CSCI3180 – Principles of Programming Languages – Spring 2020

Assignment 3 — Perl and Dynamic Scoping

Deadline: Apr 12, 2020 (Sunday) 23:59

1 Introduction

The purpose of this assignment is to offer you the first experience with Perl, which supports both dynamic and static scoping. Our main focus is on dynamic scoping.

The assignment consists of three parts. First, you are required to implement a popular card game called "Golden Hook Fishing" in Perl. Second, you need to implement a simplified Monopoly game in Perl with dynamic scoping. Third, you are required to re-implement the simplified Monopoly game using Python with static scoping. The detailed Object-Oriented (OO) design for both the Perl and Python implementations are given. In the process, you will experience both the programming flexibility and readability (good or bad) with dynamic scoping. Your implementation should be able to run on our VM environment with Perl v5.18.2 and Python 3.4.0. Besides, you are required to add "use warnings;" and "use strict;" at the start of your program. Good coding styles are expected.

IMPORTANT: All your codes will be graded on the VM. You are welcome to write and test your codes in your own computing environments, but please test them on the given VM before your submission (run your Python code with `python3` command instead of `python` on the VM).

NO PLAGIARISM! You are free to devise and implement the algorithms for the given tasks, but you must not "steal/borrow" codes from your classmates/friends. If you use some code snippets from public sources or your classmates/friends, ensure you cite them in the comments of your code. Failure to comply will be considered as plagiarism. You are not allowed to ask/hire others to do the assignment for you either. All these are considered as serious cheating acts, which will be dealt with severely.

2 Task 1: Golden Hook Fishing

In this task, you need to implement the "Golden Hook Fishing" game. You should strictly follow the specified OO design and game rules for this task. For the OO design, you have to follow the prototypes of the given classes exactly, but can choose to add new member variables and methods.

2.1 Description

There are 52 cards excluding "Joker" in a deck. At the beginning of each game, the deck is shuffled and distributed to N players evenly. In our game, we assume 52 is to be divisible by the player's number N. Besides, players cannot see the cards assigned to them. In other words, a deck of 52 cards is divided into N decks of $52/N$ cards to each player and none of them can see the cards.

Then players take turns to play cards. In each player's turn, he/she takes a card from the top of the his/her deck and adds it to the tail of the desktop card stack. If there is another same-character card in the stack, the players can take the cards between these two cards (include these two cards) and put them at the bottom of his/her deck. **A special situation is that if the player takes a "J" from the top of his/her deck, he/she can take all the cards on the desktop card**

stack to himself/herself. To make it possible to end the game in limited rounds, this situation is established if and only if the desktop card stack is not empty before a "J" is taken from the deck. In other words, if the "J" is the only card in the desktop card stack, then the "J" will be left in the desktop card stack as a regular card and the player will get nothing. A player is out when the player has no cards in his/her deck. The game will end when only one player remains. The last remaining player is the winner.

2.2 Perl Classes

Please follow the classes `Deck`, `Player`, `Game` defined below in your Perl implementation. You are free to add other variables, methods or classes. We would start the program by running: `GoldenHookFishing_sample1(2).pl`

1. Class `Deck` (**You are not allowed to modify this class**)
   An abstraction of a deck of cards. You have to implement it with the following components:

   - **Instance Variables**
     `cards`
     - This variable is the reference to a one-dimensional array recording initial cards in the deck. Each card is represented using the character in the face of this card. Suit information can be ignored since it will not affect the value of a card.
   - **Instance Method(s)**
     `new`
     - Instantiate a Deck object with a deck of 52 cards without shuffling, and return this object.

     `shuffle`
     - Shuffle all cards in the deck. Please strictly follow our design to call this function.

     `AveDealCards(num)`
     - Return an array of num reference of divided cards.

2. Class `Player`
   A super class representing a participant in the game. You have to implement it with the following components:

   - **Instance Variable(s)**
     `name`
     - This is a variable recording the name of the participant.

