

Introduction

Perl stands for “practical extraction and report language”. Arguably, it was designed as a replacement for sed, awk, tr, grep and other information processing tools, but it has grown beyond that in scope. Perl5 saw the advent of real programming language constructs such as object interfaces, enhanced typing, enforced scoping, internationalization support, and the beginnings of threading. The author of Perl (Larry Wall) sometimes claims that it stands for the “Pathologically Eclectic Rubbish Lister” while its fans sometimes say that it’s “The Duct Tape of the Internet”. Many refer to it as a procedural language, but others (myself included) ascribe the dubious honor of pseudo-object-oriented. Perl’s motto is “There’s more than one way to do it”. Perl is a lazily-typed language with late (execution time) binding. It has an enormous number of modules available for it; almost all of them are object-oriented (but some still have perl4-esque interfaces that pollute the global namespace).

Hello World in Perl:

```
#!/usr/bin/perl
# Perl Hello World
print "Hello, world\n";
```

The first line identifies the interpreter used by this script (a UNIX-ism). The second line is a comment, and the third line prints a string to the standard output. Perl is not compiled, but rather interpreted, and like many things in computing it’s not so simple to say such a thing. Perl is actually compiled, but this compilation takes place at execution time (and, yes, you can dump this compilation to a “binary” file for later execution if you really want to). Some refer to interpreted languages as “scripting languages”, and while Perl definitely descends from this lineage, it really has outgrown such a label.

Primitives in Perl

Perl has very few, and yet very powerful primitives:

Scalars

Scalars can contain integers, floats, doubles, strings, objects and references. Each scalar contains just one piece of information. The term ‘scalar’ is inappropriate for Perl’s use of these variables, but it sticks because of Perl history. All scalars are denoted by a dollar sign:

```
$i = 15;
$num = 12.3;
$name = "Craig Kelley";
$webserver = new HTTP::Daemon();
$copy_of_webserver = \ $webserver;
```

What about mixing types? How does a string plus a double make sense? Again, Perl has lazy typing and will go ahead do what it thinks you want it to do. It will attempt to automatically cast variables to perform the appropriate operation; C and C++ do this to an extent (try multiplying an int by a double sometime), Java tries to never do this, and different computer scientists have varying levels of contempt or praise for such practices. Regardless, here are some common Perl scalar expressions and their results:

```
print 1 + "dog"; # result is '1'
print 1 + 1.5;   # result is '2.5'
```

Arrays

Arrays in Perl are sometimes called vectors in other languages. They are sequential collections of scalars (and remember that scalars can be objects and references, so you can have an array of

objects or array of arrays). Arrays are denoted by an 'at' sign and begin at index zero (like C/C++):

```
@numbers = (1, 2, 3.5, 3.14159, 42, (5/3), (63%4));
@words = ("one", "two", "three", "fo" . "ur");
@refs = ($copy_of_webserver, \$name);
@objects = ($webserver, new IO::File("musik.mp3"));
$arrayref = \@numbers;
$arrayref2 = [ "one", "two", "three"]; #anonymous
$arrayref3 = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ];
@a_of_arefs = (\@numbers, \@words, \@refs, $arrayref);
print $a[0]; # prints the first element in @a
```

You'll note that when we accessed a variable inside the array, the scalar context was used *on the array*; this is because the result of the expression was a scalar, not an array. Here are some auto-casting examples Perl does with arrays:

```
@animals = ("dog", "cat", "horse");
print 1 + @animals; # result is '4'
print @animals; # result is "dogcathorse"
```

The reason why is, again, context. Perl sees you adding a number to an array, so it assumes that you want the scalar form of @animals, which for an array is the number of items inside the array. Leaving out the addition, the print function assumes that you want to see the contents of the array, instead of the number of elements instead. This is related to (but is not a good example of) object-oriented concepts commonly known as operator overloading and polymorphism. If this doesn't make sense, that's OK, you'll get used to it over time.

