PERL
LANGUAGE REFERENCE GUIDE

HARVARD COMPUTER SOCIETY

Table of Contents

Introduction

Perl stands for Practical Extraction and Report Language, or Pathologically Eclectic Rubbish Lister. Perl is a very useful programming language that is standard on almost all Unix systems. Its semantics are a lot like C, but its text processing and regular expression matching features are far more powerful. Perl is the ideal tool for the lazy programmer; almost any simple task can be performed using a Perl script that can be written in under 10 minutes. Perl is particularly powerful for scanning large quantities of text and doing pattern matching. Perl is also very useful as a scripting language for web pages (as CGI).

Invocation and #!

To create a Perl script, simply put your program into a text file. You can invoke it several ways:

1) Execute it from the command prompt with **`perl -w <file>`**
2) Put **`#!/usr/local/bin/perl –w`** at the top of your script, make the script executable, and invoke it simply by typing the script name at the command prompt. (You still have to type the path to the script, or if the script is in the current directory, type **`./scriptname`**.
   Tip: You'll need to make the permissions on the script executable to run it this way; to do so type: **`chmod 700 scriptname`**

Basic Semantics

In Perl, comments start with a # and continue to the end of the line. Otherwise, the semantics are basically very similar to C: all statements end with a semicolon; whitespace is ignored; curly braces are used for control structures; functions have arguments in parentheses following the function name.

Hello, World

Pioneered by Brian Kernighan, the Hello, world program is the traditional way to start off a new language. One of the nicest features of Perl, unlike C, is that to do something very simple requires only a very simple script. You can just go ahead and say:

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

# My first perl program
print "Hello, world!\n";
```

Scalars

Unlike C, Perl does not have separate data types for numbers, characters, strings, and so on. Instead, Perl has three primary data types: scalars, lists, and hashes. Scalars can hold one of several kinds of data:

Type	Example(s)
`Integers`	`3`
	`-9`
	`0377  # octal`
	`0x15  # hexadecimal`
`Floats`	`1.25`
	`6.55e12`
	`-2e-15`
`Strings`	`‘Perl’`
	`“This is a sentence with a newline at the end.\n”`

Also unlike C, you do not need to declare your variables ahead of time. When you want a variable, you simply use it and it springs into being. Likewise, Perl deals with allocating the appropriate amount of memory for the data you want a scalar (or anything else) to hold.

$@%&*

Perl is a very punctuation-heavy language. Regular expressions are the worst offenders. In addition, all variable names start with an item of punctuation:

$	**Scalar**
@	**Array or List**
%	**Hash**
&	**Function (aka Subroutine)**
*****	**Typeglob** (we won’t cover this)

Numerical Data and Operators

Arithmetic operators and their associated assignment operators are very much like C's. A notable exception is /, which will create a float from an integer if appropriate. Use `int ($a / 2)` for integer arithmetic.

```
$a = 4;
$b = 2.5;
$c = $a + $b;      # now $c is 6.5
$b += 2;           # $b is 4.5
$a %= 3;           # $a is 1
$a++;              # $a is 2 (same as C)
```

Strings

Strings are also held in scalars. They are not represented as arrays of characters (well, they are internally, but you never have to worry about that). Anything you want to do, you can do:

```
$a = "Harvard";
$b = 'Computer';
$c = $a . " " . $b;      # $c is Harvard Computer
$c .= " Society";        # $c is Harvard Computer Society
$d = $a x 4;             # $a is HarvardHarvardHarvardHarvard
$b x= 2;                 # $b is ComputerComputer
```

Since you never have to explicitly iterate over the characters of a string, strings are not null terminated. All perl operators know when they have reached the end of a string and stop.

Single and Double Quoting

You can put a string literal into your program with either single or double quotes. Inside a single-quoted string, all characters stay as they are and are not interpreted. The only exception is to put a single quote in the string, use `\'` and to put a backslash in the string, use `\\`.

Double quoted strings translate many more backslashed expressions, such as \n for newline. Also, any variable name placed in the double quoted string is *interpolated*; in other words, the variable name (including the $ or other identifier) is replaced by the value of that variable.

