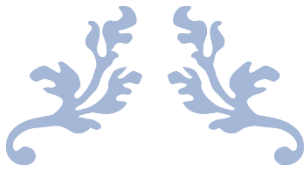


MINESWEEPER IN HASKELL

By: Engla Cederberg Marmefelt, Group 51

PROGRAMKONSTRUKTION OCH DATASTRUKTURER 2021/2022

Table of contents

1. Introduction

In order to broaden my knowledge of imperative programming and how Haskell can be used for an interactive user experience, I have made a minesweeper game for this project.

To create a different game state for each game, the mines will have to randomly be placed in the gameboard. It should also be possible to either press a spot and reveal what it is, or to flag a spot if the player think that it is a mine. If a spot is flagged, it must be possible to deflag it, and remain the original value of the spot (whether it was a mine, a number or a blank).

Since the player needs to state which spot they want to play, the row and column index must be displayed alongside the gameboard. In order to make the game more interactive and more user friendly, the gameboard should be displayed in different colours and shapes.

One of the challenges of constructing a minesweeper game is that each spot in the gameboard has a value that depends on the values of its surrounding spots. If a spot is not a mine, the spot should be the number of mines that surrounds that spot. To get around this problem, each gameboard is built by first creating a mineboard – the gameboards "mirror" that only contains the information of whether a spot is a mine or not – and use this in order to count the values of the non-mine spots. It would have been possible to skip this step, but I used it in order to lessen the chances of bugs arriving from values being calculated "over the boarder" – e.g. the last spot in a row should not affect the first spot in a row.

I also started out by doing the game hard coded, but found along the process that this was rather disadvantageous, both when it came to test and debugging the game and when writing function specifications. However, it took some time to transform the code from hard coded, partly because of the type errors that occurred when I squared integers, since there is a difference between the types of a**2 and a*a in Haskell.

2. Summary

This program will provide a standard 16x16 game of Minesweeper. When the program is run, the game will evolve by carrying out the different moves from the player. A valid format of a move is (Action, (x-coordinate, y-coordinate)). An action is either P (for pressing) or F (for flagging). Both the format and the validity of a move is checked by the program. The player will have to make a new move if the input was invalid.

The user interface has been made user friendly by taking advantage of ANSI code. Different numbers, mines and flags will have different colours and symbols in order to make it easier to distinguish between different values on the gameboard. The symbols are either numbers, blankspaces, an X, an ! or a white **O**. A number represents the number of mines surrounding the spot. A blankspace represents a spot with no surrounding mines. An X represents a mine. An ! represents a flag. A white **O** represents a spot with its value hidden.

The game is initiated by the function main, but the actual function that carries out the game is playerMove. It is technically possible to play the game with arbitrary sizes of the gameboard and arbitrary number of mines, but these variables must manually be changed in order to do that at the moment.

A game will end when a mine is pressed or when all non-mine spots have been displayed.

3. Use cases

3.1 Software requirements

To access the game, the modules Test.HUnit and System.Random must be installed. If cabal is installed, it can be done by cabal install as follows:

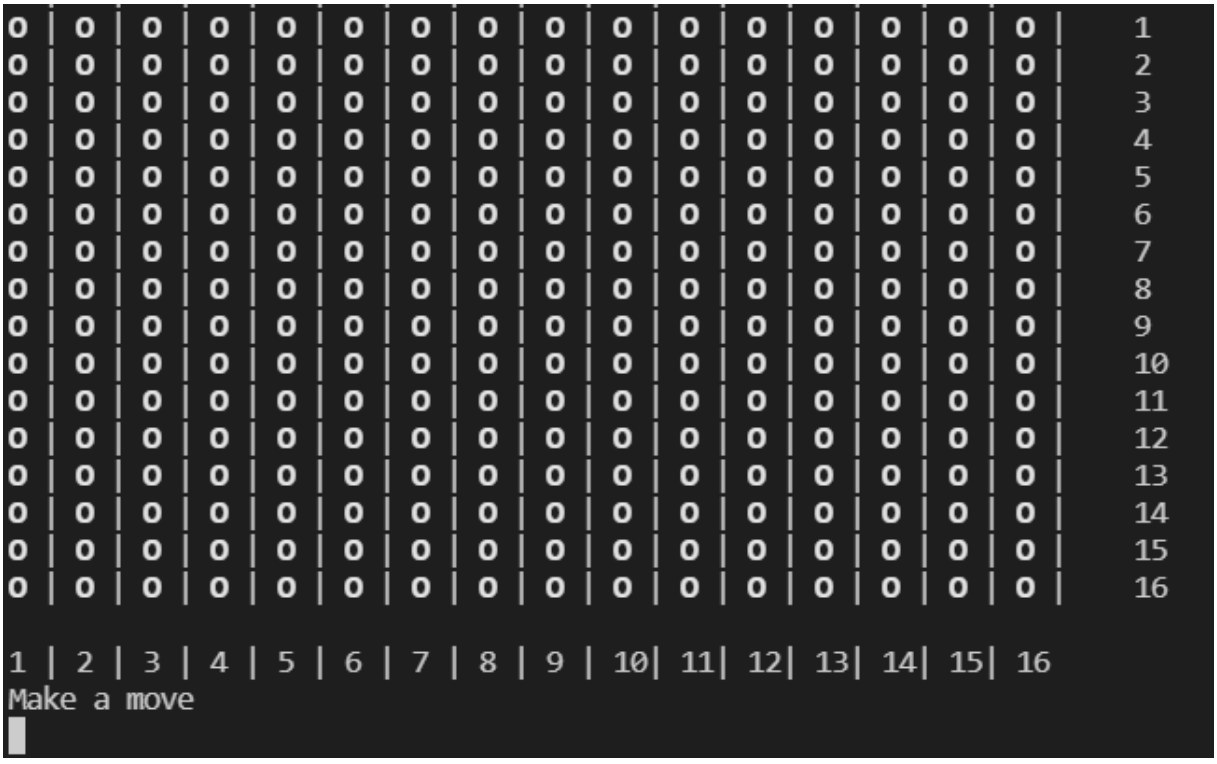
```
cabal install –lib HUnit
```

```
cabal install –lib random
```

The user should then compile the program. The game will start when the program is run.

