A Brief Introduction to Programming in Perl

By Douglas
douglas at weaknetlabs dot com

Part 1. Interpretive vs. Compiler Languages
1. Introduction

Part 2. An example program
1. Escape Characters
2. Strings

Part 3. Variables, Strings, and Arrays.
1. Syntax
2. Arrays

Part 4. Input from Users
1. Chomping Data

Part 5. Statements and Loops
1. If statements
2. Else
3. Operators
4. Loops

Part 6. File Handling
1. Reading Files
2. Writing to Files
3. Append
4. Deleting Files and Directory Listings.

Part 7. Sub Routines

Part 8. Splitting and Matching.
1. Splitting
2. Matching

Part 9. Perl GUI applications
1. Simple Example
2. Images

Part 10. Conclusion
1. Debugging
2. Illegal Names

Part 1. Interpretive vs. Compiler Languages.

Introduction

Perl is an interpretive language. You can think of it as function, like in mathematics. Here is a simple to understand analogy to what "interpretive" means. Say, we have a function

```
f(x) = x³
```

This function means that for each value we give "x" it will cube it, or multiply it by itself 2 times:

```
x * x * x
```

Perl is the function in our case, and the "code" or "application" is the "x." If you give the program to Perl, it interprets the code as a whole and redirects the output to your screen (or stdout).

For every character in the above example the interpreter or "function" does something specific to that character. For instance, the "*" character's are interpreted as "multiplication." The "x" characters are interpreted as values. So in essence, these characters are "nicknames" that the function understands and "interprets" to create meaningful output.

Perl is the same way. You give Perl a statement such as

```
print “foo bar foo man bar!”;
```

and it gets interpreted as "print the text in quotes to the stdout (or screen)." So, one can think of the above statement, relative to the function example above, in the same manner: the value is the "string" or row of characters within the quotes, and the "`print`" part is what to do with those values. The quotations and semicolon are like the multiplication operator, they tell the print function how to handle the text. We will get to all of this in our first Perl code analysis in part 2.

An example of a non-interpretive language, would be C. C is a compiler language, so all of the above functions and values are compiled at the start and there is no need for an interpreter. What you get is an "executable" once the code is compiled. The C program will still have "functions" within it to carry out tasks, but they don't need to be handed over to an interpreter, they are run right from the computer itself.

So that's how Perl works. It seems that it would be slower to hand your programs to another program to interpret and run them, and in comparison to compiled languages, it is. With the speed of modern computers even today in the beginning of 2009, Perl can interpret large amounts of code in milliseconds. When Perl uses "modules" to interact with TCP/IP, large files, or hardware in any certain way, is where you would see a significant difference.

Here's a nice example. I wrote a program that "cracks" encryption, using the MD5 method. This program opens a massive dictionary file, a file containing line by line list of words, and creates a "hash" for each line using a key or a "salt." Then, Perl matches every line that is newly hashed from the output to the hash that was given as input. If both hashes so happen to match, Perl tells you what the password is and that it has been "cracked." This code, on a Pentium 4, 3GHz computer with 1GB of RAM can only guess at about 115 wps (words per second). Now a similar program that uses a C based MD5 hash method (John the Ripper[1]) could get (on the same machine) about 3500 wps!

I am not trying to get you to want to choose another language, I just want you to understand the differences between both worlds of "interpretive" and "compiled" languages. Actually, Perl is a great language to start using if you are new to computer languages. It's easy to use, fun, and quick. You can write a Perl program very quickly to do a tedious task for you!

1 John the Ripper is a fascinating program and a system admin/security nuts must have. I really wasn't trying to reinvent the wheel here, I just wonder how things work and thought it was a good exercise. Code and video can be found here: http://weaknetlabs.com/code/perlwd

Part 2. An example program.

Let's now look at the code itself. Here I will present an easy Perl program that should help someone new to computer languages to understand them.

```
#!/usr/bin/perl
print “Hello World!\n”;
exit;
```

This program will simply print “`Hello World`” out onto the screen and exit. Let's look at this program line by line. The first line is the path to the interpreter Perl. This line is usually only necessary if we make the Perl code an executable. So when the program is ran, the shell it's run from (*bash, sh, zsh*, etc) knows to use Perl to interpret the rest of the code. The second line is the `print` function. This is another math-like function. One can think of it as

```
print()
```

and anything put into the “()” would be the value to interpret, or in our case “print to stdout” (the screen). The last line then tells Perl to stop and go back to the shell which invoked it.

