What is Perl?

Perl is the acronym for Practical Extraction and Reporting Language developed by Larry Wall as a tool for writing programs in the UNIX environment. Perl is an interpreted language which makes Perl suitable for producing quick solutions to programming problems. Perl is as powerful and flexible as a high level programming language such as C. Perl has a lot of features similar to C, which makes it easier to learn Perl.

This tutorial is directed towards those whose who want to get a headstart in learning Perl. The strategy of this tutorial is familiarize the individual with basic syntax, organization and programming constructs of the Perl language. This tutorial will focus on the topic of the powerful pattern matching provided by Perl, which is one of the most important features of Perl. Familiarity with basic programming ideas is assumed. The tutorial will start by describing the nuances of setting up and running the Perl program in the UNIX environment, and then will go on to describe the essential features of the Perl programming language. This tutorial is very simple in nature and should no way be used as a complete reference guide.

Requirements

No prior experience of Perl programming is required.

Basic knowledge in C/C++, VB or any other language will be helpful.

Basic knowledge of UNIX is assumed.

Further Reading

There are a lot of good references for Perl. The O'Reilly series have a comprehensive collection of books for Perl not only for advanced programmers but also for beginners. Two of my favorite books from that collection are:

Perl in a Nutshell
Programming Perl

More information on O'Reilly series Perl books can be found at http://perl.oreilly.com/

References

Schwartz & Tom Christiansen Foreword by Larry Wall 2nd Edition July 1997
Online Sources:
  http://docs.rinet.ru:8083/P7/ch7.htm
  http://www.perl.com
  http://www.wdvil.com/Authoring/Languages/Perl/PerlfortheWeb/perlintro2_table1.html
  http://www.cs.cf.ac.uk/Dave/PERL/node50.html
A Simple Perl Program

The following program demonstrates the very basics of writing and running a program written in Perl.

```
#!/usr/local/bin/perl
# Program that prints "Hello World"
print 'Hello world.);
```

Line 1: Tells UNIX to run the file through perl interpreter.
Line 2: Is a comment line.
Line 3: Prints 'Hello World'

Compiling And Running Program

If the program that you have written is saved in a file named "MyProg", then to tell the system that the file contains executable statements, enter the following command at the command interpretor.

```
> chmod +x MyProg
```

To run this program, just enter the following command.

```
> My Prog
```

Variables

Variables in Perl can be divided into two basic categories, scalar variables and array variables. We will first look at scalar variables.

**Scalar Variables:**

Scalar variables are the most primitive kind of variables provided by Perl. A scalar variable is designed to store a single value of any standard Perl type: a character, a number, string or a reference. The scalar variable is represented by an appropriate name preceded by a "$" sign. The length of variable name is restricted to 251 characters and can be composed of both lower and upper case alphabets, digits and underscores. The variable name cannot start with a number. Perl variables are case sensitive.

Examples:

```
$ measure = "miles";
$ number_of_months = 12;
$ short_side = 5;
$ long_side = 12;
$ hypotenuse = sqrt (($short_side)^2 + ($long_side)^2))
```

Another interesting aspect about Perl is the fact that strings and numbers are interchangeable. Numbers can be assigned to scalar variables in the form of strings.
Example:

```
$ short_side = 5;       # sets the variable $short_side = 9
$ short_side = 'sideA'; # same variable name can be assigned to a string.
$ short_side = '5';     # legal expression.
```

**Array Variables:**

An array variable stores a series of values, with each value uniquely identified by an integer index. The contents of the array can be accessed by giving the array's name, preceded by the "@" symbol. Each array element can be accessed by the name of the array preceded by the "$" sign, followed by the index.

Examples:

```
@Philosophers = ("Hume", "Descartes", Kant", "Marx");
$Philosophers[0] = "Hume";    # note that the first array index is 0
```

Elements can be individually assigned, and will be automatically created if they do not really exist.

Example:
```
$Philosophers[4] = "Socrates"; # this assignment causes the array to go automatically.
```

The variable $#Philosophers stores the last index of the array. It is always one less than the length of the array as the array indexing starts at 0. Therefore,

```
$#Philosophers = 3;
$ Length = @Philosophers;
```

Note: when the whole array is referred in a scalar context, the resulting scalar contains the length of the array. In this case, $Length = 4;

**Array Operations**

**Arrays as Stacks and Queues:**

Perl provides functions that allows the use of arrays to implement stacks and queues via the built-in functions push, pop, shift and unshift. All the four functions are described in detail below.

