Essential Perl

This document is a quick introduction to the Perl language. Perl has many features, but you can get pretty far with just the basics, and that's what this document is about. The coverage is pretty quick, intended for people with some programming experience. This document is available for free in the spirit of engineering goodwill -- that success is not taken, it is earned.

Stanford CS Education #108
http://cslibrary.stanford.edu/108/

Revised 5/2002

Contents

1. What Is Perl?

Perl is a free, open source programming language created by Larry Wall. Perl aims for adjectives like "practical" and "quick" and not so much words like "structured" or "elegant". A culture has built up around Perl where people create and give away modules, documentation, sample code, and a thousand other useful things -- visit the Comprehensive Perl Archive Network (CPAN), http://www.cpan.org/, or http://www.perl.com/ to see the amazing range of Perl material available.

Perl Niche

Perl is probably best known for text processing -- dealing with files, strings, and regular expressions. However, Perl's quick, informal style makes it attractive for all sorts of little programs. If I need a 23 line program to get some task done, I can write it in Perl and be done in 3 minutes. Perl code is very portable -- I frequently move Perl programs back and forth from the Mac to various Unixes and it just works. With Perl, you are not locked in to any particular vendor or operating system. Perl code is also robust; Perl programs can have bugs, but they will not crash randomly like C or C++ programs. On the other hand, in my opinion, Perl's easy-going style makes it less appealing for large projects where I would rather use Java.

Warning: My Boring Perl Style

Perl is famous for allowing you to write solutions to complex problems with very short, terse phrases of code. There's something satisfying about reducing a whole computation down to a single line of dense code. However, I never do that. I write Perl code in a boring, straightforward way which tends to spell out what it's actually doing step by step. The terse style is mentioned briefly in the Terse Perl section. Also, in versions 5 and 6, Perl has accumulated more sophisticated features which are not covered here. We just do simple old Perl code.

Running Perl

A Perl program is just a text file. You edit the text of your Perl program, and the Perl interpreter reads that text file directly to "run" it. This structure makes your edit-run-debug cycle nice and fast. On Unix, the Perl interpreter is called "perl" and you run a Perl program by running the Perl interpreter and telling it which file contains your Perl program...

```
> perl myprog.pl
```

The interpreter makes one pass of the file to analyze it and if there are no syntax or other obvious errors, the interpreter runs the Perl code. There is no "main" function -- the interpreter just executes the statements in the file starting at the top.

Following the Unix convention, the very first line in a Perl file usually looks like this...

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

This special line is a hint to Unix to use the Perl interpreter to execute the code in this file. The "-w" switch turns on warnings which is generally a good idea. In unix, use "chmod" to set the execute bit on a Perl file so it can be run right from the prompt...

```
> chmod u+x foo.pl  ## set the "execute" bit for the file once
>
> foo.pl     ## automatically uses the perl interpreter to "run" this file
```

The second line in a Perl file is usually a "require" declaration that specifies what version of Perl the program expects...

```
#!/usr/bin/perl -w
require 5.004;
```

Perl is available for every operating system imaginable, including of course Windows and MacOS, and it's part of the default install in Mac OSX. See the "ports" section of http://www.cpan.org/ to get Perl for a particular system.

2. Syntax And Variables

The simplest Perl variables are "scalar" variables which hold a single string or number. Scalar variable names begin with a dollar sign ($) such as `$sum` or `$greeting`. Scalar and other variables do

not need to be pre-declared -- using a variable automatically declares it as a global variable. Variable names and other identifiers are composed of letters, digits, and underscores (_) and are case sensitive. Comments begin with a "#" and extend to the end of the line.

```
$x = 2; ## scalar var $x set to the number 2
$greeting = "hello"; ## scalar var $greeting set to the string "hello"
```

A variable that has not been given a value has the special value "undef" which can be detected using the "defined" operator. Undef looks like 0 when used as a number, or the empty string "" when used as a string, although a well written program probably should not depend on undef in that way. When Perl is run with "warnings" enabled (the -w flag), using an undef variable prints a warning.

```
if (!defined($binky)) {
  print "the variable 'binky' has not been given a value!\n";
}
```

What's With This '$' Stuff?

Larry Wall, Perl's creator, has a background in linguistics which explains a few things about Perl. I saw a Larry Wall talk where he gave a sort of explanation for the '$' syntax in Perl: In human languages, it's intuitive for each part of speech to have its own sound pattern. So for example, a baby might learn that English nouns end in "-y" -- "mommy," "daddy," "doggy". (It's natural for a baby to over generalize the "rule" to get made up words like "bikey" and "blanky".) In some small way, Perl tries to capture the different-signature-for-different-role pattern in its syntax -- all scalar expressions look alike since they all start with '$'.

3. Strings

Strings constants are enclosed within double quotes (") or in single quotes ('). Strings in double quotes are treated specially -- special directives like `\n` (newline) and `\x20` (hex 20) are expanded. More importantly, a variable, such as $x, inside a double quoted string is evaluated at run-time and the result is pasted into the string. This evaluation of variables into strings is called "interpolation" and it's a great Perl feature. Single quoted (') strings suppress all the special evaluation -- they do not evaluate \n or $x, and they may contain newlines.

```
$fname = "binky.txt";
$a = "Could not open the file $fname.";  ## $fname evaluated and pasted in -- neato!
$b = 'Could not open the file $fname.';  ## single quotes (') do no special evaluation

## $a is now "Could not open the file binky.txt."
## $b is now "Could not open the file $fname."
```

The characters '$' and '@' are used to trigger interpolation into strings, so those characters need to be escaped with a backslash (\) if you want them in a string. For example: `"nick\@stanford.edu found \$1"`.