From the code above you can see that every line of actual code ends with a semicolon. This needs to be true! Semicolon's are put after each expression or function you put into Perl. They do not go after curly braces usually. If you forget the semicolon's usually Perl will tell you were you went wrong, but it's best to be careful to avoid errors!

Escape Characters

You may also notice a “`\n`” in the quotes of the value given to the `print` function. This is equivalent to pressing ENTER or creating a new line. It is called an escape character and the “n” stands for “new line.” There are many escape characters you can put in your value for the print function such as “backspace,” “tab,” “carriage return,” or anything you choose, you can escape.

If you want Perl to print a character like a quote, which Perl interprets as an operator, and you did something like this:

```
print “a quote like this “ starts or ends a value given to the print function\n”;
```

Perl would return an error. This is because the print function looks for the value (or string) between two quotes. So if a quote was in the middle of the string, `print` would think the string stopped at the middle quote and return ”a quote like this “ then return to Perl. Perl would wonder were the missing semicolon right after the middle quote was and return an error about a bare word being found were an operator is expected.

To get around this we add a backslash to the middle quote to make the print function ignore it as an operator and think of it as a character to print.

```
print “a quote like this \” starts or ends a value given to the print function\n”;
```

As I have previously mentioned, it's easy to think of the quotes, semicolons, and backslashes as “operators” in the print function. They govern what to do with the alpha character values, and ultimately what the output will be.

Part 3. Variables, Strings, and Arrays.

A variable is just like a mathematical variable. You can assign anything you want to it. In our mathematical example above, our variable was "x." Meaning we could assign anything we want to "x" and the function will carry out "x" as governed by the operators in it's rules. In Perl, you can assign values to variables as well. Here is a small example of that in action:

```
#!/usr/bin/perl
$foo = "great";
$bar = "awful";
print "Is this a $foo tutorial, or an $bar tutorial?\n";
exit;
```

The output would be to simply print out "`Is this a great tutorial, or an awful tutorial?`" followed by a new line (or enter). Here you can see, a dollar sign is to signify variable's container. Just how any number can be stored in "x," Any value, be it a number, string, letter, etc, can be stored into a name with a dollar sign in front of it.

```
$variable = what to put into variable.
```

Syntax

You can also see, that it is quite important as to what you assign to what. Meaning the "syntax" must be correct or you will assign nothing to something. Syntax is the special order in which the language must be written to get the desired output from Perl. The variable's container must be on the left side of the equation. So we assign whatever is on the right side to the left side.

Syntax is like the main rule. Spelling needs to be perfect. Punctuation, and symbols and order of all, including semicolons at the end of each complete function, all must be perfect. Computers are smart, but not smart enough to correct syntax mistakes yet. Perl does, however, tell you (usually in a straightforward manner) what's exactly wrong with the program.

Strings

I have been mentioning strings in this tutorial and would like to cover them a bit further to help one understand all of the terminology I use about programming languages. Strings are simply rows of characters. This sentence is a string. "foo the man bar foo, 44, yes ASDFG ddddddddd\n\n\n\t" is a string also. You can have even very long spaces as strings.

Now that you know what a string is, you can assign a whole string to a variable even with tabs and spaces!

```
$var = "this is a                    string, with 66666666666 stuff. And \tstuff.";
print $var;
exit;
```

This would print the $var string and exit.

Arrays

An Array is a list. Like variables they, too, have a leading special character and that character is the "@" symbol. An array is a list of strings, numbers, or characters, and each thing in the list is called an element. Each element is numbered in the list starting at zero. Here is a list:

```
   0. red
   1. green
   2. blue
   3. yellow
   4. orange
```

Here is an array with the same elements:

```
@array = ("red", "green", "blue", "yellow", "orange");
```

This array labeled "array" has the same elements as the list above. Each element is read by Perl from left to right, so the first element is labeled "0", the second "1", etc. All computers and computer programs start with zero instead of 1 when keeping track of elements.

If we were to put this array into the `print` function, it would do just that: print "`redgreenblueyelloworange`" because no spaces where put before or after the elements. You probably won't use this method much unless a single element is a paragraph, HTML code, or whatever. There is a way to print a single element of an array, and that is to specify that element's number in brackets immediately after the array's name. Here is an example of that:

```
#!/usr/bin/perl
@array = ("red", "green", "blue", "yellow", "orange");
print "the first element in \@array is @array[0]\n";
print "the second element in \@array is @array[1]\n";
exit;
```

Notice how I "escaped the "@" symbol in the print function? The output would look like this:

```
the first element in @array is red
the second element in @array is green
```

You can see the syntax for printing a single element of an array is to simply type out the array name with the special character, then put the element you want to use into the square brackets.

