Perl: A crash course

Brent Yorgey

Winter Study '03

Contents

1 Introduction and philosophy

First, as the title says, this is a crash course in Perl. It is not meant to be a comprehensive Perl reference, nor even a comprehensive introduction to Perl. Rather, it is intended to be a concise introduction aimed specifically at those with a good background of general computer knowledge. It is my hope that after reading this you will:

- be able to write simple, useful programs in Perl
- be aware of some of the more advanced constructs and techniques that are available in Perl, and where/how to learn about them if you so desire
- have a solid understanding of Perl's unique way of looking at the world

In view of these goals, I have put in specific details only where I thought they are critical or especially interesting or useful. I have, however, tried to put in footnotes[1] where appropriate to alert you to the fact that details are being omitted, and to point you in the right direction if you are interested in learning more. Also, at the end of most sections I have placed a section titled "Muffins"[2]; in it I try to give you an idea of cool features I haven't described which you could look up if you wanted. Some general places to go for more information include:

- The Perl man pages, which are quite detailed and come bundled with Perl (which usually comes bundled with any UNIX-type system). Type `man perl` at a prompt for a listing of the many, many man pages on various aspects of Perl. Throughout this document I have adopted the standard convention that *somename(1)* refers to the man page titled *somename* in section 1 of the manual.
- The O'Reilly book series is excellent in general, and in particular *Learning Perl* (the Llama Book) is a good introduction to the language (although it suffers somewhat from the let's-make-this-accessible-to-stupid-people syndrome), and *Programming Perl* (the Camel Book) is THE standard Perl reference[3], written by the creator of Perl himself along with a few others.
- CPAN (the Comprehensive Perl Archive Network) has just about everything you could ever want related to Perl: documentation, additional modules, various releases and ports of Perl... *http://www.cpan.org*.
- You can ask me, since I know everything there is to know about Perl.[4]

[1] (like this)

[2] Why "muffins", you ask? Because...uh...muffins are tasty.

[3] You can borrow mine if you take good care of it and return it in a prompt manner. I believe WSO also owns a copy which lives in the Cage.

[4] Hahahaha!! Just kidding. You will soon understand why this is funny.

Secondly, the thing that always annoys me about most tutorials is that I have to spend way too much time wading through stuff that I a) already know or b) could easily figure out myself, in order to get to the important stuff. So, in writing this tutorial I've assumed a lot of basic knowledge about programming languages in general, C and/or Java in particular, working in a UNIX environment, and computers in general. But, of course, if something doesn't make sense, isn't clear, or assumes some piece of knowledge you don't have, then please ask! I expect I will end up making rather broad revisions based on the responses I get, so input in this respect would be greatly appreciated.

Finally, I have tried to include a few project ideas at the end of some of the later sections, ranging from simple to complex, which reinforce the ideas introduced in the section. Of course you are free to try them, modify them, ignore them, or come up with your own projects as you wish. Let me know if you invent your own projects, since I would love to include them in future versions of this tutorial (with proper citation, of course).

2 Basics

Perl is a declarative[5], interpreted[6] programming language useful for things ranging from system administration to CGI programming to recreational cryptography[7]. It is extremely good at string processing (something that most other programming languages aren't), and has a unique way of looking at things that makes many tasks quite simple to program which would be awkward in many other languages. It's not good at everything, though; for example, it is not particularly useful for things that require efficiency, such as scientific computation or simulations of CPU scheduling algorithms[8]. But that's why you need a large (arsenal | toolbox | garage | stable), so you can pick the right (weapon | tool | vehicle | steed) to get the job done.[9]

In case you were wondering, Perl stands for Practical Extraction and Report Language, or to those who know it well, Pathologically Eclectic Rubbish Lister.

2.1 Running Perl scripts

All Perl scripts must start with the line[10]

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

which tells the OS that the rest of the file should be piped through the Perl interpreter.[11] Sometimes Perl is located somewhere other than `/usr/bin/` —

[5] (mostly)

[6] This is a lie.

[7] It may even be useful for real cryptography, but I kind of doubt it.

[8] ask Jeremy Redburn or Joe Masters... (=

[9] Pick your favorite metaphor.

[10] This is a lie too. You could actually leave this line off and run your script by typing `perl` *scriptname* at a prompt. But that would be silly.

[11] By the way, "start" should be taken literally. *Nothing* may come before this, including comments, blank lines, spaces, tabs...

type `which perl` at a prompt to find out where.

The `-w` is optional but turns on various warnings which can be very helpful — I definitely recommend you use it, at least while you are learning Perl.

To run a Perl script, first make sure its executable bit is set, and then, well, execute it.

2.2 Syntax

First of all, Perl is *case-sensitive*, *i.e.* `$q` is a completely different variable than `$Q`. The problem with this is that by default, Perl does not require you to declare variables before you use them. Thus, if you mis-capitalize (or misspell) a variable name somewhere, Perl will not complain, but instead will assign the misspelled variable a default value[12]. Needless to say this can lead to *very* frustrating and hard-to-find bugs. So, if you want to save yourself countless hours of pain and frustration, I *highly* recommend that you use the directive

```
use strict;
```

at the top of all your programs. This forces you to declare all your variables and causes Perl to complain when it sees a variable which has not been declared (often a misspelling).

Perl syntax bears many similarities to C or Java syntax. In particular, Perl ignores most whitespace, and every statement must be terminated by a semicolon. Syntax for things like if statements, for loops, and while loops is very similar to that of C and Java, with only a few notable exceptions (see section 5).

Comments in Perl are introduced by `#` (pound); everything from a pound symbol to the end of a line is ignored by Perl. Unfortunately, there are no multi-line comments.[13]

Muffins!

- If you really *must* know why saying that Perl is an interpreted language is a lie, see chapter 18 of *Programming Perl*, or for the truly masochistic, see *perlguts(1)*. Technically it's sort of compiled into an intermediate format first. From the outside it still *looks* like it's interpreted, the only difference being that it runs a lot faster than it would if it really were being interpreted line-by-line.

Projects

1. Write a Perl script that does nothing. (OK, just kidding, you'll have to wait for a few sections to get interesting projects...)

[12] Like zero, or the empty string, or something else you probably don't want it to be.

[13] You guessed it, another lie. If you really really must use multi-line comments, see chapter 26 of *Programming Perl* about POD. It's not pretty, though.