The dot operator (.) concatenates two strings. If Perl has a number or other type when it wants a string, it just silently converts the value to a string and continues. It works the other way too -- a

string such as "42" will evaluate to the integer 42 in an integer context.

```
$num = 42;
$string = "The " . $num . " ultimate" . " answer";

## $string is now "The 42 ultimate answer"
```

The operators `eq` (equal) and `ne` (not equal) compare two strings. Do not use `==` to compare strings; use `==` to compare numbers.

```
$string = "hello";
($string eq ("hell" . "o"))  ==> TRUE
($string eq "HELLO")  ==> FALSE

$num = 42;
($num-2 == 40)  ==> TRUE
```

The lc("Hello") operator returns the all lower-case version "hello", and uc("Hello") returns the all upper-case version "HELLO".

Fast And Loose Perl

When Perl sees an expression that doesn't make sense, such as a variable that has not been given a value, it tends to just silently pass over the problem and use some default value such as undef. This is better than C or C++ which tend to crash when you do something wrong. Still, you need to be careful with Perl code since it's easy for the language to do something you did not have in mind. Just because Perl code compiles, don't assume it's doing what you intended. Anything compiles in Perl.

4. Arrays -- @

Array constants are specified using parenthesis ( ) and the elements are separated with commas. Perl arrays are like lists or collections in other languages since they can grow and shrink, but in Perl they are just called "arrays". Array variable names begin with the at-sign (@). Unlike C, the assignment operator (=) works for arrays -- an independent copy of the array and its elements is made. Arrays may not contain other arrays as elements. Perl has sort of a "1-deep" mentality. Actually, it's possible to get around the 1-deep constraint using "references", but it's no fun. Arrays work best if they just contain scalars (strings and numbers). The elements in an array do not all need to be the same type.

```
@array = (1, 2, "hello");  ## a 3 element array
@empty = ();               ## the array with 0 elements

$x = 1;
$y = 2;
@nums = ($x + $y, $x - $y);

## @nums is now (3, -1)
```

Just as in C, square brackets [ ] are used to refer to elements, so $a[6] is the element at index 6 in the array @a. As in C, array indexes start at 0. Notice that the syntax to access an element begins with '$' not '@' -- use '@' only when referring to the whole array (remember: all scalar expressions begin with $).

```
@array = (1, 2, "hello", "there");
$array[0] = $array[0] + $array[1];    ## $array[0] is now 3
```

Perl arrays are not bounds checked. If code attempts to read an element outside the array size, undef is returned. If code writes outside the array size, the array grows automatically to be big enough. Well written code probably should not rely on either of those features.

```
@array = (1, 2, "hello", "there");
$sum = $array[0] + $array[27];    ## $sum is now 1, since $array[27] returned
undef

$array[99] = "the end";           ## array grows to be size 100
```

When used in a scalar context, an array evaluates to its length. The "scalar" operator will force the evaluation of something in a scalar context, so you can use scalar() to get the length of an array. As an alternative to using scalar, the expression `$#array` is the index of the last element of the array which is always one less than the length.

```
@array = (1, 2, "hello", "there");
$len = @array;                ## $len is now 4 (the length of @array)

$len = scalar(@array)         ## same as above, since $len represented a scalar
                              ## context anyway, but this is more explicit

@letters = ("a", "b", "c");
$i = $#letters;               ## $i is now 2
```

That `scalar(@array)` is the way to refer to the length of an array is not a great moment in the history of readable code. At least I haven't showed you the even more vulgar forms such as `(0 + @a)`.

The sort operator `(sort @a)` returns a copy of the array sorted in ascending alphabetic order. Note that sort does not change the original array. Here are some common ways to sort...

```
(sort @array)                         ## sort alphabetically, with uppercase first

(sort {$a <=> $b} @array)             ## sort numerically
(sort {$b cmp $a} @array)             ## sort reverse alphabetically
(sort {lc($a) cmp lc($b)} @array)     ## sort alphabetically, ignoring case
(somewhat inefficient)
```

The sort expression above pass a comparator function `{...}` to the sort operator, where the special variables `$a` and `$b` are the two elements to compare -- `cmp` is the built-in string compare, and `<=>` is the built-in numeric compare.

There's a variant of array assignment that is used sometimes to assign several variables at once. If an array on the left hand side of an assignment operation contains the names of variables, the variables are assigned the corresponding values from the right hand side.

```
($x, $y, $z) = (1, 2, "hello", 4);
```

```
## assigns $x=1, $y=2, $z="hello", and the 4 is discarded
```

This type of assignment only works with scalars. If one of the values is an array, the wrong thing happens (see "flattening" below).

Array Add/Remove/Splice Functions

These handy operators will add or remove an element from an array. These operators change the array they operate on...

- Operating at the "front" ($array[0]) end of the array...
  - `shift(array)` -- returns the frontmost element and removes it from the array. Can be used in a loop to gradually remove and examine all the elements in an array left to right. The foreach operator, below, is another way to examine all the elements.
  - `unshift(array, elem)` -- inserts an element at the front of the array. Opposite of shift.
- Operating at the "back" ($array[$len-1]) end of the array...
  - `pop(array)` -- returns the endmost element (right hand side) and removes it from the array.
  - `push(array, elem)` -- adds a single element to the end of the array. Opposite of pop.
- `splice(array, index, length, array2)` -- removes the section of the array defined by index and length, and replaces that section with the elements from array2. If array2 is omitted, splice() simply deletes. For example, to delete the element at index $i from an array, use splice (@array, $i, 1).

