CSCI3180 – Principles of Programming Languages – Spring 2019

Assignment 3 — Perl and Dynamic Scoping

Deadline: Apr 7, 2019 (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 two parts. First, you need to implement a game called "Maze Race" in Perl with dynamic scoping. Then, you are required to re-implement this task using Python with static scoping. The detailed object-oriented design for the Perl implementation is given, and you are asked to design a similar framework in the Python realization. In the process, you will experience both the programming flexibility and readability with dynamic scoping. Your implementation in Perl should be able to run with Perl v5.14.2. Besides, you are required to add "*use warnings;*" at the start of your program. For the Python part, Python 3.6 is demanded. Good coding styles are expected.

IMPORTANT: All your codes will be graded on the Linux machines in the Department. You are welcome to write your codes on your own machines, but please test them on the given Departments machines (the perl version in linux1 is 5.14.2) before your submission.

NO PLAGIARISM! You are free to devise and implement the algorithms for the given tasks, but you must not "steal/copy" 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.

2 Task: Maze Race

In this task, you need to implement a maze race game with 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 behavior of functions in 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 given classes exactly. You can choose to add new member variables and functions if necessary.

2.1 The Story

In ancient Greece, there was a mysterious creature called Minotaur, with a bull-like head and tail, and a humanoid body. The surrounding villagers had been suffering from the monster's bullying for a long time. Barely a man could match Minotaur's mighty power and agility, let along killing it. To seek peaceful living, Greek people came up with a temporary solution to isolate the monster. The architect Daedalus and his son Icarus designed and built the Labyrinth (a Greek term for maze). Then they seduced Minotaur to dwell at the center of the Labyrinth to prevent it from engaging with humans. But still, nobody could deliver death to Minotaur even it was trapped in the building. Years later, the mighty Greek hero Theseus rose and took the mission of terminating Minotaur. He received the blueprint of the Labyrinth from Daedalus and found a feasible path to kill Minotaur.

Thousands years after the battle between Theseus and Minotaur, rumors are spread across the world that Theseus left some treasures in that Labyrinth. You, a promising and gifted Greek explorer, make up your mind to retrieve these precious items for your homeland. Meanwhile, a heritage smuggling company employs a treasure hunter to loot for Theseus's legacy. So please be cool and quick, draw your strategies now.

2.2 Task Description

You are required to realize a treasure hunting game that involves two players: Player 1 and Player 2. Both players are located in a different entrance of a maze respectively. The maze is organized into a 2-dimensional array of cells, which can be empty or occupied by a solid wall, a player, or the treasure. Besides, they both know where the treasure is but *initially* know nothing about the maze itself except their own locations and the dimensions of the maze (height and width). The initial positions of the two players are of the same Manhattan distance to the treasure. Normally, a player can attempt to move one step to an adjacent empty cell in the north (N), south (S), east (E), and west (W) directions. A player cannot move out of the maze. A wall always block a move, except when a special move is employed. Initially, all cells are unknown, except those containing the two players and the treasure. As a player attempts to move to the next cell, the new cell will be revealed to all players.

This game is turned-based, meaning the two players make move alternately. *Player 1 moves first.* The player who *reaches* the treasure first will win this game. Note that *reaching* the treasure means the player *has just moved into* the cell containing the treasure.

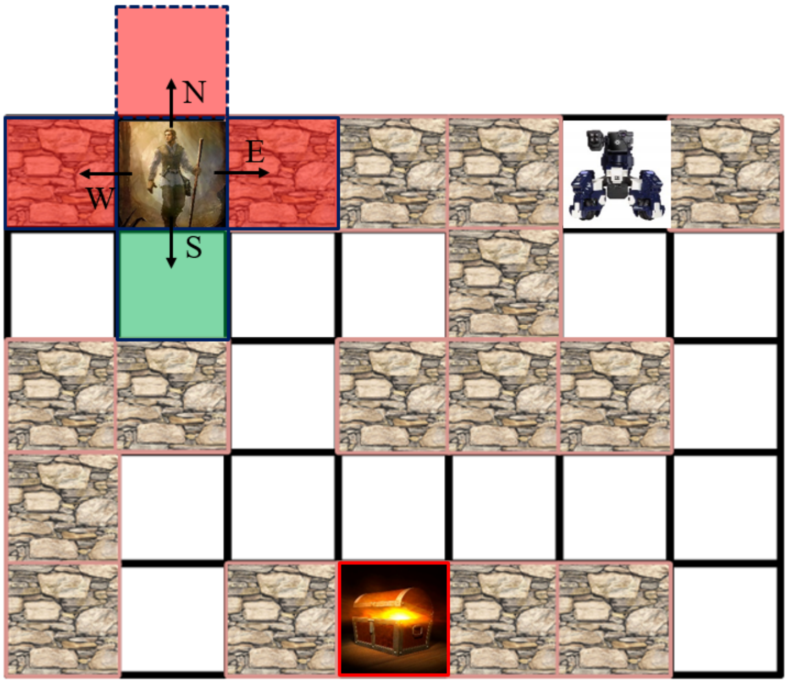


Figure 1: The neighborhood of Player 1 using *move*. The green cell is feasible while red cells are blocked. Best viewed on screen.

Players move in the maze alternately. In addition to a *normal move*, a player can make three more special moves: *rush*, *walk through a blocked cell*, and *teleport*.

