COMP3258 - Functional Programming

Pranav Talwar: 3035435462

December 2019

1 Building the Project

The project can be built by running the build script *./buildsudoku* in the terminal. This will compile the source code using the *ghc* command and generate an executable that can be used to run the program. The program can be run by typing *./Sudoku* in the terminal.

2 Functionality of the Project

2.1 Loading game

A Sudoku board can be loaded by selecting the first option when the game begins by running the *./Sudoku* command in the terminal. Following this, the game would ask for an input of the file which needs to be loaded into the game. If the file exists and the format of the file is correct, then the board is successfully loaded into the game.

Figure 1: Loading a file onto the Sudoku game

2.2 Making a move

A move can only be made if there is a valid board loaded into the game, otherwise the game would show an error.

Hence, assuming that a valid board is loaded into the game, a move can be made by selecting the fourth option. Following this, the program would ask for an input for the row, the column (0-indexed) and the number to be inputted in that space.

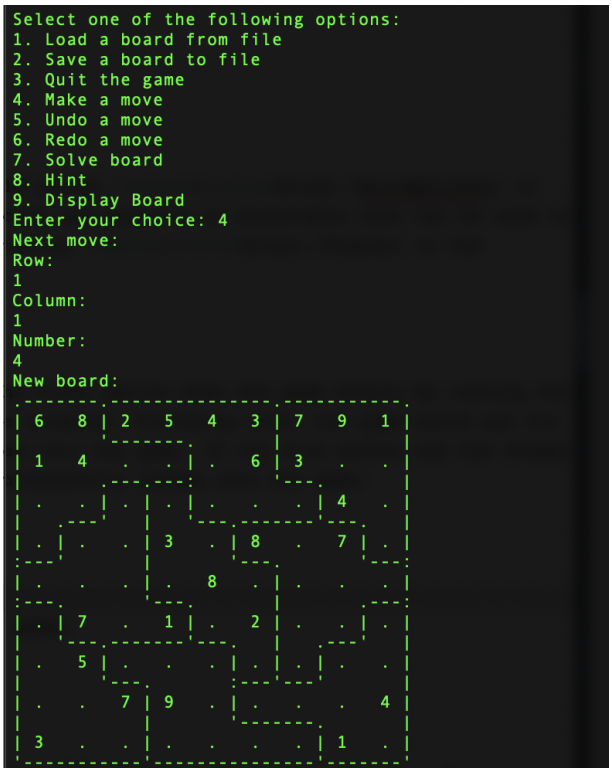

Figure 2: Making a move on a loaded Sudoku board

2.3 Saving a board

If you want to save a board to a file, you can select the second option to do that. Following this selection, the program would request for a file name to be inputted. If the file name is the same as the existing file that was loaded into the game, then it would be overwritten with the updated Sudoku board. If not, then a new file would be created with the provided name.This feature only works if a valid sudoku board is loaded into the game.

Figure 3: Saving a Sudoku board

2.4 Undoing and Redoing a move

A move can be undone by selecting the fifth option from the menu. The same move can be redone by selecting the sixth option. The undo option only works when a move has been made in the current game state, while the redo option only works if any moves have been undone. Further on, whenever a new move is made, the undone moves are lost and cannot be redone.

