programming practice. Subroutines or functions are always a useful thing. Perl does support these. As with variable declaration you can do it the unchecked, easy way or you can make things a bit more verbose and include some checking facilities. Declaring subroutines is rather simple. You simply use the `sub` keyword. For example, a perfectly good Perl function is:

```
sub foo {
    print "This is function foo\n";
}
```

How do you call the function. Well, that is rather straightforward as well. You can simply call the function by saying:

```
foo();
```

What about a more complex example. How do you pass parameters? Well, it isn't much harder. Here is an example:

```
sub foo_param {
    $param = $_[0];
    print "The parameter is: $param\n";
}
```

It is called using:

```
$the_val = "Hi";
foo_param($the_val);
```

Okay, so what did we do here. The idea is that in perl you don't have to tell the function what types or numbers of parameters to expect. You can just define the interface and be careful to not violate these assumptions in the program. This can work fine. All of the parameters passed to the function are stored in the `@_` variable and can be removed from this variable inside the function. Well, that is the quick and unchecked method. How can we be a bit more formal and try to avoid silly mistakes by misunderstanding the interface to the function etc. Even while using the strict pragma function prototypes are not required. This is something that the programmer must decide and implement on his own.

First to allow checking of the function prototype the function must be declared before it is used. For this reason you will often see the functions in a Perl program declared at the beginning of the program if function prototypes are used. This isn't always the case, and there are ways around doing this. I will leave this as an exercise for the reader.

Below is an example using function prototypes. It should be straightforward.

```

sub foo() {
    print "This is function foo\n";
}

sub foo_param($) {
    my $param = $_[0];
    print "The parameter is: $param\n";
}

foo();
my $the_val = "Hi";
foo_param($the_val);

```

Okay, these are some very simple examples. There are a few subtleties that I have ignored up to this point. One of these is how to pass an array or hash to a function. The values passed are put into an array. Well, how do you put an array into an array? The idea here is to use what Perl calls references (if you feel more comfortable using the word pointer then you can because that is basically what they are). First, I will discuss how references work and then I will move on to show how they work with functions.

The first question is what is a reference? Well, it is simply a scalar value. Arrays and hashes hold scalar values so they can also hold references. To create a 2D array or a hash that contains arrays you must use references. How do you create a reference? This is done using the \. For example:

```

@a_list = ("who", "what", "when");
$a_hash{$the_key} = \@a_list;

```

Okay, so we inserted a reference to the @a_list array into the %a_hash hash. That is very handy, but how do I retrieve the array from the hash? Well, I am glad that you asked. The code below does the trick.

```

$ref = $a_hash{$the_key};
@new_array = @$ref;

```

The idea is to get the scalar value for the reference and then tell Perl that this scalar is actually a reference to an array using the

```

@${<variable_name>} syntax.

```

Well, that may seem a bit ugly, and it is. It does work though and makes a lot of sense from the standpoint of pointers (dereferencing, etc.). How does this work with function prototypes? Well, since functions just make a list of the parameters why don't we just pass references to the functions. Here is an example:

```

sub foo_list(\@) {
    my $list_ref = $_[0];

```

```

    print "list ref is: $list_ref\n";
    my @list = @$list_ref;
    print "The list is: @list\n";
}

my @a_list = ("who", "what", "when");
foo_list(@a_list);

```

The reason that I am printing the list reference is that it is instructive to see what the reference actually looks like to Perl. This might help you when you all of a sudden see this type of output when debugging a script. You will just realize that you forgot to dereference a reference. The final item is what about multiple arguments to a function. How is that done. Another small sample code is below:

```

sub foo_list_param(@@ \%) {
    my $list_ref = $_[0];
    my @list = @$list_ref;
    print "The list is: @list\n";
    my $a_scalar = $_[1];
    print "The scalar is: $a_scalar\n";
}

my $the_val = "Hi";
my @a_list = ("who", "what", "when");
foo_list_param(@a_list,$the_val);

```

Multiple parameters in the prototype can be separated by spaces or just clumped together. I prefer spaces for readability, but you can make your own decisions.