- A *move* enables a player to move from his/her current location to a neighboring (the 4 compass direction: N, W, S, E, like where the arrows point to in Figure 1) and *feasible/available* (*empty* or *only containing the treasure*) cell. If the neighboring cell of the current player is a wall or occupied by the other player, then the current player remains in the same cell after attempting the move. The detailed logic of *move* is given in the skeleton code (`maze_race_components.pm`) within class `Player`.
- Players can *rush* through a clean (no obstacles) vertical or horizontal path until right before an edge of the maze, or a cell that is occupied by a wall or player. Check out Figure 2 for illustrations. All cells along the path of a rush will be revealed.

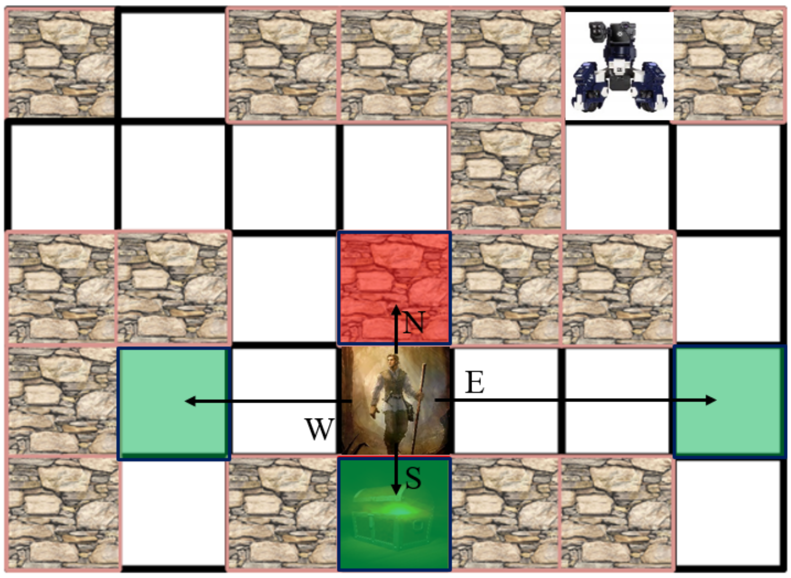


Figure 2: The neighborhood of Player 1 using *rush*. The green cells are feasible while red cell is blocked. Best viewed on screen.

- If a player is next to a wall or the other player. This player can choose to *walk through* the blocked cell (by the wall/player) to the next empty cell. This special move cannot go through more than one level of blocked cell though. On the other hand, if there is no wall/player in front of the player, then *walk through a blocked cell* degenerates to a normal *move*.

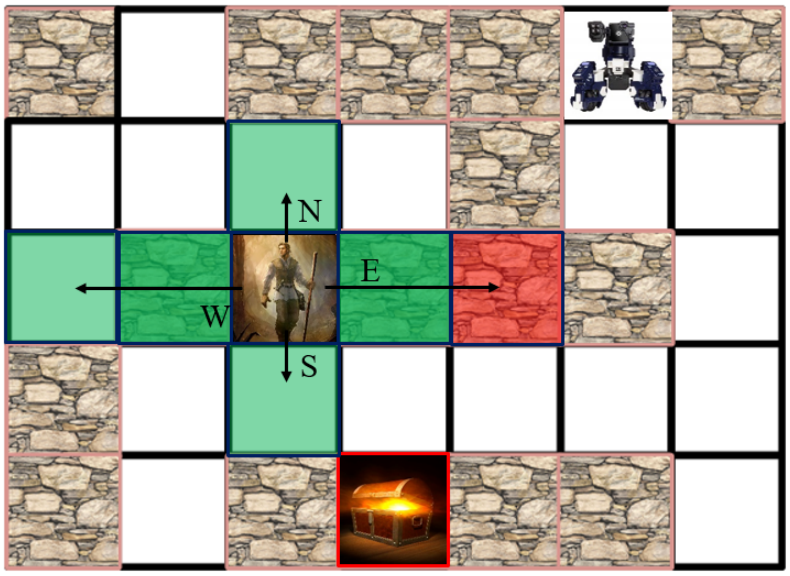


Figure 3: The neighborhood of Player 1 using *walk through a blocked cell*. The green cells are feasible while red cell is blocked. Best viewed on screen.

- Influenced by the magic in the maze, players have a choice to be teleported to a *random* cell within the maze. If the target cell is feasible then this teleportation succeeds. Otherwise, this teleportation fails and the player remains in the original cell, but the target cell is revealed on the map.

Note Each player can only make special moves at most four times. Moreover, failed move attempts would not change players' locations, but they would reveal the cell contents (feasible or not) on the map if that cell has not been explored yet.

Hint Implement special moves via dynamic scoping.

The basic logic of the game is presented in "maze_race.pl", which has been provided to you. Please check the file for more details.

2.3 Perl Classes

Please follow the classes `MazeRace`, `Player`, `Maze`, `Cell`, and `Position` 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 convention and getter/setter style too. We would start the program by running: `perl maze_race.pl`.

1. **Class** `MazeRace`

   This class realizes a simple game board for hosting the competition between two players. You have to implement it with the following components:

   - **Instance Variable(s)**

     `maze`

     - A maze instance of class `Maze` defined below, maintaining maze related information. In the following specification, all variables named as `maze` have the same meaning as here unless otherwise specified.

     `player1`

     - A player instance of class `Player` (defined below) standing for player 1.

     `player2`

     - A player instance of class `Player` (defined below) standing for player 2.

