

Jigsaw Sudoku By Haskell

[Build Project](#)

Go into the stack project root directory.

Build command:

stack setup (optional)

stack build

The file can be executed by:

stack exec jigsawSudoku-exe

The file will be stored in a dist folder inside the .stack-work folder.

In order to read text file. The text file should be in same directory of the executable.

[Functionality](#)

To Start the Sudoku game. First, call the executable. If the name of the compiled executable is jigsawSudoku-exe, then please go to the same directory of the executable in terminal and input:

```
./jigsawSudoku-exe
```

Load Function:

After executing the executable, the program will ask user to input "load" or "quit".

If "quit" is selected, user quits the game.

If "load" is selected, the program will ask user to select a Sudoku game.

User has to provide a file path to a text file that store the Sudoku game.

The text file format should be as specified in the question paper. Files with 179 characters and 180 characters (Difference is caused by the newline symbol in the text file) are both supported by the game.

If a text file is provided, the program will try to load a Sudoku game.

If the text file is a valid Sudoku game that match the format, the game will be loaded.

Sample Snapshot:

```
Please select a sudoku game
map.txt
Map is loaded successfully!
The initial map:

-----
| . 8 | . . . | . . 1 |
|-----|
| 1 . . . | . 6 | 3 . . | | |
|---|---|---|---|---|
| . . | . | . | . . | 4 . |
|-----|
| . | . . | 3 . | 8 . 7 | . |
|-----|
| . . . | . 8 . | . . . | | |
|---|---|---|---|---|
| . | 7 . 1 | . 2 | . . | . |
|-----|
| . 5 | . . . | . | . | . |
|-----|
| . . 7 | 9 . | . . . 4 |
|-----|
| 3 . . | . . . | . 1 . |
|-----|
```

If the text file is not valid or violates Sudoku rules, the program will say that the file is not valid and end.

“load” can be called during the game to load a new map as well.

Other Functions:

After successfully loading the Sudoku game. The program will ask user to input command. There are 2 commands for the basic functionalities (Ending is not triggered by command).

User can input commands repeatedly until the game ends or he/ she quits the game by giving the “quit” command.

The 2 commands are “play” and “save”.

Play (Move) Function

By giving the “play” command, user can make move on the Sudoku game. The program will show the current Sudoku grid and sequentially ask the user to input row number, column number (in 0-based index) and the value of the move.

The move must be valid in terms of range and Sudoku rules. Otherwise, the user will be asked to perform another action. (Can input other commands)

Sample Snapshot:

```

Please select your action.
play

Make a move.
The current grid:
-----
| . 8 | . . . | . . 1 | | |
|     |-----|     |
| 1 . . . | . 6 | 3 . . |
|     |-----|     |
| . . | . | . | . . | 4 . |
|     |-----|-----|
| . | . . | 3 . | 8 . 7 | . |
|-----|-----|-----|
| . . . | . 8 . | . . . |
|-----|-----|-----|
| . | 7 . 1 | . 2 | . . | . |
|     |-----|-----|
| . 5 | . . . | . | . . | . |
|     |-----|-----|
| . . 7 | 9 . | . . . 4 |
|     |-----|-----|
| 3 . . | . . . . | 1 . |
|-----|-----|-----|

Row:
0

Column:
0

Value:
5

```

User can make moves until the grid is filled.

Save Function

By save command, user can save the current game with the completed moves into a text file. To save a game, user just have to type “save” when the program ask user to make an action.

User have to provide a file path (the file name if user wants to save the game in current directory) for the program to save the output file.

The file will be in txt format. With 180 characters, including the newline symbols in every line.

Sample Snapshot

The current grid:

```
.-----  
| 5   8 | .   .   .   . | .   .   1 |  
|       '-----  
| 1   .   .   . | .   6 | 3   .   . |  
|       .-----:       '-----  
| .   . | . | . | . | .   . | 4   . |  
|       .---'   |       '-----'-----  
| . | .   . | 3   . | 8   .   7 | . |  
:---'   |       '---  
| .   .   . | .   8   . | .   .   . |  
:---  
| . | 7   .   1 | .   2 | .   . | . |  
|       '-----'-----  
| .   5 | .   .   . | . | . | .   . |  
|       '---  
| .   .   7 | 9   . | .   .   .   4 |  
|       |       '-----  
| 3   .   . | .   .   .   . | 1   . |  
'-----'-----'-----'
```

Please select your action.

save

Please specify the file path.

map777.txt

Your game is saved.

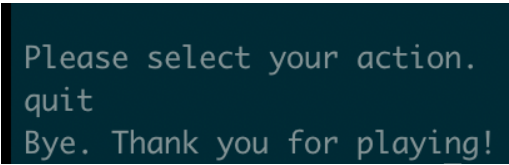
Please select your action.

█

Quit Function

By typing “quit” when program asks for command, user can quit the game.