5. Hash Arrays -- %

Hash arrays, also known as "associative" arrays, are a built-in key/value data structure. Hash arrays are optimized to find the value for a key very quickly. Hash array variables begin with a percent sign (%) and use curly braces { } to access the value for a particular key. If there is no such key in the array, the value returned is undef. The keys are case sensitive, so you may want to consistently uppercase or lowercase strings before using them as a key (use lc and uc).

```
$dict{"bart"}  = "I didn't do it";
$dict{"homer"} = "D'Oh";
$dict{"lisa"}  = "";

## %dict now contains the key/value pairs (("bart" => "I didn't do it"),
## ("homer" => "D'oh"), ("lisa" => ""))

$string = $dict{"bart"};     ## Lookup the key "bart" to get
                             ## the value "I didn't do it"

$string = $dict{"marge"};    ## Returns undef -- there is no entry for "marge"

$dict{"homer"} = "Mmmm, scalars";    ## change the value for the key
                                     ## "homer" to "Mmmm, scalars"
```

A hash array may be converted back and forth to an array where each key is immediately followed by its value. Each key is adjacent to its value, but the order of the key/value pairs depends on the

hashing of the keys and so appears random. The "keys" operator returns an array of the keys from an associative array. The "values" operator returns an array of all the values, in an order consistent with the keys operator.

```
@array = %dict;

## @array will look something like
##  ("homer", "D'oh", "lisa", "", "bart", "I didn't do it");
##
## (keys %dict) looks like ("homer", "lisa, "bart")
## or use (sort (keys %dict))
```

You can use => instead of comma and so write a hash array value this cute way...

```
%dict = (
  "bart" => "I didn't do it",
  "homer" => "D'Oh",
  "lisa" => "",
);
```

In Java or C you might create an object or struct to gather a few items together. In Perl you might just throw those things together in a hash array.

@ARGV and %ENV

The built-in array @ARGV contains the command line arguments for a Perl program. The following run of the Perl program critic.pl will have the ARGV array ("-poetry", "poem.txt").

```
unix% perl critic.pl -poetry poem.txt
```

%ENV contains the environment variables of the context that launched the Perl program. @ARGV and %ENV make the most sense in a Unix environment.

6. If/While Syntax

Perl's control syntax looks like C's control syntax . Blocks of statements are surrounded by curly braces { }. Statements are terminated with semicolons (;). The parenthesis and curly braces are **required** in if/while/for forms. There is not a distinct "boolean" type, and there are no "true" or "false" keywords in the language. Instead, the empty string, the empty array, the number 0 and undef all evaluate to false, and everything else is true. The logical operators &&, ||, ! work as in C. There are also keyword equivalents (and, or, not) which are almost the same, but have lower precedence.

IF

```
if (expr) {          ## basic if -- ( ) and { } required
  stmt;
  stmt;
```

```
}

if (expr) {          ## if + elsif + else
  stmt;
  stmt;
}
elsif (expr) {       ## note the strange spelling of "elsif"
  stmt;
  stmt;
}
else {
  stmt;
  stmt;
}

unless (expr) {      ## if variant which negates the boolean test
  stmt;
  stmt;
}
```

If Variants

As an alternative to the classic `if() { }` structure, you may use `if`, `while`, and `unless` as modifiers that come **after** the single statement they control...

```
$x = 3 if $x > 3;  ## equivalent to: if ($x > 3) { $x = 3; }

$x = 3 unless $x <= 3;
```

For these constructs, the parentheses are not required around the boolean expression. This may be another case where Perl is using a structure from human languages. I never use this syntax because I just cannot get used to seeing the condition after the statement it modifies. If you were defusing a bomb, would you like instructions like this: "Locate the red wire coming out of the control block and cut it. Unless it's a weekday -- in that case cut the black wire."

Loops

These work just as in C...

```
while (expr) {
 stmt;
 stmt;
}

for (init_expr; test_expr; increment_expr) {
 stmt;
 stmt;
}

## typical for loop to count 0..99
```

```
for ($i=0; $i<100; $i++) {
 stmt;
 stmt;
}
```

The "next" operator forces the loop to the next iteration. The "last" operator breaks out of the loop like break in C. This is one case where Perl (last) does not use the same keyword name as C (break).

Array Iteration — foreach

The "foreach" construct is a handy way to iterate a variable over all the elements in an array. Because of foreach, you rarely need to write a traditional for or while loop to index into an array. Foreach is also likely to be implemented efficiently. (It's a shame Java does not include a compact iteration syntax in the language. It would make Java a better language at the cost of some design elegance.)

```
foreach $var (@array) {
  stmt;    ## use $var in here
  stmt;
}
```

Any array expression may be used in the foreach. The array expression is evaluated once before the loop starts. The iterating variable, such as $var, is actually a pointer to each element in the array, so assigning to $var will actually change the elements in the array.

7. File Input

Variables which represent files are called "file handles", and they are handled differently from other variables. They do not begin with any special character -- they are just plain words. By convention, file handle variables are written in all upper case, like FILE_OUT or SOCK. The file handles are all in a global namespace, so you cannot allocate them locally like other variables. File handles can be passed from one routine to another like strings (detailed below).