3 Variables and data types

To declare a variable, use the `my` operator:

```
my $var;
my ($var1, $var2, $var3);
```

Note that to declare multiple variables at once, you must enclose them in parentheses as in the above example. Declaring a variable in this way creates a variable local to the current lexical scope.[14]

Perl has three data types: scalars, arrays, and hashes. That's it.[15]

3.1 Scalars

Scalar variables always start with `$` (dollar sign). The scalar is the sole primitive data type in Perl. The nice thing about scalars is that they can store pretty much any primitive bit of data, including integers, floating-point numbers, strings, even references[16] to other complex data structures. In general, Perl is pretty smart about figuring out how to use whatever data is in a scalar depending on the context. In particular:

- Perl automatically and easily converts back and forth between numbers and their string representations.
- Perl is usually smart about precision of numbers, so that, for example, it prints "25" instead of "25.000000000001" if you're using integers.

The standard arithmetic operators (`+ - * / %`) are available for working with numeric scalars, as well as `**` (exponentiation), `++` and `--` (pre- and post-increment and -decrement). Standard comparison operators are available (`== != < > <= >=`) as well as C-style logical operators (and `&&`, or `||`, not `!`). Care should be taken not to confuse `=` (assignment) and `==` (equality test). As with C, Perl will not complain if you mix them up.[17] Don't say I didn't warn you.

3.2 Strings

Strings aren't technically a separate data type (they can be stored in scalars), but they get their own section anyway, so get over it. Single-quotes are the "standard" delimiters for strings: everything inside the single quotes is taken literally. Single quotes can be obtained in a single-quoted string by using the sequence `\'` (backslash-single quote); backslashes can be obtained by `\\`.

Double-quoted strings, however, are cooler than single-quoted strings, since double-quoted strings are "interpolated": any variable names found inside the string will be replaced by their value. The program in figure 1, for example, prints the lyrics to the song "99 bottles of beer", with correct grammar.

[14] If that doesn't mean anything to you, don't worry about it.
[15] Well, unless you count file descriptors, subroutines, and typeglobs...
[16] Think "pointers".
[17] Unless you use the `-w` switch...

```
#!/usr/bin/perl -w
use strict;

my $bottles = 99;
my $bottlestr = 'bottles';
while ( $bottles > 0 ) {
  print "$bottles $bottlestr of beer on the wall,\n";
  print "$bottles $bottlestr of beer,\n";
  print "Take one down, pass it around,\n";
  $bottles--;
  if ( $bottles == 1 ) {
    $bottlestr = 'bottle';
  } else {
    $bottlestr = 'bottles';
  }
  print "$bottles $bottlestr of beer on the wall.\n";
}
```

Figure 1: Interpolation in double-quoted strings.

This program also illustrates a couple of other important things:

- the `print` function is used to print things. It works pretty much the way you would expect.[18]
- Variables are not the only things interpolated in double-quoted strings: various special backslash control sequences are interpolated as well. Important ones include `\n` (newline), `\r` (carriage return), `\t` (tab), and `\b` (backspace).

If you want to compare strings, you should not use the normal comparison operators `==` `<` `!=` *etc.*, which are for comparing numbers. If you try to compare strings with numeric comparators, Perl will not complain,[19] but you will probably get unexpected results. To compare strings, use the string comparison operators: `eq` (equal), `ne` (not equal), `lt` (less than), `gt` (greater than), `le` (less or equal), and `ge` (greater or equal).

The string concatenation operator is `.` (period).

3.3 Arrays

Array variables always start with `@` (the "at" symbol). Arrays are variable-length and can contain any sorts of scalars (even a combination of strings, numerics, *etc.*). Array literals are written with parentheses. For example:

[18] Except when you least expect it.

[19] Again, unless you use the `-w` switch. Are you beginning to see why we like the `-w` switch?

```
my @array = (3, 4, 'five', -0.0003497);
```

Note that since arrays can contain only scalars, you cannot have arrays of arrays[20]; by default, arrays included in other arrays are "flattened", which is often useful behavior. For example:

```
my @array1 = (1, 2, 3);
my @array2 = (4, 5, 6);
my @bigarray = (@array1, @array2);
```

After this code is executed, `@bigarray` contains (1, 2, 3, 4, 5, 6).

Indexing is done with square brackets; note that when indexing, the name of the array should be preceded with a `$`, since the value of the whole expression is a scalar.[21] For example:

```
my @array = ('1337', 'h4X0r', 'w4r3z');
print "The first element of \@array is $array[0].\n";
```

When run, this code will print `The first element of @array is 1337.` Note that array indexing starts with 0; also note how the `@` symbol before `array` is escaped with a backslash to prevent the value of `@array` from being interpolated into the string, since we actually want it to literally print "@array".

Negative indices count backward from the end of an array; in particular, `$array[-1]` will yield the last element of `@array`, which is often quite useful.

Assigning a value to an index past the end of an array is legal, and will just fill in the intervening array positions with the special "undefined value" (see section 4.1).

The index of the last element of `@array` is given by the special scalar `$#array`; it is always equal to one less than the length of the array. (See section 4.3, "Context", for how to directly compute the length of an array.)

3.4 Hashes

Hash variables always start with `%` (percent). A little explanation is in order, since I don't know of any other programming languages with built-in hashes — usually you have to make your own. If you already know what a hash table is, you can skip the following paragraph.

Basically, a hash is a list of key-value pairs. Each "value" is the data the hash is actually storing, and each "key" is an arbitrary scalar used to look up or index its corresponding value. One way to think about it is that a hash is like an array, except that instead of using consecutive numbers to index values, one uses arbitrary scalars. Another way of thinking about it is that each value stored in a hash has a "name" (its key). Hashes are efficient: on average, insertions, deletions, and searches take constant time. Note that unlike arrays, hashes have no inherent "order"; to be sure, the key-value pairs are stored internally in some

[20] Unless you use references.