   - **Instance Method(s)**

     `new(file)`

     - Instantiate an object of Class `MazeRace`, initialize its maze and players' information (the name of player 1 is 'E' and the name of player 2 is 'H') from `file`, and return this object. `file` is a string storing the path of the test file. In the following specification, all variables named as `file` have the same meaning as here unless otherwise specified.

     `start()`

     - Start the maze race.

2. **Class** `Player`

   A class representing a player in the game. You have to implement it with the following components:

   - **Instance Variable(s)**

     `name`

     - The name of the player.

     `curPos`

     - The current position of the player. It is an instance of class `Position`.

     `specialMovesLeft`

     - How many move times that the current player can make a special move (no matter whether this move fails or succeeds). It should be initialized to 4.

     `rshift`

     - A one-dimensional array describing the offsets related to this position in row to stand for a normal move to the S, E, N and W directions, *e.g.* our $rshift = (1, 0, -1, 0);

     `cshift`

- A one-dimensional array describing the offsets related to this position in column to stand for a normal move to the S, E, N and W directions, *e.g.* our $cshift = (0, 1, 0, -1);

- **Instance Method(s)**

  `new(name)`

  - Instantiate an object of Class `Player` with its name, and return this object.

  `setName(name)`

  - The setter of this player's name.

  `getName()`

  - The getter of this player's name.

  `getPos()`

  - The getter of this player's position. It returns an object of class `Position`.

  `occupy(maze)`

  - Update the cell content as the player moves into the current location in the `maze`.

  `leave(maze)`

  - Update the cell content as the player leaves the current location in the `maze`.

  `move(pointTo, maze)`

  - The player moves to a new location (one step away) according to the given *direction* indicated by `pointTo` in that *maze*. If he/she is blocked on the way, he/she would remain still. In our game setting, `pointTo` is an integer from 0 to 3 indicating the compass direction (0: south, 1: east, 2: north, 3: west). In the following specification, all variables named as `pointTo` have the same meaning as here unless otherwise specified.

  `next(pointTo)`

  - Return the position of the neighbor of the player in the direction `pointTo`.

  `rush(pointTo, maze)`

  - The player rushes to a new location based on the given direction (`pointTo`) in that `maze`. The player would stop at where he/she is hindered on the way.

  `throughBlocked(pointTo, maze)`

  - The player can walk through one blocked cell when he/she is facing a wall or the other player and the cell behind the wall/player is available.

  `teleport(maze)`

  - The player will be teleported to a random cell in the maze. Note that the target cell could be in any position of the maze, including this player's current location, other player's location, infeasible cells blocked by walls, available cells, and the location of the treasure.

  `makeMove()`

  - Make the current player move to a position based on the instruction of the human user. The interaction can be invoked by standard I/O.

3. **Class `Maze`**

   A class representing a maze in the game. You have to implement it with the following components:

   - **Instance Variable(s)**

     `map`

     - A two-dimensional array of object of class `Cell` recording maze information. Initially, all cells are not explored.

     `height`

     - The height of the maze.

     `width`

- The width of the maze.

`destPos`

- The position of the destination (where we can find the treasure). It is an instance of class `Position`.

- **Instance Method(s)**

  `new()`

  - Instantiate an object of Class `Maze`, initialize its instance variables, and return this object.

  `getHeight()`

  - The getter of the height of the maze.

  `getWidth()`

  - The getter of the width of the maze.

  `getCell(pos)`

  - The getter of the cell of the maze in the given position (`pos`). `pos` is an instance of class `Position`. In the following specification, all variables named as `pos` have the same meaning as here unless otherwise specified.

  `setCell(pos, cell)`

  - The setter of the cell of the maze in the given position (`pos`). `cell` is an instance of class `Cell`.

  `getCellContent(pos)`

  - The getter of the cell content of the maze in the given position (`pos`).

  `setCellContent(pos, value)`

  - The setter of the cell content of the maze in the given position (`pos`).

  `isAvailable(pos)`

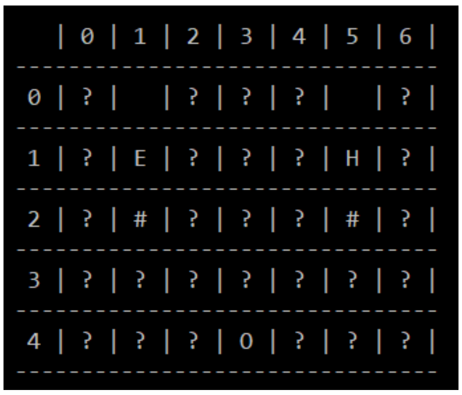
  - Check whether the cell in the given position (`pos`) is feasible or not. It will return a -1/0/1 value (-1: infeasible, 0: out of range, 1: feasible).

  `explore(pos)`

  - Mark the cell at the given position (`pos`) as explored.

  `reachDest(pos)`

  - Return whether we can find the treasures in the given position (`pos`) as a 0/1 value (0: not found, 1: found).



```
  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
-------------------------------
0 | ? |   | ? | ? | ? |   | ? |
-------------------------------
1 | ? | E | ? | ? | ? | H | ? |
-------------------------------
2 | ? | # | ? | ? | ? | # | ? |
-------------------------------
3 | ? | ? | ? | ? | ? | ? | ? |
-------------------------------
4 | ? | ? | ? | O | ? | ? | ? |
-------------------------------
```

Figure 4: A visual example about the maze information.

  `displayMaze()`

  - Print the maze information to a standard output device (screen). Specifically, players are represented by the initials of their names (like 'E' and 'H' by default), and the treasure is denoted by 'O'. For known cells, '␣' (one blank) denotes empty cell and '#' denotes wall. For unknown cells, they are all shown as '?'. An example is given in Figure 4.