     `cards`
     - This variable is the reference to a one-dimensional array recording cards in the player's deck.
   - **Instance Method(s)**
     `new(name)`
     - Instantiate a Player object with its name, and return this object.

     `getCards(card)`
     - Put the taken cards to the bottom of his/her deck.

     `dealCards`
     - Take a card from the top of his/her deck and deal it to the game.

     `numCards`
     - Return the number of player's cards.

3. Class `Game`
   This class represents the process of the cards game. You have to implement it with the following components:

   - **Instance Variable(s)**
     `deck`
     – This is a Deck object.
     `players`
     – This variable is the reference to a one-dimensional array recording all players in the game. This array naturally encodes the order of all players.
     `cards`
     – This variable is the reference to a one-dimensional array recording the desktop card stack during the game.
   - **Instance Method(s)**
     `new`
     – Instantiate a variable deck with a Deck object and an array to record players.
     `setPlayers(players_name)`
     – "players_name" is a reference to a one-dimensional array recording the name of the players. The players variable will be instantiated by the players name.
     `getReturn`
     – Calculate how many will be returned to the player from the current desktop card stack. The operation follows the rule in description.
     `showCards`
     – Show the cards on the cards stack.
     `startGame`
     – Start a new game. First, deck will be shuffled and each participant will get a deck of cards evenly. And then players will deal cards and get cards by turn. In each player's turn, he/she firstly deals a card and then get some or no return according to the rule. Finally, the winner will be shown. You are required to output very important information in this function and more details can be found in the output specifications.

2.3 Output Specification

You need to output some required information of the game. Please refer to the output sample for more details.

2.4 Grading Criteria

The package "MannerDeckStudent.pm" inherits the "Deck.pm" package and implements the shuffle method, which uses a certain algorithm to shuffle the deck. Besides, we also provide students with two input samples, "GoldenHookFishing_sample1.pl" and "GoldenHookFishing_sample2.pl", as well as their corresponding output, "output_sample1.txt" and "output_sample1.txt". Students can use them to test the correctness of their programs.

For scoring, we will modify the algorithm in "MannerDeckStudent.pm" and compare the output from students' program and standard output. **We will use several test samples and only absolutely same outputs can get marks.**

To be fair, the codes of "Deck.pm" and "MannerDeckStudent.pm" will be given to students. **You only need to implement the "Player.pl" and "Game.pl" packages.**

3 Task 2: Monopoly

In this task, you need to implement a simplified Monopoly game in Perl. Note that dynamic scoping is beneficial and required here, and you need to explain why your implementation using dynamic scoping (specifically, with the "local" keyword in Perl) is more convenient.

With dynamic scoping, you can 1) implicitly affect the behaviour of functions by function calling sequences without passing extra parameters, and 2) temporarily mask out some existing variables. Full marks on this task demands you to show both of these two properties with your code and explanations.

You should strictly follow our OO design in you implementation. You have to follow the prototypes of the classes given in the template exactly, and are not allowed to modify the existing codes in the template. You can choose to add new member variables and functions if necessary.

3.1 Background

Monopoly is a board game currently published by Hasbro. In the game, players roll two six-sided dice to move around the game board, buying and trading properties, and developing land with houses and hotels. Players collect rent (toll) from their opponents, with the goal of driving them into bankruptcy. Money can also be gained or lost through Chance and Community Chest cards, and tax squares; players can end up in jail, from which they cannot move until they have met one of several conditions. The game has numerous house rules, hundreds of different editions, as well as many spin-offs and related media. Monopoly has become a part of international popular culture, having been licensed locally in more than 103 countries and printed in more than 37 languages.

3.2 Task Description

You are required to implement a simplified Monopoly game that only involves two players: Player A and Player B. The game board is organized as a 10×10 square (36 slots in total) as shown in Figure 1, and has three kinds of slots: Bank, Land, and Jail. When players step on different slots, they will do different actions.