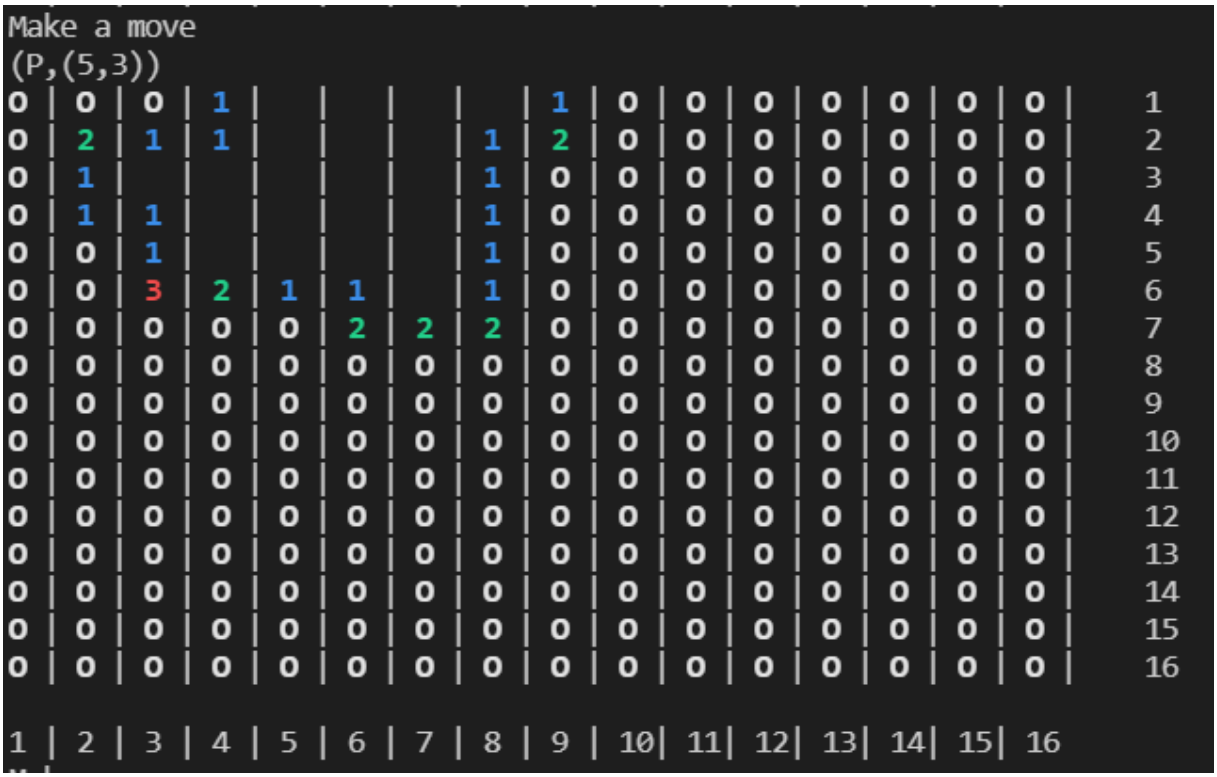
3.2 Playing the game

A newly started game will look like this:

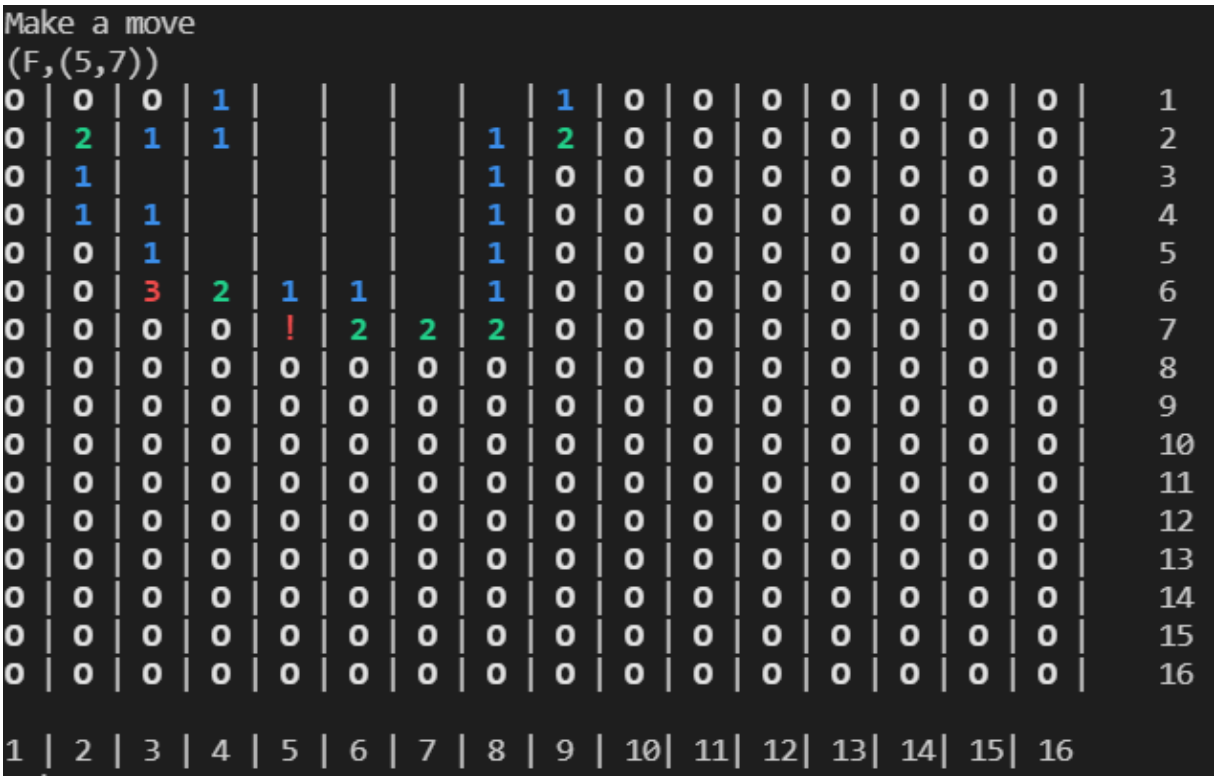


In order to play the game, the player must then provide input in the format of (action,Index). An action is either to Press (P) a spot or to Flag (F) a spot, and the index should be a tuple of the x- and y-coordinates of the spot. For example, if the player want to press the spot in the fifth column in the third row, then the input should be (P,(5,3)).

