

## Beans Gambit Part 2

2227951 | CS141 | 22/3/2023

# Contents

Beans Gambit Part 2 .....	0
Analysis.....	2
Design .....	2
Game State .....	2
Storing All Game States.....	2
Parsing .....	3
Parsing Turn Number.....	3
Parsing BGN Move.....	3
Parsing End Game Status .....	3
Parsing Turn .....	4
Identifying the First Player to Move .....	5
Implementation .....	5
Running the Code.....	5
File Structure.....	5
Code Specifics .....	5
Output .....	6
Errors .....	7
Extension – Continuing Declared Unfinished Games .....	11
Implementation .....	11
Showcase .....	12
References .....	13

## Analysis

Designing the parser for my program involved the decomposition of the file into different parts. Below an example of the contents of a BGN file for a valid game.

```
1. d2xd3 b3xb2
2. c2xc3 a3xa2
3. c3b3 a2a1
4. #
```

The file contains lines that each follow the same format and are composed of 3 main parts:

- Line number (which I refer to as turn number).
- BGN move (including the take declaration for a move).
- End game status ('D' or '#').

I have considered each of these parts separately within my design.

## Design

### Game State

The game state data structure stores fields that relate to the game's current state. The detail of each field is detailed below.

Stored data	Details
Current turn number (expected).	<ul style="list-style-type: none"><li>• Always represents the correct turn value (starts as 1 and is incremented at the end of each turn).</li><li>• Ensures that the parsed turn number is not just any number in the correct "shape".</li></ul>
The current board.	<ul style="list-style-type: none"><li>• Needed to perform value validating operations on.</li><li>• Needed to be able to output the board.</li></ul>
Current player (expected) to move for the specified current board.	<ul style="list-style-type: none"><li>• Needed to ensure that the correct player moves a piece in a move.</li><li>• Used to help declare end game state.</li></ul>
(Whether current state needs user input)	<ul style="list-style-type: none"><li>• Is used for the mentioned extension later in this document.</li></ul>

Originally, I had an extra field named `hasEnded` which represented whether the game had ended. However, I instead opted to store the current player as a `Maybe` type and to represent the game end state when the current player is `Nothing` as it reduced data storage.

### Storing All Game States

The game state changes with each move that is performed as the board changes, the player changes and possibly the turn number changes (if a new turn is being parsed).

I have made a list of all the game states so previous game states can be recalled and processed (for example to show all the boards to the user).

## State Monad

I could have used the state monad to keep track of the game state and given more time, I would have implemented functions of the shape `State GameState Parser ()` and instead have the individual GameState store the list of boards.

## Parsing

My parser has been designed in such a way that the validity of the parsed contents is checked as they are parsed and can be thought of as “playing the game” as it acquires new data. This is based on the “fail fast” design principle and means that less processing is done for an invalid file as the parser will throw an error as soon as invalid data is detected.

When writing my parsers, I made extensive use of the megaparsec documentation (Haskell, 2023).

## Parsing Turn Number

The turn number that I parse in should be able to accommodate for numbers with any number of digits and should be validated by checking if it is equal to the expected current turn number within the latest game state record.

## Parsing BGN Move

The BGN move consists of 3 main parts:

- BGN start coordinate.
- BGN end coordinate.
- Take declaration.

As the makeMove function works with normal coordinates (e.g. (0,1)), after parsing the BGN coordinate, it must be converted to a normal coordinate. I considered using a map to do this conversion, but instead I managed to spot the mathematical relationship between a BGN coordinate and a normal coordinate. These are as follows:

$$\begin{array}{lcl} \text{(Ordinal of the BGN x coordinate)} - 97 & = & \text{normal x coordinate} \\ 4 - \text{BGN y coordinate} & = & \text{normal y coordinate} \end{array}$$

However, as these conversions allow any arbitrary BGN x or y coordinate, I must also check that the resulting coordinate is the expected valid range (between 0 and 3 inclusive).

Once the normal coordinates are obtained for each part of the BGN move, the move can be carried out using makeMove and can be validated by simply pattern matching for a resulting Just board.

However, makeMove will only validate if the take move was legal and does not consider whether the move was declared as a take. Due to this, I must check that the take symbol is present if the move was a take and vice versa. Calculating the change in balance between 2 boards can be found to imply whether a take has occurred (it would have changed by 1) and then verifying that this agrees with the take status will verify whether the take declaration is correct.

As the game end parser should have dealt with any draw/loss declarations, before any move is parsed, there must also be a check to see if the game is in a loss/draw state. This is because moving when either of those states have occurred is also considered invalid.

## Parsing End Game Status

The end game status may be declared in place of a move, so should be part of the choice of parsers for a turn's action (a move or a game end declaration).

An additional consideration is when the game has been declared has ended, all subsequent turn numbers/actions after the end symbol invalidate the BGN file as the only valid expected characters are optional trailing white space.

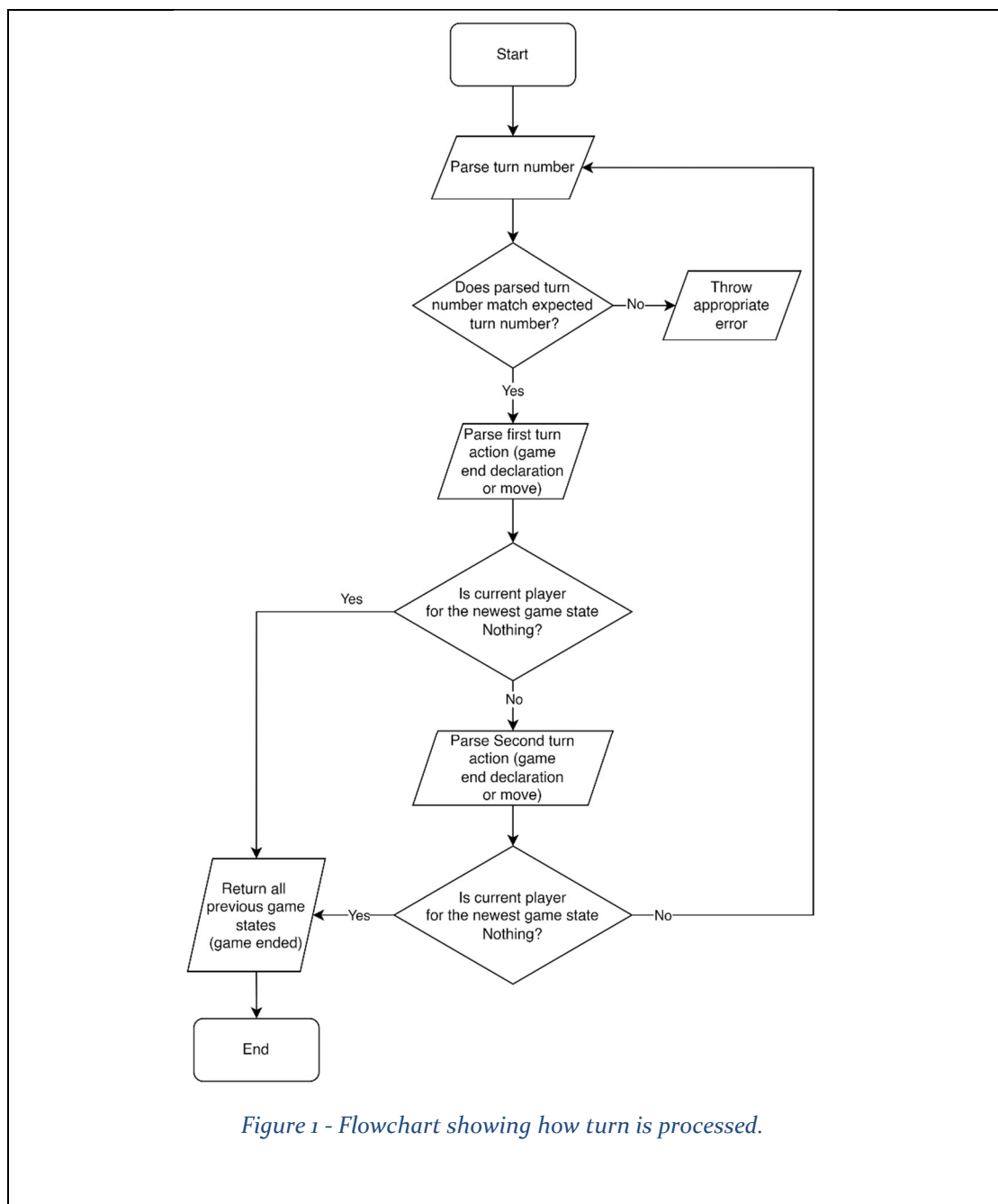
In my extension, I detail an extra game end state I added (C for continue).

## Parsing Turn

A turn action is defined as either a game end declaration or a move and each turn consists of 3 main components:

- Turn number.
- Turn action 1.
- Turn action 2.

The flowchart below highlights the sequential steps performed within parsing a turn.



*Figure 1 - Flowchart showing how turn is processed.*

I use my other parsers in the appropriate order and perform appropriate checks to determine when a game end state has occurred.

## Identifying the First Player to Move

The BGN file does not explicitly state the first player to move in the game. Due to this, the first player to move must be identified. I have done this by parsing the first BGN coordinate of the file and identifying which player moved the piece.

## Implementation

### *Running the Code*

To parse a bgn file, use `stack run -- "path to file"` (Call Main Function With Arguments, 2023)

### *File Structure*

I have added to all the files listed below.

File	Contents
src/Parsers.hs	The parsers.
src/Types.hs	<ul style="list-style-type: none"><li>• GameState data type:</li><li>• Show implementation for GameState.</li><li>• Show implementation for [GameState].</li><li>• Show implementation for Maybe Player</li></ul>
src/Game.hs	Contains the principal operations of the game and game related functions.
app/Main.hs	Contains IO functions including: <ul style="list-style-type: none"><li>• The main application.</li><li>• The continueGame IO function.</li></ul>

### *Code Specifics*

Detail	Explanation
when and unless functions	<ul style="list-style-type: none"><li>• Initially code had many nested if/case statements that decided when to throw an error.</li><li>• The use of when/unless allows for much clearer and more readable code. (Control Monad, 2023)</li></ul>
Sequential conditional statements.	<ul style="list-style-type: none"><li>• Initially in places where sequential conditional statements determine a non error output, there were many indents and the code was very messy.</li><li>• I decided to split the parts of the if statement using the where clause.</li></ul>
Non exhaustive pattern matches	<ul style="list-style-type: none"><li>• I have verified that the non-pattern matched states are impossible to reach.</li></ul>

White space considerations	<ul style="list-style-type: none"> <li>At the end of each line, there may be extra whitespace before the newline character. I deal with this by parsing an infinite optional number of spaces before the new line character.</li> <li>At the end of a valid BGN file, there may be extra lines or spaces. I deal with this by parsing an infinite optional number of newlines or spaces and then check for the end of the file by parsing eof.</li> </ul>
Avoiding the <b>try</b> function	<ul style="list-style-type: none"> <li>Using try reduces the performance of the parser as it must backtrack.</li> <li>To avoid using try, I have considered the order in which I apply parsers to ensure the correct thing is parsed.</li> <li>Within parseTurn, when parsing the actions, I apply the parseGameEnd parser before the parseMove parser as my parseGameEnd is defined for the explicitly stated characters 'D', '#' and 'C', which all do not conflict with parseMove.</li> </ul>
Recursive parsers	<ul style="list-style-type: none"> <li>Has one BGN file consisted of many turns which all conform to the same set of rules, I have decided to write the parseGame parser to just serve the purpose of calling parseTurn with the initial game state values.</li> </ul>
Use of cons and game state	<ul style="list-style-type: none"> <li>I have made extensive use of the cons operator as it allows me to control which game states which I am referring to.</li> <li>For example, within parseTurn, I use cons to construct the appropriate lists that need processing/returning. When an end game state or a needs user input game state has been reached, I ensure that the head of the list was replaced with the appropriate correct most recent board state as it differs to the original head game state (which assumed currentPlayer = opponent, etc.).</li> </ul>
Outputting errors	<ul style="list-style-type: none"> <li>I have used the &lt;?&gt; operator for parsing errors.</li> <li>I have used predicates for validity errors.</li> </ul>
Writing the parsers	<ul style="list-style-type: none"> <li>Considered the shape of what I was parsing.</li> <li>Considered the validity checks needed for parsed items.</li> <li>Constructed a parser based on the 2 principles above.</li> </ul>

## Testing

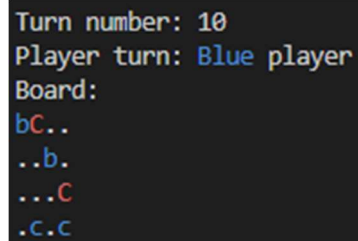
To test my parser, I have made use of the files provided by people on the course. At times I have modified valid files to contain invalid information to force specific errors to occur.

In addition to this, I have modified some valid files to act as tests for my extension. These are the \_DUGx files (declared unfinished game files).

## Output

Using the show implementations for GameState, [GameState] and Maybe Player, I output all the game states.

The output for a typical board state can be seen below.

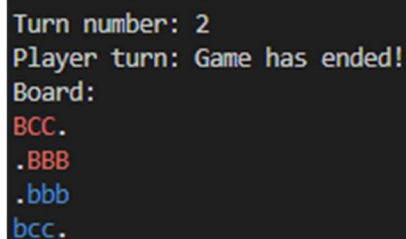
A screenshot of a terminal window showing the output of a game state. The text is as follows:

```
Turn number: 10
Player turn: Blue player
Board:
bC..
..b.
...C
.C.C
```

The text is color-coded: 'b' is blue, 'C' is red, and 'b' is blue.

*Figure 2 - Screenshot showing what a typical board state looks like*

And when the game ends.

A screenshot of a terminal window showing the output of a game state when the game has ended. The text is as follows:

```
Turn number: 2
Player turn: Game has ended!
Board:
BCC.
.BBB
.bbb
bcc.
```

The text is color-coded: 'B' is red, 'C' is red, 'B' is red, 'B' is red, 'B' is red, 'b' is blue, 'b' is blue, 'b' is blue, 'b' is blue, 'c' is blue, 'c' is blue, 'c' is blue.

*Figure 3 - Screenshot showing what an end board state looks like*

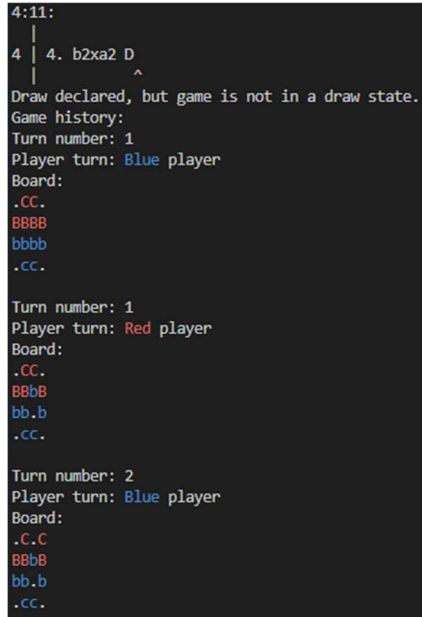
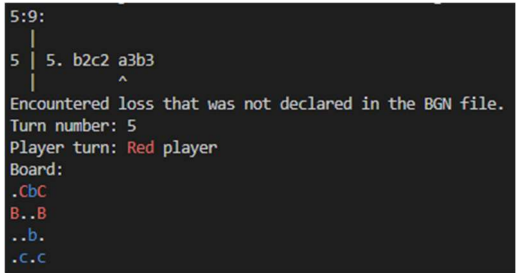
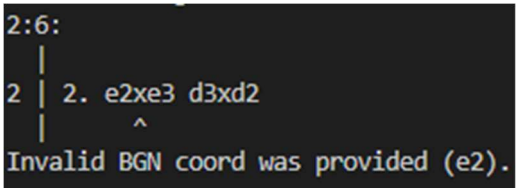
## Errors

All errors that are output because of an invalid file have an accompanying informative error message. In the error messages, I have made considerations to add information that allows the user to verify the problem with the file.


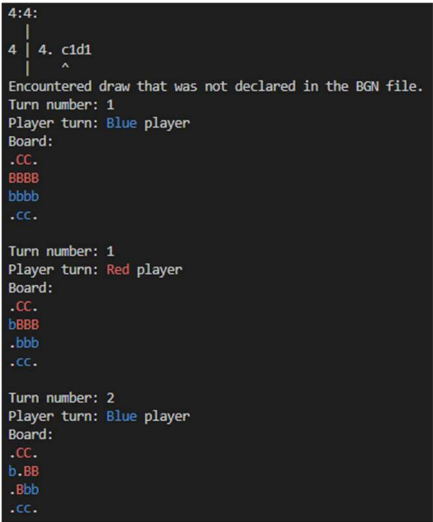
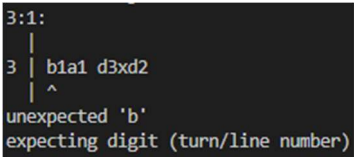
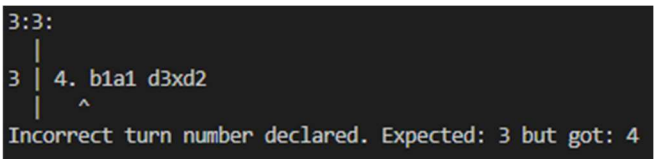
For example, I have output all the game states or the most recent game state where I have deemed it useful. For example, when the parser detects an error relating to a draw, all boards will be output as that it allows the user to verify the invalidity of that file.

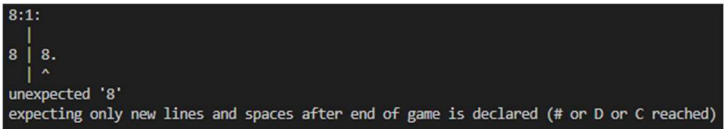
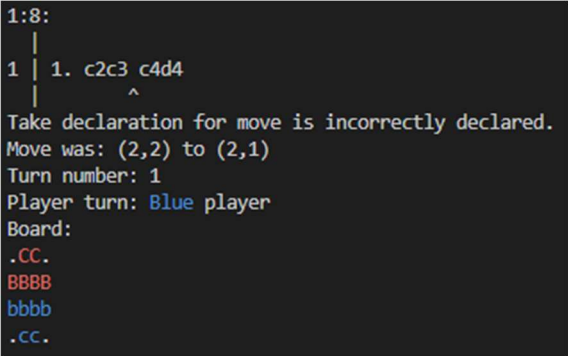
My parser is built in a way that will detect any error, however below I have listed some specific errors and their output messages.



Error	Error Message
Draw declared when the game is not in a draw state	 <pre> 4:11: 4   4. b2xa2 D     ^ Draw declared, but game is not in a draw state. Game history: Turn number: 1 Player turn: Blue player Board: .CC. BBBB bbbb .CC.  Turn number: 1 Player turn: Red player Board: .CC. BBbB bb.b .CC.  Turn number: 2 Player turn: Blue player Board: .C.C BBbB bb.b .CC. </pre> <p><i>Figure 4 - Screenshot showing the error when draw declared when not in a draw state. The actual output shows ALL boards so the user can verify why the draw state occurred.</i></p>
Loss declared when game is not in a loss state	 <pre> 5:9: 5   5. b2c2 a3b3     ^ Encountered loss that was not declared in the BGN file. Turn number: 5 Player turn: Red player Board: .CbC B..B ..b. .C.C </pre> <p><i>Figure 5 - Screenshot showing the error when a loss is declared when not in a loss state. The actual output shows the most recent board so the user can verify why the loss state occurred.</i></p>
Invalid BGN coordinate	 <pre> 2:6: 2   2. e2xe3 d3xd2     ^ Invalid BGN coord was provided (e2). </pre> <p><i>Figure 6 - Screenshot showing the error when an invalid BGN coordinate has been declared. The problematic coordinate is output to the user explicitly.</i></p>

Invalid piece moved	<p>Figure 7 - Screenshot showing the error when a player tries to move an empty square. The error contains the coordinate of the moved piece and the player that was expected to move it so the user can verify its invalidity.</p>
Invalid move	<p>Figure 8 - Screenshot showing the error when an invalid move has been declared. The move and the board are output so the user can verify its invalidity.</p>
Invalid characters sandwiched in between valid characters	<p>Figure 9 - Screenshot showing the error when an invalid character is in between valid characters.</p>
Gibberish	<p>Figure 10 - Screenshot showing error when random characters are encountered. The screenshot tells the user what the expected input was.</p>

Undeclared loss	 <p>5:9:         5   5. b2c2 a3b3      ^    Encountered loss that was not declared in the BGN file.    Turn number: 5    Player turn: Red player    Board:    .CbC    B..B    ..b.    .C.c</p> <p>Figure 11 - Screenshot showing the error when a loss state is declared in the BGN file when the game is not in a loss state. The latest board is output so the user can validate that the game is not in a loss state.</p>
Undeclared draw	 <p>4:4:         4   4. c1d1      ^    Encountered draw that was not declared in the BGN file.    Turn number: 1    Player turn: Blue player    Board:    .CC.    BBBB    bbbb    .cc.      Turn number: 1    Player turn: Red player    Board:    .CC.    bBBB    .bbb    .cc.      Turn number: 2    Player turn: Blue player    Board:    .CC.    b.BB    .Bbb    .cc.</p> <p>Figure 12 - Screenshot showing the error when a draw state is declared in the BGN file when the game is not in a draw state. All boards are output so the user can validate the game is not in a loss state.</p>
No turn number	 <p>3:1:         3   b1a1 d3xd2      ^    unexpected 'b'    expecting digit (turn/line number)</p> <p>Figure 13 - Screenshot showing the error when no turn number is encountered. The error tells the user that the turn number was expected.</p>
Non sequential turn numbers	 <p>3:3:         3   4. b1a1 d3xd2      ^    Incorrect turn number declared. Expected: 3 but got: 4</p> <p>Figure 14 - Screenshot showing the error when the turn numbers declared are not sequential. The expected turn number is shown to the user.</p>

Turn snumber/action after game declared as ending	 <p>Figure 15 - Screenshot showing the error when a turn number/action is declared after the game end declaration (C, D, #)</p>
Incorrect take declaration for a move	 <p>Figure 16 - Screenshot showing the error when the take declaration of a move is incorrect. The error contains the move and the board that caused the error.</p>

## Extension – Continuing Declared Unfinished Games

I have implemented functionality for the user to mark the file as unfinished with the letter C (just like how 'D' and '#' are used). The parser will still parse all the game information before this letter providing it is valid and will then allow the user to enter moves in the terminal until the game is finished.

The format for the moves is ((x1,y1),(x2,y2))

The parser that I have written to parse the user's moves will still consider the following aspects of the specified move:

- That it is valid.
- That the correct player is moving the piece.
- That when the game has ended, no more moves can be done.

It will also keep prompting the user until a valid move is entered using recursive calls to the IO function.

## Implementation

I updated the game state record to contain a Boolean value called needsUserInput. With this new field, I can identify when the game needs to be continued (needsUserInput = True) with the new finished game state having both needsUserInput as false and currentPlayer as Nothing.

The turn numbers of the game must be manually calculated as they are no longer parsed in. I have done this using the following conversion:

$$\text{turn number} = ((\text{number of game states}) \div 2) + 1$$

Although the logic for checking the validity of the normal move is similar to checking the validity of a BGN move, it does not check for take symbol and has to check if the game has ended after the player has made their move.

In addition to this, unlike what the flowchart earlier depicts, the `parseTurn` function must also return the list of game states when `needsUserInput = True` as this marks the end of the parsing period in a way that allows the subsequent logic to differentiate between a draw/loss state.

## ***Showcase***

Any file that has at least 1 valid move declared in it and the letter C can be continued. For example.

1. a2xa3 b3xa3
2. b2a2 a3b3
3. a2a1 b3a3
4. d2xd3 c3xd3
5. c1d1 a3a4
6. a1a2 d3c3
7. a2a3 c4d4
8. a3xa4 d4d3
9. c2xc3 d3d2
10. c3c4 C

Below is what finishing a game off looks like.

```

Turn number: 9
Player turn: Red player
Board:
bC..
..bC
....
.c.c

Turn number: 10
Player turn: Blue player
Board:
bC..
..b.
...C
.c.c

Turn number: 10
Player turn: Red player
Board:
bCb.
....
...C
.c.c

Enter move in form of ((x1,y2),(x2,y2)):
((1,0),(1,1))
Turn number: 11
Player turn: Blue player
Board:
b.b.
.c..
...C
.c.c

Enter move in form of ((x1,y2),(x2,y2)):
((1,3),(2,3))
Turn number: 11
Player turn: Game has ended!
Board:
b.b.
.c..
...C
..cc

```

*Figure 17 - Screenshot depicting the terminal for a saved game that has been declared as needing to be continued*

Given more time, I would implement saving functionality for the continued game.

## References

- Call Main Function With Arguments*. (2023, 3 16). Retrieved from Stack Overflow:  
<https://stackoverflow.com/questions/17374356/call-main-function-with-arguments>
- Control Monad*. (2023, 3 16). Retrieved from Hackage Haskell:  
<https://hackage.haskell.org/package/base-4.18.0.0/docs/Control-Monad.html#v:when>
- Haskell. (2023, 3 15). *Megaparsec*. Retrieved from Hackage Haskell:  
<https://hackage.haskell.org/package/megaparsec>