[21] This actually makes sense if you think about it.

sort of order, but it's dependent on the particular hash function being used, and not something one can depend on.[22]

Hash indexing is done with curly braces; note again that when retrieving a scalar value from a hash by indexing, one should use a dollar sign. For example, the code snippet in figure 2 would print `Brent, age 20, is a student.` Note

```
my %hash = ('name', 'Brent',
            'age', 20,
            'occupation', 'student'
           );
print "$hash{'name'}, age $hash{'age'}, ";
print "is a $hash{'occupation'}.\n";
```

Figure 2: Using hashes.

that hashes can be initialized with array literals; each pair of elements is taken to be a key/value pair. A more idiomatic way of writing the same code is exhibited in figure 3. The `=>` operator acts like a comma which also quotes whatever is

```
my %hash = (name => 'Brent',
            age => 20,
            occupation => 'student'
           );
print "$hash{name}, age $hash{age}, ";
print "is a $hash{occupation}.\n";
```

Figure 3: Using hashes idiomatically.

to its left; unquoted strings inside curly braces are automatically quoted.

The `keys` and `values` functions, when applied to a hash, return a list of the hash's keys and values, respectively. The `delete` function is used to delete entries from a hash, like this:

```
delete $hash{occupation};
```

Creative use of hashes can make many programming tasks much simpler — it is worth your while to learn how to use them.

Muffins!

- References ("pointers") are not needed for most small programs, but if you plan to write any larger projects in Perl, especially ones involving relatively complex data structures, I encourage you to read about them in chapter 8 of *Programming Perl* or in *perlreftut(1)* and *perlref(1)*.

[22] If you want to access the elements of a hash in some particular order, try sorting the keys.

- Many built-in functions for manipulating strings are available, such as `chomp`, `chop`, `chr`, `lc`, `uc`, `index`, `rindex`, `substr`, `reverse`, and `split`; see *perlfunc(1)*.
- You can easily insert multi-line literal strings into your Perl programs with "here-documents". See chapter 2 of *Programming Perl* or *perldata(1)*.
- Many more backslashed escape sequences are available in strings, including special ones, unique to Perl, that let you easily manipulate letter cases (`\u \l \U \L`).
- Several methods are available for quoting strings beyond simple single or double quotes, such as the `q//`, `qq//`, `qw//`, `qr//`, and `qx//` operators. See *Programming Perl*, chapter 2, or *perlop(1)*.
- There are quite a few more operators available, including the spaceship operator (`<=>`), the string repetition operator (`x`), and the ternary conditional operator (`?:`) — see chapter 3 of *Programming Perl*, or *perlop(1)*.
- You can grab multiple array or hash elements at once with slices. See *perldata(1)*.
- Interval notation can be used in array literals, *e.g.* `@array = (1, 5..10)`.
- Many built-in functions for manipulating arrays are available, such as `push`, `pop`, `shift`, `unshift`, `reverse`, `sort`, `join`, `splice`, `grep`, and `map` — see *perlfunc(1)*.

Projects

1. Write a "Hello, world" program in Perl! Go on, it'll be fun!
2. Write a script that starts out with a random list of integers, then prompts the user for a number, and prints out a sorted list of only those integers from the list that are greater than the number the user entered. You could do this the long, painful way, or you could learn how to use the `sort` and `grep` functions and do it with a few lines of code. Also, to read a scalar from standard input, just assign the special filehandle object `<STDIN>` to a scalar, like this:

   ```
   $value = <STDIN>;
   print "You typed $value.\n";
   ```

   More on I/O later, in section 6.[23]

[23] I am aware that this project is sort of dumb. It's hard to come up with interesting projects that don't use I/O, control structures, or regular expressions...

4 Important concepts

Now that you have a basis in basic Perl syntax and data types, it's important to digress and discuss some Important Ways Perl Sees Things, since they are quite different from the ways a lot of other programming languages see things. If you understand the concepts in this section, you will be well on your way to having a good grasp of Perl. If you don't understand the concepts in this section, you will simply be confused.

4.1 Truth and undef

What is truth? This is an important question, and one on which Perl has a definite opinion.[24] But first, a slight digression.

Perl has a special "undefined" value, often written `undef`.[25] Scalars which have been declared but not yet given an explicit value have the value `undef`, as do automatically created array elements, hash values looked up with a non-existing key, and pretty much exactly what you'd think would be undefined. You can test whether a particular scalar is defined by using the special `defined` function, which returns true iff its argument has any value *other* than `undef`. Also, you can use the special function `undef` to generate the undefined value. When used as a number, `undef` acts like 0; when used as a string, `undef` acts like the empty string. But of course it is really neither of these things. And now, on to truth.

There is no boolean type as such in Perl. Instead, Perl has a notion of the truth or falsity of any scalar. In particular:

- Zero (the number) is false.
- The empty string `''` and the string `'0'` are false.
- `undef` is false.
- Anything else is true.

Seems intuitive, right? Most of the time, it is. See if you can figure out what the code in figure 4 does.

```
my $v;
if ( $v ) { print 'abc'; }
$v = '0.0';
if ( $v ) { print 'def'; }
if ( $v + 0 ) { print 'ghi'; }
```

Figure 4: Truth.

[24] Although, unfortunately, not in a significant metaphysical sense.
[25] Think "pizza". But predictable pizza.

The answer is that it prints `def`. In the first `if` statement, `$v` has the value `undef`, which is false. In the second `if`, `$v` is the *string* `'0.0'`. The only strings that are false are the empty string, and the string `'0'`. Thus, since `$v` is neither of these, it evaluates to true. In the final `if` statement, however, `$v` is first converted to a number (resulting in 0), which is then added to 0 to produce the final result — the number 0 — which is tested for truth. Of course, the number 0 is false, so the `if` body is skipped.

4.2 Variadic subroutines and default arguments

Perl's subroutines and built-in functions are by default what is known as *variadic*, that is, they take a variable number of parameters. This is quite different from most programming languages, such as C or Java, in which subroutines must always have a particular number of parameters of particular types or else the compiler yells at you.[26] The parameters to a subroutine arrive in the special array `@_`[27], which of course is variable-length. If multiple arrays or hashes are passed as parameters to a subroutine, their elements are simply flattened into the argument list.

More importantly, many of Perl's built-in functions act on the "default scalar", `$_`[28]. Input from standard input or from a file goes to `$_` if not explicitly assigned anywhere else; functions such as `split`, `chomp`, and `print` act on `$_` by default, as do pattern matches (see section 7). If you see a function that looks like it is missing a parameter, or has none, it is probably acting on `$_`. Figure 5 gives an example of using the default scalar.

```
while ( <STDIN> ) {
  $_ = reverse $_;
  print;
}
```

Figure 5: Using the default scalar `$_`.