The standard file handles STDIN, STDOUT, and STDERR are automatically opened before the program runs. Surrounding a file handle with <> is an expression that returns one line from the file including the "\n" character, so <STDIN> returns one line from standard input. The <> operator returns undef when there is no more input. The "chop" operator removes the last character from a string, so it can be used just after an input operation to remove the trailing "\n". The "chomp" operator is similar, but only removes the character if it is the end-of-line character.

```
$line = <STDIN>;   ## read one line from the STDIN file handle
chomp($line);      ## remove the trailing "\n" if present

$line2 = <FILE2>;  ## read one line from the FILE2 file handle
                   ## which must be have been opened previously
```

Since the input operator returns undef at the end of the file, the standard pattern to read all the lines in a file is...

```
## read every line of a file
while ($line = <STDIN>) {
  ## do something with $line
}
```

Open and Close

The "open" and "close" operators operate as in C to connect a file handle to a filename in the file system.

```
open(F1, "filename");       ## open "filename" for reading as file handle F1
open(F2, ">filename");      ## open "filename" for writing as file handle F2
open(F3, ">>appendtome")    ## open "appendtome" for appending

close(F1);                  ## close a file handle
```

Open can also be used to establish a reading or writing connection to a separate process launched by the OS. This works best on Unix.

```
open(F4, "ls -l |");        ## open a pipe to read from an ls process
open(F5, "| mail $addr");   ## open a pipe to write to a mail process
```

Passing commands to the shell to launch an OS process in this way can be very convenient, but it's also a famous source of security problems in CGI programs. When writing a CGI, do not pass a string from the client side as a filename in a call to open().

Open returns undef on failure, so the following phrase is often to exit if a file can't be opened. The die operator prints an error message and terminates the program.

```
open(FILE, $fname) || die "Could not open $fname\n";
```

In this example, the logical-or operator `||` essentially builds an if statement, since it only evaluates the second expression if the first if false. This construct is a little strange, but it is a common code pattern for Perl error handling.

Input Variants

In a scalar context the input operator reads one line at a time. In an array context, the input operator reads the entire file into memory as an array of its lines...

```
@a = <FILE>;  ## read the whole file in as an array of lines
```

This syntax can be dangerous. The following statement looks like it reads just a single line, but actually the left hand side is an array context, so it reads the whole file and then discards all but the first line....

```
my($line) = <FILE>;
```

The behavior of <FILE> also depends on the special global variable `$/` which is the current the end-of-line marker (usually "\n"). Setting `$/` to undef causes <FILE> to read the whole file into a single

string.

```
$/ = undef;
$all = <FILE>;    ## read the whole file into one string
```

You can remember that `$/` is the end-of-line marker because "/" is used to designate separate lines of poetry. I thought this mnemonic was silly when I first saw it, but sure enough, I now remember that `$/` is the end-of-line marker.

8. Print Output

Print takes a series of things to print separated by commas. By default, print writes to the STDOUT file handle.

```
print "Woo Hoo\n";   ## print a string to STDOUT

$num = 42;
$str = " Hoo";
print "Woo", $a, " bbb $num", "\n";  ## print several things
```

An optional first argument to print can specify the destination file handle. There is no comma after the file handle, but I always forget to omit it.

```
print FILE "Here", " there", " everywhere!", "\n"; ## no comma after FILE
```

File Processing Example

As an example, here's some code that opens each of the files listed in the @ARGV array, and reads in and prints out their contents to standard output...

```
#!/usr/bin/perl -w
require 5.004;
## Open each command line file and print its contents to standard out

foreach $fname (@ARGV) {

  open(FILE, $fname) || die("Could not open $fname\n");

  while($line = <FILE>) {
    print $line;
  }

  close(FILE);
}
```

The above uses "die" to abort the program if one of the files cannot be opened. We could use a more

flexible strategy where we print an error message for that file but continue to try to process the other files. Alternately we could use the function call `exit(-1)` to exit the program with an error code. Also, the following shift pattern is a common alternative way to iterate through an array...

```
while($fname = shift(@ARGV)) {...
```

10. String Processing with Regular Expressions

Perl's most famous strength is in string manipulation with regular expressions. Perl has a million string processing features -- we'll just cover the main ones here. The simple syntax to search for a pattern in a string is...

```
($string =~ /pattern/)  ## true if the pattern is found somewhere in the string

("binky" =~ /ink/)  ==> TRUE
("binky" =~ /onk/)  ==> FALSE
```

In the simplest case, the exact characters in the regular expression pattern must occur in the string somewhere. All of the characters in the pattern must be matched, but the pattern does not need to be right at the start or end of the string, and the pattern does not need to use all the characters in the string.

Character Codes

The power of regular expressions is that they can specify patterns, not just fixed characters. First, there are special matching characters...

- **a**, **X**, **9** -- ordinary characters just match that character exactly
- **.** (a period) -- matches any single character except "\n"
- **\w** -- (lowercase w) matches a "word" character: a letter or digit [a-zA-Z0-9]
- **\W** -- (uppercase W) any non word character
- **\s** -- (lowercase s) matches a single whitespace character -- space, newline, return, tab, form [\n\r\t\f]
- **\S** -- (uppercase S) any non whitespace character
- **\t**, **\n**, **\r** -- tab, newline, return
- **\d** -- decimal digit [0-9]
- **** -- inhibit the "specialness" of a character. So, for example, use **\.** to match a period or **** to match a slash. If you are unsure if a character has special meaning, such as '@', you can always put a slash in front of it \@ to make sure it is treated just as a character.

```
"piiig" =~ /p...g/     ==> TRUE   . = any char (except \n)

"piiig" =~ /.../       ==> TRUE   need not use up the whole string

"piiig" =~ /p....g/    ==> FALSE  must use up the whole pattern (the g is not
matched)

"piiig" =~ /p\w\w\wg/  ==> TRUE   \w = any letter or digit
```

```
"p123g" =~ /p\d\d\dg/  ==> TRUE   \d = 0..9 digit
```