Well, that is about it for subroutines and references. It is a quick introduction to a reasonably complex topic. This is a good starting point.

10 Regular Expressions

Wow, we finally made it to the discussion on regular expressions!!! Welcome to the power of Perl. Regular expressions give Perl the parsing and munging power that many other languages lack. The problem is that they are not always obvious or easy to understand. This section hopefully will familiarize you with the basics, and you can progress from there.

The main idea behind regular expressions is that they are great for pattern matching. The regular expression defines a pattern and then text can be searched for a matching pattern. Decisions can be made based on a match.

The first type of regular expression that we will discuss is a simple matching pattern. The pattern is placed between a pair of slashes and then a letter may follow the pattern to change the properties of the matching. These letters and the corresponding changes are shown in Table 3.

Okay, now lets talk about how to create a pattern. It can be very simple or become extremely complicated. The pattern below looks for a tree in the \$forest variable. It will return true if it finds one and false if not.

/i	case insensitive
/s	let . match newline
/m	let ^ and \$ match next to an embedded \n
/x	ignore (most) whitespace
/o	compile pattern once only

Table 3: Perl Regular Expression Modifiers

Symbol	Definition
	match on LHS or RHS
(...)	group this expression
[...]	match as a set
^	must match at the beginning of the string
.	match on character
...\$	must match at the end of the string
*	match 0 or more times
+	match 1 or more times
?	match 0 or 1 times
{NUM}	match NUM times
{MIN,}	match at least MIN times
{MIN,MAX}	match at least MIN times but not more than MAX times

Table 4: Some Perl Regular Expression Metacharacters

```
$forest =~ m/tree/;
```

The syntax for Perl regular expressions is quite standard, but I have included the most commonly used symbols and their definitions in Table 4.

There are also a few escape sequences that match for certain character classes. I have listed some of the common definitions in Table 5

Okay, that is a lot to digest. Lets just do a few examples. Let's say I am looking for a word that starts with an 'a' then has another character and a 'b'. That would be:

```
$string = m/^a.b/;
```

What about starts with an 'a' and ends with a 'b'?

```
$string = m/^a*b$/;
```

Hopefully you get the idea now. It isn't exactly the easiest thing to understand at first, but it will grow on you and soon you will be a regular expressions genius. Just keep working at it (or you can take a CS theory class that discusses regular expressions!).

Escape Sequence	Definition
\0	match the null character
\NNN	match the character given in octal
\cX	match the control character control-X
\d	match any digit
\D	match any non-digit
\E	end case
\l	match the next character as lower case
\L	match lower case until the next \E
\n	match the newline character
\s	match any whitespace character
\S	match any non-whitespace character
\u	match the next character as upper case
\U	match upper case until the next \E
\w	match any word character (alphanumeric plus -)
\W	match any non-word character
\x{ffff}	match the character in hex

Table 5: Some Perl Regular Expression Metasymbols

Another great use for regular expressions is for substitution. A substitution can happen based on a match. This is done by putting the variable to be modified on the LHS of the regular expression assignment operator. An `s` precedes the the opening slash which is followed by the regular expression for the match. Then a slash which is followed by the substitution which is followed by an ending slash which can be followed by match modifiers. An example of this is shown by the following expression that removes leading white space.

```
$var =~ s/^\s+//;
```

The regular expression searches for all leading white space and replaces it with nothing. If we wanted to replace leading white space with something else then that information would be put between the ending slashes.

11 Checking Syntax & Debugging

Well, Perl is a scripting language so it is not compiled. There are some nice mechanisms to check the syntax of your script though. The way that I prefer is to use the `perl -cw {scriptname}` command. The `-c` flag causes perl to check the syntax of the script and then exit before executing the script. The `-w` flag turns on all warnings. These are very handy little switches. I have gone to the trouble of setting up an emacs command sequence to perform this operation. Almost like compiling, but not quite.

There are debuggers that do exist for Perl, but I don't have experience with any of them. I just resort to the old boring method of using `print` statements to print critical variables at important times so that I can find the improper operation. It obviously isn't the most efficient method, but that is how I write scripts.

-research debuggers and link them