To print all elements by simply doing a `print @array` and have them discrete (or separated) you can add spaces to the ends of the elements in the array. That means element 0, would be "red ", element 1 would be "green ", etc. This is probably the easiest way I could think of to introducing someone to simple arrays. It shows off how to print each individual element, or the element as a whole. Arrays will be very useful to you once you really get into the swing of things with Perl and you will learn new functions to add, remove and change elements in arrays very easily.

Part 4. Input from Users.

Programs are really all about input and output, or “IO.” A lot of acronyms you here in the computer world that have those two letters together in that manner mean “INPUT OUPUT.” With Perl, getting input from a user is easy. You simply need a variable and to assign the <STDIN> function to that variable. Usually you will want to use the `chomp` function along with the <STDIN> function. The `chomp` function cuts off the trailing newline character that gets put into the <STDIN> based variable by the end user of your program typing out the word you are asking for then pressing enter.

Say you want to get a word from someone and you want to print that word back out by simply using the print function. If that person put in “HELLO” and pressed enter, the chomp function would knock off the enter or new line so you could use that word in a sentence. Like so:

```
#!/usr/bin/perl
print “Give me a word “; $word = <STDIN>; chomp $word;
print “That word $word is great!\n”;
exit;
```

This example shows how I got the variable $word to contain the users input. The input was gathered using the <STDIN> function, then chomped to delete the new line. If it weren't chomped the output would look something like this:

```
That word olives
 is great!
```

Notice the space before “is” is still valid, and that the enter key I pressed after typing out the word *olives* was printed out too. Gathering input is quite easy. Just make sure you don't accidentally define the variable before asking the user what kind of input you are looking for, or you'll get some complaints!

Part 5. statements and loops.

If statements

The `if` statement is very easy to understand, it does a “block” of code **IF** the if statement is correct. The `if` statement is first, the rules are second and the block is in *curly brackets*. Here is a general definition of an `if` statement:

```
if (this OP that) { do this stuff here... }
```

You don't need semicolons in `if` statements as you can see. An `if` statement is good to use when you need specific IO. Let's say we want a program that takes either a yes, or a no (“y” or “n”) input from a user. Well, your **`if`** statement can do just that, and perform different tasks that you assign to either answer! Once the input from the user has been chomped Perl puts the answer into the above definition as “that.” Here is a small and easy example of an **`if`** statement in a real life program.

```
#!/usr/bin/perl
print “Do you like Nintendo Wii? (y/n)\n → “;
$ans = <STDIN>; chomp $ans;
if ($ans eq “y”) { print “Because it\'s the best!”; exit; }
if ($ans eq “n”) { print “You don't know what you\'re missing!\n”; exit; }
```

You can probably just skim over the lines and determine what the outcome of this program will be once run. If the answer you give it is “y” and enter, it prints the first **`print`** function's value. If you give the program an “n” then it prints the second **`print`** function's value. Note again, how I escape the apostrophe's in the value given to print.

Else

Now what if you want to tell the user that they didn't enter a correct value? What if they enter “h” or “lol?” That's where you would want to use an `else` statement. Think about this – you wouldn't want to write a program with an infinite amount of `if` statements! So the `else` statement takes care of all of that for you in one single statement. Here is a general definition.

```
if (this OP that) { do this stuff here... } else { do something else }
```

This will do “`something else`” if say in our program above I were to enter anything *else* besides what was expected. Let's re-code the above program with this new information.

```
#!/usr/bin/perl
print “Do you like Nintendo Wii (y/n)? \n → “;
$ans = <STDIN>; chomp $ans;
if ($ans eq “y”) { print “Because it\'s the best!\n”; exit; }
if ($ans eq “n”) { print “You don\'t know what you\'re missing!\n”;exit; }
else { print “That wasn\'t a \”y\” or a \”n\”!”; exit }
```

Great! Now, if I were to enter anything else on my keyboard[2] the else statement takes over and kills off the program complaining about how it didn't get what it wanted.

