Fairy Chess
RW144 - Project Specification
Revision 3

October 3, 2019

1 Introduction

1.1 Overview

Fairy Chess is the name given to variations on standard chess[1] where non-standard boards and pieces are used. For this project, you will implement a system that will allow for such a game of chess to be played, first by textual descriptions of a board configuration and a set of moves, and then by a graphical interface.

1.2 The Board

Our variation of chess is played on a **10×10 board**. Rows are called *ranks* and are numbered from 1 to 10. Columns are called *files* and are "numbered" from a to j. Indexing into the board is done with a combination of a file and a rank, so `d6` refers to the square at the intersection of the 4th column and the 6th row. **This indexing notation will be referred to as chess notation throughout this specification.** Figure 1 shows a chess board with the ranks and files labeled, and a White Pawn at `d6`.

1.2.1 Player Sides

As with standard chess, Black starts on the top two ranks (10th and 9th on the 10×10 board), whereas White starts on the bottom two ranks (1st and 2nd). Direction of play also remains the same, with Black Pawns only being allowed to move and capture in the direction of decreasing rank (*downwards*), and White Pawns in the direction of increasing rank (*upwards*).

For the purposes of the Elephant piece, Black's half of the board includes ranks 6 through 10, and White's half includes ranks 1 through 5.

[1] If you would like a quick chess refresher, `https://en.wikipedia.org/wiki/Chess` is a good place to start.

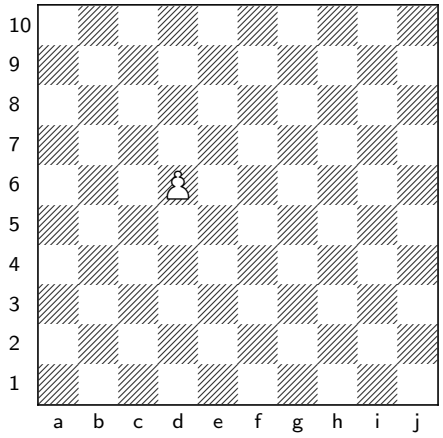

Figure 1: The 10x10 chessboard used in our variation of chess

1.3 The Pieces

Standard chess has six pieces (Pawn, Bishop, Knight, Rook, Queen, and King), and our variation introduces **five new pieces** (Drunken Soldier, Flying Dragon, Elephant, Amazon, and Princess).

1.4 The Moves

In general, pieces can move any number of steps along their trajectory, as long as the squares are not occupied by another piece. A move must end at the latest on the square *before* any own piece, or *on* the square occupied by an opposite piece; in the latter case, the opposite piece is *captured* and removed from the board. The main exception is the Knight, which can jump over occupied squares.

1.4.1 The Pawn

Unlike the other pieces, Pawns cannot move backwards. Normally a Pawn moves by advancing a single square, but the first time a Pawn moves, it has the option of advancing two squares. Pawns may not use the initial two-square advance to jump over an occupied square, or to capture. Any piece immediately in front of a Pawn, friend or foe, blocks its advance. Upon reaching the opposing color's base rank, a Pawn can be promoted to any Officer, given that at least one Officer of that type was allocated in during piece allocation (see 2.4.1).

1.4.2 The Bishop

The Bishop has no restrictions in distance for each move, but is limited to diagonal movement. Bishops, like all other pieces except the Knight, cannot

jump over other pieces. A Bishop captures by occupying the square on which an enemy piece sits.

1.4.3 The Rook

The Rook moves horizontally or vertically, through any number of unoccupied squares. As with captures by other pieces, the Rook captures by occupying the square on which the enemy piece sits. The rook also participates, with the King, in a special move called castling (see 1.4.7).

1.4.4 The Knight

The Knight moves to a square that is two squares away horizontally and one square vertically, or two squares vertically and one square horizontally. The complete move therefore looks like the letter "L". Unlike all other standard chess pieces, the Knight can "jump over" all other pieces (of either color) to its destination square. It captures an enemy piece by replacing it on its square.

1.4.5 The Queen

The Queen can be moved any number of unoccupied squares in a straight line vertically, horizontally, or diagonally, thus combining the moves of the Rook and Bishop. The Queen captures by occupying the square on which an enemy piece sits.

1.4.6 The King

A King can move one square in any direction (horizontally, vertically, or diagonally) unless the square is already occupied by a friendly piece or the move would place the King in check.

1.4.7 Castling

Castling is a move that can be performed by a King and one of the two Rooks, given that neither the chosen Rook nor the King have moved since the start of the game. Castling is either *queenside*, where the King moves a distance of three squares to the left, or *kingside*, where the King moves a distance of three squares to the right. The move is completed by the Rook then jumping over the King, landing adjacent to the King. Castling may only be done if the King has never moved, the Rook involved has never moved, the squares between the King and the Rook involved are unoccupied, the King is not in check, and the King does not cross over or end on a square in check. Figure 2 illustrates the castling opportunities.

1.4.8 The Drunken Soldier

A Drunken Soldier can move one square forward (like a Pawn) or a one square to the side. Capturing can be done one square diagonally (like a Pawn). Upon

reaching the opposing color's base rank, a Drunken Soldier can be promoted to any Officer, given that at least one Officer of that type was allocated in during piece allocation (see 2.4.1).

1.4.9 The Elephant

An Elephant moves and captures like a Rook, but is restricted to its half of the board.

1.4.10 The Flying Dragon

A Flying Dragon moves and captures like a Bishop, but is restricted to a distance of two squares per move.

1.4.11 The Princess

The Princess moves and captures like a Bishop or a Knight.

1.4.12 The Amazon

The Amazon moves and captures like a Queen or a Knight.

2 The Board File Format

2.1 Overview

The first step in creating a system that can facilitate the playing of any variation of chess is **being able to read some description of a chessboard, along with some metadata, and then validating that the description is correct.** In our variation of chess, we differ from standard chess in that we do not have a fixed configuration of pieces. Instead, **the board file specifies how many pieces of each type each player begins with**. The only fixed configuration is the initial position of the Rooks, and the King (see 2.2). Pawns are also subject to some constraints (see 2.3).

2.2 Initial Position of King and Rooks

In order to allow *castling* (see 1.4.7) in our variation, we need to fix the initial positions of the Rooks and the Kings. Rooks start in their respective colors' corners, so White's Rooks start on `a1` and `j1` and Black's on `a10` and `j10`. White's King starts on `f1` and Black's King starts on `f10`.

2.3 Initial Position of Pawns

Black Pawns start on the 9th rank, whereas White Pawns start on the 2nd rank. **Note that the Drunken Soldier is also classified as a Pawn**, and therefore also starts on the same rank as other Pawns of its color.

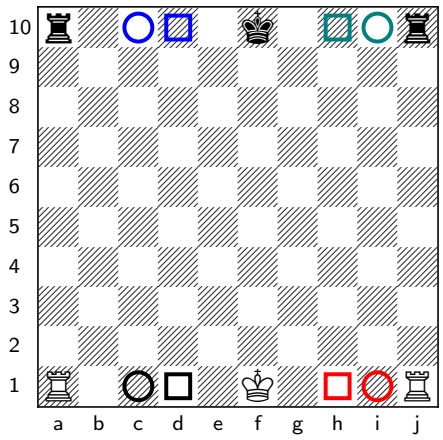

Figure 2: Initial positions of the King and the Rooks. Castling positions for each pair indicated by a different color, with the King's destination indicated by a circle and the Rook's by a square.

2.4 The File Format

The **board file is divided into three sections**: piece allocations, board configuration, and the status line. Sections are separated by fives dashes (`-----`). Lines that start with the % character (modulo, percent) are called *comment lines*, and they must be ignored. Figure 3 shows an example of a valid board file.

2.4.1 Piece Allocations

The first section describes how many pieces of each type every player starts the game with. Each row is a pair of of the format `<piece>:<count>`, where `piece` is a character representing the type of piece (see Table 4 for the mapping from pieces to characters), and `count` is the number of pieces of that type that each player start the game with. Each piece allocation must include exactly one King and at least two Rooks (see 2.2). Omitted allocations default to zero.Your program must take one argument (a path to a board file), and validate that board.

So, in Figure 3, each player starts with four Knights, two Bishops, eight Pawns, two Drunken Soldiers, one Queen, two Rooks, and one King.

2.4.2 Board Configuration

The second section describes piece positions. Each row is a sequence of 10 characters, separated by spaces. Lowercase characters indicate Black's pieces, while uppercase characters are White's pieces. The period character indicates

```
n:4
b:2
q:1
p:8
d:2
k:1
r:2
-----
r n n b q k b n n r
p p d p p p p d p p
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
P P D P P P P D P P
R N N B Q K B N N R
-----
w:++++:0:0
```

Figure 3: An example of a valid board file.

an empty square on the board.

2.4.3 Status Line

The third section describes some metadata surrounding the state of the game represented by the board file, with each piece of metadata separated by a colon. **The first character indicates the player who must move next**, with `w` representing White and `b` representing Black. The **second set of characters represent the available castling opportunities**, in the order Black Queenside, Black Kingside, White Queenside, White Kingside, with a `+` indicating that the maneuver is still available, while a `-` indicates that it is no longer available. **The third set of characters represent the Halfmove Clock.** This is an integer between 0 and 50 that is incremented after each move that is not a capture or a pawn move. Upon reaching 50, the game is considered drawn. **The last set of characters represent the move counter.** This is an integer that begins at zero, and is incremented every time black Black makes a move.

Piece	Character
k / K	King
r / R	Rook
q / Q	Queen
n / N	Knight
b / B	Bishop
p / P	Pawn
d / D	Drunken Pawn
f / F	Flying Dragon
e / E	Elephant
a / A	Amazon
w / W	Princess

Figure 4: Piece to character mappings. Uppercase characters for White pieces, lowercase for Black pieces.

3 Validating the Board File

3.1 Overview

When a board file is read, it should be validated to ensure that it represents a legal chessboard and game state. Each section of the board file is subject to its own set of validation criteria, and each section provides context for the validation of subsequent sections. A helper class, `BoardValidationErrors`, is provided. **Upon detecting that a board file is invalid, your system must report the validation error by calling the appropriate method from the helper class.**

3.2 Validation of Piece Allocations

Each player always starts with one King, and two Rooks. This leaves seven open squares on the base rank for each player, and ten open squares for the Pawn rank. Piece allocations use the lowercase representation of the piece, as given in Table 4.

3.2.1 Pawn Allocations

The sum of the number of Pawn and Drunken Soldiers allocated must equal ten. Examples of legal Pawn allocations include:

- `p:10`
- `p:8` and `d:2`
- `p:5` and `d:5`
- `d:10`

Examples of illegal Pawn allocations include:

- `p:11`
- `p:8` and `d:4`
- `d:5`
- No Pawn allocations

Upon detecting that the number of Pawn pieces allocated constitute an invalid board file, the method `illegalPieceAllocation` must be called with the current line number of the board file. In the case that no Pawn are allocated, the line number to be used must be the line number of the first section divider.

3.2.2 Officer Allocations

An Officer is any piece that is not a Pawn or a Drunken Soldier. **The sum of the number of Officer allocations must equal ten, including the two Rooks and the King.** Examples of legal Officer allocations include:

- `n:4`, `b:2`, `q:1`, `k:1`, and `r:2`
- `b:2`, `n:2`, `q:2`, `f:1`, `k:1`, and `r:2`
- `e:6`, `r:3`, and `k:1`

Examples of illegal Officer allocations include:

- `n:4` and `b:4`
- `b:2`, `n:2`, `q:2`, and `f:2`
- `e:10`
- No Officer allocations

Upon detecting that the number of Officer pieces allocated constitute an invalid board file, the method `illegalPieceAllocation` must be called with the current line number of the board file. In the case that no Officers are allocated, or there is not one King and at least two Rooks allocated, the line number to be used must be the line number of the first section divider.

3.3 Validation of Board Configuration

The board configuration describes the state of the chessboard at some point in time. There are ten ranks, where each rank should have ten characters, each separated by a space character. Each non-space character represents a piece or an empty square, as per Table 4. **Your system must use the specified methods to report the first violation it encounters, stating from position `a10`, going left to right (a $\rightarrow$ j), then top to bottom (10 $\rightarrow$ 1).**

3.3.1 Illegal Pieces

When a character is encountered that does not appear in Table 4, and / or has not been allocated in the piece allocation section, the method `illegalPiece` must be called with the position of the illegal piece in chess notation.

3.3.2 Exceeding Pawn Allocation

When a Pawn or Drunken Soldier is detected that exceeds the allocation as per the Piece allocation section, the method `pawnAllocationExceeded` must be called with the location of the offending Piece in chess notation.

3.3.3 Exceeding Officer Allocation

The maximum number of Officers that may be present on the chessboard for each color is given by:

$$n = i + 10 - (p + d) \tag{1}$$

where n is the maximum of the number Officers, i is the sum of the Officers allocated during piece allocation, p is the number of Pawns of that color present on the board, and d is the number of Drunken Soldiers of that color present on the board. This is to account for Pawns or Drunken Soldiers that have been promoted to Officers during the course of the game. Note that Pawns may not be promoted into Officers that were not allocated during piece allocation.

When a Piece is detected that causes an Officer to exceed it's legal count as per (1), the method `officerAllocationExceeded` must be called with the location of the offending Piece in chess notation.

3.3.4 Illegal Board Dimension

If the board configuration does not describe a 10×10 board, the method `illegalBoardDimension` must be called. Examples of such boards would include a board where one of the ranks only has eight Pieces, or a board that only has five ranks.

3.3.5 Illegal Pawn Position

If a Pawn or a Drunken Soldier is located behind the Pawn rank, the method `illegalPawnPosition` must be called with the position of the Piece in chess notation. An example of such an illegal position would be a Black Pawn on `a10`, or a White Drunken Soldier on `i1`.

3.3.6 Illegal Elephant Position

If an Elephant is located on the opposing player's side of the board, the method `illegalElephantPosition` must be called with the position of the Elephant in chess notation. An example of such an illegal position would be a Black Elephant on `a3`, or a White Elephant on `j10`.

3.4 Validation of Status Line

3.4.1 Illegal Next Player

If the next player marker is not `w` or `b`, the method `illegalNextPlayerMarker` must be called with the current line of the board configuration file.

3.4.2 Illegal Castling Opportunities

If a castling opportunity is listed as available, but the King or the Rook are not in their respective correct positions (as per 1.4.7) for the given opportunity, the method `illegalCastlingOpportunity` must be called with the current line of the board configuration file, and the index of the violating opportunity (0 for the first, 3 for the last).

3.4.3 Illegal Halfmove Clock

If the Halfmove clock is not a non-negative integer between 0 and 50, the method `illegalHalfmoveClock` must be called with the current line of the board configuration file. Examples of illegal Halfmove clocks are 99, -25, and `foobar`.

3.4.4 Illegal Move Counter

If the move counter is not a non-negative integer, the method `illegalMoveCounter` must be called with the current line of the board configuration file. Examples of illegal move counters are -25 and `foobar`.

4 The Move File Format

4.1 Overview

Once a board file has been read and validated, a game of chess can now be played. Moves are stored in a separate file, with a notation not too far removed from *long algebraic notation*. The first move from the file is to be applied from the perspective of the next player according to the board file (see 2.4.3).

4.2 Move Notation

4.2.1 Movement

A normal move, where a piece moves from one square to another without a capture occurring as a result. This is represented as `src−dst`, where `src` is the index of the source square, and `dst` is the index of the destination square. Examples of moves include: `a1-a2`, `c1-a2`, and `d3-g6`.

4.2.2 Capture

A capture move, where a piece moves from one square to another with a capture occurring as a result. This is represented as `src`×`dst`, where `src` is the index of the source square, and `dst` is the index of the destination square. Examples of captures include: `a1xa2`, `c1xa2`, and `d3xg6`.

4.2.3 Promotion

A promotion move, where a Pawn or Drunken Soldier reaches the opposing color's base rank through either a movement or a capture, and then promotes to another officer. This is represented as `cap_or_move=type`, where `cap_or_move` is the movement or capture that gets the Pawn on the base rank, and `type` is the type of officer that results from the promotion, using the representation from Figure 4, respecting the case of the character depending on the current player. Note, promotion to an Elephant is not allowed. Examples of promotions include: `b9-b10=Q`, `a2xb1=w`, and `d9-d10=R`.

4.2.4 Castling

A castling move, where the King and one of two Rooks perform the movement described in 1.4.7. This is represented as either `O-O-O` or `O-O`, for queenside and kingside castling respectively.

4.2.5 Check

If a move results in a situation where the King is in check, that move must be have `+` as a suffix. Examples of moves with a check prefix include: `a1-a2+`, `c1xa2+`, and `d3xg6+`.

5 Validating the Move File

5.1 Overview

When a move file is read, it should be validated to ensure that it represents a legal game of chess, given the state of the game as described by the board file. Every line of the move file is either a move in the notation described in Section 4.2, or a comment line as described in Section 2.4. A helper class, `MoveValidationErrors`, is provided. **Upon detecting that a move file is invalid, your system must report the validation error by calling the appropriate method from the helper class.**

5.2 Illegal Move

A move is illegal if:

1. The source square is not occupied by a piece of the current player's color.

```
% Input board file
n:4
b:2
q:1
p:8
d:2
k:1
r:2
-----
r n n b q k b n n r
p p d p p p p d p p
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
P P D P P P P D P P
R N N B Q K B N N R
-----
w:++++:0:0
```

```
% Input moves file
e2-e4
g9-g8
d1-i7+
h9-h8
i1-h3
h8xi7
j1-i1
```

```
% Output board
n:4
b:2
q:1
p:8
d:2
k:1
r:2
-----
r n n b q k b n n r
p p d p p p . . p p
. . . . . p . . .
. . . . . . . d .
. . . . . .
. . . . . .
. . . . P . . . .
. . . . . N . .
P P D P . P P D P P
R N N B Q K B N R .
-----
b:+++-:1:3
```

Figure 5: An example of a valid board file, valid move file, and expected output.

2. The destination square is occupied (in this case, the capture notation should be used).
3. The piece at the source square is not able to move to the destination square according to its movement rules.
4. The current player's King is in check, and the result of the move does not remove the King from check.
5. The Halfmove clock is equal to 50, and the piece being moved is not a Pawn or Drunken Soldier.
6. The move puts the current player's King in check.
7. The move puts the opposing player's King in check, but the appropriate suffix is not present.

When an illegal move is encountered, the method `illegalMove` must be called with the current line of the move file.

5.3 Illegal Capture

A capture is illegal if:

1. The source square is not occupied by a piece of the current player's color.
2. The destination square is not occupied by a piece of the opposing player's color.
3. The piece at the source square is not able to capture the destination square according to its movement rules.
4. The current player's King is in check, and the result of the capture does not remove the King from check.
5. The move puts the current player's King in check.
6. The move puts the opposing player's King in check, but the appropriate suffix is not present.

When an illegal capture is encountered, the method `illegalCapture` must be called with the current line of the move file.

5.4 Illegal Castling

A castling move is illegal if:

1. It does not adhere to the rules for castling in Section 1.4.7.
2. The corresponding castling opportunity is listed as unavailable in the status line (as per Section 2.4.3).
3. The Halfmove clock is equal to 50.
4. The move puts the current player's King in check.
5. The move puts the opposing player's King in check, but the appropriate suffix is not present.

When an illegal castling move is encountered, the method `illegalCastlingMove` must be called with the current line of the move file.

5.5 Illegal Promotion

A promotion move is illegal if:

1. The move or capture does not result in a Pawn or Drunken Soldier of the player's color on the opposing player's base rank.
2. The piece being promoted to does not occur at least once during piece allocation.

3. The piece being promoted to is an Elephant.
4. The move puts the current player's King in check.
5. The move puts the opposing player's King in check, but the appropriate suffix is not present.

When an illegal promotion is encountered, the method `illegalPromotion` must be called with the current line of the move file.

5.6 Illegal Check

If a check suffix (`+`) is present on a move, but the move did not result in the King being left in check, that move is illegal. When an illegal check suffix is encountered, the method `illegalCheck` must be called with the current line of the move file.

6 Visualisation of Fairy Chess Games

The final phase of the project is to provide a suitable visualisation of your Fairy Chess system, and to extend your implementation to support input from a user via a text and visual interface, in addition to move files.

6.1 Visualise Board

Read a board file, produce a graphical representation of the board using Princeton's `StdDraw`, or your graphics library of choice. Your visualisation should include a representation of the status line and piece allocations. **This functionality accounts for 10% of the 25% for Submission 3.**

6.2 Handling Illegal Moves

When an illegal move is encountered (either in a move file, or a move that was provided by the user), the user must be warned through some graphical notification (a dialog box or text message appearing). The system should not exit upon encountering an illegal move, and the state of the board and the graphical representation must not be otherwise modified. Illegal boards must cause your system to print out the validation error, and then exit, as with Submission 1.

6.3 Accept Text Input

Read a board file, visualise it (implemented in Section 6.1), and allow the user to input moves via a text input widget such as a text box or dialog. Legal moves must be applied to the board and the graphical representation must be updated accordingly. **This functionality accounts for 5% of the 25% for Submission 3.**

6.4 Accept Visual Input

Modify your implementation from 6.3 to allow the user to input moves by clicking on the desired source and destination squares. Legal moves must be applied to the board and the graphical representation must be updated accordingly. **This functionality accounts for 5% of the 25% for Submission 3.**

6.5 Step Through a Move File

Give the user the option to select a move file after reading a board file. Supply the user with a button that can be used to apply a move from the move file each time the button is pressed. **This functionality accounts for 5% of the 25% for Submission 3.**

7 Bonus Functionality

If you have the time and energy on your hands to go above and beyond, feel fry to attempt implementing some of the additional functionality listed below. Each piece of bonus functionality is an additional 5%, with the total bonus marks that we can allocate capped at 20%.

Undo/Redo Allow users to undo moves made during games played using the inputs methods described in Sections 6.3 and 6.4.

Checkmate Support Supporting the checkmate assertion in move files.

Save Board and Moves Save the starting board and a list of moves played during a game in your visualisation.

Board Creator Construct starting boards, including the custom piece allocations, through a graphical interface, and save the resulting board to file.

Computer Opponent The ability to play a game of Fairy chess against a computer opponent.

Fairy Chess Online The ability to play a game of Fairy Chess against another player over the network.

Better Graphics Increase the graphical fidelity of your Fairy Chess visualisation significantly.

8 Submission and Marking

8.1 Format of Submission

You have been assigned a Git repository hosted on the University's GitLab system. All the code you write for this assignment must be committed and

```
1 #!/bin/bash
2
3 # Compile all the files
4 javac ./*.java
5
6 # Run the test boards
7 for testBoard in ./testBoards/*.board; do
8     echo "------------------"
9     echo "Running $testBoard"
10
11     # Capture output
12     OUTPUT="$(java FairyChess "$testBoard")"
13
14     # Compare output with expected output
15     EXPECTED="$(cat "$testBoard.out")"
16
17     if [ "$OUTPUT" == "$EXPECTED" ]; then
18         echo "Correct! :)"
19     else
20         echo "Incorrect! :("
21     fi
22
23     echo
24 done
```

Figure 6: An example of a stripped down marking script for Submission 1.

pushed into that repository's `master` branch. We expect a class file called `FairyChess.java` to be in the root of your repository[2]. This class must contain your main method, which will be used to run your program against our board file test suite, as per the script in Figure 6.

8.1.1 Submission 1

Your program must take one argument (a path to a board file), and validate that board. The script in Figure 6 illustrates how your program will be called, with line 4 showing the compilation, and line 12 showing the invocation of your program.

8.1.2 Submission 2

Your program must take two arguments (a path to a board file, and a path to a move file), and validate the board file. The move file will then be read line by line, with each move being validated and applied to the chessboard if it is determined to be valid. After the last move has been applied, your system must print out the state of the board to `stdout`, **producing a board file in the same format as described in Section 2.4, including**

[2]You are allowed to use multiple Java files, but the main method must be in the `FairyChess.java` file.

piece allocations, an updated board layout, and an updated status line.

8.2 Mark Allocation

There will be three project submissions in total.

1. The first submission, which covers the Board Validation, has a **submission date of the 5th of September, 2019, 17:00, and counts 25% of the project mark**.
2. The second submission, which covers the Move Validation, has a **submission date of the 3rd of October, 2019, 17:00, and counts 50% of the project mark**.
3. The third and final submission, which covers Visualiation, has a **submission date of the 24th of October, 2019, 14:00, and counts 25% of the project mark**.

8.3 How We Mark

8.3.1 Submission 1

We mark by means of an automated marking script. The script will first clone your repository, and compile all of the Java files. **Submissions that do not compile will receive zero marks.** Your program must take one argument (a path to a board file), and validate that board. Then, for each of our test board configuration files, the script will execute your program with the board file as input, and compare the output your program produces with our expected output. Figure 6 should give you a fairly representative idea of how the marking script will work. We will set up a Continuous Integration system that runs similar tests every time you push a commit into the GitLab repository.

8.3.2 Submission 2

As with Submission 1, marking is done by means of an automated marking script. **Submissions that do not compile will receive zero marks.**

8.3.3 Submission 3

Submission 3 will be marked by means of a live demonstration. Markers will visit students individually, and there will be a fixed rubric of functionality that you will be expected to demonstrate. After the fixed rubric has been filled in, students will have a chance to demonstrate additional functionality.