The University of Hong Kong

COMP3258: Functional Programming

Final Project (Jigsaw Sudoku Game)

Deadline: 23:55, Dec 31, 2019 (HKT)

---

1 Sudoku

Wikipedia:

Sudoku, originally called Number Place, is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contain all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution.

In this project, you are required to **design and implement** an **interactive** Jigsaw Sudoku game.

A Jigsaw Sudoku does not have 3x3 boxes, but regions with an irregular shape.

For example, a board might look like

```
.-------.---------------.-----------.
| .   . | .   .   .   . | .   .   . |
|       '-------.       |           |
| .   .   .   . | .   . | .   .   . |
|       .---.---:       '---.       |
| .   . | . | . | .   .   . | .   . |
|   .---'   |   '---.-------'---.   |
| . | .   . | .   . | .   .   . | . |
:---'       |       '---.       '---:
| .   .   . | .   .   . | .   .   . |
:---.       '---.       |       .---:
| . | .   .   . | .   . | .   . | . |
|   '---.-------'---.   |   .---'   |
| .   . | .   .   . | . | . | .   . |
|       '---.       :---'---'       |
| .   .   . | .   . | .   .   .   . |
|           |       '-------.       |
| .   .   . | .   .   .   . | .   . |
'-----------'---------------'-------'
```

One way to represent the shape of a board (like the above) is to use digits (from 0 to 8) to represent a connected block:

```
001111222
000011222
003411122
033445552
333444555
633344558
667774588
666778888
666777788
```

Here the numbers from 0 to 8 represent the different blocks in a board (there are 9 blocks). The occurrences of a number **n** specifies to what block does the row/column belong to.

2 Basic game functionality

You're required to implement four basic functionalities for this game:

1) **Load/Save**: You should implement load and save operations that load and save boards from a given text file.

2) **Make Move**: Given that a board is loaded, you should allow a player to make a move by specifying the row, column and number to be added to the board.

3) **Error Handling**: When a player makes a move the program should detect errors, such as violations of the Sudoku constraints or checking for ranges of numbers and positions.

4) **Ending**: When a board is complete (i.e. all the numbers have been filled in), the program should determine whether the player has won. That is if the Sudoko board is a valid one, which does not violate any constraints of the game.

One way to develop the user-interface of the program is to think of it as consisting of a few different commands. For example, for the basic functionality above, we could have the following commands:

1) **load file**: Loads a Sudoku board
2) **save file**: Saves a Sudoku board
3) **quit**: Quits the game
4) **move**: Makes a move on the current Sudoko board

3 Example of the game play

Although we do not require you necessarily follow the example, your program should have basic 4 functions pointed out previously:

Loading/saving file When the program loads it should allow the option to read/write a board from/to a text file (we provide an example called *map.txt*).

Format of the text file:

1. The first 9 lines describe the block and boundary of a initial board (as section 1 shows)
2. The following 9 lines describe the number placed at the board. The hole is denoted as a dot.

For example, the content of *map.txt* file might be:

```
001111222
000011222
003411122
033445552
333444555
633344558
667774588
666778888
666777788
.8......1
1....63..
.......4.
...3.8.7.
....8....
.7.1.2...
.5.......
..79....4
3......1.
```

Initial board:

```
Read board successfully!
Initial board:
.-------.---------------.-----------.
| .   8 | .   .   .   . | .   .   1 |
|       '-------.       |           |
| 1   .   .   . | .   6 | 3   .   . |
|       .---.---:       '---.       |
| .   . | . | . | .   .   . | 4   . |
|   .---'   |   '---.-------'---.   |
| . | .   . | 3   . | 8   .   7 | . |
:---'       |       '---.       '---:
| .   .   . | .   8   . | .   .   . |
:---.       '---.       |       .---:
| . | 7   .   1 | .   2 | .   . | . |
|   '---.-------'---.   |   .---'   |
| .   5 | .   .   . | . | . | .   . |
|       '---.       :---'---'       |
| .   .   7 | 9   . | .   .   .   4 |
|           |       '-------.       |
| 3   .   . | .   .   .   . | 1   . |
'-----------'---------------'-------'
```

Move Ask player to provide a location and a number of for the next step. (Assume it is 0-based index)

```
<we omit some previous steps>
Next move:
Row:
0
Column:
0
Number:
6
New board:
```

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

Here, line 3, 5 and 7 are your input and other lines are program output.

Error handling You program should handle some error cases.

```
<we omit some previous steps>
Next move:
Row:
2
Column:
1
Number:
6
Sorry, there is a conflict existing in your board.
Your current board:
```

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

In this case we try to insert the number 6 in the 2nd row, 1st column, but there is already a number 1 at that position! Therefore, the move is rejected.

There are various other checks that can be done. For instance check that rows and columns are within the correct bounds; or that the number itself is within bounds. Furthermore you can do basic checks, such as checking that the same line/row/box does not contain the number you are trying to insert.

Ending Determine if a game ends.