Operators

The “OP” I put into the general definition, that translated to “eq” in my examples, acts just like an operator in mathematics. In mathematics we can have things like “<” “>” “=” or combinations of these. These same characters can be used when dealing with numbers. Say we want a number as the input from a user. Here we can simply do:

if ($ans = 2) { do stuff here... }

The *block* of code “do stuff here” would be executed if the input from the user was “2.” When dealing with strings, you need to specify “eq” as the operator. You can also have “ne” meaning “does not equal.” You can have “lt” for “less than,” or “gt” for “greater than.” There are many operators that you should familiarize yourself with while learning Perl programs because, as I said, programming is all about input and output!

Loops

Loops are great. You can have loops do tedious tasks for you that require lots of repetition. Say we get input from a user that is a number and we want to count to that number from zero. We could simply define a variable $n as equal to zero, and issue a while loop that says “while $n is less than one plus the input value, print $n, add 1 to $n, then when complete (**`while`** loop is true), exit.” This will do just that:

```
#!/usr/bin/perl
print "Give me a number ";$m = <STDIN>; chomp $m;
$n = 0;
while ($n < (1 + $m)) { print "$n "; $n++; }
print "\n";
exit;
```

The reason we add a one to the user input is so that it gets printed out to the screen as well. Then I added a newline character, so that when the shell comes back from Perl it looks clean by starting [bash] on a new line.

2 Except CTRL+C that kills a captive Perl program like this were it's not specified what to do when the signal is received.

There's new syntax in the above program that you aren't aware of, it's the incremental operator. The incremental operator adds one to the value of a variable, in our case $n, for each turn the loop takes. This is very important to understand, as loops are another crucial part of constructional programming.

The extra parenthesis in the above `while` loop are to simply contain the expression "1 + <the input variable>" This is very much like mathematics. Things in parenthesis's[3] get calculated first as a rule to keep every answer correct and the equations and expression organized!

There is another loop I'd like to cover, and that is the `foreach` loop. The `foreach` loop is very useful when dealing with sets of things, or arrays, along with repetition. Say we have an array, and we want to to print the elements of the array as a single word in a sentence. You could write out each line using the print function, or you could tell Perl "`foreach` (element in array) { print the element in a sentence}" Here the block in the curly brackets is carried out for each element in the array.

A new special variable I'd now like to introduce is the `$_` variable. You can use this variable to represent the current line of a read file, or array. Watch how I use this variable in a real life program below.

```
#!/usr/bin/perl
@array = ("lol", "hello", "mhmm", "pieces");
foreach (@array) { print "The world revolves around the word $_\n"; }
exit;
```

The output of that program looks like this:

```
The world revolves around the word lol
The world revolves around the word hello
The world revolves around the word mhmm
The world revolves around the word pieces
```

Here you can see that the `$_` Perl special variable was the current line as the `foreach` loop went through each element of the @array array. So wherever `foreach` printed the line $_, it then changed $_ to the next line and printed that line. This is also true with reading from files. I will cover reading from files in the next part of this tutorial.

The two above examples should be good enough to get someone new to Perl familiar with loops. Loops are very easy to understand and, at times, can save you a lot of typing. Once you get good with basic loops you might want to start reading into how to nest loops and statements. For a beginner this is very confusing and I really don't recommend trying it right off the bat!

3 Remember PEMDAS? If not maybe you should search Google for it!

Part 6. File Handling.

Files play an important role in most operating systems. The Unix and Linux operating systems are all based off of plain text files! The configurations for most server services in said operating systems can usually be found in the /etc directory. Perl runs text files! You write the text files and Perl turns them into applications! You can certainly see that files are very important, and Perl knows this. Perl can open files for reading, writing and even appending.

If you are familiar with the Unix and Linux operating systems, you most likely already know this. You are probably familiar with the “>” and “>>” output operators for printing to an appending to files. Well, Perl uses the same syntax! Here is how to open a file for reading:

```
open (variable name, “real file name”);
```

Say we want to open the file “test.txt” from our current working directory, and give it the name “FILE0” we would do:

```
open (FILE0, “test.txt”);
```

Here's how we open a file to write to it.

```
open (FILE0, “>test.txt”);
```

Please take note that syntax will overwrite or “clobber” the file “text.txt” if anything is printed to it. This next example is how we open a file to append data to the end of the document.

```
open (FILE0, “>>text.txt”);
```

All of these, if you have been using Linux or Unix, you should already be familiar with.