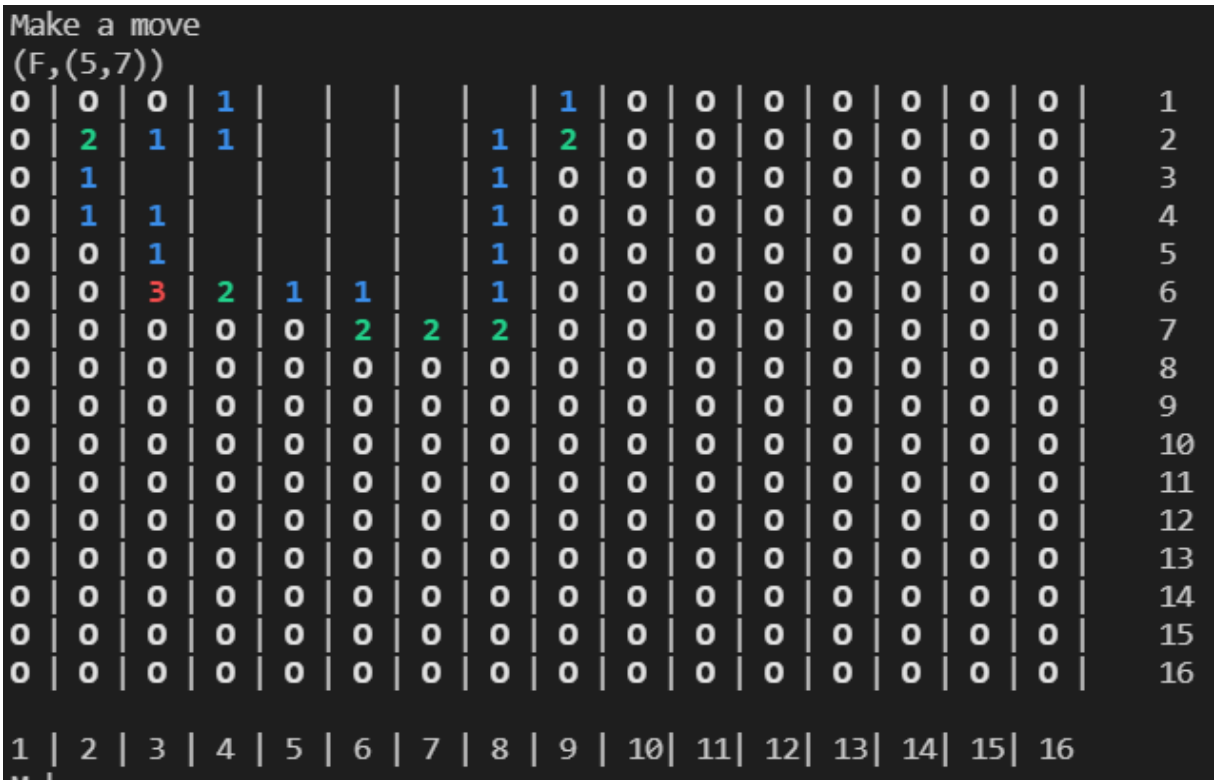
This is a valid move:



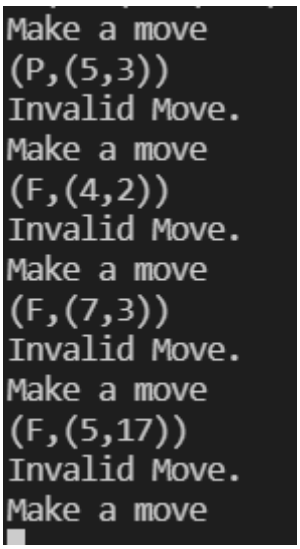
The player can also flag and deflag a spot. If the spot on the fifth column and the seventh row should be flagged, the input should be (F,(5,7)):



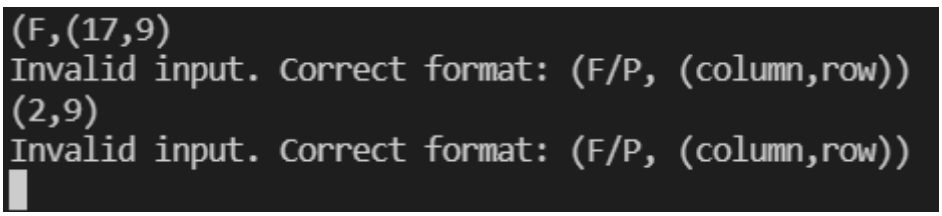
In order to deflag, the same command is used:



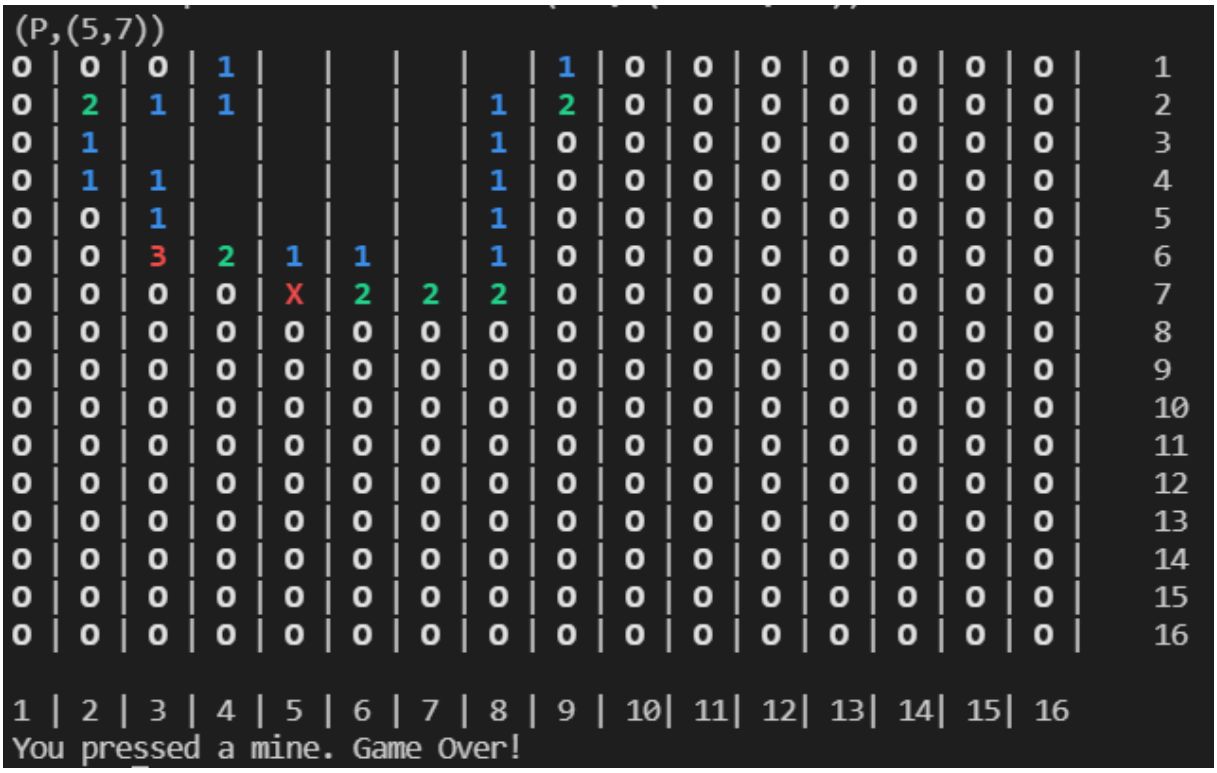
However, making the move (P,(5,3)) again, making a move for an index that is not in the gameboard or making a move on an already displaying spot will result in Invalid Move, and the player will be asked to provide a new move:



If the format is incorrect, the program will catch this and clarify which format it requires of the input.



The game will end if the player presses a mine or if all non-mine spots have been displayed.



In a victorious game state, the mines can either be flagged or left white.

4. Program documentation

4.1 Control flow

The code can be summarised into this pseudo code:

1. genMines
2. mineBoard
3. genGameBoard
4. displayGameboard
5. playerMove
    a. While not victory
        i. Check if victory
        ii. If validMove
            1. If action == P then playMove else flagMove
            2. displayGameBoard
            3. If action == P, and move is the index of a mine then End game
        iii. playerMove
    b. End game

mineBoard will use the return value from genMines, and genGameBoard will use the mineboard from mineBoard. displayGameboard will then display the generated gameboard, and playerMove will initially play with that gameboard, but make a new gameboard with slight modifications with each playing loop. End game is to display what the end result in the game is – if it is a victory, or if a mine was pressed.