`loadMaze(file)`

- Load the maze related information from a given `file` (to be explained in further details).

4. **Class `Cell`**

   A class representing a cell in the maze. You have to implement it with the following components:

   - **Instance Variable(s)**

     `pos`

     - The position of this cell. Here `pos` is an instance of class `Position` defined below.

     `content`

     - A character representing the content of this cell. '#' denotes wall, and '*' denotes an empty cell, 'E' denotes Player 1, 'H' denotes Player 2, and 'O' denotes the treasure.

     `explored`

     - A 0/1 value indicating whether this cell is explored (1) or not (0).

   - **Instance Method(s)**

     `new()`

     - Instantiate an object of Class `Cell`, initialize its instance variables, and return this object.

     `setExplored(value)`

     - The setter of the variable `explored`.

     `getExplored()`

     - The getter of the variable `explored`.

     `setPos(pos)`

     - The setter of the variable `pos` which is an object of class `Position`.

     `getPos()`

     - The getter of the variable `pos`.

     `setContent(value)`

     - The setter of the variable `content`.

     `getContent()`

     - The getter of the variable `content`.

     `isAvailable()`

     - Return whether this cell is available (1) or not (0) as a 0/1 value.

5. **Class `Position`**

   A class representing a position in the maze. You have to implement it with the following components:

   - **Instance Variable(s)**

     `r`

     - The row number of this position.

     `c`

     - The column number of this position.

- **Instance Method(s)**

  `new()`

  - Instantiate an object of Class `Player`, initialize its instance variables, and return this object.

  `getC()`

  - The getter of the column number `c`.

  `getR()`

  - The getter of the row number `r`.

  `setC(value)`

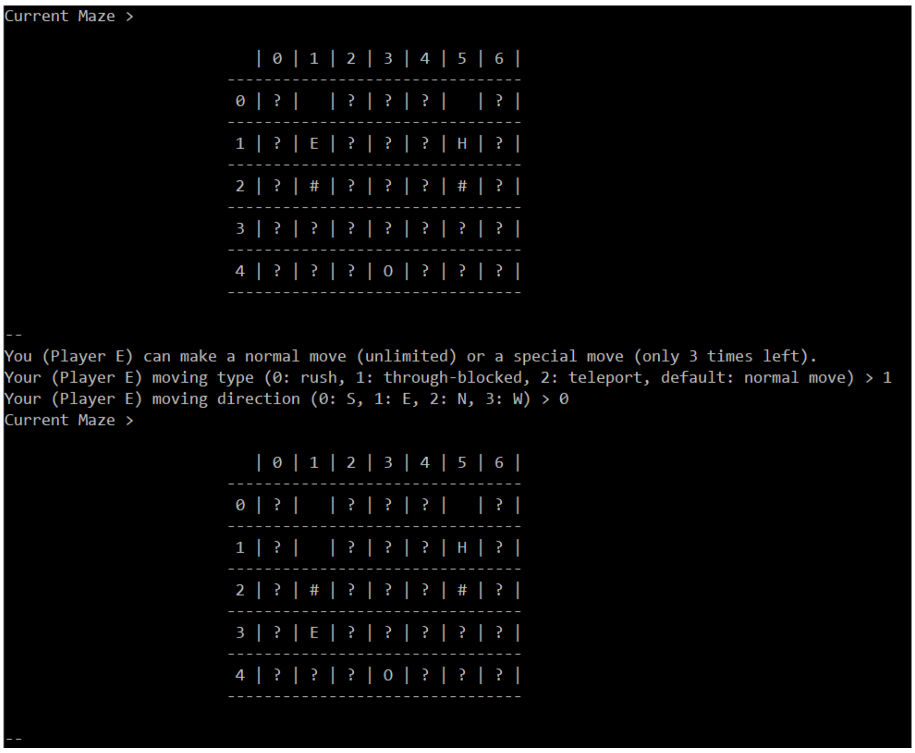
  - The setter of the column number `c`.

  `setR(value)`

  - The setter of the row number `r`.

2.4 Execution Context

We provide you with the skeleton code in "maze_race.pl" and "maze_race_components.pm". You are not allowed to change our code, but can only insert your own code. Your implementation should be able to run with the main program in "maze_race.pl".



```
Current Maze >

                        | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                      -------------------------------
                      0 | ? |   | ? | ? | ? |   | ? |
                      -------------------------------
                      1 | ? | E | ? | ? | ? | H | ? |
                      -------------------------------
                      2 | ? | # | ? | ? | ? | # | ? |
                      -------------------------------
                      3 | ? | ? | ? | ? | ? | ? | ? |
                      -------------------------------
                      4 | ? | ? | ? | O | ? | ? | ? |
                      -------------------------------

--
You (Player E) can make a normal move (unlimited) or a special move (only 3 times left).
Your (Player E) moving type (0: rush, 1: through-blocked, 2: teleport, default: normal move) > 1
Your (Player E) moving direction (0: S, 1: E, 2: N, 3: W) > 0
Current Maze >

                        | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                      -------------------------------
                      0 | ? |   | ? | ? | ? |   | ? |
                      -------------------------------
                      1 | ? |   | ? | ? | ? | H | ? |
                      -------------------------------
                      2 | ? | # | ? | ? | ? | # | ? |
                      -------------------------------
                      3 | ? | E | ? | ? | ? | ? | ? |
                      -------------------------------
                      4 | ? | ? | ? | O | ? | ? | ? |
                      -------------------------------

--
```