Reading

So once these files are open, what do you do with them? Well, first I'll cover how to print out the lines of the file, line by line. You may have already guessed, that the $_ Perl special variable will be used. Here is a quick example of how I read from a file and print out each line.

```
#!/usr/bin/perl
open (FILE, “file.txt”);
while (<FILE>) { print $_; }
close $FILE;
exit;
```

Writing

This will print out each line of "file.txt" to stdout (my screen) and exit. See how easy this is? Now let's do an example of printing to a file. Let's an existing file and print the line "Hello word!" to it and exit.

```
#!/usr/bin/perl
open (FILE, ">file.txt");
print FILE "Hello World!\n";
close $FILE;
exit;
```

It is **VERY** important that you remember to close the file after writing to it. If you don't the changes will not be kept! To close the file you simply do:

```
close (file variable name);
```

as I have done in the above program.

To print to the file I simply used the `print` function, but I placed the files variable name before the value given to `print`. Then I called the exit function to notify Perl that I am done. If the file "file.txt" contained anything, it would be overwritten (all gone) with the line "Hello World!" If the file "file.txt" doesn't exist, Perl creates it with the specified ">" or open to write operator. Again, you must remember to call the close function with the files variable to save changes!

Append

Let's look at an example where I open a preexisting file and write to the end, or add to the end, of the file. In this example I will add the line "This is the end of the file" to the file "file.txt."

Let's say that the file "file.txt" contains the following:

```
Hello,
I am inquiring about a System Admin job, and would like to be hired.  Can you please call
me at 1.foo.foo.barr?  Thanks!
–  Douglas.
```

And our perl program is like this:

```
#!/usr/bin/perl
open (FILE, ">>file.txt");
print FILE "This is the end of the file.\n";
close $FILE;
exit;
```

If we run the Perl program, the file "file.txt" would now look like this:

```
 Hello,
I am inquiring about a System Admin job, and would like to be hired.  Can you please call
me at 1.foo.foo.barr?  Thanks!
- Douglas.
This is the end of the file.
```

If we run the Perl program once again, the file “file.txt” would then look like this:

```
 Hello,
I am inquiring about a System Admin job, and would like to be hired.  Can you please call
me at 1.foo.foo.barr?  Thanks!
- Douglas.
This is the end of the file.
This is the end of the file.
```

You can see how if Perl opens the file with the “>>” or, append operator, it does just what it's name implies.

Deleting files and Directory Listings

The Unix and Linux Files systems contain links to files in an index. So when you delete a file you are just deleting its place holder in the FS (File System). The data stays there until something else write it over. So Perl uses a function called “unlink” to delete an unwanted file. Here is an example of how I delete a file with Perl that is in my current working directory.

```
#!/usr/bin/perl
unlink (“file.txt”);
exit;
```

Someone would probably look at this and think “wow, what a pointless program!” Well, this is just showing off the function `unlink` itself. You can use the function to get rid of created files, temp files, or all of your files if you wanted – in any program.

Part 7. Sub Routines

All of the above code and code samples you have learned from so far can be further enhanced with better functionality as a whole for your programs. Let's say you want a specific answer like in the `if` statement above that quits if the user inputs anything that is not a “y” or an “n.” What if you coded a huge program that you didn't really want the end user to have to run each time he/she makes a mistake?
Wouldn't it be easy if we could just call the <STDIN> function again and again until we get what we want? Well, you can with sub routines!

If you think of a program as set of instructions to be carried out, then you can easily understand the concept of sub routines. In this way of thinking you can “group” sub-sets of instructions together. To do so we simply use curly braces like we did with “blocks” of code. Here is a general definition of a sub routine:

```
sub <subroutine name> { set of code to be carried out }
```

You define the sub routine like you do with a variable, but instead of using a “$” symbol, you use the word “sub” and a space. Then you announce what you want the subroutine to do by putting that information into the curly braces, like a *block* of code. To call a sub routine, simply put an ampersand in front of the sub routines name. Let's take a look at the above if program re-coded to ask the question over and over until we get either a “y” or an “n” as our input.

```
#!/usr/bin/perl
&ask;
sub ask {
print “Do you like Nintendo Wii (y/n)? \n → “;
$ans = <STDIN>; chomp $ans;
if ($ans eq “y”) { print “Because it\'s the best!\n”; exit; }
if ($ans eq “n”) { print “You don\'t know what you\'re missing!\n”;exit; }
else { print “That wasn\'t a \”y\” or a \”n\”!”; &ask; }
        }
```

In the above example, you can see I tell Perl to direct its attention to the “ask” routine, then I define the sub routine using the general definition syntax. This will now output the following when ran:

```
Do you like Nintendo Wii (y/n)?
 --> j
That wasn't a "y" or an "n"!Do you like Nintendo Wii (y/n)?
 --> ef
That wasn't a "y" or an "n"!Do you like Nintendo Wii (y/n)?
 --> y
Because it's the best!
```

This is very convenient. Remember the example where I asked you if you would rather write out an infinite amount of `if` statements or use the `else` statement? This is the same way. Nesting `if` statements can be very confusing and wastes a lot of time. Use the else statement and sub routines to make your programs look better, more compact, and easier to debug!

