

CS306 – Introduction to Perl  
Summer 2006  
Homework Assignment #3  
Due: July 12<sup>th</sup>, 11:20am

Please follow the guidelines on HW0 on how to submit the homework. Remember, one zip file like this:

lastname-firstname-hw3.zip

answers.txt

hw3p1.pl

hw3p2.pl

etc....

This week I will use an automated grading script, so please pay attention to the naming and zip file structure.

### Homework #3

**Question 1. (15 points)** What is anonymous data?

**Question 2. (20 points)** Explain the meaning of each of the following special characters as they are used in regular expressions. `.` `*` `?` `+` `[]` `()` `{}` `^` `$` `|`

**Question 3. (15 points)** What's the difference between a symbolic reference and a hard reference?

**Question 4. (15 points)** Define and give an example of autovivification. Be careful here – make sure that what you are doing is really autovivification.

Programs begin on next page.

## **Program 1. Extend Your Contact List Program (50 points)**

In HW2's Program 7, you wrote a program to maintain a contact list that was stored in a flat file. You probably found it awkward to complete tasks like sorting by alphabetical order, due to the “scalar values only” limitation of arrays and hashes. Each record consisted of 4 fields, but an array could only hold one of the fields at a time. With the hash, you could have one field as the key and another as the value, but that did not do much to solve the problem.

Now that we have learned about references and complex data structures, we can improve this program significantly. By using a multi-level hash to represent each entry in the contact list, we can more tightly couple all of these related fields into a single variable. Consider the following hash structure:

```
$list->{Smith}->{firstname} = 'Joe';  
$list->{Smith}->{phone} = '555-1212';  
$list->{Smith}->{email} = 'jsmith@example.com';
```

Note how we captured all four pieces of information about Joe in one multi-level hash. We used his last name as the top level key – since we assume these are unique and since this is the field we most often want to sort by, this is a good choice to differentiate between our records. Then, at each of the lastname keys, we store a reference to a hash with three fields: firstname, phone and email. So, now, the value at each lastname key in the top level hash is a reference to another hash that has the rest of the information about that user. Here's another way to construct this hash, that might illustrate the data structure more clearly:

```
$list->{Smith} = {  
    firstname => 'Joe',  
    phone => '555-1212',  
    email => 'jsmith@example.com'  
};
```

If you take this concept one step further by inserting variables in place of the 4 values and inserting this into a loop, you can see how it will be possible to read the entire contact list into a multi-level hash with just a few lines of code.

Note the subtle difference below:

```

# FAILS - hash values must be a scalar
$list{Smith} =
  {fname => 'Joe', phone => '555-1212', email => 'jsmith@example.com'};

# WORKS. Creates %list with one key, Smith, which has a reference to
# another hash as its value.
$list{Smith} =
  {fname => 'Joe', phone => '555-1212', email => 'jsmith@example.com'};
# Use it like this
print $list{Smith}->{fname};

# BEST. Note that no %list exists! The only hashes here are
# anonymous. $list is a # scalar which has a reference to an anonymous
# hash as its value. That anonymous hash has one key, Smith, which
# holds a reference to a second anonymous hash with the other three
# keys
$list->{Smith} =
  {fname => 'Joe', phone => '555-1212', email => 'jsmith@example.com'};
# Use it like this
print $list->{Smith}->{fname};

```

Use this new technique to revise your contact list program. You should write it so that you use one multi-level hash structure to read the contact list into, work on within the program, and write back out to the file when you are done. Keep the same file format as we used in the last exercise. If you wrote a well-modularized program last time, you should be able to reuse a lot of your code.

Now that we have a better data structure, let's put our regular expressions to the test. Add some validation code so that when I add a new user, it ensures that the phone number and email address are valid.

For the phone number, it should allow the following formats:

```

123-456-7890
(123) 456-7890
123.456.7890
(123)-456-7890

```

Start with the phone number regular expression we talked about on the slides and modify

from there.

For the email, I'm just looking for a regular expression that will match email addresses of the following style:

[example@cis.uab.edu](mailto:example@cis.uab.edu)

[example@uab.edu](mailto:example@uab.edu)

[example.user@gmail.com](mailto:example.user@gmail.com)

[example@yahoo.com.in](mailto:example@yahoo.com.in)

Valid characters for the part before the @ are letters, numbers, periods and underscores. After the @, there are one or more “words”, separated by periods, and the last “word” is between 2-4 characters long (.us, .edu, .info and similar).

A truly accurate regular expression for pattern matching for email addresses is actually quite a famous problem. I won't ask you to solve that, but take a look at:

<http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html>

and

<http://search.cpan.org/src/CWEST/Email-Address-1.80/lib/Email/Address.pm>

### **Program 2. Checkers (85 points)**

Write a program to play checkers! Use a two-dimensional array to represent the checkers board (8x8), and you will need to become familiar with all of the rules of the game. Here is the web site I will use as the reference:

[http://boardgames.about.com/cs/checkersdraughts/ht/play\\_checkers.htm](http://boardgames.about.com/cs/checkersdraughts/ht/play_checkers.htm)

I will make one exception to the above rules – there is no need to check if all of a player's pieces are blocked in (described in rule 13). We'll just live with the risk that sometimes that might happen and we won't be able to finish a game. However, an easy way to code for this is to have a special move where the player issues a “concede” move, when they themselves recognize they have no move. It's up to you whether you add that feature or not.

I suggest you design a set of character labels to represent the different states of a spot on

the board: maybe WH for while, BL for black, WK for white king, and so on. You will use these throughout your program. It's best if they are all the same number of characters – it makes printing out the game board easier. You will also need a system for entering moves. I suggest that the rows are letters and the columns are numbers, so that you can enter a move like: C4 – B5 to indicate that you want the piece at C4 to move to B5.

A key to this program is good functional design. You should think of this program as a main loop that calls out to helper functions which each return to the main loop. The logical flow should be something like:

```
initialize game
start game loop
  show game board
  figure out whose turn it is
  get the move request until valid request entered according to rules
  update the game board with the move
  check for a winner, if found, notify and exit loop
end loop
```

You will need helper variables to track whose turn it is and to keep track of the number of pieces left and other maintenance tasks.

Note that in some cases, the player has no choice: if a jump is available, they must take it. So you might want to optimize the logical flow to do some pre-analysis. There may be an instance where the player has a choice between two jumps – don't force one on them, just ensure that they choose one of their jumps.

This program represents the culmination of everything you've learned to date. In just one month, you've learned enough Perl to write a substantial and highly-structured program. While developing this program, focus on making it easy to use for all of your audiences – the user interface should be clean and self-explanatory for the end user, the code should be easily understandable through commenting and good use of functions for code reviewers, and most of all, the design of the program should facilitate ease of development for yourself. Achieve this by using smart data structures, breaking your code into small manageable pieces, and understanding the big picture before you start coding. You should leverage all of the advanced techniques – references, functions, complex data structures, perhaps even regular expressions (parsing user's move requests?), to make this code as elegant as possible.