How does this code work? First of all, one line of input at a time is read from standard input, and since it is not assigned anywhere else, it is assigned to `$_`. (Note that when EOF is reached, the input operator `<STDIN>` returns `undef`, which conveniently evaluates to false, breaking out of the loop.) The first line of the loop reverses the contents of `$_`; then, since `print` has no arguments, it prints `$_` by default. And just to show you how cool Perl is, I will mention that the following line of code accomplishes exactly the same thing as the code in figure 5:

```
print scalar reverse while <STDIN>;
```

[26] Don't even talk to me about C's va_lists, which are an ugly, cheap hack.
[27] Pronounced "them".
[28] Pronounced "it".

In addition, certain functions that expect arrays, such as `shift`, act by default on `@_` if it exists.

4.3 Context

Perl has a notion of two different "contexts" in which evaluations can take place: scalar context, and list context.[29] Things behave differently depending on what context they're in, so it's important to know the difference. Essentially, an expression is in "scalar context" if the value it generates will be used as a scalar; similarly, an expression is in "list context" if the value it generates will be used as a list (*i.e.* an array or hash). The difference is illustrated in figure 6.

```
my @things = ('peach', 2, 'apple', 3.1415927);
my @thing1 = @things;
my $thing2 = @things;
my ($thing3) = @things;
print "\@thing1: @thing1\n";
print "\$thing2: $thing2\n";
print "\$thing3: $thing3\n";
```

Output:

```
@thing1: peach 2 apple 3.1415927
$thing2: 4
$thing3: peach
```

Figure 6: Context.

In the first assignment, `@thing1 = @things`, `@things` is evaluated in a list context, since its value will be used to assign to an array. Evaluating an array in list context simply returns the entire array. Thus this line does exactly what you might think: it creates `@thing1` as a copy of `@things`. (As an aside, note that it really is a copy: Perl doesn't use any of those silly Java-esque reference semantics. If you want reference semantics, you have to do it yourself.[30])

But weird things start happening in the second assignment, `$thing2 = @things`. In this case, since its value will be assigned to a scalar variable, `@things` is evaluated in a scalar context. As it turns out, evaluating an array in a scalar context results in the array's length. Thus `$thing2` is assigned 4, the length of `@things`.

What about the third assignment, `($thing3) = @things`? You can probably guess by now that `@things` is evaluated in a list context, since the result will be assigned to the array `($thing3)`, an array with a single element. So the

[29] Technically there is also a "void context", but it's really a special sort of scalar context.

[30] Sigh... the lies are just flying, aren't they? Perl uses reference semantics in two situations: foreach loops (see section 5.2), and subroutine parameters (see section 5.3).

question is, what does Perl do when you try to assign an array of length m to another array of length n, where m and n are not the same? It turns out that the array being assigned to just gets assigned the first n elements of the other array. So in this case, `$thing3` is assigned the first element of `@things`, namely `'peach'`.

Variations on this behavior include:

```
my ($first, $second) = @array;
```

`$first` and `$second` are assigned the first and second elements of `@array`, respectively.

```
my ($first, @rest) = @array;
```

`$first` is assigned the first element of `@array`, and `@rest` is assigned everything else.

```
my (@copy, $dummy) = @array;
```

I know what you're thinking, and it's wrong. In this case `$dummy` is *not* assigned the last element of `@array`; in fact, `$dummy` ends up with `undef`, since `@copy` "eats up" the entire available array, leaving nothing to be assigned to `$dummy`. Remember, arrays are variable-length, so `@copy` has no reason to stop.

If you want to force something to be evaluated in scalar context, you can use the special `scalar` function. For example, you can compute the length of an array with `scalar @array`.

4.4 TMTOWTDI

The title of this section is an abbreviation of the Perl mantra: There's More Than One Way To Do It. You see, Perl was first developed by a UNIX geek[31], not by experts in programming language theory. Thus the strengths of Perl are its extreme diversity, flexibility, and plain usefulness, rather than how well it lends itself to verifying static type safety[32], or anything like that. Part of the fun of Perl is figuring out not just how to do something, but how to do it elegantly, and with the least amount of code such that it is still readable.

5 Control structures

For the most part, Perl control structures have similar syntax to the corresponding structures in C or Java, with the notable exception that curly braces must *always* be used to enclose conditional and loop bodies, even if the bodies only consist of a single line.[33]

[31] Larry Wall, in 1987.

[32] Bleagh.

[33] Actually, this restriction is dropped when you use conditional and loop constructs as infix operators, which is kinda nifty...

5.1 Conditionals

The syntax for `if` statements in Perl is as follows:

```
if ( test ) {
  # do stuff
} elsif ( test2 ) {
  # do other stuff
} else {
  # do other other stuff
}
```

Note the strange spelling of `elsif`! Of course, you may use as many `elsif` clauses as you want (including none at all), and the `else` clause is always optional. Remember that the curly braces are required, even if the `# do stuff` is only a single line.

There is also an `unless` statement, which executes its body if its test is false. Note that you may use an `else` clause with `unless`, but there is no such thing as `elsunless`.

5.2 Loops

The syntax of `while` and `for` loops is exactly the same as in C or Java, except that, of course, curly braces are required. The `until` loop is also provided, which continues executing as long as its test is false (*i.e. until* it becomes true).

Perl also provides the nifty `foreach` loop.[34] The syntax is as follows:

```
foreach var (array) {
  # do stuff
}
```

It works by setting `var` to successive values of `array` each time through the loop. If `var` is omitted, `$_` is used instead. Figure 7 shows two examples of using a foreach loop.

```
foreach (reverse ('Blastoff!', 1..10) ) {
  print "$_\n";
  sleep(1);
}

foreach $key (keys %hash) {
  print "$key -> $hash{$key}\n";
}
```

Figure 7: `foreach`-style loops.

[34] Actually, `foreach` is just an alias of `for`, which has two different syntaxes (syntacies?). Just use whichever name you think makes things more readable in a particular situation.

The first prints out a launch countdown; the second illustrates the standard method for printing out a hash, or in general doing something to each element of a hash. (If `(sort keys %hash)` is used instead, the key/value pairs can be printed out sorted by key.)