Part 8. Splitting and Matching

Splitting

Strings can be split into parts. The same way we use “cut” or “awk” in Unix or GNU Linux. When Perl separates a string of text, it uses special dividers. These dividers are called “delimiters” in the programming world. Take the string “Here is a sentence.” for example. If I use the spaces as a delimiter, then I can add the elements “Here,” “is,” “a,” and “sentence.” into an array. Then I could use a single part of that string (or element from the array) using the @array[n] syntax[4]. To split the string you can simply use the `split` function. Here is the general definition I came up with for the `split` function:

@array = split(/<delimiter>/, $string);

What you choose to put into the “/ /” in the print functions value, is called a “regular expression” or “regexp” for short. Regular Expressions go a long way in the Unix/GNU Linux/Programming world, and are used for many things. They are similar to an actual programming language themselves, in essence, using special characters to declare rules. If you are new to them I'd suggest finding a good book on them! Here is a quick example of a program that uses the split function.

```
#!usr/bin/perl
$string = "hello this is a string";
@array = split(/ /,$string);
foreach (@array) { print $_."\n";}
exit;
```

This program will simply take the input $string and split it up dividing the string by every space. Then adds the elements “hello,” “this,” “is,” “a,” and “string” to the array @array. Then for each element in that array, it prints out the string followed by a newline (to keep it clean in the bash shell). Here's what that output would look like:

```
hello
this
is
a
string
```

4 N being the number of the element. Remember that the elements start at “0” and not “1!”

Splitting text up can make your programs much more efficient and practical. You can use the `split` function for many different applications.

Matching Text

If you want to match text, similar to the way "grep" works with Unix/GNU Linux, you can use the "=~" operator, and the same "/ /" regexp syntax. Let's take a look at how we would search each line in an array for a certain string. Let's say that string is "apple."

```
#!/usr/bin/perl
@array = ("applebutter", "applecore", "application", "applicant", "apple");
foreach (@array) { if ($_ =~ /apple/) { print "$_ \n"; } }
exit;
```

This example is a little more advanced, as I have nested an `if` statement into a `foreach` loop. But you can easily see how the "=~" operator tells Perl to search for the string inside of the "/ /" This program will only print out the words that contain the string "apple." And by string I mean, not the word "apple." The way grep and the "=~" operator work is by matching each letter, letter by letter. Perl has no idea that there's a real word "apple" or that it even has any meaning as a noun. Computer's work in this way for almost every application you use.

Here is another useful example that uses a users input.

```
#!/usr/bin/perl
print “give me a word “;$word = <STDIN>; chomp $word;
if ($word =~ /apple/) { print "Apples are great\n"; } }
exit;
```

This will take the users input and do something useful with it. You can modify this code (mainly put whatever you want into the "block" part (curly braces)) to do anything you want now!

Part 9. GUI applications

Simple Example

Perl uses a module to display graphics in your Window Manager or GUI (graphical User Interface). A Perl "module" is an add-on piece of code that allows Perl to behave in ways that it can't by itself. There are modules for interacting with the network layer TCP/IP, hashing passwords (unix style) with the MD5 algorithm, using threads (or multiple child processes in one application), interacting with network – and other hardware, and much more. Usually, if you can think of something you want Perl to do that it doesn't in it's minimal form, you can do it with a Perl module.

Modules can be found at CPAN, and can be installed using the `cpan` program via the command line. Some package managers, such as aptitude for Debian and ports for FreeBSD, will allow you to install the modules as pre-compiled packages. This is very useful when you get an error returned from the `cpan` program.

