

## CS193i Perl Review Section

### Running Perl

Perl is an interpreted language. A Perl script can be executed on any of the Leland system machines with:

```
elaine36:~> perl foo.pl
```

A Perl interpreter starts executing statements at the top of the file. Unlike c/c++, not "main" function is needed...just start writing Perl code.

### Variables

Scalar variable names begin with \$ and hold a number or string. It is a good idea to "use strict 'vars'". This will enforce global/local variable declarations (otherwise all variables will be global). Variables will need to be declared with "my" (eg my \$num = 1) before they are used.

### Basic String Processing

```
$num = 193 ;
$string = "CS" . $num . "i" ;
## use == to compare numbers.
## use eq and ne to compare strings.
($num == 193 ) ==> true
($string eq "CS193i") ==> true
($string eq "cs193i") ==> false
(lc($string) eq "cs193i") ==> true
## difference between " and '
$string = "test" ;
$x = "this is a $string" ;
## $x evaluates to: this is a test
$y = 'this is a $string' ;
## $y evaluates to: this is a $string
```

### Arrays

Arrays are specified using () and elements are separated by commas. Array variable names begin with @. The assignment operator (=) works for arrays (unlike c/c++).

Square brackets are used to reference elements in arrays. Remember that scalar expressions begin with a \$, so when referring to an element in an array use \$array[0]. Use @array when referring to the whole array.

```
## array examples
@array = (1, 2, "foobar") ;
$len = @array ;
$num = $array[0] ;
$one = shift(@array) ; ## now array is (2, "foobar")
unshift(@array, 1) ; ## now array is back to (1, 2, "foobar")
## pop and push operate on the end of the array
## splice replaces a segment of an array (or deletes it)
splice(@array, 0, 1, @another_array)
## @ARGV is a global array containing all the command line arguments
```

## Subroutines

Unlike c/c++ you don't need to define a subroutine before it is used.

```
## sub routines
sub foo
{
    return "foo" ;
}
```

Parameters are passed to subroutines in a single array called `@_`. Usually subroutines will simply copy the elements of the `@_` array into local variables.

```
sub sum
{
    my ($x, $y) = @_ ;
    return ($x + $y) ;
}
```

## while, for, if

Syntax is similar to c/c++, except that `{}` are always needed. There is no "true" or "false" keywords in Perl. The empty string, the empty array, the number 0, and `undef` all evaluate to false, and everything else is true. The logical operators work the same as in c/c++.

```
if ( expr )
{
}
elsif( expr )
{
}
else
{
}
```

The `foreach` construct can be used to neatly iterate over all the elements in an array.

```
foreach $var (@array)
{
    print $var ;
    ## do some other processing
}
```

## Files

Variables that represent file handles are treated differently from other variables. File handles are not preceded by any special characters and they are all in the global namespace. They can be passed to subroutines just like a scalar.

<> is an expression that returns one line from a file (including the \n). <> returns undef when there is not more input.

```
# by convention all uppercase.
open(FILE, "filename.txt") ;
print <FILE> ## prints one line from the file to STDOUT
close(FILE) ;
```

### Regular Expressions

Perl has very powerful string processing features. Some very basic examples are provided below, but you should check out Nick Parlante's guide for more detailed examples. Learning how to use regular expressions will make writing Perl code much easier.

```
$string =~ /pattern/ ==> true if pattern is somewhere in $string
```

Regular expressions allow you to match patterns using character codes and control codes.

```
"piiiig" =~ /pi*g/ ==> true
"pabcg" =~ /p.*g/ ==> true
"pg" =~ /pi*g/ ==> true
"pg" =~ /pi+g/ ==> false
```

If =~ is true, the special variables \$1, \$2, ... will be substrings that match the parts of the pattern is parenthesis.

```
"lastname, first" =~ /(\w+) (\s*,*\s*(\w+))*/
## $1 = "lastname", $2 = ", first", $3 = first
"lastname, first" =~ /(\w+) [\s,]*(\w+)* /
## $1 = "lastname", $2 = "first"
"lastname,first" =~ /(\w+) [\s,]*(\w+)* /
## $1 = "lastname", $2 = "first"
"lastname first" =~ /(\w+) [\s,]*(\w+)* /
## $1 = "lastname", $2 = "first"
```

\* and + are “greedy”, meaning they will match the largest sequence. This greedy behavior can be suppressed using the ? operator.

```
#!? trick to supress greedieness of *, and +
"{foo} {bar}" => /({.*})/ ## $1 = "foo} {bar"
"{foo} {bar}" => /({.*?})/ ## $1 = "foo"
```

What would a regular expression being for extracting the url from a html hyperlink statement (eg extract cs.stanford.edu from <a href="cs.stanford.edu">CS Department</a>)? How could you make the pattern robust against extra white space?

### File Processing Example

```
#!/usr/bin/perl -I/usr/class/cs193i/pm -w
```

```

## CS193i Perl section example, summer 2004
## This script opens quote.txt, and searches for a quote that contains
## the string passed at the command line.

use FileHandle ;
use strict 'vars' ;

my $fname = "quotes.txt" ;
my $string_to_find = shift(@ARGV) || die "please provide a string" ;
my $line ;

open(FILE, $fname) || die "unable to open file" ;
while( $line = <FILE> )
{
    if ( $line =~ /$string_to_find/ )
    {
        print $line ;
        close(FILE) ;
        exit(0) ;
    }
}
print "no quote found that contains \"" . $string_to_find . "\"\n" ;
close(FILE) ;

```

How could this program be modified to take a list of files on the command line to search (eg. perl searcher.pl find\_this\_string quotes.txt, more\_quotes.txt, even\_more.txt)?