For example, if you find the board fills up, you should terminate the game.

```
<we omit some previous steps>
New board:
```

```
.-------.---------------.-----------.
| 6   8 | 2   5   4   3 | 7   9   1 |
|       '-------.       |           |
| 1   4   9   2 | 7   6 | 3   5   8 |
|       .---.---:       '---.       |
| 7   3 | 5 | 6 | 9   1   8 | 4   2 |
|   .---'   |   '---.-------'---.   |
| 5 | 2   4 | 3   1 | 8   9   7 | 6 |
:---'       |       '---.       '---:
| 9   6   3 | 4   8   7 | 1   2   5 |
:---.       '---.       |       .---:
| 4 | 7   8   1 | 5   2 | 6   3 | 9 |
|   '---.-------'---.   |   .---'   |
| 2   5 | 1   7   6 | 9 | 4 | 8   3 |
|       '---.       :---'---'       |
| 8   1   7 | 9   3 | 5   2   6   4 |
|           |       '-------.       |
| 3   9   6 | 8   2   4   5 | 1   7 |
'-----------'---------------'-------'
Congratulations! You win the game!
```

When the game terminates, the player wins if the Sudoku constraints are satisfied (all columns, rows and blocks have distinct numbers).

4 Extra Features

Besides the four basic features, your program can provide more functions. Here are some ideas for additional features:

- **Support undo/redo operations**: you could support an undo and redo operations that, respectively, undo and redo moves.
- **Hints**: You could support hints, allowing the user to ask for a hint. The hint should of course always be valid (i.e. you should not guess a number that violates the constraints of the game).
- **Solver**: You could support a solver functionality that given a board, it automatically finds a solution for the board (if one exists).
- **Board generation**: You could support a functionality that allows the automatic generation of new boards (with different shapes and number configurations). Your generator should only generate **solvable** boards.
- **Good-looking user interface**: you could support a better user interface. For example, instead of a command-line interpreter you could develop a graphical interface using one of the various graphic libraries for widgets available in Haskell.
- **Other features**: There are several other features that you could implement. We will value your creativity and non-trivial features that you design and implement.

Most of the extra features above (except the UI) can be added via new commands to the user interface. For example, you could have (some) of the following commands:

1) **new**: random board generation
2) **solve**: for automaticaly solving a board
3) **undo**: for undoing the last move
4) **redo**: for redoing the previous undone move
5) **hint**: to ask for a hint

If you develop a graphic user interface (or some other more elaborated form of interface) you can adapt the commands into other things (like items in a menu).

5 Final Report

You should write a short final report (in pdf format) that:

1. Describes how to build your project. It is highly recommended that your project builds with 1 or 2 commands (for example by employing a Makefile or some other build scripts). If your project cannot be easily build, you may be penalized.

2. Describes the functionality of your program (how to play a game, how to save/load files, etc).

3. Explains your choice of data structures for representing Jigsaw Sudoko boards.

4. Explains how your code deals with error cases and ending.

5. Explains the additional features that you implement. You should start by listing **all** the additional features that you have implemented, and then explain those features and how their implementation works.

6 Grading

Grading criteria:

1. Correctness (50%): your program should implement the 4 basic requirements (and optional extra features) and give the correct feedback to user.

2. Lecture understanding (30%): We will evaluate good use of functional programming techniques (such as recursion, list comprehensions, higher-order functions and data and type declaration).

3. Code specification (20%): your code should have good naming convention/code reuse/comments. See https://wiki.haskell.org/Programming_guidelines for details.

4. Extra features: If you only implement the basic functionality (no extra features) the maximum grade that you can have is capped at 75 points. Therefore a perfect project solution scoring 100 points should implement the basic functionality correctly, with elegant well-document Haskell code, some interesting extra features and a well-written report that describes the whole project in a good manner.

7 Advice

The main advice is to get the basic functionality for the project correct, together with a good report. This alone will probably guarantee a positive (pass) grade for the course. If time permits you can try to design some extra features for extra marks.

Also do not forget that we have plenty of Haskell code from the Lectures and Tutorials that can be helpful to get you started with the project. Moreover it is not hard to find Sudoku related code online, that you can try to adapt to the JigSaw Sudoku. If, however, you base your code on code available online, please make sure to acknowledge the source in your comments and your final report to avoid plagiarism!

8 Tips

- How to read and save files in Haskell?

  Haskell functions `readFile` and `writeFile` might help you.

- How to get a random number in Haskell?

  `System.Random` library might help you.

- I can interact with my program in GHCi, is it acceptable?

  No. You should ensure that your program is able to be compiled by `ghc` command easily. Here is a tutorial about `main` entrance function for you: http://learnyouahaskell.com/input-and-output

- What if my program using a third-party library?

  In this case, judge machine might not have such a third-party library, i.e. `ghc` cannot compile your work properly. As a result, your makefile should tell judge machine where to find the library. More precisely, you should use stack tool for developing your Haskell project.

Submission

Please submit your solution on Moodle before the deadline. All the files should be compressed into a Zip file. **Please note that the deadline is strict and set by the University. The Lecturer and TA's have no power to extend the deadline!** Failure to submit on the deadline will mean that you'll score 0.