Note that the index variable is actually an implicit reference to the elements of the array being iterated over, so that the actual array contents can be modified by modifying the index variable inside the loop. For example:

```
foreach $price (@price_list) {
  $price *= 0.8;      # All items 20% off!
}
```

5.3 Subroutines

Subroutines can be defined with the syntax

```
sub subname {
  # do stuff...
}
```

where `subname` is the name you want the subroutine to have. Note that there is no parameter list,[35] since all the parameters to subroutines always come in the special parameter array `@_`. It is common to grab parameters (if there are any) like this:

```
my ($param1, $param2, ...) = @_;
```

Parameter passing is done with reference semantics; in other words, you can actually change the values of parameters if you directly modify the contents of the `@_` array. However, copying parameters into local variables enforces value semantics; for example, in the above example, modifying `$param1` would have no effect on the original parameter. The example in figure 8 should hopefully make all of this clear.

Subroutines can also be used as functions which return scalar or list values; the `return` keyword works exactly the same as in C and Java. Additionally, the `wantarray` function can be evaluated to determine whether the subroutine was called in a scalar or list context; it returns a true value if the subroutine was called in a list context, and false if in a scalar context.

Subroutines are called just as in C or Java, with parentheses to enclose a comma-separated parameter list. The parentheses are optional for subroutines which take no parameters.[36]

Muffins!

- Since Boolean operators use short-circuit evaluation, they can be used as conditional statements. Think about it.

[35] Unless you want there to be...

[36] Or for subroutines that do take parameters when you don't feel like giving them any! Remember, subroutines are variadic.

```
sub swap {           # WRONG! This does not actually swap
  my ($a, $b) = @_;  # the parameters - it just swaps $a
  my $temp = $a;     # and $b, which are copies of the
  $a = $b;           # parameters and go away when the
  $b = $temp;        # subroutine exits
}

sub swap {           # That's more like it!
  my $temp = $_[0];  # Access the parameters directly as
  $_[0] = $_[1];     # elements of @_
  $_[1] = $temp;
}
```

Figure 8: Parameter semantics.

- Conditional and loop keywords may be used as infix operators, for example:

  ```
  print "GAME OVER" if $game_status == 0;
  print $k++ while $k < 10;
  ```

  These forms can greatly increase readability of code if used well.

- It is possible to specify subroutine prototypes, in order to have a bit more control over what sort of things are passed as parameters. See chapter 6 of *Programming Perl* or *perlsub(1)*.

- Using references and anonymous subroutines, it is possible to make "closures"[37], and from there it is not too hard to make the jump to function generators (functions that make new functions) and function templates (a method for easily generating a whole family of similar functions). Which is totally ninja. See chapter 8 of *Programming Perl* or *perlref(1)*.

6 I/O

A file can be opened using the `open` function, which has the following syntax:

```
open ( filehandle, mode, filename );
```

`filename` can be any scalar value, *e.g.* a scalar variable, a double-quoted string with variables interpolated into it, *etc.* Note that filehandles have no special character in front of them. `mode` is a string specifying the mode in which the file should be opened; table 1 lists some of the available modes. (Pipes will be covered in more detail in section 8.) You may often see the mode string and

[37] If you've taken a programming languages course you'll know what that is — it's basically some code along with the environment that was in effect at the time it was defined.

Mode string	File mode
<	read
>	write
>>	append
|-	output pipe
-|	input pipe

Table 1: File modes.

filename concatenated into a single parameter. This is an older syntax for `open` which is still supported; its only drawback is that it does not allow you to open files which actually begin with one of the mode strings.

Closing files is easy:[38]

```
close ( filehandle );
```

Writing to a filehandle can be accomplished by passing the filehandle as the first argument to the `print` function, WITHOUT any commas or parentheses to separate the arguments:

```
print filehandle "Print me!\n";    # Right!
print filehandle, "Erm...\n";      # WRONG
print (filehandle, "Uh...\n");     # WRONG
```

The special pre-opened filehandle `STDOUT` corresponds to standard output. The `print` function prints to `STDOUT` by default. The special pre-opened filehandle `STDIN` corresponds to standard input.[39]

Reading from a filehandle is accomplished by placing the filehandle inside angle brackets (the "line input operator", or, often, the "angle operator"). When evaluated in scalar context, the next line (up to and including the end-of-line character) is returned. When evaluated in list context, the entirety of the remaining file is returned in an array, one line per element.

```
open ( inFILE, $fileName );
my $line = <inFILE>;           # get the first line
my @rest = <inFILE>;           # get the rest of the file
```

Conveniently, a line read from an angle operator will always evaluate to true (since it always includes the end-of-line character), until EOF is reached, at which point `undef` is returned, which evaluates to false. So you can do something like the following:

```
while ( $line = <inFILE> ) {
  # do something with $line
}
```

[38] Actually, it's even easier than that: files close automatically when their filehandle goes out of scope. But I didn't tell you that.

[39] Three guesses what `STDERR` is.

If an angle operator is the only thing inside a while loop test, it will put each line read into the default scalar, `$_`. This *only* works inside while loop tests, but is an extremely common idiom:

```
while ( <inFILE> ) {
  # do something with $_
}
```

Muffins!

- Formats provide an incredibly easy way to print reports, tables, charts, or other similar types or repetitive output. See chapter 7 of *Programming Perl* or *perlform(1)*.
- The empty angle operator, `<>`, provides input from files listed on the command line, or from `STDIN` if no command-line arguments were given. (Not accidentally, this is exactly how many standard UNIX filters[40] function, so that they can operate on files, or as part of a pipeline.)
- The `printf` function is quite similar to the standard C function of the same name, and provides formatted output. See *Programming Perl*, pp. 766-7, or *perlfunc(1)*.

Projects

1. Write a utility to provide frequency counts for a file specified by the user (*hint*: use a hash). As an extension, you might look up normal letter frequencies for the English language, and have your utility say whether it suspects the processed file to be a Caesar cipher. As a further extension, you might have your utility guess at the correct decryption of a suspected Caesar cipher.[41]
2. Write a script that searches through a user-specified file and prints all the maximal palindromes that it finds.