Both players start from the "Bank" with \$100,000 and move clock-wisely. This game is turned-based, meaning the two players make moves alternately. Player A moves first. In each round, the players roll a six-sided dice in turn to decide how many steps to take. A Fixed cost of \$200 will be charged per round, even if the player is in Jail. Notice that if the player does not have enough money to pay the fixed cost, the amount he/she has to pay is the amount he/she has. The player can pay an extra \$500 to roll two dice and add up the numbers. When one of the players runs out of money, the game ends with the other player as the winner.

Bank	Land	Land	Land	Land	Land	Land	Land	Land	Jail
Land									Land
Land									Land
Land									Land
Land									Land
Land									Land
Land									Land
Land									Land
Land									Land
Jail	Land	Land	Land	Land	Land	Land	Land	Land	Jail

Figure 1: The game board

We explain the types of state and relevant rules as follows:

- Bank: A player who steps on the "Bank" collects \$2,000 from the bank.
- Jail: A player who steps on a "Jail" stops for two rounds by default, but the player can stop for only one round if he/she pays \$1,000.
- Land:
  1. If a player steps on an unowned piece of land, the player can buy the land with \$1000, but can also choose to decline this purchase. The development level of the land is initially 0.
  2. If a player steps on the land owned by him/herself, the player can choose to upgrade the development level of the land or not. There are four levels of land: 0, 1, 2 and 3. The land must be upgraded gradually, i.e., the player can only upgrade the land to level i when it is level $i-1$.
  3. If a player steps on the land owned by the other player, the player must pay the owner a given toll; the amount depends on the development level of the land.
  4. More explanation of "Land" is shown in Figures 2-4.
- Tax: Whenever a player receives income (e.g., receiving money when steps on Bank or receiving toll from the other player), tax will be charged. The tax rate varies according to different types of income as shown in Table 1.
- Handling fee: Whenever a player needs to pay (e.g., paying for toll or reducing prison rounds in Jail), handling fee will be charged. The handling fee rate varies according to different types of payments as show in Table 1.

Action	**amount**	**tax rate**	**handling fee rate**
Buy land	1000	/	0.1
Upgrade to level 1	1000	/	0.1
Upgrade to level 2	2000	/	0.1
Upgrade to level 3	5000	/	0.1
Pay toll of level 0 land	500	/	0
Pay toll of level 1 land	1000	/	0
Pay toll of level 2 land	1500	/	0
Pay toll of level 3 land	3000	/	0
Receive toll of level 0 land	500	0.1	/
Receive toll of level 1 land	1000	0.15	/
Receive toll of level 2 land	1500	0.2	/
Receive toll of level 3 land	3000	0.25	/
Pay to throw two dice	500	/	0.05
Pay to reduce prison rounds	1000	/	0.1
Receive money from Bank	2000	0	/
Fixed cost for every round	200	/	0

Table 1: The cost/income of different actions in the game.

