Assignment 3: Dating with Perl

Due: 19:00, Fri 15 Apr 2016 Full marks: 100

1 Introduction

In this assignment, you will gain experience in *Perl*, a classic scripting language originally designed for Unix. The assignment consists of three parts. Task 1 is to implement a game called *Reversi* (or *Othello*, 黑白棋) in object-oriented Perl with some simple artificial intelligence. Task 2 provides three Perl scoping implementations of a credit card system for a bank. You have to understand the scoping in Perl and re-implement them in C/C++, and a Perl implementation with correct variable scoping. In the process, you will learn the difficulty of understanding codes written with dynamic scoping. Task 3 is to write a ***two-page*** report to tell us your experience in using Perl.

2 Task 1: Reversi

In this task, you will write a program in Perl for the game *Reversi*. The following subsections describe the game rules and the object-oriented design.

2.1 Game Description

Reversi is a two-player (called Black and White respectively) game which is played on an 8x8 chess board with some chess pieces. A piece is a disc with two sides in black and white colors respectively. We shall use X and O to denote black and white respectively. At the beginning of the game, there are four pieces placed on the board as shown in Figure 1. Black (X) and White (O) alternately put a piece of their color on an unoccupied position in the board. Black plays the first move. The newly placed piece must be in a position such that it forms at least one straight (horizontal －, vertical ｜, or diagonal ＼ ／) occupied line so that *the new piece and another piece of the same color are enclosing one or more contiguous pieces of the opposite color*. After the move, those contiguous pieces will all be flipped to the other color. Figure 2 shows an example move (shaded in green) by Player X. Note that a move can form more than one straight line; the relevant pieces (shaded in yellow) on *all* the lines will be flipped.

```
  0 1 2 3 4 5 6 7
0 . . . . . . . .
1 . . . . . . . .
2 . . . . . . . .
3 . . . O X . . .
4 . . . X O . . .
5 . . . . . . . .
6 . . . . . . . .
7 . . . . . . . .
Player X: 2
Player O: 2
```

Figure 1: Initial configuration of Reversi

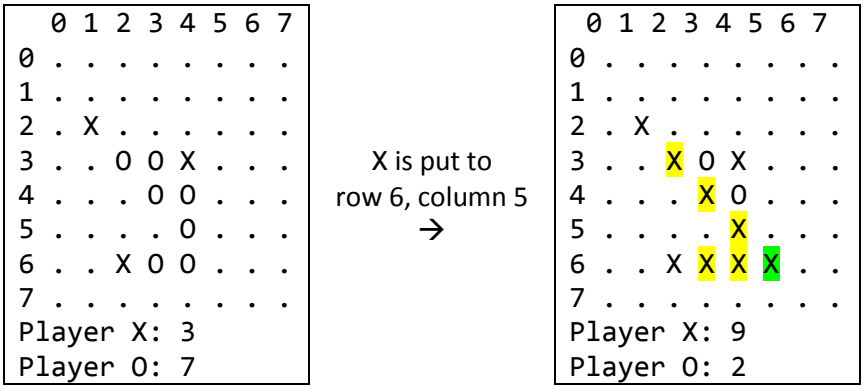


Figure 2: An example move by X, forming two occupied lines. All relevant pieces are flipped to Black.

The game finishes when either the board is full or both players have no valid moves. (Nonetheless, we shall handle "no valid moves" a bit differently. See the next section for details.) The player having more pieces of his/her color than the opponent wins the game. If both players have the same number of pieces, the game is a draw.

2.2 Program Design and Specification

Your program file name should be `reversi.pl`. You have to write your program in the classes `Reversi`, `Player`, `Human`, and `Computer` defined below. You are *free to add other extra members* (instances variables, methods, etc.) to these classes. You are *free to add extra classes* as well. You are also *allowed to design extra parameters* for the methods described below.

2.2.1 Class `Reversi`

This class models the Reversi game, with the following members:

Instance Variables

`board`

A two dimensional array of size 8x8 representing the game board.

`black`, `white`

Player X and Player O respectively.

`turn`

The player in the current turn. The first turn is Player X.