4.2 Key data structures

MineData is the data type that is used to represent the value of a spot and its displaying value. By using for example Flag instead of "\ESC[31;1m!\ESC[0m" every time a flag is used, it will be easier to change the ANSI code later on if that is desirable. Each MineData is mapped to an ANSI code in the variable mineDataAnsi, so the specific ANSI code is only stated once in the program for each MineData. It is also a more human friendly format to call a flag "Flag" instead of "\ESC[31;1m!\ESC[0m".

Action is a data type which purpose is to make it quicker for the player to communicate its intentions. With Action, the player only has to input P when wanting to press a spot, instead of writing "press", or "p", or having some kind of numeric system for different actions (e.g. 1 represents press and 2 represents flag).

fromList is provided by the module Data.Map. This module has been used since it is advantageous in performance, compared to regular lists, when dealing while larger amounts of data. No minesweeper game of this scale would probably cause poor performance regardless, but the module provides a handy way of looking up, inserting and deleting values.

4.3 Key functions

playerMove is the function that plays the game and keeps the game running. It is described in 4.1 Control flow, as it is the function that connects many of the playing functions.

validMove is a function that will make sure that a given move is a playable move in the given game state. It will check the validity based on which action has been done, since (de)flagging can be done on white or flag spots, whereas pressing can only be done on white spots. It first checks that the move is within the gameboard. It thereby excludes all indexes that are either too high or too low. Then it checks whether the move is played on a valid spot (white or, if deflagging, flag spot).

playMove's purpose is to find the MineData of the given move and change the display of the spot of the move into the corresponding ANSI code of that MineData. However, if the move is played on a blank spot, it needs to play all the surrounding spots as well. If the move is out of the gameboard's range, the function will return the same game state as it was given.

flagMove will on the other hand not work if the move is out of the gameboards range and it will only operate on the specific index of the move. If the spot played on is white, it will continue to store the underlying MineData, but change the ANSI code to that of a flag. Otherwise, it will do the same thing but change to ANSI code of white.

mineBoard will create base for a gameboard by mapping between a gameboard's index and whether this index should be a mine or not, according to a given list of indexes. It takes an Int and a list of Ints as parameters. The first Int (n) will set the size of the mineboard/gameboard as nxn. The list should contain unique indexes in range 1 to n*n. mineBoard will use the auxiliary function placeMines to create the indexes alongside with checking if this corresponds to a value in the list. If that is the case, the index maps to True, otherwise it maps to False.

genGameBoard uses a mineboard and **countSurroundingMines** to check if its indexes are mines or not, and what other MineData it is otherwise. It maps between the index and the displaying value of a white spot and this MineData.

countSurroundingMines will first identify if the given index corresponds to a mine. If so, it will return the MineData for a mine, Bomb. Otherwise, it will count all mines surrounding the index. If the index is 0, it will count all the X:s as below:

X X X
X 0 X
X X X

If no of the X:s is a mine, it will return the MineData Blank. Otherwise it will return the MineData for the number of mines, like one mine has the MineData One, two mines has Two, and so on.

victory is the function that will determine if the game state is victorious or not by counting all spots that are displayed. It then compare this number to the difference between the

number of spots in the gameboard and the number of mines. If they are equal, it is a victory – the function returns True.

5. Known shortcomings of the program

Later in the process of coding, I realised that the mineboard was not necessary, but could have been worked around by just checking if the index of a spot was on the edge of the gameboard, and then only check the spots that were actually surrounding it. However, lack of time was a contributing factor to why that was not removed.

The final game is a 16 by 16 gameboard with 40 mines. It would have been easy to add the possibility for the player to choose these numbers themselves, but I decided quite late to remove all hard coded parts in the code, and decided that the game was finished like this.

Another feature that would have been beneficial for the program would be to have a third Action E. So that if E was the input, the game would end. It takes a lot of time to play through a whole game of minesweeper, which is not always desirable. When working with this project, I always displayed the key before the game started so that I could more easily found errors about MineData and so on. I could always so the lack of a possibility to end the game didn't occur to me until the key was removed from the code. An Action A could also exist if the player wants to play again. This could either be an option when a game has finished, or a possible input at any point during the game.