The modifier "i" after the last / means the match should be case insensitive...

```
"PiIIg" =~ /pIiig/     ==> FALSE
"PiIIg" =~ /pIiig/i    ==> TRUE
```

String interpolation works in regular expression patterns. The variable values are pasted into the expression once before it is evaluated. Characters like * and + continue to have their special meanings in the pattern after interpolation, unless the pattern is bracketed with a \Q..\E. The following examples test if the pattern in $target occurs within brackets < > in $string...

```
$string =~ /<$target>/        ## Look for <$target>, '.' '*' keep their special
meanings in $target

$string =~ /<\Q$target\E>/    ## The \Q..\E puts a backslash in front of every
char,
                              ## so '.' '*' etc. in $target will not have their
special meanings
```

Similar to the \Q..\E form, the quotemeta() function returns a string with every character \ escaped. There is an optional "m" (for "match") that comes before the first /. If the "m" is used, then any character can be used for the delimiter instead of / -- so you could use " or # to delimit the pattern. This is handy if what you are trying to match has a lot of /'s in it. If the delimiter is the single quote (') then interpolation is suppressed. The following expressions are all equivalent...

```
"piiig" =~ m/piiig/
"piiig" =~ m"piiig"
"piiig" =~ m#piiig#
```

Control Codes

Things get really interesting when you add in control codes to the regular expression pattern...

- **?** -- match 0 or 1 occurrences of the pattern to its left
- ***** -- 0 or more occurrences of the pattern to its left
- **+** -- 1 or more occurrences of the pattern to its left
- **|** -- (vertical bar) logical or -- matches the pattern either on its left or right
- parenthesis **( )** -- group sequences of patterns
- **^** -- matches the start of the string
- **$** -- matches the end of the string

Leftmost & Largest

First, Perl tries to find the leftmost match for the pattern, and second it tries to use up as much of the string as possible -- i.e. let + and * use up as many characters as possible.

Regular Expression Examples

The following series gradually demonstrate each of the above control codes. Study them carefully -- small details in regular expressions make a big difference. That's what makes them powerful, but it

makes them tricky as well.

Old joke: What do you call a pig with three eyes? Piiig!

```
#### Search for the pattern 'iiig' in the string 'piiig'
"piiig" =~ m/iiig/ ==> TRUE

#### The pattern may be anywhere inside the string
"piiig" =~ m/iii/ ==> TRUE

#### All of the pattern must match
"piiig" =~ m/iiii/ ==> FALSE

#### . = any char but \n
"piiig" =~ m/...ig/ ==> TRUE

"piiig" =~ m/p.i../ ==> TRUE

#### The last . in the pattern is not matched
"piiig" =~ m/p.i.../ ==> FALSE

#### \d = digit [0-9]
"p123g" =~ m/p\d\d\dg/ ==> TRUE

"p123g" =~ m/p\d\d\d\d/ ==> FALSE

#### \w = letter or digit
"p123g" =~ m/\w\w\w\w\w/ ==> TRUE

#### i+ = one or more i's
"piiig" =~ m/pi+g/ ==> TRUE

#### matches iii
"piiig" =~ m/i+/ ==> TRUE

"piiig" =~ m/p+i+g+/ ==> TRUE

"piiig" =~ m/p+g+/ ==> FALSE

#### i* = zero or more i's
"piiig" =~ m/pi*g/ ==> TRUE

"piiig" =~ m/p*i*g*/ ==> TRUE

#### X* can match zero X's
"piiig" =~ m/pi*X*g/ ==> TRUE

#### ^ = start, $ = end
"piiig" =~ m/^pi+g$/ ==> TRUE

#### i is not at the start
"piiig" =~ m/^i+g$/ ==> FALSE

#### i is not at the end
"piiig" =~ m/^pi+$/ ==> FALSE

"piiig" =~ m/^p.+g$/ ==> TRUE

"piiig" =~ m/^p.+$/ ==> TRUE

"piiig" =~ m/^.+$/ ==> TRUE

#### g is not at the start
```

```
"piiig" =~ m/^g.+$/ ==> FALSE

#### Needs at least one char after the g
"piiig" =~ m/g.+/ ==> FALSE

#### Needs at least zero chars after the g
"piiig" =~ m/g.*/ ==> TRUE

#### | = left or right expression
"cat" =~ m/^(cat|hat)$/ ==> TRUE

"hat" =~ m/^(cat|hat)$/ ==> TRUE

"cathatcatcat" =~ m/^(cat|hat)+$/ ==> TRUE

"cathatcatcat" =~ m/^(c|a|t|h)+$/ ==> TRUE

"cathatcatcat" =~ m/^(c|a|t)+$/ ==> FALSE

#### Matches and stops at first 'cat'; does not get to 'catcat' on the right
"cathatcatcat" =~ m/(c|a|t)+/ ==> TRUE

#### ? = optional
"12121x2121x2" =~ m/^(1x?2)+$/ ==> TRUE

"aaaxbbbabaxbb" =~ m/^(a+x?b+)+$/ ==> TRUE

"aaaxxbbb" =~ m/^(a+x?b+)+$/ ==> FALSE

#### Three words separated by spaces
"Easy      does it" =~ m/^\w+\s+\w+\s+\w+$/ ==> TRUE

#### Just matches "gates@microsoft" -- \w does not match the "."
"bill.gates@microsoft.com" =~ m/\w+@\w+/ ==> TRUE

#### Add the .'s to get the whole thing
"bill.gates@microsoft.com" =~ m/^(\w|\.)+@(\w|\.)+$/ ==> TRUE

#### words separated by commas and possibly spaces
"Klaatu,   barada,nikto" =~ m/^\w+(,\s*\w+)*$/ ==> TRUE
```