7 Pattern matching with regular expressions

Regular expressions[42] and pattern matching are the bread and butter of Perl. Pattern matching allows you to search for, replace, capture, or otherwise mess around with arbitrarily complicated patterns in strings, and is one of the major

[40] *e.g. grep*, *sed*, *etc.*

[41] You might be surprised to know that even for messages of as few as 100 characters, such methods are quite accurate in automatically decrypting Caesar ciphers!

[42] I have been informed that Perl "regular expressions" are to theory of computation "regular expressions" as rocket launchers are to pocket knives. So if you've taken theory of computation and find yourself exclaiming "Hey! That's cheating!" at various points, well, you've been warned. Remember, Perl was invented by geeks, not theoreticians.

reasons that Perl is so good at things that require lots of string processing, such as CGI programming. Plus they are totally ninja and look like modem noise to the uninitiate:

```
s/\b([a-m]{4,})\b/*$1*/gi;
```

That line of code highlights all words of length at least 4 that only use letters from the first half of the alphabet, by placing asterisks on either side of them. Really.

7.1 Concepts

If you've seen regular expressions before, then you can probably skip this section...

A regular expression is simply a pattern which implicitly generates a family of strings, expressed in a special notation. For example, such a pattern might be "one or more copies of the character 'a', followed optionally by the character 'z' ".[43] This pattern generates an infinite family of strings, including 'a', 'az', 'aa', 'aaz', 'aaaaaaaaz', *etc.* Another such pattern might be "either the exact characters 'cat', or the exact characters 'dog' ".[44] This pattern generates a family of strings with exactly two elements, namely, 'cat' and 'dog'.

Given some pattern and some string, a naturally arising question is: is this string generated by this pattern? It turns out that Perl has lots of complicated machinery to efficiently[45] answer this question. Given a pattern (regular expression) and a string, we say that the string *matches* the pattern if some substring of the string is generated by the pattern. For example, the strings 'a', 'aaaz', 'bazaar', and 'cat' would all match the first pattern given above; 'cat', 'dog', 'caterpillar', and 'bulldogs' (but NOT 'dodge' or 'cast') would match the second pattern.

7.2 Pattern-matching and binding operators

There are two pattern-matching operators,[46] which by default operate on ("bind to") the default scalar, `$_` (although they can be bound to other string expressions with a binding operator — more on those later):

- The matching operator, `//`, returns true iff the string it is bound to matches the regular expression it contains. For example:

```
$_ = 'caterpillar';
print "Found 'pill'\n" if /pill/;
print "Found 'car'\n" if /car/;
```

[43] If you're curious (read: impatient), the Perl regular expression for this pattern would be '`a+z?`'.

[44] The Perl regular expression for this would be '`cat|dog`'.

[45] Well, certain patterns might take exponential time to match (or not to match), but that's not Perl's fault.

[46] Unless you count `tr///`. But it doesn't really do any pattern matching. Read more about `tr///` in the "Muffins" section.

This would only print `Found 'pill'`, since 'car' is not an exact substring of 'caterpillar'.

- The substitution operator, `s///`, tries to match the string it is bound to with the regular expression between the first pair of slashes. If it succeeds, it replaces the first matched substring with whatever is in between the second pair of slashes, and returns true. Otherwise, it does nothing to the string and returns false. For example:

```
$_ = 'caterpillar';
print "\$_ is now $_\n" if s/pilla/waule/;
print "\$_ is now $_\n" if s/car/bus/;
```

  This would print `$_ is now caterwauler`; the second print statement would not execute since the replacement `s/car/bus/` fails (since in particular the match `/car/` fails).

Note that both pattern-matching operators act like double-quoted strings, in that variables can be interpolated into them.

There are two "binding operators", `=~` and `!~` (equals-tilde and bang-tilde), which both serve to bind a string expression on their left to a matching operator on their right. The only difference is that `=~` returns the same logical value that the match on its right returns, whereas `!~` returns its logical negation (*i.e.* `!~` could be thought of as the "does not match" operator).

Appending the character 'i' to the end of either match operator causes the match to be case-insensitive (matches are case-sensitive by default). Also note that by default, the substitution operator replaces *only* the first (leftmost) match it finds, even if multiple matches exist. Appending the character 'g' to the end of the substitution operator causes it to replace *all* matches. See figure 9 for an illustration of these options.

```
my $str = 'Alfalfa';
$str =~ s/al/-/;        # $str is now 'Alf-fa'

$str = 'Alfalfa';
$str =~ s/al/-/i;       # $str is now '-falfa'

$str = 'Alfalfa';
$str =~ s/al/-/gi;      # $str is now '-f-fa'
```

Figure 9: Using `i` and `g` options with `s///`.

One other thing — the binding operators have rather high precedence, so be careful about parenthesizing things. For a table of operator precedence, see *perlop(1)*.

7.3 Metacharacters, metasymbols, and assertions, oh my!

Most characters in a regular expression simply represent themselves, as we've seen in previous examples. However, there are some characters, called *metacharacters*, that have special meanings. They are as follows:

```
\ | ( ) [ { ^ $ * + ? .
```

Metacharacters can be turned into normal characters, however, by prepending a backslash. Thus if you wanted to match an actual '+' character, you would write `/\+/`. As it turns out, backslash works in reverse on many normal characters: prepending a backslash turns many normal characters into *metasymbols* with special meanings.

One way to think about regular expressions is that each element is making some sort of assertion about strings that the pattern matches. For example, the character '`a`' simply asserts that if a string matches, it has the character 'a' at the position in question. However, not all assertions are "equal" — some, like the character '`a`', have "width" — that is, they match an actual portion of a string. Such assertions are called *atoms*.[47] Other assertions, however, have no width; for example, the assertion that "this position in the string has a lowercase character on either side of it"[48] is such a zero-width assertion. It takes up no "space" in the string, but can certainly be tested for truth. Perl regular expressions support such zero-width assertions as well, and they are often referred to simply as *assertions*.

7.4 Metacharacters

The first set of metacharacters we will look at are called *quantifiers*. They each apply to the atom immediately preceding them, and indicate how many repetitions of the atom are allowed. The quantifiers and their meanings are shown in table 2.

Quantifier	Meaning
`?`	0 or 1 time
`*`	0 or more times
`+`	1 or more times
`{`m`}`	Exactly m times
`{`m`,}`	at least m times
`{`m`,`n`}`	at least m but not more than n times

Table 2: Quantifiers.