Instance methods

`new`

A constructor for initializing all the instance variables. You have to prompt the user to choose the player type (computer or human) for Player X and then for Player O. *You can assume that the input here is always either 1 or 2* (for computer or human respectively). You also have to initialize the board as the same configuration in Figure 1. Player X takes the first turn.

`startGame`

Starts a new Reversi game and play until the game finishes. The flow of this method is as follows:

1. Obtain a position from the current player for putting the piece to.

2. Check whether the position refers to a valid move. If the obtained position is not valid (because it is out of range or already occupied or cannot flip any opponent's pieces), then the player's round will be "passed" to the opponent.
3. If the move is valid, update the board by flipping all relevant pieces.
4. Repeat Steps 1–3 until either the board is full or two consecutive passes are made (one by each player). Alternate Players X and O in each round.
5. Once the game finishes, display the message "Player X wins!", "Player O wins!", or "Draw game!" accordingly.

`printBoard`

Prints out the game board in the console, in the format shown in Figure 1.

2.2.2 Class `Player`

This class is an "abstract" superclass for modeling players of the Reversi game.

Instance Variable

`symbol`

The symbol of the player. It is either "X" or "O".

Instance Methods

`new`

Parameter: `sym`

A constructor for initializing the player symbol as the parameter `sym`. You can design extra parameters for this constructor if you see fit.

`nextMove`

An "abstract" method *to be implemented in subclasses*. You just need to provide a stub here. In all subclass implementations, this method should return an array of length two, in which the first and second array elements are the row and column numbers of the next move respectively. For example, suppose the next move by the player is row 6, column 5, then calling the method should return the array `(6, 5)`. Note that the position returned by this method *may be an invalid move*, because the player may want to "pass".

2.2.3 Class `Human`

The `Human` class models a human Reversi player and is a *subclass of* `Player`.

Instance Method

`nextMove`

This method obtains the next move of a human player from user input. First, prompt the user to enter a row number and a column number for placing a piece. You can assume that *the user input is always two integers separated by a space*. Second, return the input position as an array of length two (e.g., `(6, 5)` for row 6, column 5). You do *not* have to validate the input here, because a human player may want to "pass".

2.2.4 Class `Computer`

The `Computer` class models an AI Reversi player and is a *subclass of* `Player`.

Instance Method

`nextMove`

This method determines the next AI move of a computer player. You can choose to implement either one of the following two requirements. Please clearly indicate which version you are implementing as comments in your source code.

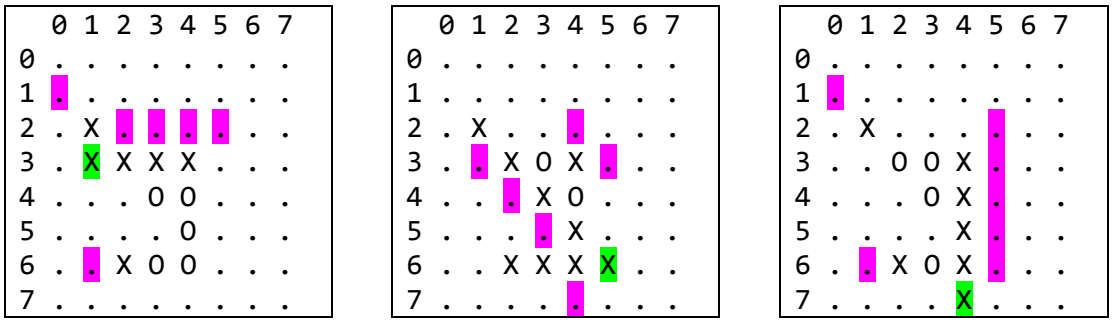
Version 1 (Easier, but you get full mark 95 only):

1. In the game board, if there is at least one valid move that can be made, then return a randomly generated position among these valid moves. For example, suppose Player X is a computer player and is given the board configuration in the left-side of Figure 2. Player X has three possible moves (a) row 3, column 1, (b) row 6, column 5, and (c) row 7, column 4. This method should randomly pick either one of the three to return.
2. If there is no valid move, then this method returns the array `(-1, -1)` to denote a "pass".