Character Classes

Square brackets can be used to represent a set of characters. For example [aeiouAEIOU] is a one character pattern that matches a vowel. Most characters are not special inside a square bracket and so can be used without a leading backslash (\). \w, \s, and \d work inside a character class, and the dash (-) can be used to express a range of characters, so [a-z] matches lowercase "a" through "z". So the \w code is equivalent to [a-zA-Z0-9]. If the first character in a character class is a caret (^) the set is inverted, and matches all the characters **not** in the given set. So [^0-9] matches all characters that are not digits.

The parts of an email address on either side of the "@" are made up of letters, numbers plus dots, underbars, and dashes. As a character class that's just `[\w._-]`.

```
"bill.gates_emporer@microsoft.com" =~ m/^[\w._-]+@[\w._-]+$/ ==> TRUE
```

Match Variables

If a =~ match expression is true, the special variables $1, $2, ... will be the substrings that matched parts of the pattern in parenthesis -- $1 matches the first left parenthesis, $2 the second left parenthesis, and so on. The following pattern picks out three words separated by whitespace...

```
if ("this      and that" =~ /(\w+)\s+(\w+)\s+(\w+)/) {

  ## if the above matches, $1=="this", $2=="and", $3=="that"
```

This is a nice way to parse a string -- write a regular expression for the pattern you expect putting parenthesis around the parts you want to pull out. Only use $1, $2, etc. when the if =~ returns true. Other regular-expression systems use \1 and \2 instead of $1 $2, and Perl supports that syntax as well. There are three other special variables: $& (dollar-ampersand) = the matched string, $` (dollar-back-quote) = the string before what was matched, and $' (dollar-quote) = the string following what was matched.

The following loop rips through a string and pulls out all the email addresses. It demonstrates using a character class, using $1 etc. to pull out parts of the match string, and using $' after the match.

```
$str = 'blah blah nick@cs.stanford.edu, blah blah balh billg@microsoft.com blah
blah';

while ($str =~ /(([\w._-]+)\@([\w._-]+))/) { ## look for an email addr
  print "user:$2 host:$3  all:$1\n";          ## parts of the addr
  $str = $';       ## set the str to be the "rest" of the string
}

output:
user:nick host:cs.stanford.edu  all:nick@cs.stanford.edu
user:billg host:microsoft.com  all:billg@microsoft.com
```

Substitution

A slight variation of the match operator can be used to search and replace. Put an "s" in front of the pattern and follow the match pattern with a replacement pattern.

```
## Change all "is" strings to "is not" -- a sure way to improve any document
$str =~ s/is/is not/ig;
```

The replacement pattern can use $1, $2 to refer to parts of the matched string. The "g" modifier after the last / means do the replacement repeatedly in the target string. The modifier "i" means the match should not be case sensitive. The following example finds instances of the letter "r" or "l" followed by a word character, and replaces that pattern with "w" followed by the same word character. Sounds like Tweety Bird...

```
## Change "r" and "l" followed by a word char to "w" followed
## by the same word char
$x = "This dress exacerbates the genetic betrayal that is my Legacy.\n";
$x =~ s/(r|l)(\w)/w$2/ig;    ## r or l followed by a word char
## $x is now "This dwess exacewbates the genetic betwayal that is my wegacy."
```

The ? Trick

One problem with * and +, is that they are "greedy" -- they try to use up as many characters as they can. Suppose you are trying to pick out all of the characters between two curly braces { }. The simplest thing would be to use the pattern...

```
m/{(.*)}/  -- pick up all the characters between {}'s
```

The problem is that if you match against the string "{group 1} xx {group 2}", the * will aggressively run right over the first } and match the second }. So $1 will be "group 1} xx {group 2" instead of "group 1". Fortunately Perl has a nice solution to the too-aggressive-*/+ problem. If a ? immediately follows the * or +, then it tries to find the **shortest** repetition which works instead of the longest. You need the ? variant most often when matching with .* or \S* which can easily use up more than you had in mind. Use ".*?" to skip over stuff you don't care about, but have something you do care about immediately to its right. Such as..

```
m/{(.*?)}/ ## pick up all the characters between {}'s, but stop
           ## at the first }
```

The old way to skip everything up until a certain character, say }, uses the [^}] construct like this...

```
m/{([^}]*)}/ ## the inner [^}] matches any char except }
```

I prefer the (.*?) form. In fact, I suspect it was added to the language precisely as an improvement over the [^}]* form.

Substring

The `index(string, string-to-look-for, start-index)` operator searches the first string starting at the given index for an occurrence of the second string. Returns the 0 based index of the first occurrence, or -1 if not found. The following code uses index() to walk through a string and count the number of times "binky" occurs.

```
$count = 0;
$pos = 0;
while ( ($pos = index($string, "binky", $pos) != -1) {
 $count++;
 $pos++;
}
```

The function `substr(string, index, length)` pulls a substring out of the given string. Substr() starts at the given index and continues for the given length.