Sample Snapshot



```
Please select your action.  
quit  
Bye. Thank you for playing!
```

Data Structure

Data structure of this jigsaw Sudoku is inspired by regular Sudoku game. The data is structured to facilitate pruning of cells.

Reference: <https://abhinavsarkar.net/posts/fast-sudoku-solver-in-haskell-1/>

The Structure for representing the jigsaw Sudoku boards:

```
data CellV = Fixed Int | Possible [Int] deriving (Show, Eq)  
type Cell = (CellV, Char)  
type Row = [Cell]  
type Grid = [Row]
```

CellV is a new data representing the value of a cell. It can be fixed or not fixed.

For fixed cell, its value is represented by Fixed Int. The value of a fixed cell cannot be changed in a Sudoku game (except redo or undo).

For non-fixed cell, it is represented by Possible [Int]. It means that the cell is not assigned with value and it is possible to assign it with values inside the integer list.

In order to adapt the data to jigsaw Sudoku, Cell is represented by a tuple of (CellV, Char) in this project. CellV is the value of the cell and Char represents the block number which is from 0 to 8. Cell with number value in the text file will be read as a Fixed cell and cell with '.' value will be read as a Possible cell to the program. The information about a cell can be presented by Cell type sufficiently.

Then, a Row type is used to represent a row in a Sudoku grid (board). There are 9 cells in a row.

After that a Grid is used to store a list of Row. There are 9 rows in 1 grid, representing each row in a Sudoku game.

By the above structure, a Cell can be specified by its Row number (index of the list it belongs to in Grid) and the column number (index of it in the Row it belongs to). Other information can be obtained by looking into the cell's CellV and block number.

In order to build the basic functionalities of jigsaw game, the data structure of a grid (game board) should allow the program to draw the board and generate new board after receiving new cell information. Hence, the grid should hold all the information of location of cells, value of cells and block number of cells. The above structure fulfils all these criteria.

On the other hand, the above data structure is chosen because it facilitates validation and pruning of cells as well. Specifying cell as fixed and non-fixed with possibilities can let the program easily determined whether a value can be assigned to a cell.

Talking about pruning, the structure allows a possible cell to holds all the possibilities it has, it makes guessing of value more efficient. More would be explained in the “solve” function section.

Error Cases And Ending

There are several error cases to handle in the Sudoku game.

First, when the execution is first called, it asked for a command but user may input invalid command. These are handled by the startSudoku function in the project. It only response to “load” or “quit” command, otherwise, it call itself recursively until the user input the valid command.

Second, error occurs when the text file provided by the user is in wrong format. It is handled by readGrid function in the project. The function return Nothing when any value-block-number pair extracted from the text file cannot be transformed to a Cell type. Traverse is used to achieve the goal. When Nothing is return by the function, the program will ask the user to choose a command (load or quit) again. In successful case, the program load a grid from the text file and start the game.

Third, error occurs when user input invalid command in opt function where the program asks the user to specify an action. As the startSudoku function, it asks user to input command recursively, until a valid command is input.

Then, error may occur when user make a move on certain cell. This is handled by a isValidMove function. It checks whether a move is valid in terms of range and Sudoku rules before a move can be made.

Check handled:

The value of row number, column number and value of cell are within bound.

The cell is not fixed yet.

The row does not have same number.

The column does not have same number .

The sub grid (block) does not have same number.

By these validation, only valid move can be made in the game.

After that, error may occur when a text file with correct format but wrong value (violating Sudoku rules) is provided by the user. It is handled by checking the validity of a grid after it is successfully transformed into a grid. The operation is done in the loadGrid function as well. It asks user to load or quit if the text file provided is not valid. The validity is checked by function isGridInvalid. Generally speaking, it checks each row has no duplicated value. Then transpose the grid, so the column become row. Then check the transformed rows to check duplicates in column. After that, it checks duplicates in each sub grid by filtering by block number. The function return true if the grid is invalid.

Ending

As all kind of invalid input, both in terms of format and Sudoku rules are prevented by the checking above, it is easy to check the end of game.

The game ends when the grid is filled up. It is checked after every move.

As the grid translated from text file by the program must be valid and all the moves made by the user must be valid. The game ends and player wins when all the grid is filled up with fixed cell.

The checking: Scanning through the whole grid. If there is no Possible Cell in the grid, the game ends and player wins.

The game would not end even there is no possibilities for the player to win as redo and undo function is available in this game.

Although Sudoku board with no solution is not prevented from input, User can check whether the Sudoku has a solution by the solve command. (Explained in later chapter)

The game only ends when the grid is filled up correctly.

The operation is handled by the function gamelsEnd.

Additional Features:

Implemented additional features:

1. Undo
2. Redo
3. Solve
4. Hint

Undo and Redo

The idea is inspired by:

<https://stackoverflow.com/questions/40411645/is-it-useful-to-create-a-collection-of-generic-function-types-in-haskell>

As undo and redo are implemented similarly, they would be explained together.
The command to use the features.