A module that Perl can use for a GUI application is "Tk." Tk is a widget based GUI developing environment. This means that it can be used to create buttons, progress bars, labels, frames, menus, and message boxes that look "native" to your current operating system. To use this module, or any other modules for that matter you simple declare it at the beginning (or before you start referencing functions that are intrinsic to the module) of the Perl program. To declare them you simply use the following syntax:

```
use <Perl modulename>;
```

If that module has a second name with two colons between it, it must be declared in full such as in the case of `LWP::Simple`, an HTTP based module for Perl:

```
use LWP::Simple;
```

Perl modules can be thought of as sets of functions. The syntax for the functions are all pretty much the same. You can simply check the CPAN page for any module if you forget the syntax. Perl shows the graphics you specify using these functions, and they are similar to other functions but with a change of syntax in the actual value you pass to the function. For instance, now you can pass several expressions to the function in one value, and you use different symbols and commas to separate them.

Also, there are two new functions you should become familiar with, `grid` and `pack`. These functions govern how the widgets in the window you create are arranged. In my examples I will simply stick with the `grid` function.

Now instead of explaining everything (that's what books are for) I will simply give you example

code, and go over how the code acts withing Perl, and why the outcome is what it is. The following program will display a message box with one button. The button, when pressed, will exit or close the window.

```
#!/usr/bin/perl
use Tk;
$mw = MainWindow->new(-title=>"Test application");
$mw->geometry("300x100");
$mw->Label(-text=>"My Program")->grid(-row=>"1");
$mw->Button(-text=>"Exit", -command=>sub{exit} )->grid(-row=>"2");
MainLoop;
```

The output of this program looks something like this:

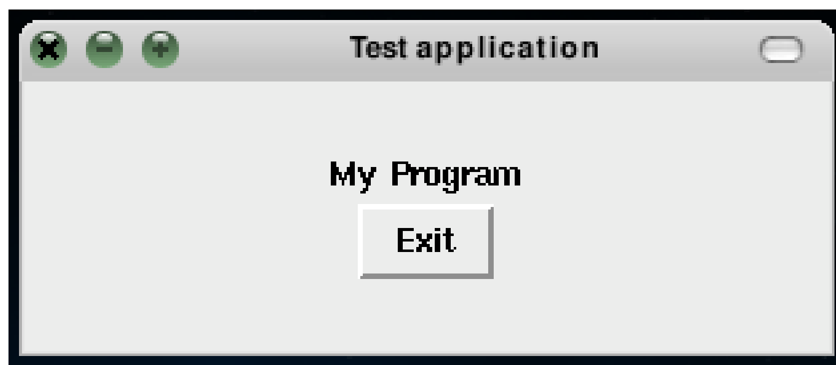

figure 0: the simplest Perl TK GUI program.

This program is the simplest Perl Tk GUI program. It has only one function for IO and that is to exit once the button is pressed.

Let's take a closer look at the program line by line. The first line is the path to the interpreter, as we've covered, tells the shell we are using to run Perl with the remaining lines of code (or LOC). The second line declares the use of the `Tk` module. The third starts the variable $mw which is our whole window or "MainWindow." Then you see new syntax using arrows or "`->`" to tell the function `MainWindow` to create a "new" window. The `new` function then has values passed to it "(<inside here>)" which is, once more, new syntax. Within the value passed to `new`, I declare a window title with a starting hyphen "-," a larger arrow "=>," then the title within quotes. This syntax has to be perfect for this function `new` (which is a child function of the Tk Module) to complete. Next, line four, declares a window size in pixels. Again, the arrow simply means to call the function `geometry` to the variable "$mw." The fifth line puts text in the new window, "My Program" using the `Label` function called for, once again, our main window "$mw." There's something new on this line, the `grid` formatting function. This function arranges the items in the main window in a matrix like style. It uses rows and columns. I could have easily called the "-columns=>'n'" value and put the `Label` "My Program" into another column. But, here I simply declare the "row" value, once again with the syntax being:

1. A starting hyphen.
2. The name of the function.
3. A large arrow
4. And the value within quotes.

The sixth line calls the `Tk` function `Button` which creates a native looking button. This button is given a few values itself using the above syntax. I pass the `exit` function to Perl, which causes it to exit, whenever the button is pressed. You can see the value (as a function) command is passed to Button. Everything is quite nested nicely now. It may be a bit confusing at first, but you will soon get the hang of it. I coded that above by memory. Meaning, by rote memorization by typing it out so many times I got a good general feel for how the `Tk` module works. Soon after using more and more Perl modules I got even a better feel as to how they all have a general syntax and how they all can be used in almost the same way!

The last line calls the function `MainLoop` which tells Perl you are finished with declaring functions and putting items into the main window. And that's it. That's a GUI application in seven lines of code!