```
$a = 'Hello'
$b = 'He said, \'Hi.\'' # $b is: He said 'Hi.'
$c = "Hello"            # same as first example
$c = "$a, world!\n"     # $c is: Hello, world! plus a newline (interpolation)
$d = '$a, world!\n'     # $d is: $a, world!\n (no interpolation)
```

Comparison Operators: Arithmetic vs. String

Comparison	Numeric	String
Equal	==	eq
Not Equal	!=	ne
Less Than	<	lt
Greater Than	>	gt
Less Than or Equal To	<=	le
Greater Than or Equal To	>=	ge

It makes a difference which one you pick. All of them will work in all cases, but Perl's behavior is different.

```
5 < 30                              # is true
5 lt 30                             # is false
"hcs" gt "hascs"                    # is true
"hcs" > "hascs"                     # is false (they are both equal to 0)
"5 golden rings" eq "5 fingers"     # is false
"5 golden rings" == "5 fingers"     # is true (both are the number 5)
```

The gt and lt tests compare in alphabetic order, so 30 comes before 5. If you do a numeric comparison on strings, Perl tries to convert them to numbers. It will take any numeric digits from the front and discard the rest; if there are no numeric digits, it converts to 0. This is an example of how Perl rarely gives errors based on type usage; instead, it will try to interpret the data in the appropriate way. Sometimes it does what you want; sometimes it does not. If you stick to the main path, you will never be surprised; but if you understand the behavior of a certain operator you can often take shortcuts and save yourself time.

undef

There is a special scalar value called `undef`. If you access any variable that has not yet been assigned to, you get `undef`. You can set any variable to `undef` to clear it. If used as a number, `undef` looks like 0, if used as a string, it looks like "" (the empty string). Some operators/functions return `undef` under various circumstances; this will usually look like 0 or "" if you try to use it, which is typically not a problem. Perl also includes a function called `defined` that tests to see if a value is `undef`.

Lists

Perl's second type of data structure is the list, or array. A list is an ordered series of scalars. List variables start with the @ character. In your program, you can write arrays as a sequence of comma-separated values in parentheses.

```
@a = (1, 2, 3);
@b = ("Lewis", "Epps", 4); # you can mix types – they're all scalars anyway
@c = ($a + $b, $c . "\n"); # expressions are evaluated
@d = ();                   # an empty array with 0 elements
@e = 2;                    # it must be an array, so Perl makes it (2)
@f = (@a, 5, @b)           # no multi-level arrays
                           # this becomes (1, 2, 3, 5, "Lewis", "Epps", 4)
```

Subscripting

You can access individual elements of an array using square brackets. The index of an array starts at 0:

```
$a[0]       # is 1
$a[2]       # is 3
$a[3]       # is undefined
$b[$a[0]]   # is $b[1], which is "Epps"
@a[1,2]     # is (2, 3)
```

`$a[0]` is written with a $ because the value of that expression is a scalar. Likewise, if you take multiple values from an array, you get an array, so `@a[1,2]` is written with an @. Whatever type you get out is the type of identifier you use.

Note that `$purple` is very different from `$purple[2]`. The first is a scalar named $purple; the second refers to the third element of an array named `@purple`.

You can also assign to individual elements or whole arrays:

```
$b[2] = "Nathans";               # now @b is ("Lewis", "Epps", "Nathans")
($b[0], $b[1]) = ($b[1], $b[0]); # swap first two items
$a[0]--;                         # now @a is (0, 2, 3)
$d[3] = "Knowles";               # now @d is (undef, undef, undef, "Knowles")
```

As you can see from the last example above, Perl automatically changes the sizes of arrays to accommodate the data that is in them. Basically, you can do just about whatever you want to, and Perl will make it work out.

Array Sizes

The expression `$#array` evaluates to the subscript of the last element of `@array`. In other words, it is equal to the number of elements in `@array`, minus one. If you assign to `$#array`, the size of the array will change; if you make an array smaller, values on the end will be removed. You don't often want to do this, though, since you can easily lose data, so be careful.

```
@conc = ("CS", "Government", "English");
$size_conc = $#conc;    # now $size_conc is 2
$#conc = 1;             # we don't need the humanities
                        # now @concentrations is ("CS", "Government")
$#conc = 2;             # the last value was lost
                        # now @concentrations is ("CS", "Government", undef)
```

Some Array Functions

`push()` and `pop()` access an array like a stack, adding and removing things on the right side. `unshift()` and `shift()` do the same on the left, respectively.

```
@example = (1, 2, 3);
push(@example, "Spam");    # now @example is (1, 2, 3, "Spam")
$a_meat = pop(@example);   # now $a_meat is "Spam" and @example is (1, 2, 3)
$single = shift(@example); # now $single is 1
unshift(@example, "One");  # now @example is ("One", 2, 3)

@empty = ();
$hmm = pop(@empty);
```

What is the value of `$hmm` after the assignment above? Since Perl usually returns something appropriate, if possibe, rather than crashing, in this case `$hmm` will be set to `undef`. `shift()` exhibits the same behavior.

Operators that change the order of elements in an array include `reverse()` and `sort()`. `Sort()` will sort an array alphabetically, *not* numerically. (Later you will learn how to change the sort order).

```
@elpmaxe = reverse(@example);       # now @elpmaxe is (3, 2, "One");
@jumble = (1, 2, 4, 8, 16, 32);
@bejlmu = sort(@jumble);            # now @bejlmu is (1, 16, 2, 32, 4, 8)
```

Context

Some Perl operators expect to be given a scalar, and others expect to be given an array. What happens if you give the wrong type? By now, you probably know that Perl will make an attempt to deal with it. You already know how to see a scalar as an array: simply make it a one-element array. If an array is evaluated in a scalar context, then it is evaluated as the number of elements in the array. This will pop up occasionally later.

Control Structures

If

```
      if ($who eq "HCS") {
            print "The HCS is cool.\n";
      } elsif ($who eq "HASCS") {
            print "Your UA is $user_assistant.\n";
      } else {
            print "I don't understand.\n";
      }
```

This construct is like C's except for two major differences. First, in C, you can omit the curly braces if you have only one line following the if or the else. In Perl, you *must* use the curly braces. Also, because of this, you cannot write `if {…} else if {…}`. Instead, you use the word `elsif`.

Unless

If you want to test if something is not true:

```
      unless ($school eq "Harvard") {
            print "You should transfer.\n";
      }
```

There is no elsunless.

Doing Them Backwards

You can write them backwards, just as you could write an English sentence:

```
      $b += 5 if $b < 9;
```

This form has no `else` clauses. You also do not need parentheses after the `if` here.

And, Or

Another way to do this, also borrowed from natural language patterns:

```
      $age >= 0 or die "Aargh!  Your age is negative!\n";
      $age >= 120 and die "That's awfully old… do you have a fake ID or
something?\n";
```

This works because the `or` and `and` operators evaluate the left sides of their expressions. If it then has enough information to evaluate itself (such as if the left side of an `or` is already 1) then it does not execute the right side.

(The `die` operator quits the current program with an error condition after printing out its argument as a string. This form is often used for error checking, because it puts the contingency part on the right side of the screen away from the normal visual flow. But you can use it for things besides error trapping if style makes this form more aesthetic.)

What is Truth?

For any scalar, Perl figures out whether it is true or false. Most of the time, this is intuitive. 0 is false. Any other integer is true. `undef` is false. The empty string "" is false, and any other string except "0" is true. This last one is because what Perl actually does is to convert its value to a string, and if it becomes "" or "0", it's false; otherwise, it's true. This means that the strings "00" and "0.00" are actually true. But you can force any variable to be converted to a number first by putting a + in front of the variable. But this kind of pathological condition almost never occurs.

While

This is the simplest type of iteration:

```
while ($age < 21) {
      print "Get older first.\n";
      sleep(365 * 24 * 60 * 60);
      $age++;
}
```

For

Very much like C's `for`:

```
for ($t = 10; $t > 0; $t--) {
      print "$t\n";
}
print "Boom!\n";
```

Foreach

`Foreach` iterates over each element in a list:

```
@publications = ("Crimson", "Indy", "Perspective", "Salient");
foreach $p (@publications) {
      print "I read $p.\n";
}
```

Note that $p becomes an alias for the element in the list; thus if you change the value of $p somewhere in the loop, you change the array. You can also leave out the `$p`, just writing `foreach (@publications)`. In this case, Perl will assign each value to the special variable $_. `$_` is the default for a lot of operations. For example, if not given anything to print, the print operator will print $_; `length` will test the length of $_, and many more:

```
@hds_comments = ("Get a FroYo machine.\n", "The chicken is overcooked.\n",
                 "I love the cereal selection.\n", "HDS sucks!\n");
foreach (@hds_comments) {
      print if length > 15;
}
```

Here, if the string is longer than 15 characters, the operator print is called without any arguments; this will make it print the current string since each string is placed in $_.

last, next, and redo

These operators change the flow of a loop. `Last` immediately exits the loop, just like C's `break` statement. `Next` skips to the end of the loop and begins the next iteration; in a `for` loop, variables are updated just as if the loop had naturally reached the end. `Redo` goes back to the top of the loop and starts the current iteration over again without updating the variables.

```
# Example using last
@donuts = ("glazed", "powdered", "honey-dipped")
$favorite_donut = "powdered";

# The question is, is my favorite donut in the list?
foreach (@donuts) {
      if ($_ eq $favorite_donut) {
            print "Hurrah!  You have $_ donuts!\n";
            last; # no need to keep looking
      }
      print "I don't like $_ donuts.\n";
}

# Example using next
# Print all non-square numbers from 1 to 10
for ($i = 1; $i <= 10; $i++) {
      next if (int sqrt($i) == sqrt($i));
      # we only want the non-squares

      print "$_ is not a square number.";
}
```

Hashes

Hashes are the third major data type in Perl. Hashes are denoted using the % sign. Hashes also contain scalars, but instead of storing them in a fixed order, a hash stores them in pairs: the key and the value. Given a scalar as a key, Perl will find the corresponding pair in the hash and return the value.

You can assign directly to a hash just as with an array. The => operator is just like the comma, but it makes the code more readable.

```
%profs = (   "Ec 10" => "Feldstein",
             "CS 121" => "Lewis",
             "Justice" => "Sandel");

%sizes = (   "Ec 10" => 900,
             "CS 121" => 120,
             "Justice" => 750);
```

Referencing a hash is just like referencing an array, except you use curly braces instead of square brackets:

```
$a = $profs{"Ec 10"}                            # $a is "Feldstein"
$b = $sizes{"CS 121"} + $sizes{"Justice"}       # $b is 870
```

Keys() and Values()

Two important operators on hashes are `keys()` and `values()`. The first returns a list containing all of the keys in the hash. Their order in the list is arbitrary (it is based on Perl's internal representation), but you can always `sort` the list if you need to. The `values()` operator returns a list of the values in the hash. These operators are frequently used for iteration with `foreach`:

```
@names = keys(%profs)         # @names is ("CS 121", "Ec 10", "Justice")
@numbers = values(%sizes)     # @numbers is (120, 900, 750)

foreach (keys %sizes) {
      if ($sizes{$_} > 300) {
            print "Wow, $_ is a big class!\n";
      } else {
            print "$_ is a reasonable size.\n";
      }
}
```

Deleting

If you want to get rid of an item in a hash, use the `delete()` operator:

```
delete $profs{"Ec 10"}  # now %profs only has two key/value pairs
```

Note that the parentheses for most operators are optional. I usually omit them for operators taking only one argument unless it is part of a more complex expression and could be confusing. In the `foreach` loop above, you could write `foreach (keys (%sizes))`, but in this case the parentheses don't add much. On the other hand, if you were writing `foreach (reverse sort keys %sizes)`, then throwing in one or two extra sets of parentheses could make the code more readable. Your call.

Filehandles and I/O

Perl provides simple input and output facilities using a mechanism known as *filehandles*. A filehandle, unlike variables that you have seen so far, does not have a special punctuation mark preceding it. For this reason, as well as for readability, it is customary to name all filehandles with `CAPITAL LETTERS`.

<STDIN> and <>

Any Perl program automatically creates several filehandles, including `STDIN`, for standard input, which contains keystrokes the user types. Reading from a filehandle is accomplished by enclosing the filehandle name in angle brackets. In a scalar context, the expression <`STDIN`> returns one line of input:

```
print "What is your name?\n";
$name = <STDIN>;
if ($name) {
      print "Good to meet you, $name.\n";
} else {
      print "There's no need to be shy.\n";
}
```

The diamond operator, <>, normally acts like <`STDIN`>, but if the Perl program was run with file names on the command line, <> will read through those first. Therefore, you can write the Unix program cat this way:

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

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

In a list context, any filehandle in angle brackets returns a list of all lines of input. Note that the `print` operator takes a list as input, so you can do something like `print "Hi ", "there"`. Therefore, you probably do not want to write `print <STDIN>`, because in an array context, <> will grab all of the lines:

```
@input = <>;
print "The command-line files you specified have a total of  ",
      length @input, " lines.\n";
```

But beware that if this script is not called with any command-line arguments, then it will keep asking for lines from the user until the user sends an end-of-file character with control-D. The same thing would happen if you wrote `print <STDIN>`.

Make Your Own Filehandles

You can create filehandles yourself using the `open()` and `close()` operators. Giving just the name of the file opens a file to be read; prepending an > opens the file for writing; and prepending >> opens the file to append. A filehandle is closed the same way regardless of its read/write status.

```
open(WEBPAGE, "index.html");        # creates a filehandle WEBPAGE to be read from
open(MAIL, ">/etc/aliases");        # wipes out the given file and allows you to
                                    # write to the file using filehandle MAIL
open(ERROR, ">>logs/error.log");    # lets you write to filehandle LOGFILE to
                                    # append data to the end of the file
close(MAIL);                        # closes this filehandle
```

If you `open()` an existing filehandle, then any file previously attached to it is automatically closed first.

Chopping

When reading in a line, you get the entire line, including the newline at the end. To get rid of it, pass the variable name to the `chop()` operator, which will remove the last character of the line. `chop()` also returns the character it removed. Another function is `chomp()`, which removes the last character *only* if it is a newline character.

```
      $_ = "suey";            # use of "suey" comes from the Llama book
      chop($_);               # now $_ is "sue"
      chop;                   # also works: now $_ is "su"
      $newstr = chop($_);     # wrong; now $newstr is "u", not "s"
```

Printing to Filehandles

To print to a filehandle, place the name of the filehandle (no angle brackets) after the word print. There should not be a comma after the filehandle; this lets Perl know that it is a filehandle and not a string that it should print.

```
      print "Hello, world!\n";            # these two are equivalent
      print STDOUT "Hello, world!\n";

      print FILE "$book: \$$price\n";     # prints Programming Perl: $39.95
      print FILE, "$book: \$$price\n";    # probably generates an error
```

Handling File Errors

Sometimes, opening a file fails (because of a disk error, a mistyped filename, etc.) You can test for this by checking the return value of `open()`, which is false if the open failed. The backwards `if` syntax using the `or` operator, described above, is usually used here to avoid having a long if statement at every file open.

```
      open (JUNK, ">/scratch/my.garbage") or die "I can't dump my garbage!\n";
```

File Tests

You can check for the existence of files and other attributes by using the file test operators. For example, to see if a file exists:

```
if (-e "johnharvard.txt") {
      open(JOHNNIE, "johnharvard.txt");
} else {
      print "Sorry, I can't find John.\n";
}
```

Other file tests include `-r`, which is true if the file is readable by you; `-w` if the file is writeable, `-x` if it is executable; `-o` if you own it; `-f` it is a normal file; `-d` if it is a directory; `-l` if it is a symbolic link; and many more.

Some operators return values beyond just true or false. `–s` returns the size of the file; `-M` gives the number of days since the file was modified, `-A` the days since it was accessed, and `-C` the days since it's inode (name, permissions, etc.) was changed. `-M`, `-A`, and `-C` return a decimal value.

If you test a file for some attribute and want to test it for something else, you can use the special filehandle _ which refers to the last file tested. Testing against _ is also faster.

```
if (-r $file and –w _) {
      print "Good news!  $file is both readable and writeable.\n";
}
```

Pipes

You can also read input from another program and/or write output to another program by opening pipes. To read from a program, the second argument to open() should be the program name, followed by any arguments, just as you would enter them on the command line, followed by a pipe symbol **|**. To write to a program, place the pipe symbol before the command name:

```
      open(FINGER, "finger \@fas.harvard.edu |");
      open(MORE, "| more");
      while (<FINGER>) {
            # parse finger input somehow
            print MORE $_;
      }
```

Regular Expressions

Perl's most versatile and powerful feature is its regular expression matching. Regular expressions allow you to pull apart strings, analyze them, and extract pieces. The basic type of regular expression is the matching operator, which consists of two forward slashes with a regular expression in between, like this: `/…/` Regular expressions can include these characters:

a	matches the letter a
.	matches any character except newline
\n	matches a newline
\w	matches any word character (upper- or lowercase letters, digits, and _)
\d	matches digits 0-9
\s	matches white space (space, tab, newline)
\W \D \S	the negation of \w, \d, and \s, respectively

\w, \d, and \s above are examples of *character classes*, which match one of a certain group of characters. You can make your own by enclosing the characters in square brackets. Therefore `[aeiou]` would match any lowercase vowel. Adding a `^` to the beginning of the class negates it; so `[^5]` matches any character but the numeral 5. You can also use a dash (–) to give a range, so `\w` would be equivalent to `[a-zA-Z0-9_]`.

Any character, character class, or group of characters placed in parentheses can be followed by a multiplier, which allows the regular expression to match multiple repeated occurrences of that item:

b*	matches zero or more b's
b+	matches one or more b's
b?	matches zero or one b's
b{m,n}	matches at least m and at most n b's
b{x}	matches exactly x b's
b{y,}	matches y or more b's
b{0,z}	matches up to z b's

Anchoring patterns do not specifically match characters, but force the matching to take place at a certain place within the string:

^	matches only at the beginning of the string
$	matches only at the end of the string
\b	matches only at a word boundary (between a \w and a \W or a \w and the beginning or end of the string)
\B	matches only when not at a word boundary

The vertical bar | means "or" and allows a match with either the item before or the item after. Parentheses have highest precedence, followed by the multipliers +, `*`, ?, and `{}`; then sequence `(abc)` and the anchoring patterns; and finally the alternator `|` is the lowest. This is consistent with other definitions of regular expressions.

ab*	an a followed by any number of b's
(ab)*	the sequence ab repeated zero or more times
a|b*	an a or any number of b's
(a|b)*	any number of a's and b's

You can precede any special character by a `\` to turn off its special meaning.

Examples

```
/low/             matches "lower", "pillow", or "Yellowstone"
/ab...ed/         matches "aborted" or "labelled"
/\d+ [\w ]*/      matches "12 Monkeys" or "5 golden rings"
/(sick)*/         matches anything
/low$/            only matches "pillow", "marshmallow", etc.
/^\D{2,}ever/     matches "forever" or "not ever"; not "never" or "1234ever"
/\Bing/           matches "laughing" or "sing" but not "ingenious"
```

Using the Matching Operator

By default, the matching operator compares against $_. One of the simplest and most common constructs in Perl iterates through a file and does some sort of match on each line. Here is a variant of `grep` that looks for romantic lines:

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

while (<>) {
      chomp; # customary to get rid of the newline
      if /love/ {
            print "Here's a sweet line: $_\n";
      } elsif /sex/ {
            print "For young adults only: $_\n";
      }
}
```

If you want to match against some other variable, you use the =~ operator. To negate the results of the match (i.e. to return true if there is no match), use `!~`.