Hashes

Hashes in Perl are called dictionaries or associative arrays in other languages. They are randomly ordered collections of scalars (and remember, again, that scalars can be objects and references, so you can have a hash of objects or hash of arrays of hashes of object references containing scalars). Hashes are denoted by the percent sign:

```
%friends = ( "Homer" => [ "Marge", "Moe"],
             "Buzz" => [ "Woody", "Potato Head"]);
@an_array = ("Kenny", "Kyle", "Cartman");
$friends{"Stan"} = \@an_array;
$friends{"Bender"} = ["Lela", "Fry", "Zoidberg"];
@people = keys %friends;
foreach $person (@people) {
    print $person . "'s friends are: ";
    $buddies = $friends{$person};
    foreach $buddy (@$buddies) {
        print "$buddy ";
    }
    print "\n";
}
```

The output of the above looks like this:

```
Buzz's friends are: Woody Potato Head
Stan's friends are: Kenny Kyle Cartman
Homer's friends are: Marge Moe
Bender's friends are: Lela Fry Zoidberg
```

Notice that the keys were chosen randomly. They were placed into the hash in this order (Homer, Buzz, Stan, Bender) but they came out in this order (Buzz, Stan, Homer, Bender); you need to explicitly sort the keys if you expect some sort of order to pertain to them. I also demonstrated the anonymous constructor for arrays above ([]); it's a shortcut so that you don't have to name an

array, but just create one that is tied to a reference instead. There is an anonymous hash operator as well:

```
$scalar = {
    "Homer" => ["Marge", "Moe"],
    "Kenny" => ["Kyle", "Cartman"]
};
```

Abstract Data Types (ADT)

With the three basic types given above, one can create very complicated data structures. Representations that would take pages of C code can be expressed in a few lines of Perl:

```
$users{"Craig"} = {
    "Name" => "Craig Kelley",
    "Job" => "IS",
    "Family" => [ "Jenny",
                 "Kyle" ],
    "Managers" => [$users{"Paul"},
                  $users{"Joe"}],
    "Degrees" => new Degree("CS")
};
```

So, this fictional collection of users is contained in a hash called %users, and the above entry for "Craig" has his full name (a scalar), his job (another scalar), his family (an anonymous array), his managers (references to other places in the %users hash) and his degree (a new instance of a Degree object, created by passing in an argument for "CS").

Dereferencing ADTs in Perl is done with the arrow operator (->). If I wanted the full name of Craig's first manager, it would look like this:

```
print $users{"Craig"}->{"Managers"}->[0]->{"Name"};
```

Note that the type of grouping denotes the primitive data type: {} for hashes and [] for arrays. There is another dereferencer for subroutines and functions, but let's leave that alone for now. I could also explicitly dereference it a piece at a time like this:

```
$craig = $users{"Craig"};
$craigs_managers = $craig->{"Managers"};
$first_manager = $craigs_managers->[0];
$name_of_first_manager = $first_manager->{"Name"};
```

Looks like object-oriented code, no? Well, it *is*, because this is basically how Perl manages object-oriented systems. If you need to debug ADTs, the Data::Dumper module often comes in handy:

```
use Data::Dumper;
print Data::Dumper->Dump([ $some_variable ]);
```

Which will print out the entire data structure to the screen.

Casting

If you have a reference to an array, or a reference to a hash, you can cast them to real arrays and hashes using this syntax:

```
$array_ref = [1, 2, 3, 4];
$hash_ref = { "joe" => 1, "mary" => 2 };
@real_array = @{$array_ref};
%real_hash = %{$hash_ref};
```

Subroutines & Functions

Perl's subroutine or function system is simple, even too simple for strict enforcement of type information. A subroutine is created using the keyword `sub`, and accepts an array of arguments, while returning either a scalar or an array of results. Most Perl subroutines just return scalars, and simply pass references back if more information is needed.

```
sub sum {
    @arguments = @_;
    foreach $argument (@arguments) {
        $total = $total + $argument;
    }
    return $total;
}
```

Then you could sum a bunch of numbers like this:

```
$aggregate = sum(1, 2, 3, 4, 5);
print $aggregate;
15
```

Some things to note about subroutines: They accept their arguments from the default array (`@_`) which is a left over shell-ism from older versions of Perl. Many built-in Perl functions will operate on default variables and so you could see a subroutine like this:

```
sub sum {
    foreach $argument (@_) {
        $total = $total + $argument;
    }
    $total;
}
```

Where the `@_` is used or even left out (because the default variable used by the `foreach` operator is `@_`). This is bad form, and propagates the ugly code that Perl is famous for. You'll also note that I didn't use `return`; the default returned value is the last expression (another bad form), so `$total` will make it back out of this second version just fine. You should try to avoid using the "garbage globals" as much as possible, which is why I don't discuss them here, except where you need to know about them.