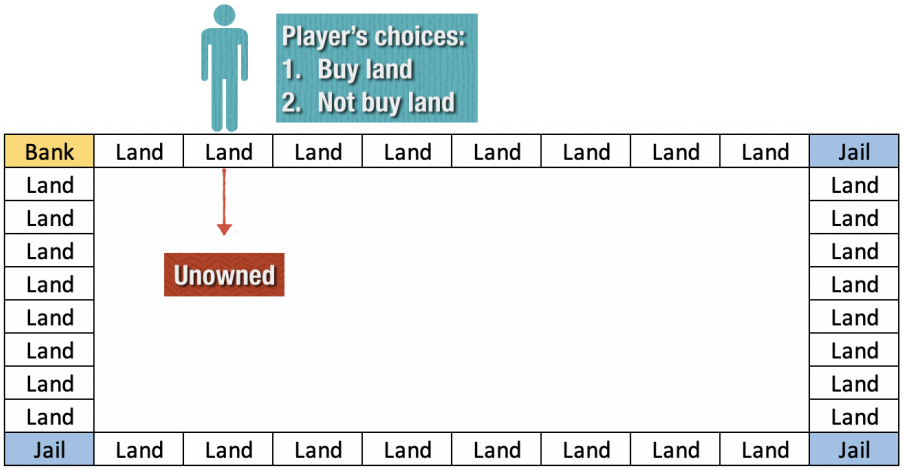

Figure 2: If the player steps on an unowned piece of land

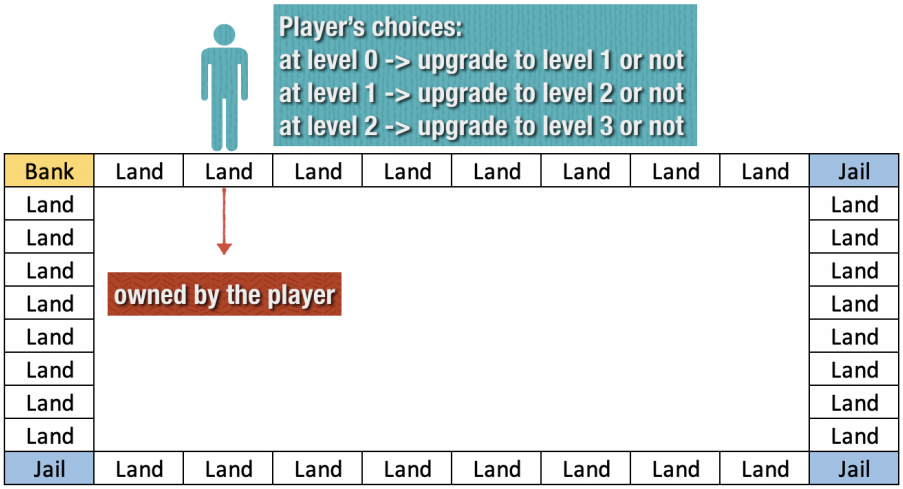

Figure 3: If the player steps on the player's own land

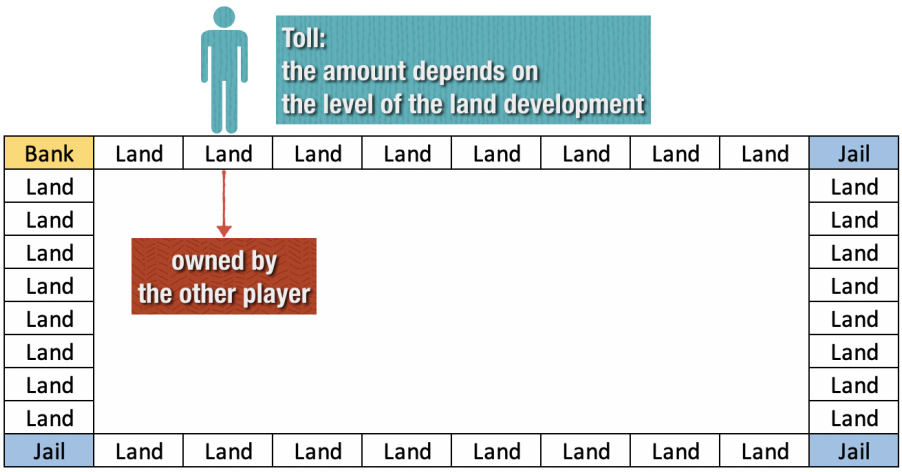

Figure 4: If the player steps on the other player's land

Hint Implement non-default cases via dynamic scoping. The basic logic of the game is presented in "monoply.pl", "Bank.pm", "Jail.pm" and "Land.pm" and "Player.pm", the templates of which have been provided to you. Please check the files for more details.

3.3 Perl Implementation Template

Please follow the classes `Bank`, `Jail`, `Land`, and `Player` defined below in your Perl implementation. You are free to add other variables, methods or classes, but cannot modify/delete ours. Please stick to our naming conventions. We would start the program by running: perl monopoly.pl. The template for the Python implementation is also provided in "monoply.py" and follows exactly the same structure of the Perl implementation.