Thus we can now write:

[47] More technically, an atom is anything that is quantifiable, since `(a*)` is an atom that could have zero width. But the intuition about width is usually good enough.

[48] Note that when talking about string matching, a "position" is thought of as being in between two adjacent characters.

```
print "Matches pattern 1\n" if /a+z?/;
```

Note that by default, quantifiers are "greedy", that is, they try to match as much as possible (such that the whole match still succeeds). To make a quantifier match as little as possible, append a question mark to it.

The `|` (pipe) metacharacter specifies "alternation", *i.e.* a choice between several alternative patterns. For example, `cat|dog|mouse` would match exactly one of 'cat', or 'dog', or 'mouse'.

Square brackets indicate a "character class", and match exactly one of the characters listed inside them. For example, `[aeiou]` matches exactly one vowel; `[aeiou]+` matches a string of one or more arbitrary vowels. If the first character following the opening bracket is `^` (carat), then it becomes a negated character class and matches any characters *not* listed. For example, `[^aeiou]+` matches any string of length one or more that contains no vowels. Character classes may also contain ranges; for example, `[A-Z0-9]` matches any uppercase letter or any digit.

`^` (carat) is an assertion which is true at the beginning of a string; `$` is the corresponding assertion to check for the end of a string. Thus `^cat` would match 'cat' and 'caterpillar' but not 'concatenate'; `cat$` would match 'cat' and 'bobcat' but not 'caterpillar'; `^cat$` would match 'cat' and nothing else.

Finally, the special metacharacter `.` (period) matches any character except newline. For example, `/near.*far/` matches strings which contain the words 'near' and 'far', separated by any number of characters in between.

7.5 Grouping and capturing

How are we to encode the pattern "one or more copies of the string 'ab' ", *i.e.* the pattern that generates the strings 'ab', 'abab', 'ababab', and so on? `ab+` will not work, since quantifiers only apply to the most recent atom; thus, in this case the `+` only applies to the `b`, and the pattern matches strings like 'ab', 'abb', 'abbb', and so on. Thankfully, parentheses serve to group several atoms together into one larger atom. So, the pattern that we are looking for is `(ab)+`.

A pair of parentheses also has the property that it "captures" whatever piece of a string matches its contents. These captured pieces are available after a match in the special variables `$1`, `$2`, ... — the first set of parentheses places what it captures into `$1`, the second into `$2`, *etc.*[49] This is a very useful mechanism for extracting information from a string, as figure 10 illustrates.

Figure 10 also shows what a pattern match returns in a list context: it conveniently returns a list of the captured subpatterns, *i.e.* the list `($1, $2, ...)`. As it turns out, the `s///g` operator also returns more than just 'true' or 'false' in scalar context, too; in particular it returns the number of matches that it made.[50]

[49] In the case of nested parentheses, order is determined by the order of the opening parends of each pair.

[50] Which, you'll note, evaluates to false when the match failed, and true otherwise. Handy, no?

```
$data = <dataFILE>;
$data =~ /name: (.*) age: (.*) occupation: (.*)/i;
print "$1, age $2, is a $3.\n";

my ($vowelpart, $conspart) = /([aeiou]+)([^aeiou]+)/;
```

Figure 10: Capturing.

Often, you may want to group certain things without also capturing: in this case you can use the grouping construct `(?:...)`, where the ellipsis is replaced by whatever you wish to group.

7.6 Metasymbols

Table 3 lists some of the most commonly useful metasymbols.

Symb	Atom?	Meaning
`\0`		Match the null character (ASCII NUL).
`\`n		Match the nth previously captured string.
`\A`	N	True at the beginning of a string.
`\b`		Match the backspace character (in a character class).
`\b`	N	True at a word boundary.*
`\B`	N	True when not at a word boundary.
`\d`		Match any digit (*i.e.* `[0-9]`).
`\D`		Match any non-digit (*i.e.* `[^0-9]`).
`\n`		Match the newline character.
`\s`		Match any whitespace character (*i.e.* `[␣\t\n]`).
`\S`		Match any non-whitespace character.
`\t`		Match the tab character.
`\w`		Match any "word" character (*i.e.* `[A-Za-z0-9_]`).
`\W`		Match any non-word character.
`\z`	N	True at the end of a string.
`\Z`	N	True at the end of a string, maybe before `\n`.

Table 3: Common metasymbols.

* A "word boundary" is defined to be a position between `\w\W` or `\W\w`.

Muffins!

- There are several more options which can be put at the end of the pattern-matching operators to further customize their operation. One cool one in particular is the `e` option to the `s///` operator, which causes the replacement portion to be evaluated as Perl code, thus allowing you to dynam-

ically generate a replacement based on the results of the match. For example:

```
s/(-?\d+)/$1 + 2/eg;  # add 2 to any integers found
```

See *Programming Perl* pp. 150 and 153, or *perlop(1)*.

- Frontslashes are not the only characters which may be used as delimiters for the pattern-matching operators; sometimes it is nice to have different delimiters, especially when your regexp contains frontslashes (if the delimiters are not frontslashes, you don't have to backslash-escape the frontslashes in your regexp). See *Programming Perl* p. 63, or *perlop(1)* (search for "Quote-like Operators").

- The transliteration operator, `tr///`, lets you specify a set of characters to replace, and their individual replacements. It also lets you easily do things like count the number of occurrences of a certain set of characters, condense repeated sequences of the same character into a single character, delete certain characters from a string, and many other randomly useful operations. See *Programming Perl*, pp. 155-7, or *perlop(1)*.

- There are many more metasymbols as well as extended grouping symbols (although I did cover all the metacharacters). In particular you can do crazy stuff like include arbitrary Perl code in the middle of a regexp, although why you would ever need to is beyond me. See *Programming Perl*, pp. 160-4, or *perlre(1)*.

- Since matching operators interpolate things like[51] double-quoted strings, you can match against dynamically generated regexps. You can even "compile" them using the regexp compile operator, `qr//`, so that they run just as fast as a regexp known at compile-time. See *Programming Perl* pp. 190-7 or *perlop(1)* (search for "Regexp Quote-Like Operators").