Undo : undo

Redo: redo

The commands can be used after a grid is loaded successfully.

A new data to support the undo redo features:

```
data Game a = Game [a] [a]
```

There are two list in a Game.

The two lists are basically two stacks.

The first stack stores a list of Grid, which is the history of a Sudoku game.

The second stack stores a list of Grid as well, which is the history of popped (undo) steps.

By passing a Game instead of a single Grid, the program can always pop (undo) the top Grid in the history stack to reverse the game.

More detailed explanation of the functions:

undo:

Popped the top grid in the history stack, put it on the top of the redo (popped) stack. If there is only one Grid in the history stack, then undo make no change to the stack.

redo:

Popped the top grid in the redo stack, put it on the top of the history set. If there is no Grid in the redo stack then does nothing.

In order to make change to this new data structure, changes are always apply to the top Grid in the history stack. And each new move should clear the redo stack as the redo stack is no longer the future of the current Grid.

A function update is used to handle the operation in this project.

Solve

The command to call the solve feature:

solve

The command can be used after a grid is loaded successfully.

The implementation of the function is mainly based on:

<https://abhinavsarkar.net/posts/fast-sudoku-solver-in-haskell-1/>

There are mainly few repeating steps in the approaching:

1. Pruning the Grid by each row, each column and each sub-grid.
2. Recursively Prune the Grid until the Grid is fixed. (Nothing will be changed even pruning is carried out)
3. Making guess on Cell with least possibilities of value in the Grid.
If the cell has two possibilities, the next two guess to explore would be the two grids created by setting the 2 possibilities as fixed value in the 2 grids respectively.
If the cell has more than 2 possibilities, the two guess grids will be a grid with a guessed fix value and a grid with a guessed reduced possible cell
4. Check whether the guess is invalid or completed. If invalid, return Nothing. If completed, the grid is solved. Otherwise, the program will keep on guessing based on the existing guesses until Nothing returned or the grid is solved.
Invalid grid appears when possibilities of cells are pruned to empty list.
5. At last, the solution will be shown.

Functions to support the above steps:

`pruneCells :: [Cell] -> [Maybe Cell]`

The function prune a row of cells by comparing fixed and possible cells inside a row. Fixed value will be removed from possibilities of possible cells.

`gridToRow :: Grid -> Grid`

The function takes a Grid, output a grid with rows that contain cells in the same sub-grid(block).

Turning a sub-grid into a row, the sub-grid can be pruned by `pruneCells` as well.

`pruneCells' :: Grid -> Maybe Grid`

Similar to `pruneCells` but adapted for pruning cells by comparing them with cells in same sub-grid. Instead of taking a row, it takes a whole grid as cells in same block can be filtered out easily by the block number.

`pruneGrid' :: Grid -> Maybe Grid`

A combination of the above pruning function. It first prune a grid by row. Then prune it by column by transposing the grid before pruning it. After pruning, the grid will be transposed again and turn back to the original structure. After that, the pruned will passed to `pruneCells'` for sub-grid pruning.

`pruneGrid :: Grid -> Maybe Grid`

A function run `pruneGrid'` recursively to get the fixed value of the pruned grid. The function calls `pruneGrid'` until the function can make no change to the grid.

`nextGrid :: Grid -> (Grid, Grid)`

Guessing function of the solve functionalities. It guess the next grid by assign value to the cell with least possibility. The tuple output from the function holds the possible cases of the selected cell. Each will be further assessed in the solve function. One of the tuple must guess the cell by assigning a fixed value.

`isGridInvalid :: Grid -> Bool`

A function checking whether a Grid is invalid in terms of Sudoku rules and the data structure. i.e. the Grid should align to Sudoku rules and there should be values inside the possible list of a possible cell. This is called in solve function to decide whether further guessing should be done for a guessed grid.

`solve :: Grid -> Maybe Grid`

The main function of solve features.

The function recursively prune and make guess on grids until solution is found or Nothing is returned in all guesses.

It generates a guessing tree by calling two solve with two grid generated by the guessing function `nextGrid` at the end and takes the result of the Just value if one exists.

It is a rather brute force approach that making guess based on the pruned grid.

Hint

A hint feature is offered in the game.

The command of the feature:

`hint`

Instead of providing the correct value of a cell, it finds the cell with least possibilities in the current game and suggest to the player.

Hence although the value is always valid (not violating the constraints of Sudoku), it may not be the correct answer in some situation as incorrect moves are already done before.

When there is no possible move, the hint function will remind the user that there is no possible move and it is better for he/ she to redo the game.

The idea is inspired by the solve feature implemented before. Making used of the `pruneGrid` function, the program can narrow down the possibilities of each cell. After that, the program is able to suggest a cell with least possibilities to the user.

If Fixed cell is found during pruning , the feature will directly suggest the cell to the user.

List difference between pruned grid and non-pruned grid is used to find out the newly discovered Fixed cell.

All the related function is written in Hint.hs. More can be checked from the comment in the code.