Version 2 (More difficult, but you can get full mark 100):

1. Among all possible valid moves allowed to the computer player, choose the position which minimizes the opponent's *mobility*. The mobility of the opponent is defined as the number of valid moves allowed to the opponent after the player puts a piece to a position. Recall the example in Version 1, if Player X chooses (a), then Player O (the opponent) would have six possible moves (mobility 6). If Player X chooses (b), then Player O would also have mobility 6. If Player X chooses (c), then Player O would have mobility 7. Then we consider (a) or (b) is better than (c). This method should return the position with the smallest opponent mobility. If there is more than one position with smallest mobility, you can pick *any one* of them to return. Figure 3 shows the mobility allowed to the opponent after choosing (a), (b), and (c).
2. If there is no valid move, then this method returns the array `(-1, -1)` to denote a "pass".

Note that in either version, if there is at least one valid move, you should *not* return `(-1, -1)`.



(a) Mobility 6 (b) Mobility 6 (c) Mobility 7

Figure 3: Mobility of Player O (shaded in pink) after Player X moves to (a) row 3, column 1, (b) row 6, column 5, and (c) row 7, column 4

2.2.5 "Main" Program

The whole program should be started using the statements:

```
my $game = Reversi->new(); # Create object; set up human/computer players
$game->startGame;          # Start playing game
```

Some sample program outputs are provided in Blackboard for your reference.

3 Task 2: Credit Card System

Three programmers developed three similar implementations of a credit card system for a bank in Perl. They have exactly the same logic and algorithm but differ in variable scoping declarations only. One uses all "global" variables; one uses all "my" variables; one uses all "local" variables. These kinds of variables have different scoping rules in Perl. However, among the three implementations, only one of them gives the correct execution behavior. Furthermore, even the one with correct execution behavior has improper variable declarations.

The three Perl programs are provided in Blackboard. You need to understand their respective behaviors, and during the process, understand *dynamic scoping* in Perl. Your specific missions are as follows:

1. You have to re-implement the three Perl programs in C or C++ of your choice to produce identical execution behaviors. Name your programs `global.c`, `my.c`, and `local.c` (or `global.cpp`, `my.cpp`, and `local.cpp`) respectively.
2. Identify the version with correct execution behavior, and rewrite that program (still in Perl) with proper variable declarations. Name your program `correct.pl`.

In this task, you are *not* allowed to touch the program logic. You can only fiddle with the variable scoping and type declarations and initializations. We will evaluate your programs not just in terms of execution but also programming style.

4 Task 3: Written Report

Your written report (`report.pdf`) should answer the following questions within ***two*** A4 pages (*all questions in **two** pages total*).

1. Using your code for Task 1, discuss features of object-oriented Perl that are different from the object-oriented features of Ruby.
2. Using your codes for Task 2, compare the differences of scoping in C/C++ and Perl. Also, explain and justify the variable type declarations of your correct Perl program.
3. Using your codes for Task 2, explain whether dynamic scoping is needed in a programming language.

5 Submission and Marking

- Your Perl programs will be graded in Perl 5.14+ in Windows. Your C/C++ program will be graded in Visual Studio 2013+.
- Prepare a zip file `assg3.zip` containing six files: (a) `reversi.pl`, (b) `global.c`, `my.c`, `local.c` (or `.cpp`), (c) `correct.pl`, and (d) `report.pdf`.
- Submit the file `assg3.zip` in Blackboard (https://elearn.cuhk.edu.hk/). Besides, your *report* has to be submitted to VeriGuide (http://www.veriguide.org/) as well.
- You can submit your assignment multiple times. Only the latest submission counts.
- *Plagiarism is strictly monitored and heavily punished if proven.* Lending your work to others is subjected to the same penalty as the copier. You have to insert the following statement as comments in your Perl and C/C++ programs.

CSCI3180 Principles of Programming Languages

--- Declaration ---

I declare that the assignment here submitted is original except for source material explicitly acknowledged. I also acknowledge that I am aware of University policy and regulations on honesty in academic work, and of the disciplinary guidelines and procedures applicable to breaches of such policy and regulations, as contained in the website http://www.cuhk.edu.hk/policy/academichonesty/

Assignment 3

Name:

Student ID:

Email Addr: