

CS306 – Introduction to Perl
Fall 2007
Homework Assignment #4
Due: November 19th, 4pm

Please follow the previous guidelines on how to construct the zip file and submit the homework.

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-hw4.zip* and should contain one text file called *answers.txt* for the written answers, and then files named *hw4p1.pl*, *hw4p2.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 #4

Question 1. (10 points) What are two different ways to run an external program in Perl? What are the key differences between the methods?

Question 2. (10 points) Explain the concept of reference counting and how it relates to Perl's memory management.

Question 3. (15 points) Given the arrayref:

```
$arrayref = ["fee", "fi", ["fo", "fum", ["eeny", "meeny", "miny", "moe"]]];
```

What are the three different syntaxes for dereferencing the element containing "meeny"?

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

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. Connect Four (75 points)

Write a program to play Connect Four. Both players will be human, the program simply takes moves, checks that they are valid, and updates the game board. The game board must be represented and accessed by a single variable (either an array or an arrayref, your preference).

Connect Four is played on a single vertical board game, consisting of 7 rows and 7 columns. The pieces are round discs that are dropped into the board from the top of each column, and fall down into the column until they hit bottom or rest on top of another piece. Each team has a different colored set of pieces, one team red and the other black. The goal of the game is to get four of your own colored pieces in a row, either vertically, horizontally, or diagonally.

Your program should represent the game board using a two-dimensional array. Users will indicate which column they wish to place their piece into by indicating the column with a number, 1-7. The program will then insert the piece into the appropriate location on the board (simulating gravity and sliding the piece as far down the column as it will go. :-)

Before the game starts, the game board looks like this:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | — | — | — | — | — | — | — |
| 2 | — | — | — | — | — | — | — |
| 3 | — | — | — | — | — | — | — |
| 4 | — | — | — | — | — | — | — |
| 5 | — | — | — | — | — | — | — |
| 6 | — | — | — | — | — | — | — |
| 7 | — | — | — | — | — | — | — |

It is Black's move. Enter choice (1-7, s, q): 5

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | — | — | — | — | — | — | — |
| 2 | — | — | — | — | — | — | — |
| 3 | — | — | — | — | — | — | — |
| 4 | — | — | — | — | — | — | — |
| 5 | — | — | — | — | — | — | — |
| 6 | — | — | — | — | — | — | — |
| 7 | — | — | — | — | B | — | — |

It is Red's move. Enter choice (1-7, s, q): 5

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | — | — | — | — | — | — | — |
| 2 | — | — | — | — | — | — | — |
| 3 | — | — | — | — | — | — | — |
| 4 | — | — | — | — | — | — | — |
| 5 | — | — | — | — | — | — | — |
| 6 | — | — | — | — | — | — | — |
| 7 | — | — | — | R | B | — | — |

It is Black's move. Enter choice (1-7, s, q): 5

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | — | — | — | — | — | — | — |
| 2 | — | — | — | — | — | — | — |
| 3 | — | — | — | — | — | — | — |
| 4 | — | — | — | — | — | — | — |
| 5 | — | — | — | — | — | — | — |
| 6 | — | — | — | — | B | — | — |
| 7 | — | — | — | R | B | — | — |

... and so on.

The program just has to get the moves from the user, check that it is valid, and update the game board for each move. It does not have to check to see if there are four in a row (the humans playing the game can check that themselves – but see the BONUS below)

To check for a valid move, the following rules must be observed:

1. The columns are numbered 1-7.
2. Each column can only hold 7 total pieces.

If you try to put a piece into a column which is full, the user must be alerted that it is an invalid move, and asked for a move again.

There is one more twist. The game should allow the users to save a game to come back to later. (This is the 's' option on the menu). It should do this by:

1. Writing out the current state of the game to a file and exit when “save game” is chosen.
2. When the program is started, it should check for saved games, and ask the users if they want to restore a saved game.
3. It should allow multiple saved games at the same time. So, you'll need to have unique names for each saved game file, and a way to present the users a list from which they can choose the game they want to resume (or none at all, starting a new game).
4. To make it easier for the users to distinguish, it should also allow them to save a

short note to indicate which game they are saving: "joe and tom, tuesday" which is presented back to them when they start the program again.

So a save might look like this:

```
It is Red's move. Enter choice (1-7, s, q): s
Please enter a note for this saved game: need to go to work
Ok, the game "need to go to work" has been saved into file
cfour12.txt. Goodbye.
```

And the start of a program might look like:

```
Welcome to Connect Four!
Three saved games detected. Please make your choice:
  1. "joe and tom, tuesday"
  2. "missy and mom"
  3. "about to lose"
  N. New Game
```

Please make your choice (1-3 or N):

The format of the saved-game file is up to you, as long as you can read the associated "note" and restore the two-dimensional game board from the saved file, the format is acceptable.

The 'q' menu option simply quits the game without saving it.

BONUS (25 Points). Make the game check for victory conditions as well. Victory conditions are:

1. Four of the same color either horizontally, vertically, or diagonally.
2. There is no wrapping – For instance, A6, A7, A1, A2 is NOT a win.

Program 2. Family Tree (75 points)

You will be given a data file with lines in the following format:

```
Firstname|Lastname|DateOfBirth|Parent
```

Your goal is to read this input file, construct a multi-level hash of the family tree, and then output the results. To make your job easier, you can assume that all Firstname's will be unique and that a child will never appear in the data file before their parent. We will also not be incorporating spouses into the tree.

You should first construct a base hash for each person – this is the person "object". This object should look like:

```
FirstName => Jane
```

```
LastName => Smith
DOB => 19570123
Children => [child1ref, child2ref, etc...]
```

You should then construct a hierarchy of such objects, connecting parents with their children through the Children array.

As soon as you are done constructing the family tree, you should use [Data::Dumper](#) to print out the entire data structure to a file called tree.txt.

```
open OUT, ">tree.txt";
print OUT Dumper(\%tree);
close OUT;
```

This file is so that we can grade you on correct construction of the multi-level hash easily.

Finally, enter a loop where you ask the user for a first name, and then display that person and their children, sorted in order of their birth. For instance:

```
Enter first name: Nancy
Nancy Forrester (5/7/46)
  Jessica Forrester (1/24/74)
  Craig Forrester (7/16/77)
Enter first name: etc....
```

Note, the format the DOB was entered above makes it easier for sorting – comparing two dates numerically will tell you which one comes first. You will need to define a custom sort routine for this, similar to last homework. Remember the format:

```
@sorted = sort {$a <=> $b} @$children;
```

The trick here is that instead of comparing \$a and \$b directly (\$a and \$b are each a reference to a child in the children hash for this particular object), you want to look into the hash and compare the two children's DOB. So, you'll need to dereference into the object that \$a represents.

BONUS. (25 points) Print -all- descendants of the name entered. This will require a recursive algorithm.