You can also have anonymous subroutines (a concept needed for *completion*, which you should already know about) and references to subroutines:

```
$subref1 = sub { return "boo" }; # anonymous subroutine
$subref2 = \&previously_defined_sub; # reference to a sub
$array_of_subs = ( \&sub1, \&sub2, \&sub3 );
print $array_of_subs[0] -> ("arg1", "arg2");
$subref3 = $array_of_subs[2];
print $subref3->("arg1", "arg2");
```

Boolean Values and Truth Tests

Perl inherits much of what C considers Boolean values. Any positive or negative number is considered to be 'true' while the number zero is considered 'false'. Any string of any positive length is also 'true' while an empty string is 'false'. All other defined data types are true (objects, references, hashes with values, arrays with values). Perl has an `undef` keyword with a corresponding defined tester. An "undefined" value is always false; but a "defined" value of zero or the empty string is *also* false. When in doubt, don't allow Perl to decide truth tests for you, but do the explicit comparison:

```
if (1) {} # always true
```

```

unless (0) {} # always true
$a = ""; # empty string; could use zero here too
if ($a) {} # false
if (defined $a) {} # TRUE!
$a = 1;
if ($a) {} # true
$a = undef();
if ($a) {} # false
if (defined $a) {} # FALSE!

# here is a better way to test

if ($a != 0);
if ($a > 0);
if ($a ne "");

```

See how this is much easier to read, and leaves nothing to chance? But what does ne mean? Well, since Perl will convert between types, it was decided (erroneously??) that string comparisons should use special operators:

Numerical Operator	String Operator
< >	lt gt
<= >=	le ge
== !=	eq ne
<=>	Cmp

That's enough of primitive types.

Perl I/O

Let's communicate with the outside world:

Syntax	Meaning
open (FOO, "/path/to/file")	Opens the file pointed to by that path for reading
open (FOO, ">blah")	Opens the file "blah" for writing only
open (FOO, "<>blah")	Opens the file "blah" for reading and writing
open (BAR, "grep ink /etc/passwd ")	Causes perl to run the shell command given and return the results through the filehandle PIPE
open (MAIL, " mail ink@isu.edu")	Causes perl to run the shell command given and passes any output through that command
\$file = new IO::File()	The (much better) object-oriented version of the above
\$network = new IO::Socket()	Open up a network socket
\$line = <STDIN>	Read in a line from standard input
\$line = <PIPE>	Read in a line from the passwd file (see the PIPE above)
\$dir = opendir ("/path/to/dir");	Open a directory for reading
\$dirent = readdir(\$dir);	Read from a directory
close (FILE); \$file->close();	Both forms of close (traditional and object-oriented)
closedir (\$dir);	Close a directory handle

If you've coded in C/C++, it should seem very familiar – although the power and simplicity looks much better than the corresponding C code. Perl filehandles are usually written in all-caps and are really a special type of primitive the dates back to early versions. The object-oriented file code is better to use because they are simple scalars (objects) that can be treated like any other scalar. You'll see both, though, which is why I present them. You read from file handles and objects using the <> operator, which works on a line-by-line basis:

```
$line = <STDOUT>; # read from standard input
```

```

print STDOUT "blah\n"; # write to standard output
print FILE "blah\n"; # write to the FILE filehandle
$line = <$file_object>; # read from an OO-style file
$file_object->print("blah\n"); # newer OO-style printing

```

There are also low-level read() and write() calls, but these are rarely used outside of low-level networking or specialized binary file access.

Control Structures

Most of the control structures are from C:

```

if (condition) {
    # a one-time condition goes here
}
elsif (condition) {
    # another one-time condition; like C switch()
}
else { # and so on }
unless (condition) {
    # syntactic sugar; just like if (!condition)
}
while (condition) {
    # repeating condition
}
do {
    # repeating condition with tail test
}
for (initial setup; condition; iterator)
    # just like C/C++ for() loops
}
condition ? true code : false code; # a C ternary operator
foreach scalar (list) {
    # iterates through the items in list
}

