

CS306 – Introduction to Perl
Fall 2007
Homework Assignment #3
Due: Monday, October 29th, 2007, 4pm

Guidelines for Submitting Homework

All homework should be submitted in the form of a zip file which contains one program for each question. Your zipfile should be named *lastname-hw3.zip* and should contain one text file called *answers.txt* for the written answers, and then files named *hw3p1.pl*, *hw3p2.pl*, *hw3p3.pl*, etc... for the programming assignments. **Make sure your name also appears in all source code files!** Email this zip file to cs306@cis.uab.edu. **When emailing this file email a copy to yourself so you know the email went through ok.**

Homework #3 (200 total points)

Question 1. (10 points) Give at least three advantages of using subroutines in your code.

Question 2. (10 points) Why might you want to use a hash as the argument to your subroutine? Provide an example of a subroutine that uses a hash as its parameter (do not copy code in the book or slides).

Question 3. (10 points) The following code does not work as expected. What is the problem?

```
my @arr1 = (1,2,3);  
my @arr2 = (4,5,6);  
foo (@arr1, @arr2);  
sub foo {  
    my (@a1, @a2) = @_;  
    ...  
}
```

Question 4. (10 points) What is a package variable and how is it different from a lexical variable?

Question 5. (20 points) Given the pattern, indicate which part of the following strings will match the pattern.

Ex. The pattern is `/\wello/`

The string is “Kids love Jello”

The part of the string that matches the pattern is “Jello”

Pattern: `/\w+ Blazers/`

String: The UAB Blazers are located in Birmingham.

Pattern: `/\d{3}/`

String: My number is 205-555-1212.

Pattern: `/\w.*\w/`

String: The sun is really bright today, isn't it?

Pattern: `/^\w+\s+d?\w+/`

String: I think I've eaten 6 tacos already.

Pattern: `/^\d.*?\d/`

String: 1234567890

Question 6. (15 points) Explain the relationship between `<` and `@ARGV` and what happens when `<` is used when `@ARGV` is empty.

Notes on programs: While writing these programs below, remember to comment thoroughly, including your name near the top as well as comments throughout explaining what you are doing in your program.

Also, turn on warnings and use strict; in all of your programs. Here's how:

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

The `/usr/bin/perl` part may change depending on your platform; it's the `-w` that is important, as it is what enables warnings.

Read each problem description carefully, and be sure to address everything that is asked for in the problem.

A NOTE ON DEBUGGING: WHEN YOU GET STUCK, PRINT OUT YOUR VARIABLE VALUES AT VARIOUS POINTS IN YOUR PROGRAM. THIS IS OFTEN THE EASIEST WAY TO FIND WHERE THE CODE IS BROKEN.

Program 1. (25 points) Movie Facts Rewrite

Rewrite HW2 Program #4 to implement each menu option as its own subroutine. The rule is that the only variables you can interact with inside a subroutine are those which have either been passed into the subroutine or defined within the subroutine itself.

Program 2. (50 points) Address Book Syntax Checker

This program reads in a file that represents an address book. The address book lines have the following format:

```
FirstName|LastName|Street Address|City|State Code|Zip Code|Phone|Email
```

This program will read in the input file `addresses.txt` (link to be provided on the web site) and syntax-check each field for correctness.

Your program should use a series of regular expressions to verify that the data meets the formatting rules. The rules are:

- #1. The first and last name should both start with a capital letter. There should be only letters, spaces and ' allowed in a name.
- #2. The street address starts with one or more digits, then a space, then a street name which can contain letters only, then a space then the code for the type of street (allowable types are St. Rd. Dr. Cir. Ln. and Blvd. for simplicity). We are only going to test for one-word street names – things like 123 Cedar Lake Dr. do not need to be accounted for.
- #3. The city is simply words and spaces where the words have only letters, no numbers.
- #4. State code is always two capital letters
- #5. Zip code is either five numbers or five numbers followed by a dash and four more numbers.
- #6. The phone number with area code has this format: (XXX) XXX-XXXX
- #7. The email address should contain at least an @ sign and at least one . after the @ sign. (The real pattern for a valid email is famously complicated; this is a major simplification)

As each record (line of the file) is read in, check the input. If it is valid, print a success message that says “The entry for John Smith is valid.”. If it is invalid, inform the user of the problem(s): “The entry for John Smith has the following problems: Invalid Street Address, Invalid Phone Number.”

Finally, you should produce two output files: valid.txt and invalid.txt, which should contain the valid and invalid entries, respectively.

Program 3. (50 points) Word and Letter Counter

You will be given a text file as input (link provided on web site). Your goal is to output the following:

1. a list of words sorted by the number of times they appear in the text file (most common word first) along with the number of times they have appeared.
2. a list of letters sorted by the number of times they appear in the file (most common letter first) along with the number of times each has appeared. We're only interested in letters, no punctuation or spacing. A capital B and a lowercase b should be counted the same – they're both b's. In other words, I don't want a separate count for B and b – just one for all b's.

So, for each line of input, you'll be doing two simultaneous counts.

Tips and Tricks

1. Remember the trick we learned in class about using hashes for counts.
2. `split //`, `$line` splits a line on the empty string. See `perldoc -f split` for more.
3. You'll want to sort a hash based on the values of the hash (most common word first, etc...). This can be accomplished by defining a custom sort.

```
sort {CUSTOM SORT DEFINITION GOES HERE} (keys%hash);
```

When you use the sort function, you get access to two special variables: `$a` and `$b`. These variables represent “the two things currently being compared” from the list to be sorted. You can use these variables inside your custom sort definition. The end result of your sort is that the left item is first in the sort, the right item is first in the sort, or the left and right items are equivalent.

To do this, you need to use the “three way comparison” operators: `cmp` and `<=>` which are documented in “`perldoc perlop`”

In the example above, our list to be sorted is the output of `(keys %hash)`. So, `$a` and `$b` are going to contain the names of keys within the hash. Since our goal is to sort by the hash values (e.g. the number of times each word has appeared), you can use something like this as your sort definition:

```
my @sortedkeys = sort {$hash{$a} cmp $hash{$b}} (keys %hash);
```

Now, you'll get the list of keys in the order of their sorted values. Tricky, but very handy! This isn't *quite* the code needed to do what is asked in this program, but it will get you close enough to figure it out on your own.