Images

If you want to display images in your GUI applications you can use `Tk::PNG`; or any other Perl modules and the same syntax above to display the image. Here is a quick example of how to show an image “perlfoo.png:”

```
#!/usr/bin/perl
use Tk;
use Tk::PNG;
$mw = MainWindow->new(-title=>”image!”);
$mw->geometry(“500x400+200+100”);
$image = $mw->Photo(-file=>“/path/to/perlfoo.png”);
$mw->Label(-image=>”$image”)->grid();
MainLoop;
```

This will display the image perlfoo.png. Let's go over this example quickly line by line:

- Line 1: Path to interpreter.
- Line 2: Tell Perl to use the Tk module.
- Line 3: Tell Perl to also use the Tk::PNG png extension module.
- Line 4: Declare our new window and give the window a title in the title bar.
- Line 5: Declare the geometry of the new window.[5]
- Line 6: Declare our image. This is the first use of the Tk::PNG module here with the `Photo` function.
- Line 7: Put the image into a "Label" (show it in the window somewhere) and use the `grid` format function.
- Line 8: Finish our window.

This may be a bit daunting to look at the first time. But this too, was written by memory (tested on a Debian system) as I was writing this paper. What I mean is, after writing code in Perl over and over and learning new things along the way, you will start to just remember how this is all written, just like you would with any other language. I have had a 6 month period were I didn't write a single line of Perl and was able to write a text based application without using any references before. Give it time, have patience, and most importantly of all, be positive!

5 Notice something new here? The addition symbol and two new values! These values tell Perl where on your screen to put the window. This syntax says "500 pixels by 400 pixels in size, and at 200 pixels horizontal, and at 100 pixels vertical. This syntax is easy and this can be helpful when dealing with nested windows from the same application.

Part 10. Conclusion

Perl

Perl has tons of cool functions! `chown()` and `chmod()`, `gethostbyname()`, `length()`, `int()`, `rand()`, `sort()`, `system()`, `sleep()`, and many, many others. The list would just go on and on. Syntax must be perfect when using these functions and how they are used. I can't imagine remembering all of the different ways to use them, so sometimes I check on-line for help! To find out more about these extra functions, it may be handy to pickup a big Perl reference guide.

As stated, Perl is lightweight, easy to understand, not compiled (source is easy to edit on the fly), and is over all my favorite programming language. It does, however, have it's cons as well. Rather than ramble them off, I'd like to leave it up to you as the student to find them. This is, by no means, a replacement to a "*learn Perl in 5 hours!*" or "*Perl by examples*" book. If you want to further learn Perl it would be a good idea to get a Perl book. A few books I felt were helpful were *Perl for Dummies,* and *Learning Perl, Mastering Perl,* and even *Mastering Regular Expressions*.

When coding, if you need a quick reference to syntax, or syntax to a certain Perl module, you can check out CPAN, Perl Monks, or try help channels on IRC. Google is another great help, just remember to surround your search strings with quotes to find better results!

Debugging

Debugging drives me nuts sometimes, and it could be seriously frustrating to anyone new to programming. I found that about %20 of the time I use to write a program is actually writing that program. The other %80 is spent debugging and cleaning up output. After a while you will start to understand what to look for by Perl's output.

When coding a very large script I find it better to dedicate a continuous amount of time from the beginning of the program to end. Distractions only seem to take more time because I have to read through the code once more, each time. It's good to add comments in large programs too, to help remind you what's going on. It also helps other people to help you, or to improve your code. Comments begin with a "#" sign and Perl ignores everything after them on that line.

Illegal Names

Perl has many special characters, and many rules about how to name variables. The best way to not come into contact with an error returned from using a special name or character in the name of an array, variable, or whatever, is to just keep the names simple. Just use letters and numbers until you have totally become familiar with Perl's syntax.

Well, that about wraps up all I can think of for someone new to the Perl programming language.

I hope this paper helps someone, or inspires someone to start coding right away!

References:

John the Ripper:
http://www.openwall.com/john/
Google:
http://www.google.com
CPAN:
http://www.cpan.org/
Perl Monks:
http://www.perlmonks.org/
Perldoc
http://perldoc.perl.org/

Book References:

O'Reilly:
http://oreilly.com/pub/topic/perl
Dummies Guide:
http://www.dummies.com/store/product/Perl-For-Dummies-4th-Edition.productCd-0764537504.html