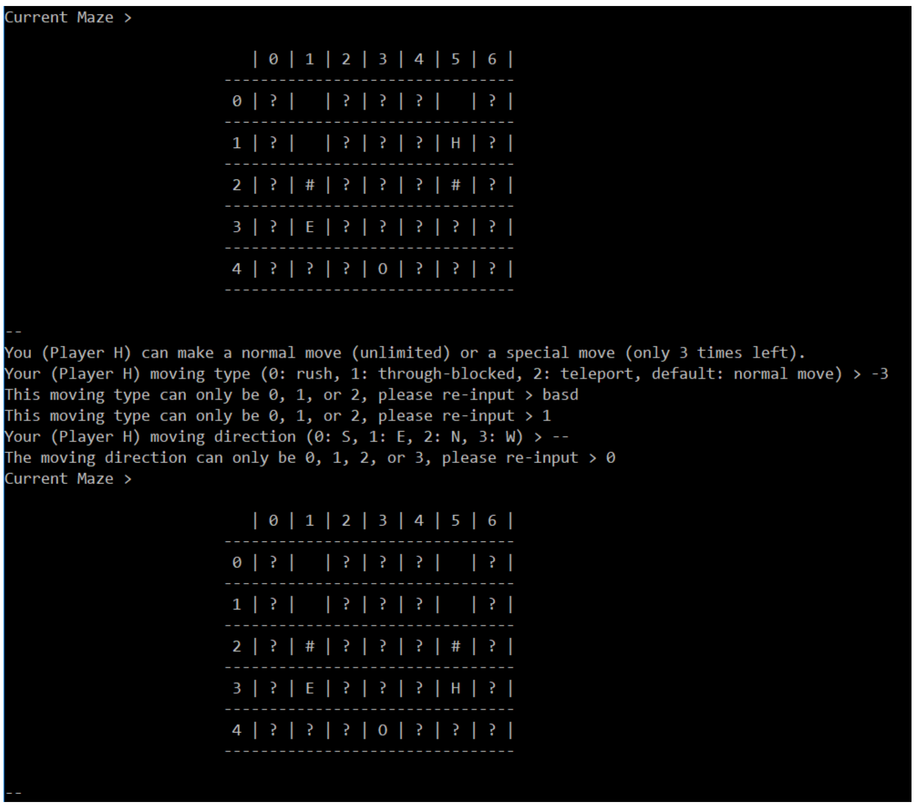
Figure 5: The input of Player 1.

2.5 Input/Output Specification

All possible mazes with the positions of the treasures and two players are given in the specification file. The data format is given below.

- **Input**

```
H W
#1###2#
```



```
Current Maze >

                          | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                        -------------------------------
                        0 | ? |   | ? | ? | ? |   | ? |
                        -------------------------------
                        1 | ? |   | ? | ? | ? | H | ? |
                        -------------------------------
                        2 | ? | # | ? | ? | ? | # | ? |
                        -------------------------------
                        3 | ? | E | ? | ? | ? | ? | ? |
                        -------------------------------
                        4 | ? | ? | ? | O | ? | ? | ? |
                        -------------------------------

--
You (Player H) can make a normal move (unlimited) or a special move (only 3 times left).
Your (Player H) moving type (0: rush, 1: through-blocked, 2: teleport, default: normal move) > -3
This moving type can only be 0, 1, or 2, please re-input > basd
This moving type can only be 0, 1, or 2, please re-input > 1
Your (Player H) moving direction (0: S, 1: E, 2: N, 3: W) > --
The moving direction can only be 0, 1, 2, or 3, please re-input > 0
Current Maze >

                          | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                        -------------------------------
                        0 | ? |   | ? | ? | ? |   | ? |
                        -------------------------------
                        1 | ? |   | ? | ? | ? |   | ? |
                        -------------------------------
                        2 | ? | # | ? | ? | ? | # | ? |
                        -------------------------------
                        3 | ? | E | ? | ? | ? | H | ? |
                        -------------------------------
                        4 | ? | ? | ? | O | ? | ? | ? |
                        -------------------------------

--
```

Figure 6: Check the input for Player 2 and request the input again if that is invalid.

```
****#**
##*###*
#******
#*#O##*
```

The first line gives the height (H, where $2 \leq H \leq 70$) and width (W, where $2 \leq W \leq 70$) of the maze. The remaining H lines give the maze information. The detailed maze description is given in a 2D matrix form, where '#' denotes wall and '*' denotes an empty cell, 'O' denotes treasure, '1' and '2' denote players 1 and 2 respectively. Note 'O', '1' and '2' only show up once in the maze information.