1. Class `Player` (**You are not allowed to modify this class**)
   - **Instance Variables**

     `name`
     - Name of the player, which will be displayed on the game board.

     `money`
     - Total amount of money the player has, which is initially $100,000.

     `position`
     - The position of the player. The value of position is 0 when the player is at the Bank slot of the game board. The value increments clock-wisely and is reset to 0 at Bank slot.

     `num_rounds_in_jail`
     - The number of rounds remained in which the player cannot move.
   - **Instance Method(s)**

     `new`
     - Initializer of the Player instance.

     `move`
     - If the player is not in jail (i.e., `num_rounds_in_jail` = 0), move the position of the player according to the number of dice. Otherwise, decrease the `num_rounds_in_jail` by 1.

     `payDue`
     - Update the player's money according to values stored in `income`, `due`, `tax_rate`, `handling_fee_rate`. See the code template for concrete formula.

     `putToJail`
     - This method will be called when the player steps on a Jail slot. The instance variable `num_rounds_in_jail` is updated according to the value stored in `prison_rounds`.

     `printAsset`
     - Display the asset of the player.
2. Class `Bank` (in "Bank.pm")
   - **Instance Method(s)**

     `new`
     - Initializer of the Bank instance. You can add additional instance variables here if needed.

     `print`

– This method determines how Bank is displayed on the game board. Please do not modify this method.

`stepOn`

– This method is called when a player steps on the corresponding Bank slot. You are required to implement the rule of Bank according to Section 3.2. Your program should output "You received $2000 from the Bank!" when a player receives money from the Bank.

3. Class `Jail` (in "Jail.pm")

- **Instance Method(s)**

`new`

– Initializer of the Jail instance. You can add additional instance variables here if needed.

`print`

– This method determines how Jail is displayed on the game board. Please do not modify this method.

`stepOn`

– This method is called when a player steps on the corresponding Jail slot. You are required to implement the rule of Jail according to Section 3.2. When a player steps on it, your program should ask for user's input by printing "Pay $1000 to reduce the prison round to 1? [y/n]" and call `putToJail` method in `Player` class according to the player's answer. If the player does not have enough money, the program will output a warning message "You do not have enough money to reduce the prison round!".

4. Class `Land` (in "Land.pm")

- **Instance Method(s)**

`new`

– Initializer of the Land instance. You can add additional instance variables here if needed.

`print`

– This method determines how Land is displayed on the game board. Please do not modify this method.

`buyLand`

– This method is called when a player wants to buy land. If the player does not have enough money, the program will output a warning message "You do not have enough money to buy the land!", and the player cannot buy the land. You are required to implement the rule of buying land according to Section here

`upgradeLand`

– This method is called when a player wants to upgrade the land. If the player does not have enough money, the program will output a warning message "You do not have enough money to upgrade the land!", and the player cannot upgrade the land. You are required to implement the rule of upgrading land according to Section here

`chargeToll`

– This method is called when a player steps on a land owned by the other player and has to pay a toll. You are required to implement the rule of charging toll according to Section here. Notice that if the player does not have enough money to pay the toll, the amount he/she has to pay is the amount he/she has.

`stepOn`

– This method is called when a player steps on the corresponding Land slot. If the land is unowned, ask whether the player wants to buy the land by printing "Pay $1000 to buy the land? [y/n]" and take actions according to the players answer. If the land is owned by the player, ask whether the player wants to upgrade the land by printing "Pay $[upgrade fee] to upgrade the land? [y/n]" (replace [upgrade fee] with the corresponding value) and take actions according to the player's answer. If the land is owned by the other player, your program needs to output "You need to pay player [owner_name] $[amount]" (replace [owner_name] and [amount] by corresponding values) and charges the toll.

5. Main file, "monoply.pl"

- `printCellPrefix` and `printGameBoard`
  Utilities to print the game board and show the position of players on game board. Do not modify these two functions.
- `throwDice`
  Return the number of steps the player can move according to the dice. You are not allowed to modify this function.
- `terminationCheck`
  Check whether game has ended according to the rule. Return 1 if the game has not ended. Otherwise, return 0.
- `main`
  The entry of the game. Player A moves first. Each round starts by printing the game board and the players' money. A message "Player [current_player]'s turn." (replace [current_player] with the corresponding value) is also printed to indicate who is moving in this round. Your program will ask the player "Pay $500 to throw two dice? [y/n]" and throw the dice accordingly. If the player does not have enough money, the program will output a warning message "You do not have enough money to throw two dice!", and the player cannot throw two dice. The points of the dice is shown by printing "Points of dice: [points]" (replace [points] by the corresponding value). Afterwards, the player's position is updated accordingly and the new game board is printed. According to where the player steps on, different actions will be activated according to the rule. A typical round is shown in Figure 5. After each round, your program should check whether the game has ended. When the game is ended, print "Game over! winner: [winner_name]." (replace [winner_name] with corresponding value). Otherwise, continue to the next round.

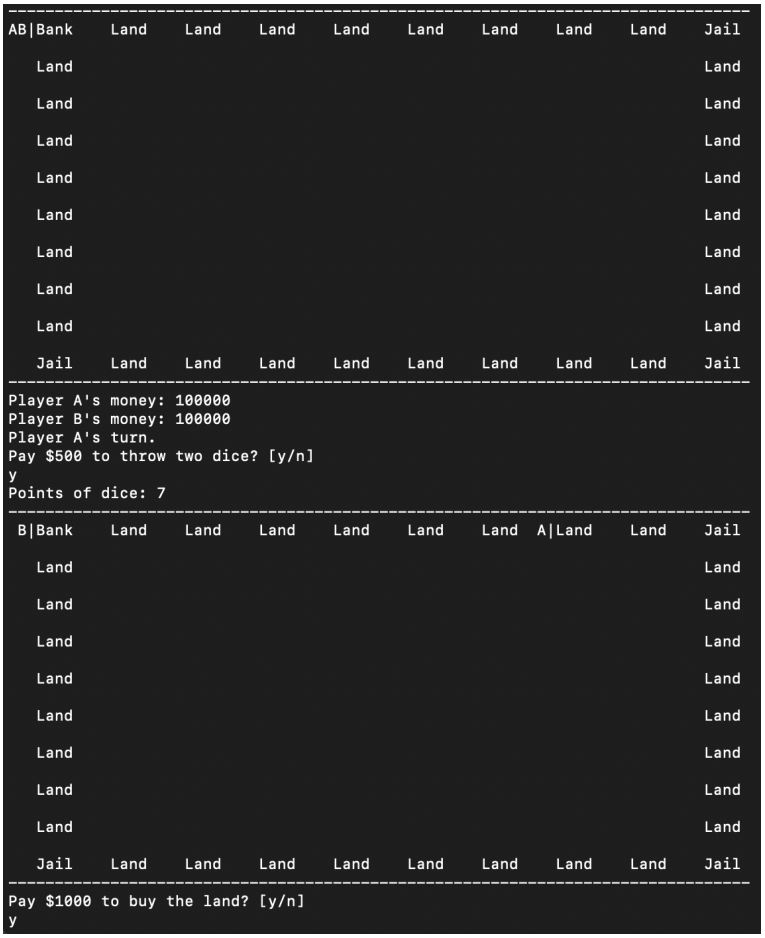

Figure 5: A typical round

3.4 Input/Output Specification

- **Input**
  The input is simply "y" or "n" representing the players decision. Notice that if the player enters an invalid answer, i.e., neither "y" nor "n", your program should ask the player again until he/she enters a valid answer.

- **Output**
  The codes for printing the game board are given in the skeleton. Do not change them.
  The name printed in each slot of the game board represents the type of the slot. The name of Bank and Jail is simply "Bank" and "Jail". But notice that we use "Land" to represent an unowned land and "A:Lv0" to represent Player A owns the land of level 0. Similar notations are used for Player B and other levels of the land.