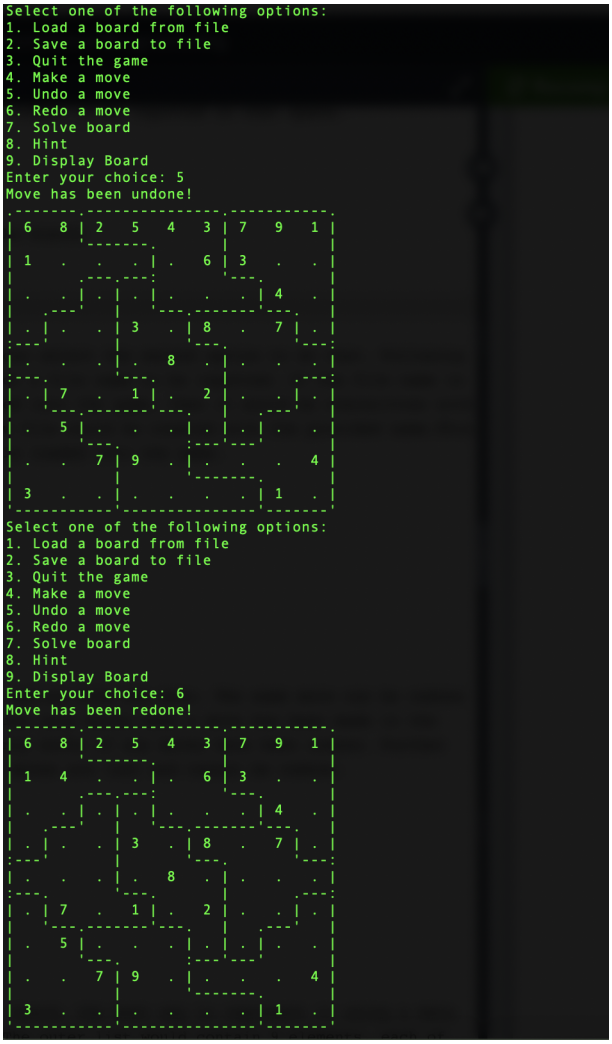

Figure 4: Undoing and Redoing a move

2.5 Quit

The game can be easily quit/exited by selecting the third option.

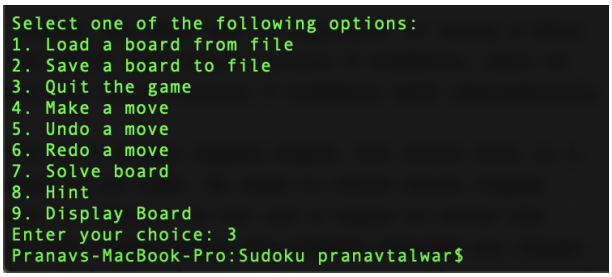

Figure 5: Quitting a game

2.6 Solver

In case we have difficulty in solving a board, we can select the seventh option and ask the game to provide us with a solution to the JigSaw Sudoku Board. The game will tell us in case there is no solution possible for the current state of the board.

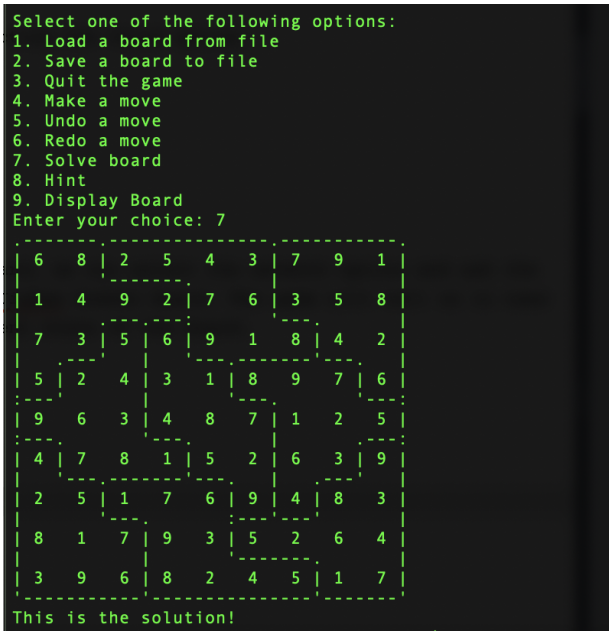

Figure 6: Using solver to solve a board

2.7 Hint

The game also provides us with a functionality where we can select the eighth option, which will provide us with a valid hint to fill up one of the positions on the board. In case there is no hint possible for a particular state of the board, the game will let us know by generating a message.