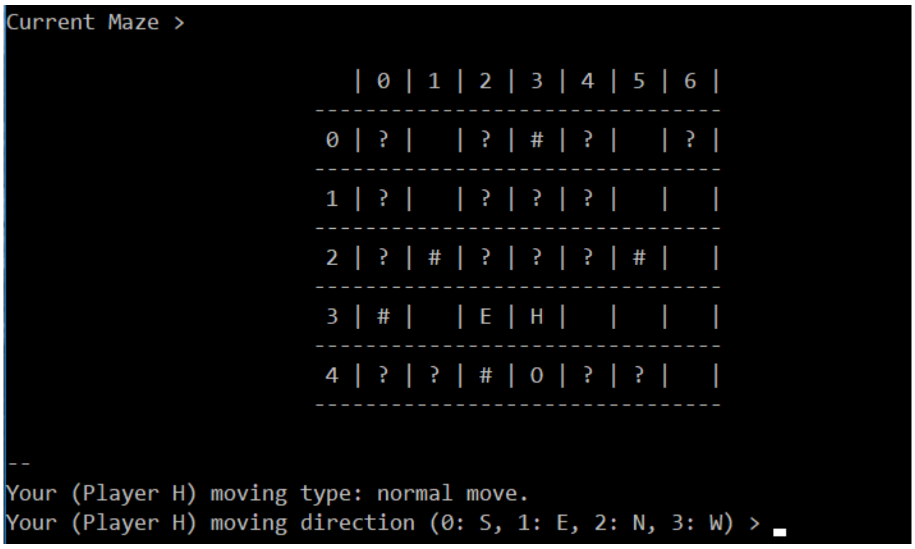
Note each line in the given test file is ended with '\n'. Once again, we use '*' to denote an empty cell in the input file while '␣' (a blank) is used to show an empty cell on the screen (Figure 4).

You can assume the input file to be error free.

How to control a Player's Move Players require user-inputs to determine their next moves. When it is one player's turn during the game, your program should request for the type and direction of the next move. The input should be integers from 0 to 2 for the type and from 0 to 3 for the direction. When inputting the moving type, hitting ENTER directly without giving any number will control the current player to make a normal move.

If the target cell is available, the player will move to that cell. If not, the information of the target cell will be uncovered. An example is presented in Figure 5. After this, the system switches to the other player's turn automatically.

Moreover, any invalid input should be forbidden and the user is requested for input again until a valid move is given. See Figure 6. Note we will also display the number of remaining special moves that a player can make in each turn. When a player runs out of all his special moves,



```
Current Maze >

                        | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                     -------------------------------
                      0 | ? |   | ? | # | ? |   | ? |
                     -------------------------------
                      1 | ? |   | ? | ? | ? |   |   |
                     -------------------------------
                      2 | ? | # | ? | ? | ? | # |   |
                     -------------------------------
                      3 | # |   | E | H |   |   |   |
                     -------------------------------
                      4 | ? | ? | # | O | ? | ? |   |
                     -------------------------------


--
Your (Player H) moving type: normal move.
Your (Player H) moving direction (0: S, 1: E, 2: N, 3: W) > 
```

Figure 7: Player 2 used up all his/her special moves, and he/she is defaulted to make a normal move from now on.

this player needs no more user-input for choosing the moving type (the moving direction still requires interaction) as shown in Figure 7.

If you choose 3 (*teleport*), then there is no need for direction input.

Your implementation should output exactly the same message as shown in the examples in the figures.

- **Output**
  Figures 8 and 9 show partial output during the game. Specifically, at the beginning of the game, only the locations of Player 1 and 2 and the treasures are visible, and the remaining cells are invisible, denoted by '?'. No matter whether the movement of a player succeeds or fails, the information (replacing '?' with '#', '␣', 'E', or 'H') of the target cell will be revealed permanently.

  When a player reaches the treasure first, the game will output the information like Figure 10 and then end.

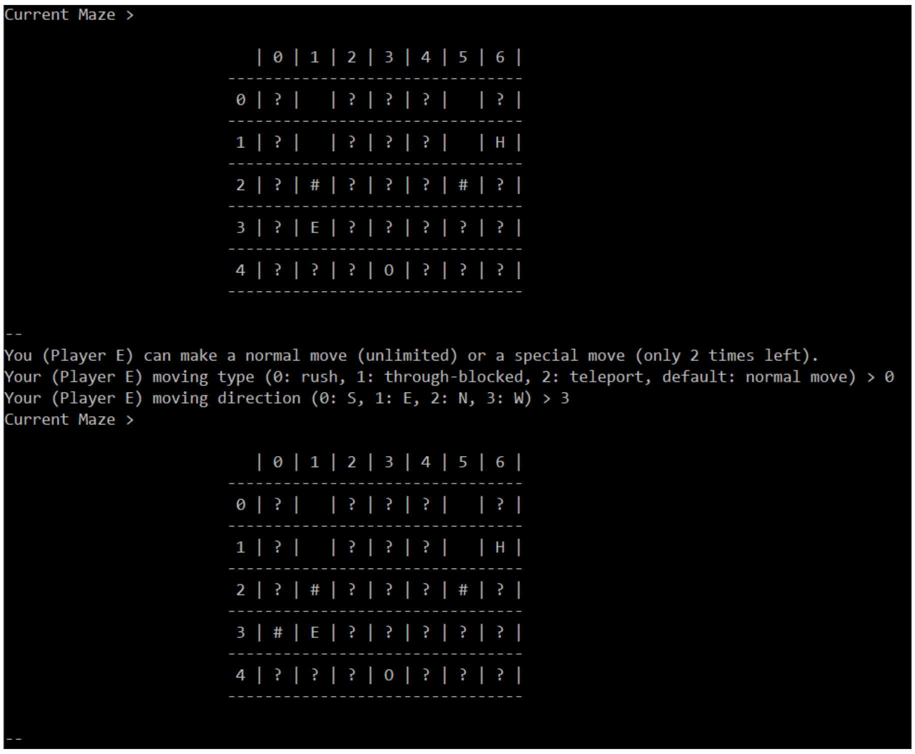
  Note all the input and output statements are provided to you in the skeleton code. Do not modify them in any way.

2.6 Grading Criteria

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

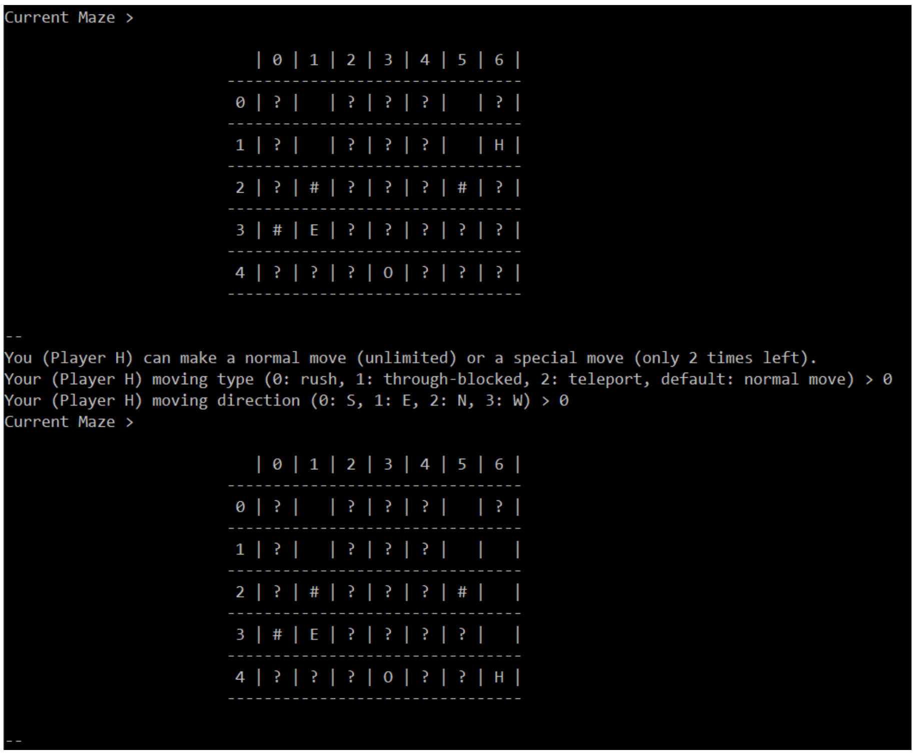
3 Implementing Maze Race via Python

In this task, you are asked to implement Maze Race using Python with exactly the same class design. You should adopt and adapt to the class design and skeleton code in Section 2 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 "maze_race.py".



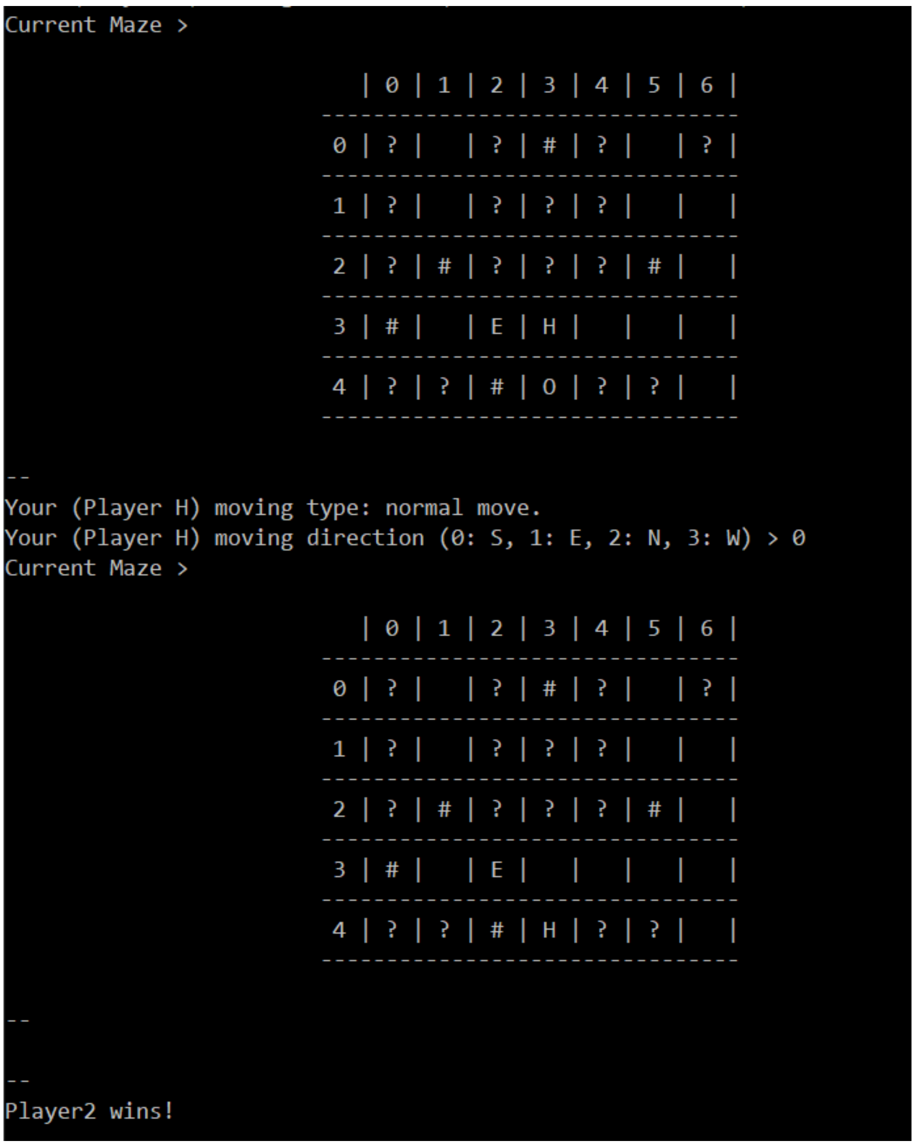
```
Current Maze >

                          | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                        -------------------------------
                        0 | ? |   | ? | ? | ? |   | ? |
                        -------------------------------
                        1 | ? |   | ? | ? | ? |   | H |
                        -------------------------------
                        2 | ? | # | ? | ? | ? | # | ? |
                        -------------------------------
                        3 | ? | E | ? | ? | ? | ? | ? |
                        -------------------------------
                        4 | ? | ? | ? | O | ? | ? | ? |
                        -------------------------------

--
You (Player E) can make a normal move (unlimited) or a special move (only 2 times left).
Your (Player E) moving type (0: rush, 1: through-blocked, 2: teleport, default: normal move) > 0
Your (Player E) moving direction (0: S, 1: E, 2: N, 3: W) > 3
Current Maze >

                          | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                        -------------------------------
                        0 | ? |   | ? | ? | ? |   | ? |
                        -------------------------------
                        1 | ? |   | ? | ? | ? |   | H |
                        -------------------------------
                        2 | ? | # | ? | ? | ? | # | ? |
                        -------------------------------
                        3 | # | E | ? | ? | ? | ? | ? |
                        -------------------------------
                        4 | ? | ? | ? | O | ? | ? | ? |
                        -------------------------------

--
```

Figure 8: The partial output of the game state in one turn. The turn of Player 1.



```
Current Maze >

                          | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                        -------------------------------
                        0 | ? |   | ? | ? | ? |   | ? |
                        -------------------------------
                        1 | ? |   | ? | ? | ? |   | H |
                        -------------------------------
                        2 | ? | # | ? | ? | ? | # | ? |
                        -------------------------------
                        3 | # | E | ? | ? | ? | ? | ? |
                        -------------------------------
                        4 | ? | ? | ? | O | ? | ? | ? |
                        -------------------------------

--
You (Player H) can make a normal move (unlimited) or a special move (only 2 times left).
Your (Player H) moving type (0: rush, 1: through-blocked, 2: teleport, default: normal move) > 0
Your (Player H) moving direction (0: S, 1: E, 2: N, 3: W) > 0
Current Maze >

                          | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                        -------------------------------
                        0 | ? |   | ? | ? | ? |   | ? |
                        -------------------------------
                        1 | ? |   | ? | ? | ? |   |   |
                        -------------------------------
                        2 | ? | # | ? | ? | ? | # |   |
                        -------------------------------
                        3 | # | E | ? | ? | ? | ? |   |
                        -------------------------------
                        4 | ? | ? | ? | O | ? | ? | H |
                        -------------------------------

--
```

Figure 9: The partial output of the game state in one turn. The turn of Player 2.



```
Current Maze >

                  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                -------------------------------
                0 | ? |   | ? | # | ? |   | ? |
                -------------------------------
                1 | ? |   | ? | ? | ? |   |   |
                -------------------------------
                2 | ? | # | ? | ? | ? | # |   |
                -------------------------------
                3 | # |   | E | H |   |   |   |
                -------------------------------
                4 | ? | ? | # | O | ? | ? |   |
                -------------------------------


--
Your (Player H) moving type: normal move.
Your (Player H) moving direction (0: S, 1: E, 2: N, 3: W) > 0
Current Maze >

                  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                -------------------------------
                0 | ? |   | ? | # | ? |   | ? |
                -------------------------------
                1 | ? |   | ? | ? | ? |   |   |
                -------------------------------
                2 | ? | # | ? | ? | ? | # |   |
                -------------------------------
                3 | # |   | E |   |   |   |   |
                -------------------------------
                4 | ? | ? | # | H | ? | ? |   |
                -------------------------------


--


--
Player2 wins!
```

Figure 10: Player 2 wins the game.

4 Report

Your report should explain the following questions within TWO A4 pages.

- Provide example code and necessary elaborations for demonstrating the advantages of Dynamic Scoping in using Perl to implement Maze Race as compared to the corresponding codes in Python.
- 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. So please start your work early!

1. Your submissions will be accepted the latest by 11:59 p.m. on Apr 7, but submissions made after the original deadline would be considered as LATE submissions and penalties will be imposed in the following manner:

- Late submissions before 11:59 p.m. on Apr 8: marks will be deducted by 20%.
- Late submissions before 11:59 p.m. on Apr 9: marks will be deducted by 50%.

2. 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*.

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

```
/*
 * 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
```

4. Late submission policy: less 20% for 1 day late and less 50% for 2 days late. We shall not accept submissions more than 2 days after the deadline.

5. The report should be submitted to VeriGuide, which will generate a submission receipt. The report should be named “report.pdf”. The VeriGuide receipt of the report should be named “receipt.pdf”. The report and receipt should be submitted together with codes in the same ZIP archive.

6. `Tar` your source files to `username.tar`
   Just `tar` all your source files, report.pdf, and receipt.pdf for Assignment 3.

```
tar tmchan.tar cvf maze_race.pl maze_race_components.pm maze_race.py report.pdf
receipt.pdf
```

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

```
gzip tmchan.tar
```

8. `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
```

9. Please submit your assignment using your Unix accounts.

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

11. You can check your submission status at

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

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

13. Enjoy your work :>