```

The conditions are declared true or false based on the Boolean tests described previously. Some common control structures look like this:

```

unless ($age > 18) {
    # process children's code here
}
foreach $image_path (@all_images) {
    # print "<img src=\"$image_path\">\n";
}
while ($line = <FILE>) {
    chomp $line; # take off newline
    print uc($line); # make it upper-case
}
if ($command eq "run") {
    # ...
}
elsif ($command eq "walk") {
}
elsif ($command eq "jump") {
}
else {
    # default code goes here
}

```

Regular Expressions

The bailiwick and bastion of Perl is its regular expression code. It's a very handy information processing tool used in the UNIX `grep` (`egrep`), `awk` and `sed` tools. Perl understands the same syntax, and is able to filter information into native Perl structures by using regular expressions and a few valuable keywords. I'll cover the basics here, but more information can be gleaned from the Perl man pages (`man perl` or `man perlre`). A regular expression generally comes in one of two forms:

Matches

Matches are expressed with the 'm' operator:

```
$a = "Portneuf Valley";
if ($a =~ m/neuf/) # TRUE
if ($a =~ m/NEUF/) # FALSE
if ($a =~ m/^neuf/) # FALSE
if ($a =~ m/^Port/) # TRUE
if ($a =~ m/neuf$/) # FALSE
if ($a =~ m/lley$/) # TRUE
if ($a =~ m/NEUF/i) # TRUE
```

Matches are so popular that you can even leave out the 'm' and Perl will note that the expression should be a match. The `=~` test is a special test for regular expressions; it tests the rvalue on the lvalue and produces either an undef or the number 1 (false or true). A trailing 'i' means that the match should NOT be case sensitive. Some more examples:

```
while ($a =~ /valley/i) {}
for ($a="AAAAAAA"; $a =~ /AA/; chop $a) {}
if ($a =~ /^AB?.*M.+Z$/) {}
```

That last one says that \$a must match something that "starts with an A, may have a following B, may or may not be followed by any number of characters until an M is found, must have at least one more character or more than one character before ending in a Z. Note the absence of the 'm' operator on these expressions.

Search/Replace

Search and replace regular expressions are done with the 's' operator. It takes two arguments, separated by /:

```
$a = "Swan Valley";
$a =~ s/Swan/Sun; # $a = "Sun Valley"
$a =~ s/[unl]\/\*/g; # $a = "S** Va**ey"
```

Note that the asterisk needed to be escaped in that last one because it is a special character inside regular expressions.

Parenthetical State

Perl regular expressions can capture any arbitrary match into special variables numbered 1 through 9 (Perl6 removes this limitation of only 9) by using parenthesis in the pattern:

```
$a = "If you haven't been following Natural Selection's
rise, I'll give you a little update. Basically, you have 2
groups; the Humans and the Aliens.";

if ($a =~ /^(.)\.\s*(.)\..*$/) {
    print "Sentence 1 is $1\n";
    print "Sentence 2 is $2\n";
}
```

The `\s` above matches any whitespace. You can also use this mechanism to do replacements:

```
$a =~ s/(.*)little(.*)Humans(.*)/\1big\2Bipeds\3/;
```

Safe, Pragmatic Perl

As mentioned earlier, Perl has sort of “grown up” from a text processing tool on the order of `awk` and `sed` into a full-fledged, multi-threaded, object-oriented programming language. Some of its ancestry is left in place to remain backward compatible with older Perl programs, and some of this code is dangerous. A quick 12-line Perl *script* has the tendency to grow into a 12 hundred page monster, and having a lot of global variables with no declarations can make things... a bit messy. Fortunately, Perl5 gave us the tools to make our code safer.

Safe Pragma and Scope

Perl has a keyword called `use` that is much like the C pre-processors `#include` coupled with the C++ `using` command. Basically, any time you wish to bring in external help for your Perl program you “use” it (there is an older “require” imperative, but that should be avoided in my opinion). Most of the things that you use are Perl modules (libraries), but some of them are internal directives that make the Perl compiler and run-time act differently; the `safe` pragma forces the author to declare all variables. Take this program for example:

```
$abc = 1;
if ($abc == 1) {
    $abc = 3;
    $def = 2;
    print "abc = $abc  def = $def\n";
}
print "abc = $abc\n";
```

This program will run just fine under Perl; but if we turn on the strict pragma:

```
use strict;
$abc = 1;
if ($abc == 1) {
    $abc = 3;
    $def = 2;
    print "abc = $abc  def = $def\n";
}
print "abc = $abc\n";
```