Figure 7: Using hint to help solve the board

2.8 Display

This is a feature that allows to see the current state of our JigSaw Sudoku board. It can be used by using the ninth option in the menu.

Figure 8: Displaying a board

3 Choice of Data Structure

Since the Sudoku board is a two dimensional board, the best way to represent it using a data structure would be to use a list of lists. The outer list would contain 9 elements, each of which would be a list representing a row. Each row would contain 9 elements each representing, one number in a row.

A simple integer as each element would suffice in a normal Sudoku board, but since this is a Jigsaw Sudoku board, a different strategy needs to be used. We need to store which Jigsaw piece each element in the Sudoku board belongs to. Hence, we can use a tuple to store the number in that place as well as the number of the Jigsaw piece the element belongs to. Since some elements might be empty as the user might not have made a move, we can use the *Maybe Int* in Haskell to encapsulate an optional value.

Finally our data structure would be of the type:

```
[[(Maybe Int, Int)]]
```

, where *Maybe Int* represents an optional number in the position. If the element is empty then it would be equal *Nothing* else it would be equal to a *Just Int*. The second represents the JigSaw piece the element belongs to.

4 Dealing with Errors and Ending

4.1 Menu

In case an invalid option is inputted while selecting an option from the menu, the program prints an error message onto the screen and prompts the user to enter another option. This is done by recursively calling the *game* function with the same arguments.

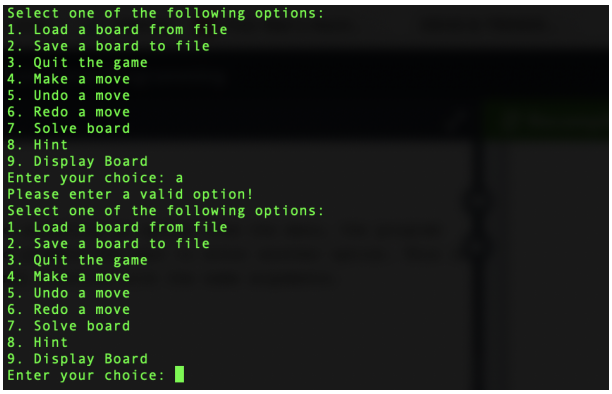

Figure 9: Handling Wrong Option

4.2 Loading a file

In case the file entered to be loaded does not exist (using the *doesFileExist* function from the *System.Directory* library), the program will respond with an error message and ask us to enter the name of the file again which is implemented by recursively calling the function itself. Further on, if the contents of the file are invalid, an error message would be shown on the console along with a prompt to enter the name of a file that has valid contents with respect to the game using recursion again. The validity of the contents of the file will be checked by a *checkFile* function that checks various aspects of the contents such as number of lines, length of each line and the contents of each line along with the board structure.

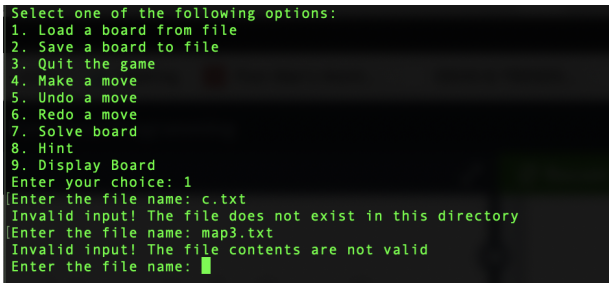

Figure 10: Handling non-existent file and wrong file contents

4.3 Making a move

When the player makes a move, the games asks for three details of the move which include the row, the column and the number to be inserted into the board. When the player inputs the row or the column, the game uses the *getRowCol* function to check if it's a digit or not. If it is not, then the game prints an error message onto the console and asks the player to input the row or the column number again by recursively calling the function again with the same arguments. If it is a digit, then the game checks whether it is between 0 and 8 (inclusive). If it is not, a similar procedure is followed.

