Assignment #4

Instructions

- In this assignment you can create as many source files as needed. Make sure to give the files meaningful names. Each file name must end with .hs file extension.
- Each file must start with a compiler directive and two comment lines which contains your **full name** (in English) and **ID** number

```
{-# OPTIONS -Wall #-}
-- John Doe
-- 654321987
```

- You must put type annotations to each of your functions in all your files.
- All files must pass compilation cleanly with 0 errors and 0 warnings.

task

In this assignment you will create a mine sweeper game. In case you are not familiar with the game here is a link to an online version of the game: http://minesweeperonline.com. Our implementation will be a textual version of the original game. Here are the requirements:

1. The game is run from command-line with the following command-line arguments:
   1. width and height are both integer numbers in the range 10 - 20
   2. mine-count is an integer number in the range 4 - 199
   3. example of running the game with a 10x10 board and 16 mines:
      ```
      stack run 10 10 16
      ```
   4. You must validate all command line argument values. If the player provide invalid values, too many arguments or too few arguments you must output a relevant error message and then exit.
2. If all command line arguments are valid the player should be greeted with a textual representation of the initial state of the game. Here is an example of an initial state of a game with the size board of 10x10:

```
   001 002 003 004 005 006 007 008 009 010
001 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
002 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
003 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
004 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
005 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
006 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
007 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
008 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
009 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
010 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
what is your next move?
```

3. Note that a prompt such as: `"what is your next move?"` must be displayed.
4. At this point your program must wait for input from the player. In mine-sweeper game the player can choose between two possible actions:
    1. `Dig x y` - this is similar to a player left-click in the GUI version of the game.
    2. `Flag x y` - this is similar to a player right-click in the GUI version.
5. Since our program doesn't support mouse input, the player will have to choose the row and column numbers textually.
6. All player input must be validated for correctness. If the player types an invalid action, invalid row or column values, the program will output an error message and the player will be given a chance to re-type his next move.
7. The game must function as a real playable mine-sweeper. If the player typed the action: `Dig 5 5` (which is considered a valid input) the program must output the new state of the game.
8. The following state is displayed after a `Dig 5 5` and `Flag 9 8` actions:

```
    001 002 003 004 005 006 007 008 009 010 011 012 013
001 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
002 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
003 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
004 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
005 [ ] [ ] [ ] [ ] [2] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
006 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
007 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
008 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [!] [ ] [ ] [ ] [ ]
009 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
010 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
011 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
012 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
013 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

9. The number 2 in the example above is the amount of mines that lie at adjacent cells to the cell at location `(5,5)`. If there are zero mines around the chosen location then the program must automatically dig all around the cell revealing more values and continue to reveal more and more cells recursively until non-zero values are revealed just like in the original game.
10. When a player assume a mine to be hidden in a certain location he is expected to flag the suspected location using the Flag action. This action will be denoted in the game with a `'!'` character at the relevant location like in the example above at location `(9,8)`
    1. Note that if you flag a location that is already dug nothing happens. The state of the game must remain the same.
    2. Note that if you dig in a location that has a flag, nothing happens.
    3. You CAN flag a location that already has a flag in it. The program automatically understand this as un-flag action.

4. The Flag x y action serve as a toggle, it will either flag or un-flag based on the state of the game.
5. It is allowed to put more flags than the known mine count.
6. Note that a game cannot be won with more flags than mines.

11. If a player dig in a location where a mine is hiding, the program must output a board with all mine locations revealed to the player using `'*'` characters to denote where the mines have been hidden. You must also add a message such as `"Boom! game is over".` Here is an example of a board with 10 mines and some cells already revealed in previous moves:

```
    001 002 003 004 005 006 007 008 009 010 011 012 013
001 [0] [0] [0] [1] [*] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
002 [0] [0] [0] [1] [1] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
003 [1] [1] [1] [0] [1] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
004 [ ] [*] [1] [0] [1] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
005 [ ] [ ] [1] [1] [2] [*] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
006 [ ] [ ] [ ] [*] [ ] [ ] [ ] [ ] [ ] [ ] [*] [ ] [ ]
007 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [*] [ ] [ ]
008 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [!] [ ] [ ] [ ] [ ]
009 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [*] [ ] [ ] [ ] [ ] [ ]
010 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
011 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
012 [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [*] [ ]
013 [ ] [ ] [ ] [*] [*] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
BOOM! game is over
```

12. Note that the flag at `(9,8)` was wrongly placed and is still shown after the explosion.
13. If the player managed to reveal the whole board without exploding, the program must display the message: "you win! all mines cleared" and exit.
14. The program must always respond to player input correctly, and never get into an infinite loop or throw exceptions.

Hints and suggestions

You are free to implement the game in any way possible, as long as the program compile and run according to the requirements above. The following hints and suggestions are here to help, but are not part of the requirements and considered optional:

1. It is enough to split the code into two modules. the Main module will handle all input / output operations, and a MineSweeper module will handle all game logic. The MineSweeper module can be implemented with pure code only, without any IO actions.
2. In the MineSweeper module I suggest to implement the following functions:
    1. validateArgs - this function get all values from command line (as function arguments) and make sure the values are valid. Please note that a game board cannot contain more than `width * height - 1` mines, otherwise

the game is not playable. The function return type should be `Either String ()` where it will return a `Left "error message"` when one or more values are out of range or `Right ()` when all values are valid.
2. generateMines - this function will randomly generate locations of all mines inside some data structure. Note that this function needs a random number generator to do its task.
3. newGame - this function will create an empty game board which will serve as the initial state in the game.
4. showGame - this function will generate a multi-line string (a string with newline characters) that visualise the current state of the game.
5. dig - this function will implement the logic of a dig action. I suggest that this function will receive the current state of the game as one of the parameters of the function, and will calculate the next state of the game as the return value.
6. toggleFlag - this function will implement the Flag action. In a similar manner this function will calculate the next state based on the current state and the flag action data (the location of the flag).
7. act - this function will receive an Action value and will call either dig or toggleFlag accordingly
8. isGameOn - this function will analyse the state of the game and return False only when the game is over (whether winning or loosing).
9. gameOver - this function will analyse a game state that has ended and return a string the show the final state of the game.
    1. If there was an explosion this function will return a string that visualise the game with all revealed cells and all mine locations.
    2. If the player wins then this function simply return a string with a "you win! all mines cleared"
10. isValidAction - a function that will make sure the parsed action is valid, which means the x and y values are within the boundaries of the game board.

3. In the mineSweeper module I suggest to define various data types:
    1. Define Action data type like this:

```
data Action = Dig Int Int
            | Flag Int Int deriving (Read)
```

    2. This will make it easier for you to read player input with minimal effort.
    3. You may use any of the data structures we learned about: Map, Set, lists, tuples to represent mine locations and game state. I can assure you that you don't have to invent a new data structure to implement this game correctly, but if you enjoy doing that you can.
4. In the main module:
    1. Remember that preduled's read throws exception if the input string isn't valid.
    2. I used `readMay` function that reside in module named "Safe" (using "safe" package) instead of prelude's read function. You can find documentation about this function in hoogle website.
    3. I implemented only 3 functions in the main module
        1. `readAction :: Board → IO Action`

2. `gameStep :: Board → GameState → IO GameState`
3. `main :: IO ()`

4. The readAction function reads a string, parse it and validate the values. If the action is valid it will return an IO Action object otherwise it will keep trying to read more and more strings and parse them.
5. The gameStep function is given a Board (which contain the width, height and mines data) and a GameState structure which contains all the revealed cells and flags. This function calls the readAction function and will also call act (from mineSweeper module) to calculate the next state of the game. If the game is over it will return the final state of the game, otherwise it calls itself to allow the player to make more and more moves and progress from one state to another until it either win and lose.
6. The main function get command line arguments, validate the command line arguments values. If the arguments are valid, it calls generateMines and newGame and then it calls gameStep. Finally it will output the final state of the game (using the gameOver function) and exit.