**German University in Cairo**
**Department of Computer Science**
**Dr. Nourhan Ehab**

**Concepts of Programming Languages**, Spring Term 2022
**Project 2: Minesweeper Robot**

*Due:* $17^{th}$ *June 2022*

## 1. Project Description

In this project you are going to implement the AI module for a minesweeper robot in Haskell. The environment in which the robot operates is an $4 \times 4$ grid of cells. Initially, a cell on the grid is either empty, contains the robot, or contains a mine. The robot can move in all four directions and is able to collect a mine only if it is in the same cell as the mine. Your program will take as input the initial position of the robot and the positions of all of the mines. The objective of the AI module is to compute a sequence of actions that the robot can follow in order to go to all the mines and collect them. Below is an example grid.

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | X | |
| 2 | | | X | |
| 3 | R | | | |

The robot R starts at (3,0) while the mines are at (2,2) and (1,2). One possible generated sequence of actions by your program is:
`["up","right","right","collect","up","collect"]`

**3. Implementation Description.** Your implementation must contain the following type definitions.

```
type Cell = (Int,Int)
data MyState = Null | S Cell [Cell] String MyState
```

A `Cell` represents a position on the grid with the first coordinate in the pair representing a row number, and the second coordinate representing a column number. The `MyState` structure represents the state of the robot at any given time. It is either `Null` or the data constructor `S` followed by a cell representing the robot's position, a list of cells representing the positions of the mines to be collected, a string representing the last action performed to reach this state, and the parent state. The parent state is the last state the robot was in before doing the last performed action. The initial state of the robot in the above grid is accordingly represented as: `S (3,0) [(2,2),(1,2)] "" Null`

Your implementation must also contain the following functions. Implement the functions in the order they are written below to make your task easier. You are not allowed to change the given type definitions of the functions, but you can add any other helper functions you need.

- `up:: MyState -> MyState`
  The function takes as input a state and returns the state resulting from moving up from the input state. If up will result in going out of the boundaries of the grid, `Null` should be returned.
  Examples:
  ```
  up (S (3,0) [(2,2),(1,2)] "" Null)
  = (S (2,0) [(2,2),(1,2)] "up" (S (3,0) [(2,2),(1,2)] "" Null))

  up (S (0,0) [(2,2),(1,2)] "" Null) = Null
  ```
- `down:: MyState -> MyState`
  The function takes as input a state and returns the state resulting from moving down from the input state. If down will result in going out of the boundaries of the grid, `Null` should be returned.
  Examples:
  ```
  down (S (3,0) [(2,2),(1,2)] "" Null)
  = Null

  down (S (2,0) [(2,2),(1,2)] "up" (S (3,0) [(2,2),(1,2)] "" Null))
  = (S (3,0) [(2,2),(1,2)] "down" (S (2,0) [(2,2),(1,2)]
   "up" (S (3,0) [(2,2),(1,2)] "" Null)))
  ```
- `left:: MyState -> MyState`
  The function takes as input a state and returns the state resulting from moving left from the input state. If left will result in going out of the boundaries of the grid, `Null` should be returned.
  Example:
  ```
  left (S (3,0) [(2,2),(1,2)] "" Null)
  = Null
  ```
- `right:: MyState -> MyState`
  The function takes as input a state and returns the state resulting from moving right from the input state. If right will result in going out of the boundaries of the grid, `Null` should be returned.
  Example:
  ```
  right (S (3,0) [(2,2),(1,2)] "" Null)
  = S (3,1) [(2,2),(1,2)] "right" (S (3,0) [(2,2),(1,2)] "" Null)
  ```
- `collect:: MyState -> MyState`
  The function takes as input a state and returns the state resulting from collecting from the input state. Collecting should not change the position of the robot, but removes the collected mine from the list of mines to be collected. If the robot is not in the same position as one of the mines, `Null` should be returned.
  Example:
  ```
  collect (S (3,1) [(2,2),(3,1)] "right" (S (3,0) [(2,2),(3,1)] "" Null))
  = S (3,1) [(2,2)] "collect"
  (S (3,1) [(2,2),(3,1)] "right" (S (3,0) [(2,2),(3,1)] "" Null))
  ```
- `nextMyStates::MyState->[MyState]`
  The function takes as input a state and returns the set of states resulting from applying `up`, `down`, `left`, `right`, and `collect` from the input state. The output set of states should not contain any `Null` states.

Example:
```
nextMyStates (S (3,0) [(2,2),(1,2)] "" Null) =
[(S (2,0) [(2,2),(1,2)] "up" (S (3,0) [(2,2),(1,2)] "" Null)),
S (3,1) [(2,2),(1,2)] "right" (S (3,0) [(2,2),(1,2)] "" Null) ]
```

- `isGoal::MyState->Bool`
  The function takes as input a state, returns `true` if the input state has no more mines to collect (the list of mines is empty), and `false` otherwise.
  Example:
  ```
  isGoal (S (3,1) [] "collect" (S (3,1) [(3,1)]
  "right" (S (3,0) [(3,1)] "" Null))) = True

  isGoal (S (3,1) [(3,1)] "right" (S (3,0) [(3,1)] "" Null)) = False
  ```
- `search::[MyState]->MyState`
  The function takes as input a list of states. It checks if the head of the input list is a goal state, if it is a goal, it returns the head. Otherwise, it gets the next states from the state at head of the input list, and calls itself recursively with the result of concatenating the tail of the input list with the resulting next states (take care that the order of the concatenation here is important, the next states must be placed by the end of the list).
- `constructSolution:: MyState ->[String]`
  The function takes as input a state and returns a set of strings representing actions that the robot can follow to reach the input state from the initial state. The possible strings in the output list of strings are only `"up"`, `"down"`, `"left"`, `"right"`, and `"collect"`.
  Example:
  ```
  constructSolution (S (3,1) [] "collect" (S (3,1) [(3,1)]
  "right" (S (3,0) [(3,1)] "" Null))) = ["right","collect"]
  ```
- `solve :: Cell->[Cell]->[String]`
  The function takes as input a cell representing the starting position of the robot, a set of cells representing the positions of the mines, and returns a set of strings representing actions that the robot can follow to reach a goal state from the initial state.
  Example:
  ```
  solve (3,0) [(2,2),(1,2)] =
  ["up","right","right","collect","up","collect"]
  ```

For up to 10 **bonus marks**, optimize your implementation to solve grids bigger than 4×4 with more mines. To do this, you are allowed to define any types you need and/or change any type definition of any function other than `solve`. In case you work on an optimized implementation, you must submit it separately. The original basic implementation with the above type definitions must also be submitted. The bonus marks will be added to your coursework.

3. **Teams.** You are allowed to work in teams of four members. You must stick to the same team you worked with in Project 1. IDs for the submitted teams are posted on the CMS.

4. **Report.** You should submit a short report with your code containing a brief description of the implemented functions and screenshots of two different grid configurations with the returned solutions. If you work on the bonus optimized implementation, then you must include a run on a bigger grid.

5. **Deliverables.** You should submit a single `.hs` file named with your team ID containing the basic implementation, and a single `.pdf` file with your report also named with your team ID. In case you work on the bonus implementation, you must submit an additional separate file named `bonus_teamID.hs`. The submission link will be posted on the CMS prior to the submission.