For entering the digit, the *getDigit* function is invoked and a similar procedure is followed except the range of checking the player input is between 1 and 9 (inclusive).

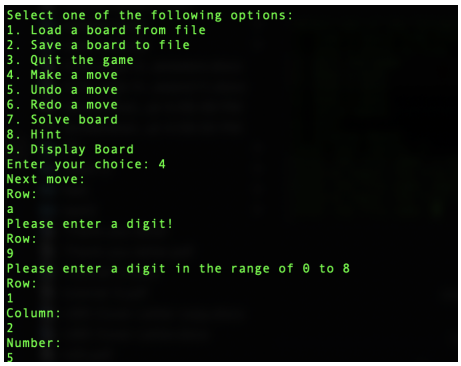

Figure 11: Handling row, column and number input

4.4 Checking Validity of Move

The validity of a move is checked when the player has inputted valid values for the row, column and the number. The *checkMove* function checks whether the element that is supposed to be filled up is empty and the number inputted does not occur in that row, column or jigsaw piece. In case there is a conflict, an error message in printed and the player is brought back to the main menu by recursively calling the *game* function again with the same arguments.

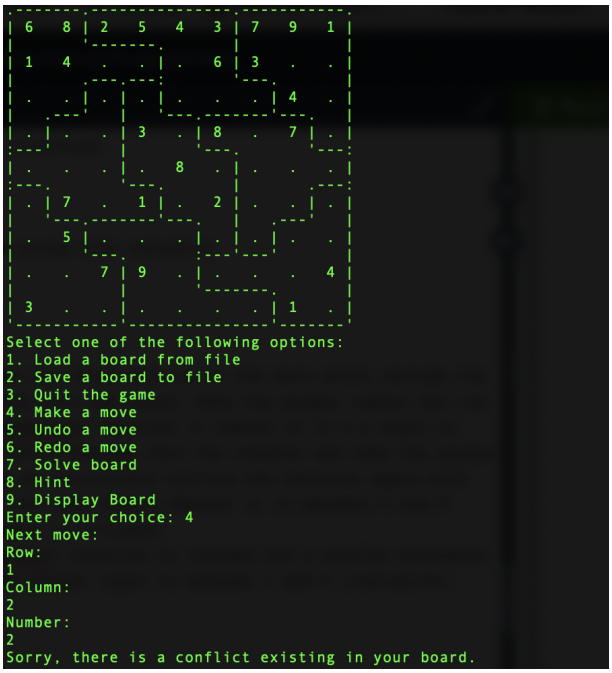

Figure 12: Handling row, column and number input

4.5 Ending the game

The game can be ended in three ways:

- **The player solves the board.**
  When the player makes a move, the *checkSudokuOver* function runs on the new board that is generated. If all the elements of that sudoku board are filled up, that means that the player has won. A win message is printed and the game ends.

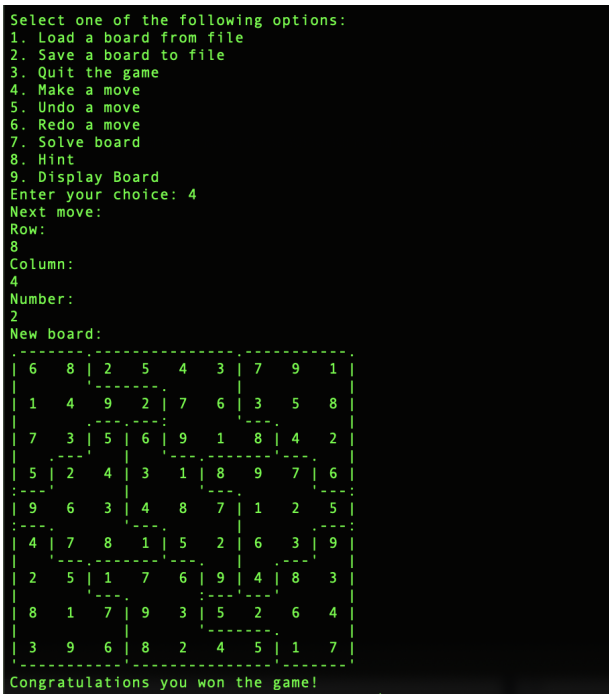

Figure 13: Ending game when player wins

- **The player quits the game.**
  If the player selects the third option from the menu (as explained in Section 2.5), the *game* function just returns an empty value and the game ends.