- You'll note that nothing I've said really answers the question of exactly which match is found when a string could match a pattern in multiple ways. The simple answer is "the leftmost one"; the real answer is, "it's a completely deterministic result of how the pattern-matching engine matches patterns[52]; read pp. 197-202 of *Programming Perl* or see *perlre(1)*."

Projects

1. Write a script that changes commonly misspelled words to their mispelled versions, and then run it on your friend's paper when they leave themselves logged in. Hee hee!

[51] Well, almost like...

[52] Namely, with a nondeterministic finite-state automaton. Don't ask me what that is.

2. Here, once again, is the sample regexp I gave at the beginning of this section:

   ```
   s/\b([a-m]{4,})\b/*$1*/gi;
   ```

   It makes sense now, doesn't it? Pat yourself on the back, and then make up your own complicated-looking regexp to show off to your friends.

3. Write a script to grab the WSO homepage source (using `lynx -source`) and extract the current temperature. Set it to run every time you start a bash process. Better yet, put it in your prompt![53]

4. Solve the following cryptogram: `ABCDE BDCAE`. If you can't solve it on your own, write a script to solve it by searching through a dictionary for words that fit the given pattern. I should warn you, however, that it would be easy to write a correct solution that took forever to run. With a little cleverness it's possible to write a script that finds the solution in a few seconds. If you've already seen this particular cryptogram, I apologize. Wait, no I don't...

5. Write a script to solve the word-ladders game. Given two words of the same length, the word-ladders game is to find the shortest sequence of words that "transforms" the first into the second, *i.e.* a sequence of words in which any two adjacent words differ only in a single letter. For example, a solution for `POLE-MINT` would be `POLE => PILE => PINE => PINT => MINT`.

8 Interacting with UNIX

Of course, if you're using Perl to do system administration-type things, you'll want to know how to interact with the OS through Perl. Thankfully, since Perl was originally developed on a UNIX platform, it's really easy to do.[54]

First of all, any command-line arguments given when your script was run are available in the special array `@ARGV`.

Next, the `system` function lets you run arbitrary shell commands. On UNIX-like systems, it forks a copy of `/bin/sh` and passes its argument along for processing.[55] On other systems it probably does something equivalent. It then waits for the forked process to finish, and returns the return value of the process as reported by *wait* (to get the actual return value, divide by 256).

If you want to do something with the output of a shell command, however, you should use backtick-quoting (or an input pipe, which I'll describe in a

[53] By putting it in single backquotes.

[54] Of course, this applies to all UNIX-like systems, including BSD, Linux, Mac OS X, *etc...* I've never used Perl on a non-UNIX system, so I'd be interested to know how well the stuff in this section works.

[55] Technically, if there are no pipes or redirections or anything like that, it splits up the arguments and calls `execvp` directly. But, same difference.

minute). Any string enclosed in backticks will be run as a shell command, and the output of the command will be returned, either as a huge string in scalar context, or as an array with one line per element in list context. Figure 11 illustrates the `system` call and backtick-quoting.

```
print 'Enter a command to run: ';
my $command = <STDIN>;
system ( "sudo $command" );

my @file = `ls -l`;
foreach (@file) {
  print "Found a world-writable file\n" if /(r|-)w(x|-) /;
}
```

Figure 11: Running shell commands.

I recommend not writing scripts like the first part of figure 11, especially if they run setuid root...

Also, it's a little hard to tell with this particular font, but the `ls -l` is in backticks, *i.e.* the character located to the left of '1' on most standard keyboards.

I mentioned in section 6 that it is possible to use the `open` function to create input or output pipes, which can be an incredibly useful tool. Essentially, you can run an arbitrary shell command and read its output exactly as if it were a file open for reading (in the case of input pipes), or provide input to a command exactly as if it were a file open for writing. Instead of providing a filename to the `open` function, you provide an arbitrary shell command. For example,

```
open ( inPipe, '-|', "decrypt $filename" );
```

would allow you to read the contents of `$filename`, assuming that `decrypt` were some filter which decrypted its contents. Output pipes work in a similar fashion. For example,

```
open ( outPipe, '|-', "gzip > $outfile" );
```

would cause anything written to `outPipe` to be compressed before it was written to disk.

Muffins!

- There are a number of built-in "unary file-test operators", which test various properties of files. For example, you can test for the existence of a file like so:

  ```
  print "$file exists\n" if ( -e $file );
  ```

  For a list of all the available file test operators, see *Programming Perl*, p. 98, or *perlfunc(1)*.

- Many standard UNIX utilities are also available as Perl functions, *e.g.* `chmod`, `mkdir`, `symlink`, *etc.* See *perlfunc(1)*.
- The `glob` function returns a list of files in the current directory that match a specified pattern; it uses standard filename wildcards like `*`, `?`, ∼, braces for alternation, *etc.* See *Programming Perl*, p. 727, or *perlfunc(1)*.

Projects

1. Write a script which recursively processes a directory subtree, replacing all "strange" characters in file names (*i.e.* ones that a web browser would choke on) with underscores.[56]
2. Implement a shell-wrapper in Perl, which displays a shell-like prompt and passes most things through to a shell, but also provides some extra features, such as context-sensitive tab completion, fast cycling through a circular list of directories, or other such 'extras' you come up with.

And now for some extra muffins that didn't really fit anywhere else:

Muffins!

- I may at some point in the future write an appendix on CGI programming with Perl. Until then, however, you're on your own. Well, almost. I will say that there exist many nice modules that do a lot of the dirty work for you. In particular I have found the `CGI` module (see *CGI(3)* or *CGI(5)* or an appendix to *Programming Perl* or documentation on CPAN) and the `HTML::Template` module (see CPAN) to be incredibly helpful and time-saving. If you're interested I'd also be happy to let you look at a few scripts I've written using these modules.
- Apparently, Perl supports such things as object-oriented programming, modules, packages, *etc.* I've never really had occasion to use such features, but if you're interested you might want to check them out. See chapters 10-15 of *Programming Perl*, and also *perlboot(1)*, *perltoot(1)*, *perltootc(1)*, *perlobj(1)*, *perlbot(1)*, and *perltie(1)*.
- There are so many more muffins out there I don't know where to begin. If you're at all serious about learning and using Perl well, I suggest you get your own copy of *Programming Perl*. I think it's about $40 or $50, not cheap, but it's an excellent book and a good investment.

[56] And then give me a copy, so I can use it on my mp3 collection...