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

print "What is your name?\n";
$name = <STDIN>; # get name

print "What is your quest?\n";
$quest = <STDIN>; # get quest

unless ($quest =~ /grail/) {
      print "Then why are you in this movie?\n";
      exit; # stop executing the program
}
```

The unless line above could also be written `if ($quest !~ /grail/)`.

Case Insensitivity

If you want an expression to match regardless of case, add the letter i after the last slash:

```
/^cab/i           # matches "cabinet", "Cabot", and "cAbBaGe"
```

The Magic of Parentheses

Parentheses are useful to alter precedence for a regular expression, but they do much more than that. First, they remember what the sub-expression inside matched, and you can refer to that elsewhere in the expression. If you number the pairs of parentheses sequentially, then later in the expression `\1` matches the same thing that the first set of parentheses matched; `\2` matches the same thing the second set matched, etc. If parentheses are nested, then they are numbered in the order in which the left parentheses appear (so an outer set is before an inner set).

```
/(.{3})\1/                  # matches "jonjon" but not "jonjoe"
/I (\w+) you; you \1 me\./    # look for Barney songs
```

These stored values are not lost after the regular expression matching is over; `\1` is stored in the variable `$1`, `\2` in $2, etc. This allows you to use regular expressions to dissect a string. Because of this feature, regular expressions are often used where their truth value (whether they match or not) is less important than the values of these variables afterward.

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

# This program reads in a list of e-mail addresses and gives info
# about the Harvard e-mail addresses in the list.

while (<>) {
      chomp;
      if (/(\w{2,8})\@(\w+)\.harvard\.edu/) {
            print "$1 is logged into $2\n";
            # $1 is the username, and $2 the host
      }
}
```

Substitution

In addition to the regular matching operator, you can make substitutions in a string using the `s///` operator. This operator has three forward slashes. Between the first and second slashes is a regular expression that matches where the substitution should take place. Between the second and third slashes is a string, which is treated as a double-quoted string for purposes of interpolation, that should replace the part of the string that matched.

```
$a = "Yale is the best";      # inaccurate statement
$a =~ s/Yale/Harvard/;        # $a now is "Harvard is the best"

$b = "Mary had a little lamb";
$b =~ s/(\w+)/$1/g;           # $b now is "<Mary> <had> <a> <little> <lamb>"
```

The `i` flag works as with matching. If you would like to make the substitution multiple times throughout the string, you can add the g (global) flag which makes all possible substitutions. The e flag treats the right side of the substitution as a Perl expression rather than a string.

```
$c = "11 drummers drumming, 10 pipers piping, 9 lords a-leaping";
$c =~ s/(\d+)/$1 + 1/eg;      # add 1 to each number in the string
```

split() and join()

If you have a string with several pieces of data separated by a delimiter (often found in simple text databases), the `split()` operator allows you to easily divide the string using a regular expression.

```
      $data = "CS 121:Harry Lewis:Boylston Auditorium";
      ($class, $professor, $location) = split(/:/, $data);
      # this syntax makes it easy to see what each item represents

      $lincoln = "Fourscore and   seven years ago";
      @gettysburg = split(/\s+/, $lincoln);
      # the \s+ matches the long space, so that is treated as all one delimiter
```

To put a string together in the same way, use `join()`, which doesn't use regular expressions at all, really.

```
      $lincoln = join("==", reverse @gettysburg);
      # now $lincoln is "ago==years==seven==and==Fourscore"
```

Functions

While many Perl programs can be written using only one main section of execution, you can define functions within your program. Anywhere in the script (location does not matter, but most people place them at the end), you can define a function using the keyword `sub` (short for subroutine), followed by a name, followed by a block of statements.

Any arguments that are passed to the function when it is called are placed in `@_` (not to be confused with `$_`). `@_` is the default argument for array operators like shift, so it is common to get a function's argument(s) using `shift`; but you can assign a whole list to various scalars by writing `($a, $b, $c) = @_`.

```
sub add_two {
      my($a) = shift;  # ‘my’ makes $a local to the function
      $a + 2;
}

sub add_numbers {
      my($sum) = 0;  # initialize $sum to 0
      foreach (@_) {
            $sum += $_;
      }
      return $sum;
}
```

The function returns the value of the last expression that was evaluated. You can force the function to exit with a particular value in the middle of its execution using the `return` operator, but this is not necessary (unlike C).