- **The player uses the solver to solve the board.** If the player uses the solver functionality to solve the board, the game function does not recursively call itself and prints the solution and exits the game.

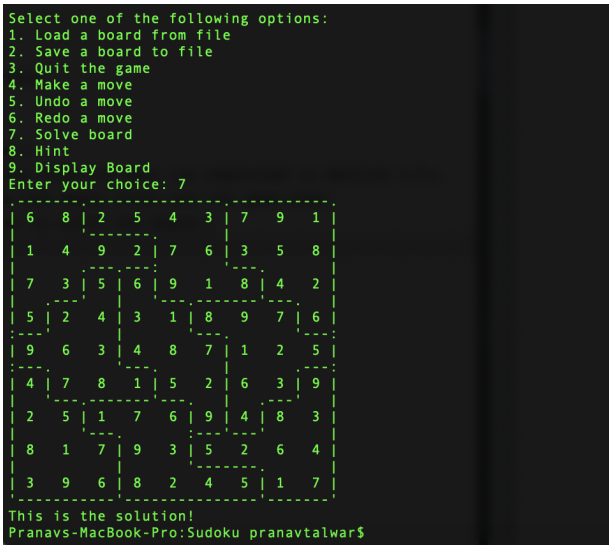

Figure 14: Ending game when player uses solver

5 Additional Features

The additional features implemented by me are:

- Undo and Redo a move
- Check File Contents being loaded
- JigSaw Sudoku Board Solver
- Hint for a move

5.1 Undo and Redo a move

Implementation: The main function calls the game function with a blank Sudoku board and two empty lists which signify the moves and the undone moves. Whenever the player makes a valid move, the move is added to the beginning of the *moves* list. In the case where a player picks the undo option in the menu, the first element of the *moves* list is removed and added to the beginning of the *undoneMoves* list. In the case when the player selects the redo move option, the first element of the *undoneMoves* list is removed and added to the beginning of the *moves* list. Also whenever the player makes a new

move, the *undoneMoves* list is emptied by passing an empty list to the game function when it is recursively called. This is done to avoid conflicts with the new move and previous undone moves. The redo moves option only works if the the *undoneMoves* list is not empty and similarly, the undo option works only if the *moves* is not empty.

5.2 Check File Contents being loaded

Implementation: Whenever a file is loaded using the load option, this function checks the validity of the contents of the file. It checks if the file has 18 lines, each line's length is 9, each Jigsaw piece in the Sudoku board has a size of 9, if the contents of the first 9 lines are just numbers between 0-8 and the next 9 lines are just digits between 1-9 and dots ('.'). This is achieved by using simple list comprehensions and conditions.

5.3 JigSaw Sudoku Board Solver

Implementation: The game finds the first empty space in the Sudoku Board starting from the top and finds the possible numbers that can be inputted into that space based on what is present in its JigSaw piece, row and column. The *solve* function tries every possible number for an empty space in the board and backtracks if no possible number is present for an empty cell until such a combination is found that satisfies the JigSaw Sudoku constraints. All possible combinations are recursed through and the first combination is the answer if a solution exists. If no solution is possible with the current configuration of the board, then a message will be printed and the game would continue. If there is a possible solution, then the game would end.

5.4 Hint for a move

Implementation: The hint option when invoked by the player, solves the board with respect to the current state of the board using the solver function. It then finds the first element that is empty in the user's board and using the solved board, it provides the user with a hint to solve the board. If there is no solution possible with the current configuration of the board, then no hint will be provided.