Split

The split operator takes a regular expression, and a string, and returns an array of all the substrings from the original string which were separated by that regular expression. The following example pulls out words separated by commas possibly with whitespace thrown in...

```
split(/\s*,\s*/, "dress ,      betrayal    ,  legacy") ## returns the array
   ("dress", "betrayal", "legacy")
```

Split is often a useful way to pull an enumeration out of some text for processing. If the number -1 is passed as a third argument to split, then it will interpret an instance of the separator pattern at the end of the string as marking a last, empty element (note the comma after the last word)...

```
split(/\s*,\s*/, "dress ,      betrayal    ,  legacy,", -1) ## returns the array
   ("dress", "betrayal", "legacy", "")
```

Character Translate -- tr

The tr// operator goes through a string and replaces characters with other characters.

```
$string =~ tr/a/b/;  -- change all a's to b's
$string =~ tr/A-Z/a-z/; -- change uppercase to lowercase   (actually lc() is
better for this)
```

11. Subroutines

Perl subroutines encapsulate blocks of code in the usual way. You do not need to define subroutines before they are used, so Perl programs generally have their "main" code first, and their subroutines laid out toward the bottom of the file. A subroutine can return a scalar or an array.

```
$x = Three();  ## call to Three() returns 3
exit(0);       ## exit the program normally

sub Three {
 return (1 + 2);
}
```

Local Variables and Parameters

Historically, many Perl programs leave all of their variables global. It's especially convenient since the variables do not need to be declared. This "quick 'n dirty" style does not scale well when trying to write larger programs. With Perl 5, the "my" construct allows one or more variables to be declared. (Older versions of perl had a "local" construct which should be avoided.)

```
my $a;                  ## declare $a
my $b = "hello"         ## declare $b, and assign it "hello"
my @array = (1, 2, 3);  ## declare @array and assign it (1, 2, 3)
my ($x, $y);            ## declare $x and $y
my ($a, $b) = (1, "hello");    ## declare $a and $b, and assign $a=1, $b="hello"
```

The "my" construct is most often used to declare local varaiables in a subroutine...

```
sub Three {
  my ($x, $y);  # declare vars $x and $y
  $x = 1;
  $y = 2;
  return ($x + $y);
}


# Variant of Three() which inits $x and $y with the array trick
sub Three2 {
  my ($x, $y) = (1, 2);
  return ($x + $y);
}
```

@_ Parameters

Perl subroutines do not have formal named parameters like other languages. Instead, all the parameters are passed in a single array called "@_". The elements in @_ actually point to the original caller-side parameters, so the called function is responsible for making any copies. Usually the subroutine will pull the values out of @_ and copy them to local variables. A Sum() function which takes two numbers and adds them looks like...

```
sub Sum1 {
 my ($x, $y) = @_; # the first lines of many functions look like this
                   # to retrieve and name their params
 return($x + $y);
}


# Variant where you pull the values out of @_ directly
# This avoids copying the parameters
sub Sum2 {
  return($_[0] + $_[1]);
}


# How Sum() would really be written in Perl -- it takes an array
# of numbers of arbitrary length, and adds all of them...
sub Sum3 {
  my ($sum, $elem); # declare local vars
  $sum = 0;
  foreach $elem (@_) {
    $sum += $elem;
  }
  return($sum);
}


## Variant of above using shift instead of foreach
sub sum4 {
  my ($sum, $elem);
```

```
  $sum = 0;
  while(defined($elem = shift(@_))) {
    $sum += $elem;
  }
  return($sum);
}
```

File Handle Arguments

The file handles are all in a global namespace, so you cannot allocate them locally like other variables. File handles can be passed from one routine to another like strings, but this amounts to just passing around references to a single global file handle...

```
open(FILE, ">file.txt");
SayHello("FILE");
close(FILE);

## Here, the file handle FILE is passed as the string "FILE"
sub SayHello {
  my($file_handle) = @_;

  ## Prints to the file handle identified in $file_handle
  print $file_handle "I'm a little teapot, short and stout.\n";
}
```

Actually, the file handle doesn't even need to be quoted in the call, so the above call could be written as `SayHello(FILE);`. This is the "bareword" feature of Perl where a group of characters that does not have another syntactic interpretation is passed through as if it were a string. I prefer not to rely on barewords, so I write the call as `SayHello("FILE");`.

Returning Multiple Values to the Caller

How to communicate back to the caller? Returning a single value to the caller works great when that's all you need. What about the case where there are multiple pieces of information to communicate back to the caller? The subroutine could communicate by modifying actual arguments in @_ , but that gets ugly. A better approach is to pack multiple things into an array and return that. The caller can catch the return array and use it as an array, or assign it to a my() expression that puts the values into named variables...

```
# Suppose this function returns a (num, string) array
# where the num is a result code and the string is
# the human readable form
sub DoSomething {
  # does something
  return(-13, "Core Breach Imminent!!"); # return an array len 2
}

# so a call would look like...
my ($num, $string) = DoSomething();
if ($num<0) {
```

```
  print "Panic:$string\n";
}
```

The values returned must be scalars — if they themselves are arrays, they will be flattened into the return array which is probably not what you want.

Flattened Arguments

Perl arrays are always "1-deep" or "flat". The arguments into a function all get "flattened" into the single @_ array, so it is not possible to pass an array as one of several arguments since it gets flattened out with the other arguments...

```
Sum3(1, 2, (3, 4));
## returns 10 -- the arg array is flattened to (1, 2, 3, 4)
```

This flattening can hurt you if you try to assign to an element which is an array...

```
my(@nums, $three) = ((1, 2), 3);
```

You might think that this assigns (1, 2) to @nums and 3 to $three. But instead the right hand side gets flattened to (1, 2, 3) which is then assigned to @nums, and $three does not get a value. Only use the my($x, $y) = (...); form when assigning a bunch of scalar values. If any of the values are arrays, then you should separate out all the assignments, each on its own line...

```
my (@array, $x);
@array = ...;
$x = ...;
```

You can get around the 1-deep by storing references to arrays in other arrays -- see the References section.

Global Vars and 'use strict'

You need to be careful to keep local and global variables straight, since by default, the compiler does not warn about possibly erroneous code. Suppose a subroutine has a $string local variable, except it is mistyped as $strning in one place. By default, this just declares a global variable named $strning.

Each of the following declarations causes Perl to enforce bug-reducing rules.

```
use strict 'vars';    ## enforce local/global var declarations
use strict;           ## like above, but with some additional style checks
```

With strict vars, variables inside functions must be declared with my(). Variables which are intended to be global must be referred to with two colons (::) in front of their name or must be declared with a global my(). Violating these rules results in a compile-time error.

```
## With strict vars...

## 1. Undeclared global vars must begin with "::" at all times
```

```
$::global = 13;

## 2. Or a global may be declared with a my(), in which case
## the :: is not necessary
my $global2 = 42;

sub foo {
  my $sum;
  $sum = $::global + $global2;
  ## $sum and $global2 work without extra syntax
  return($sum);
}
```

Both the "-w" option and "use strict" are good ideas or any Perl program larger than a page or two in size. Without them, you will inevitably waste time debugging some trivial variable name mixup or syntax error.

11. Running External Programs

Perl can be used to invoke other programs and mess with their input and output. The most straightforward way to do this is with the `system` function which takes a command line string (or an array of strings), and has the operating system try to run it (this makes the most sense in a Unix environment). System returns 0 when the program successfully completes, or on error the global variable `$?` should be set to an error description.

```
system("mail nick < tmp.txt") == 0 | die "system error $?";
```

The file-open function can also be used to run a program -- a vertical var (|) at the end of the filename runs the filename as a process, and lets you read from its output...

```
open(F, "ls -l |");                ## run the ls -l process, and name its output
F
while (defined($line = <F>)) {     ## read F, line by line
    ...
```

The same trick works for writing to a process -- just put the vertical bar at the beginning. Writing on the file handle goes to the standard input of the process. On Unix, the mail program can take the body of an email message as standard input...

```
$user = "nick\@cs";
$subject = "mail from perl";
open(MAIL, "| mail -s $subject $user");
print(MAIL, "Here's some email for you\n");
print(MAIL, "blah blah blah, ....");
close(MAIL);
```

If a programmer ever uses this technique to send Spam email, then all the other programmers will hunt that programmer down and explain the tragedy of the commons to them before the traditional beheading. Also, when writing a CGI, it's important that you control the sorts of strings that are

passed to system functions like open() and system(). Do not take text from the user and pass it directory to a call to system() or open() -- the text must be checked to avoid errors and security problems.

12. References

I'm happiest writing Perl code that does not use references because they always give me a mild headache. Here's the short version of how they work. The backslash operator (\) computes a reference to something. The reference is a scalar that points to the original thing. The '$' dereferences to access the original thing.

Suppose there is a string...

```
$str = "hello";  ## original string
```

And there is a reference that points to that string...

```
$ref = \$str;    ## compute $ref that points to $str
```

The expression to access $str is $$ref. Essentially, the alphabetic part of the variable, 'str', is replaced with the dereference expression '$ref'...

```
print "$$ref\n"; ## prints "hello" -- identical to "$str\n";
```

Here's an example of the same principle with a reference to an array...

```
@a = (1, 2, 3);  ## original array

$aRef = \@a;     ## reference to the array

print "a: @a\n";        ## prints "a: 1 2 3"
print "a: @$aRef\n";    ## exactly the same
```

Curly braces { } can be added in code and in strings to help clarify the stack of @, $, ...

```
print "a: @{$aRef}\n";  ## use { } for clarity
```

Here's how you put references to arrays in another array to make it look two dimensional...

```
@a = (1, 2, 3);
@b = (4, 5, 6);
@root = (\@a, \@b);

print "a: @a\n";             ## a: (1 2 3)
print "a: @{$root[0]}\n";    ## a: (1 2 3)
print "b: @{$root[1]}\n";    ## b: (4 5 6)

scalar(@root)         ## root len == 2
```

```
scalar(@{$root[0]})   ## a len: == 3
```

For arrays of arrays, the [ ] operations can stack together so the syntax is more C like...

```
$root[1][0]           ## this is 4
```

13. Terse Perl

Perl supports a style of coding with very short phrases. For example, many built in functions use the special scalar variable $_ if no other variable is specified. So file reading code...

```
while ($line = <FILE>) {
    print $line;
}
```

Can be written as...

```
while (<FILE>) {
    print;
}
```

It turns out that <FILE> assigns its value into $_ if no variable is specified, and likewise print reads from $_ if nothing is specified. Perl is filled with little shortcuts like that, so many phrases can be written more tersely by omitting explicit variables. I don't especially like the "short" style, since I actually like having named variables in my code, but obviously it depends on personal taste and the goal for the code. If the code is going to be maintained or debugged by someone else in the future, then named variables seem like a good idea.

Revision History

This document started its life in 1998 and 1999 as a "quick perl" handout for Stanford's Internet Technologies course. In the year 2000 I greatly expanded the document and added it to the Stanford CS Education Library as document #108. In 4/2001 and again in 5/2002 I did some moderate edits and additions. Thanks to Negar Shamma for the many suggestions.