We’ll get these errors:

```
Global symbol "$abc" requires explicit package name at
./blah.pl line 2.
Global symbol "$abc" requires explicit package name at
./blah.pl line 3.
Global symbol "$abc" requires explicit package name at
./blah.pl line 4.
Global symbol "$def" requires explicit package name at
./blah.pl line 5.
Global symbol "$abc" requires explicit package name at
./blah.pl line 6.
Global symbol "$def" requires explicit package name at
./blah.pl line 6.
Execution of ./blah.pl aborted due to compilation errors.
```

The solution is to declare them using the ‘my’ keyword:

```
use strict;
```

```

my $abc = 1;
if ($abc == 1) {
    $abc = 3;
    my $def = 2;
    print "abc = $abc  def = $def\n";
}
print "abc = $abc\n";

```

Try putting a second ‘my’ on the `$abc = 3` in the inner loop and see what happens. The last print statement prints out a different value! This is because you’ve now declared that the `$abc` value on the inner block is different from the `$abc` in the global block. Every block of code that is delimited by the `{` braces `}` is a different scope. This allows you to lock variables into different contexts so that they don’t trample other contexts; said another way, global variables are evil and you should only have variables *visible* within the block that they pertain to.

Warnings

Perl has an additional safety net if you start it up with the `-w` flag, which enables warnings. This allows for further checking of the code for safety. See the `perllexwarn` man page for more information. Many programs will enable this at the beginning of the file:

```

#!/usr/bin/perl -w
use strict;
# good, now we’re running with strict pragma and warnings

```

Where to go from here

All the Perl documentation is available online. Under Windows operating systems, the help will be installed in the “Start Menu”. Under UNIX, you can use the manual pages to get information about Perl. The Perl man page (`man perl`) will give an overview to the available documentation; each section is available as a separate man page.

In addition, there are two excellent books that I recommend:

Programming Perl (3rd Edition) by Larry Wall, et al
 Algorithms with Perl by John Macdonald, Jon Orwant and Jarkko Hietaniemi

There are a couple web sites that may also help:

<http://www.perl.com>
<http://www.activestate.com>

Your awk assignment done in Perl

```

#!/usr/bin/perl

use strict;                                # use strict pragma
my ($url_prefix, $expression) = @ARGV; # command line args are in @ARGV array

while (my $line = <STDIN>) {
    $line =~ s/\s+$/;/;                    # get rid of any trailing whitespace on line
    if ($line =~ /href="\?([^\s]+)\"/?) {
        # if we see something like href=blah or href="blah", grab it
        my $url = $1;                      # give $1 a name to make it easier to read
        if (defined $expression) {# do we need to filter urls?
            if ($url =~ /$expression/) {
                system ("wget", $url_prefix . $url);
            }
        }
        else {
            system ("wget", $url_prefix . $url);
        }
    }
}

```

```
}
```

Note that this code is more robust than the awk code. The `$url` and other variables are only visible inside the scope that they were declared in. It would be very easy to slap this code into a subroutine for use by other Perl programs, even if those programs used the same variable names. Using a Perl module named `HTML::TokeParser`, it would look like this:

```
#!/usr/bin/perl

use strict;                                # use strict pragma
use HTML::TokeParser;
my ($file, $url_prefix, $expression) = @ARGV;

# create a new HTML token parser object
my $p = HTML::TokeParser->new( $file ) or die $!;
while (my $token = $p->get_token()) {
    next unless (($token->[0] eq "S") &&
                 ($token->[1] eq "a")); # look for starting anchor tags <a>
    my $url = $token->[2]->{href} || "-";
    if (defined $expression) {
        if ($url =~ /$expression/) {
            system ("wget", $url_prefix . $url);
        }
    }
    else {
        system ("wget", $url_prefix . $url);
    }
}
}
```

In addition to being easier to understand, this code will be even more robust because it can now handle `i8n` (internationalization) characters, whitespace in the URL and URLs that have strange characters such as newlines (`\n` or `\r`) in them. Additionally, it will work with future versions of HTML as the protocol and this Perl module are updated. The last thing we'd want to get rid of is our dependence on the "wget" UNIX program so that our URL retriever would work on a Mac or Windows machine. The Perl modules `Net::HTTP` and `Net::FTP` would let us do that for web and FTP requests.