&calling(them)

A function is basically yet another data type, with its own special punctuation mark: the ampersand (&). To call a function, precede its name with an ampersand, and follow it with the list of arguments in parentheses. This is a Perl list, which (as always) is a simple list of scalars. If you put a list into the argument list, it will be interpolated into the list:

```
@my_list = (2, 3, 4);
&add_numbers(1, @my_list, 5); # will return 15
```

When `add_numbers` is run, its `@_` will be equal to the list `(1, 2, 3, 4, 5)`. Passing lists, hashes, and other data types by reference requires using Perl references or typeglobs (beyond the scope of this seminar).

my() oh my()

Most variables you use in Perl are global variables. When writing a function, however, you may want to be able to paste the function into different programs without having it clobber any of your variables. To create a local variable, use `my()`. `my()` can create several local variables by giving all of their names to `my()` as arguments.

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

$what = "Lions";
$what_else = "tigers";

$line = &and_bears($what_else, "oh, my()!");
print "$what and $line\n";
```

```
sub and_bears {
      my($what, $end) = @_;
      "$what and bears, $end";
}
```

This program prints "Lions and tigers and bears, oh my()!" The value of $what inside the function (here, "tigers") will not interfere with the global value ("Lions").

Special Sort Functions

If you would like to sort in a different way than the default alphabetic search, you can create your own search comparison function. To do this, create a function that simply compares the value of $a and $b (which are set locally especially for this function). It should return -1 if $a < $b, 1 if $a > $b, and 0 if $a = $b. There are two operators that already do this: the numeric comparison operator <=> and the string comparison operator cmp.

To use it in a sort, simply put the function name (without the &) after the sort command and before the list that should be sorted. Many people name their sort functions something like by_value so that the command reads sort by_value @my_list.

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

@list = (12, "fan", 7, "email");
%example = (12 => "dozen", "fan" => "cool", 7 => "lucky", "email" => "pine");

@s = sort standard @list;           # is (12, 7, "email", "fan")
@n = sort numeric @list;            # is ("email", "fan", 12, 7)
@l = sort by_length @list;          # is (7, 12, "fan", "email")
@h = sort by_hash_value @list;      # is ("fan", 12, 7, "email")

sub standard {
      $a cmp $b;
}

sub numeric {
      $a <=> $b or
      $a cmp $b;  # this is 2nd tier sort: it only gets evaluated if
                  # the first sort above returns 0 (same)
}

sub by_length {
      length($a) <=> length($b)
}

sub by_hash_value {
      $example{$a} cmp $example{$b};
}
```

Formats

Formats are useful to create well-formatted output of data. While you can always use lots of `printf()` statements (yes, `printf()` exists in Perl, but it is used very rarely; the normal `print()` is almost always better), it is much easier and more visually readable to create a format.

A format, like a function, can be anywhere in the file but is usually at the end. It starts with the keyword format, followed by the name of the format, which is usually a filehandle, and then an equal sign. The next line is called the picture line, and it basically is a picture of what the output should look like. It has a series of fields, each starting with an @ and followed by zero or more <'s, |'s, >'s, or #'s. The picture line is followed by a comma-separated list of Perl expressions, one per field in the picture line. This is complicated, so here is an example:

```
format STDOUT =
@<<<<<<<<< @>>>>>     Price: \$@###.##
$_, $status_code{$_}, $price{$_} * $profit_margin
```

Each line will start with up to 12 characters of the name, left-justified, followed by some spaces, then the status code right justified, then a lot of space, the word `price` (with colon), a dollar sign, and then the price times the profit margin to two decimal places.

To generate output, simply call the `write` operator. By default, it writes to `STDOUT` using the format called `STDOUT`. If you pass write a different filehandle as an argument, it will write to the chosen filehandle using the format with the same name as the filehandle. One line (or more, if a format has multiple picture lines) will be printed by evaluating the expressions and formatting them appropriately.

```
$profit_margin = 1.5
%price = ("Glee Club" => 12.50, "Collegium" => 5.00, "RCS" => "10.20");
%status_code = ("Glee Club => "PC", "Collegium" => "TDO", "RCS" => undef);

foreach keys %price {
      write; # see how easy it is
}
```

This would create the following output:

```
Glee Club     PC     Price: $  18.75
Collegium    TDO     Price: $   7.50
RCS                  Price: $  15.30
```