  The positions of players are shown by prefixing the slot name with the players' name. For example, in Figure 5 at the beginning, both Player A and Player B are at the Bank slot, and later, Player A is moved to postion 7 which is a Land slot.

  For other outputs, you should strictly follow the format specified in Section 3.3 (The blue-colored texts).

3.5 Grading Criteria

Your program should run by calling "perl monopoly.pl" and will be tested this way. You will also be evaluated on your implementation of the given classes (with dynamic scoping when necessary) and programming style.

3.6 Implementing Monopoly via Python

In this task, you are asked to implement the Monopoly game using Python with exactly the same class design. You should adopt and adapt to the class design and skeleton code in Section 3 for this task. Also, the input/output specifications and grading criteria are the same as those in the previous task. All your implementations are required to be written into a single Python file named as "monopoly.py". Note that Python does not support dynamic scoping.

4 Report

You should give a simple report to answer the following questions within two A4 pages:

1. Provide example code and necessary elaborations for demonstrating the advantages of Dynamic Scoping in using Perl to implement the simplified Monopoly game as compared to the corresponding codes in Python.
2. Discuss the keyword *local* in Perl (*e.g.* its origin, its role in Perl, and real practical applications of it) and giving your own opinions.

5 Submission Guidelines

Please read the guidelines CAREFULLY. If you fail to meet the deadline because of submission problem on your side, marks will still be deducted. The late submission policy is as follows:

- 1 day late: -20 marks
- 2 days late: -40 marks
- 3 days late: -100 marks

So please start your work early!

1. In the following, **SUPPOSE**

   your name is *Chan Tai Man*,
   your student ID is *1155234567*,
   your username is *tmchan*, and
   your email address is *tmchan@cse.cuhk.edu.hk*.

2. In your source files, insert the following header. REMEMBER to insert the header according to the comment rule of Perl and Python.

```
/*
 * 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 : Chan Tai Man
* Student ID : 1155234567
* Email Addr : tmchan@cse.cuhk.edu.hk
*/
```

The sample file header is available at

`http://course.cse.cuhk.edu.hk/~csci3180/resource/header.txt`

3. The report should be submitted to VeriGuide, which will generate a submission receipt. The report and receipt should be submitted together with your Perl and Python codes in the same ZIP archive.

4. For task 1, the Perl source should have the filename "Deck.pm", "Player.pm", "Game.pm" and "MannerDeckStudent.pm". For task 2, the Perl source should have the filename "Land.pm", "Bank.pm", "Jail.pm", "Player.pm" and "monopoly.pl". The Python source should have the filename "monopoly.py". The report should have the filename "report.pdf". The VeriGuide receipt of report should have the filename "receipt.pdf". All file naming should be followed strictly and without the quotes.

5. `Tar` your source files to `username.tar`. Don't change the directory structure of the given code skeleton.

```
tar cvf tmchan.tar task1 task2 report.pdf receipt.pdf
```

6. `Gzip` the `tarred` file to `username.tar.gz` by

```
gzip tmchan.tar
```

7. `Uuencode` the `gzipped` file and send it to the course account with the email title "HW3 *studentID yourName*" by

```
uuencode tmchan.tar.gz tmchan.tar.gz \
| mailx -s "HW3 1155234567 Chan Tai Man" csci3180@cse.cuhk.edu.hk
```

8. Please submit your assignment using your Unix accounts.

9. An acknowledgement email will be sent to you if your assignment is received. **DO NOT** delete or modify the acknowledgement email. You should contact your TAs for help if you do not receive the acknowledgement email within 5 minutes after your submission. **DO NOT** re-submit just because you do not receive the acknowledgement email.

10. You can check your submission status at

`http://course.cse.cuhk.edu.hk/~csci3180/submit/hw3.html`.

11. You can re-submit your assignment, but we will only grade the latest submission.

12. Enjoy your work :>