**push:** This function appends an element to the end of the array.

Example:
```
@colors = ("red", "blue", "yellow", "green");
push ( @colors, "white");
@colors = ("red", "blue", "yellow", "green", "white");
```

**pop:** This function removes the last element of the array and returns the element.
Example:

```perl
$color = pop(@colors); # $color = "white"
@colors = ("red", "blue", "yellow", "green");
```

**unshift**: This function adds an element to the beginning of the array.

Example:

```perl
unshift( @colors, "black");
@colors = ("black", "red", "blue", "yellow", "green");
```

**shift**: This function removes and returns the first element from the array

Example:

```perl
$color = shift(@colors); # $color = "black"
@colors = ("red", "blue", "yellow", "green");
```

**The splice Function**: This function is a more general function that changes the elements of an array. The function takes four arguments:

1. The array to be modified.
2. The index at which it's to be modified.
3. The number of elements to be removed (starting at the index specified in the previous argument).
4. A list of new elements to be added in place of the elements removed.

Example:

```perl
@favorite_colors = ("magenta", "lilac");
@removed_colors = splice (@colors, 1, 2, @favorite_colors);
```

Now,

```perl
@colors = ("red", "magenta", "lilac", "green");
@removed_colors = ("blue", "yellow"); #@removed_colors array stores the elem. replaced by splice
```

---

**Control Structures**

Perl supports various kinds of control structures which are very similar to C like for, while, do while, etc. We will discuss the control structures which are not included in C.

**The Foreach Statement**: is used to go through each element of an array. It is a simpler and a more elegant form of the "for" statement.

Example:

```perl
@colors = ("red", "blue", "green", "yellow");
foreach $color( @colors){
    if ($ words eq "red"){
        print ("Color is red \n");
    }
}
```
**The Until Statement:** The until statement loops until its conditional expression is true (i.e., loops as long as its conditional expression is false).

Example:

```perl
print("Enter number of days in a leap year
n");
$NumberOfDays = <STDIN>;
chop($NumberOfDays); # the chop function removes the last char of a string (in this case, newline)

until ($NumberOfDays == 365){
    print("Sorry, wrong answer . Try again!!!
");
    $NumberOfDays = <STDIN>;
    chop($NumberOfDays);
}
print("Success !! Correct Answer");
```

The for, while and do while statements in Perl work exactly the same as they do in C/C++.

**Associative Arrays**

**Representing an Associative Array:**

Associative Arrays are also called hashes. They can be viewed as a table with two columns, the left column stores a key and the right column stores the associated values. A hashing algorithm is used to map each key string to an internal index into the table. A hash variable is prefixed with the percentage "%" sign.

Example:

```perl
%offspring = ("cat","kitten","dog","puppy","lion","cub");
```

The offspring list contains an alternate sequence of keys and their associated values. In this case, "cat" is associated with the value "kitten", "dog" is associated with "puppy" and "lion" with "cub". The manner in which the hash is represented is not very intuitive. Another way of representing a hash would be to use the "=>" operator to show the associativity in a better fashion.

Example:

```perl
%offspring = ("cat"=>"kitten","dog"=>"puppy","lion"=>"cub");
```

This way of representing a hash makes it easier to see exactly which values are associated to each key.

**Accessing Entries of an Associative Array:**

Each value of an associative array is called an entry. The individual entries are accessed using the following syntax.

```perl
print "The offspring of a cat is:", $offspring {"cat"};
```

Note: The key is specified in curly braces.
We can also access the value associated with the key in the following manner:

```perl
$adult = "lion";
print "The offspring of a lion is:" , $offspring{$adult};
```

**Iterating an Associative Array:**

Since, the entries in a hash or associative array are not ordered by their sequential indices like ordinary arrays, iterating through hashes is much more challenging. Luckily, Perl provides three built in functions that do the job very elegantly. We will discuss each of the three built in functions in detail.

**The "keys" Function:** The "keys" function takes a hash as its argument and returns a list of its keys in an apparently random order.

Example:

```perl
foreach $key (keys %offspring)
{
    # prints the keys and then prints the values accessed using the keys in $offspring{$key}
    print "The $key has an offspring called $offspring{$key}\n";
}
```

**The "values" Function:** The "values" function takes a hash as its argument and returns a list of its values in an apparently random order.

Example:

```perl
foreach $val (values %offspring)
{
    print "$val , \n";           # prints only the values without the keys.
}
```

**The "each " Function:** The "each" function takes a hash as its argument and returns one distinct key from the hash every times it's called. When successive calls to each have returned every possible key from the has, the next call returns `undef`, after which the cycle repeats.

Example:

```perl
while (defined ($nextkey = each %offspring))
{
    print "The key $nextkey has the value $offspring{$nextkey}\n";
}
```

**Pattern Matching**

Pattern Matching is one of the most important features of Perl. Pattern matching is often the very basis of inspiration of using Perl in programming problems. Perl provides a powerful but complex method of creating pattern matching templates, and the syntax used is known as regular expression syntax. Regular expression syntax is inherited from the
UNIX world, and it is not limited to Perl. Typically, regular expressions are referred as "regexps".

Let us look at some of the ways Perl does Pattern Matching.

The "=~" Operator: The syntax for using the "=~" operator is as follows:

$$\text{Variable} =~ /\text{PatternToMatch}/;

Example:
$\text{var} = ~/\text{xyz}/;

The '=~' operator returns a nonzero value if the $ var contains "xyz". If the pattern 'xyz' is not found then a value of 0 is returned. SecretCode is a simple Perl program that uses the '=~' operator.

The "+" Operator: The "+" operator matches "one or more characters preceding the '+' operator.

Example:
/\text{ab}+c/  # matches any of the fol.. strings: abc, abbc, abbbbc etc.

We can write a simple program that counts the number of words in in a line using the '+' operator. WordCount is a program that demonstrates the use of the "+" operator.

The "*" & "?" Operator: The " * " operator matches zero or more occurrences of the preceding character. The "?" operator matches zero or one occurrence of the preceding character.

Example:
/\text{ab}*c/  # matches ac, abc, abbc, abbbbc.
/\text{ab}?c/  # matches either ac or abc

Note: It doe not match abbc etc. as the "?" operator matches only 0 or 1 occurrence of the preceding char.

Escape Sequences: As we saw earlier, "*", "?", ",/" have special meaning in pattern matching. How do we tell Perl to use these characters as normal characters in certain cases. To match special characters such as '/' or "" the pattern has to be preceded by a backslash character.

Example:
/\text{*}/  # This expression tells Perl to treat the asterisk as an ordinary char. Perl
# no longer treats it as a char used to match zero or more occurrences of a char.

Pattern Matching with "//":

We can match strings in Perl by placing them between two slashes.

Example:
/\text{[PatternToMatch]}/
Note: The "^" sign works as a negation operator.

The "^" & "$" Operator: The "^" ensures that the pattern is matched only at the beginning of a string. The "$" ensures that the pattern is matched only at the end of a string.

Example:
/^pre/ # matches words like predefined, preferred etc.
/$tion/ # matches words like nation, creation, celebration etc.

Note: The expression /^ABC$/ matches only the string "ABC".

The Word Boundary Pattern Anchors: The word boundary pattern \b, ensures that the pattern is at the beginning or the tend of a word. On the other hand, \B ensures that the patterns is at the ends of the word.

Example:
/\b abc/ # matches "abcdef" but not "defabc".
/abc \b/ # matches "defghi" but not "abcdef".
/abc \B/ # matches "abcdef".
/\B abc\B/ # matches "xabcd" but not "abc".

Perl provides very powerful ways to perform pattern matching. It is not always easy to remember the action of each of the special character. Therefore a table which summarizes, the basic regular expression syntax is very helpful. The Basic Regular Syntax Table should be printed out and used as a quick reference especially for beginners.

Now that we have seen the basics of pattern matching, we should utilize its power to write a small utility program which uses the above constructs. ValidEmail is a simple perl program that checks to see if the user has entered a valid email address. The definition of a valid email address in this context is an address that ends with either a ".edu" or ".org". No other extensions are to be accepted.

Subroutines

Subroutines: Like any other programming language, Perl allows the programmers to define their own subroutines. A subroutine is a small, user-defined subprogram. Subroutines are defined by using the "sub" keyword, followed by the subroutine code in curly braces. A subroutine may return a scalar or a list value.

Example:

sub HelloWorld #This program prints out the famous "HelloWorld"
{
    print "Hello World !! \n";
}

Calling Subroutines: The subroutines can be called in the following ways.
Example:

&HelloWorld;  # call the function without any parameters
&HelloWorld ($_);  # call the function with one parameter.

**Passing Parameters:** The arguments passed to the subroutine can be accessed via the special "@_" array. In Perl, return values are built in. It is optional to specify a return value. If none is specified, the subroutine automatically returns the value of the last statement it actually executed.

Example:

```perl
sub PrintArguments {
    print "@\n";
}
```

We can call the function in the following manner:

&PrintArguments ("Hello", "World");  # prints "Hello World"
&PrintArguments ("Hello", "World", "!!!");  # prints "Hello World !!!"

Individual arguments can be accessed in the following manner:

Example:

```perl
sub PrintArguments {
    print "The first argument =>, \n";
    print "The second argument =>, \n";
}
```

&PrintArguments ("Hello", "World");

The call to the function has the following output:
The first argument =>, Hello
The second argument =>, World.