Concepts Project 2 Report
Team 13

Team Members:

52-16824	Zeyad AlaaEldeen Hassan
52-2201	Haneen Khaled Hussein Ahmed
52-7516	Mostafa Abdelraheem Saad ElAmory
52-4013	Hamza Gehad

Functions:

type Cell = (Int,Int):

Creates a type synonym of Int called Cell which consists of (x,y) where x and y are coordinates on the grid.

data MyState = Null | S Cell [Cell] String MyState deriving (Show,Eq):

Creates a new data type called MyState which is either null or the constructor S followed by:

- Current location
- List of mine positions
- String containing the last action performed
- Previous state

(up, down, left, right) :: MyState -> MyState:

Takes the robot's current state as input and returns the state after any change in location. Returns Null if the next movement would result in leaving grid bounds.

collect :: MyState -> MyState:

Takes the robot's current state as input, removes the mine in the robot's position from the list of mines by calling the delete function and returns the new state after the mine is collected. Returns Null if the robot is not in the same position as a mine.

delete :: Eq a => a -> [a] -> [a]:

Takes an element and a list as inputs, returns the list resulting from removing the given element from the given list.

nextMyState :: MyState -> [MyState]:

Receives a state as an input and returns a list of all possible non Null states that could result from a movement in any of the four directions or collecting a mine. Removes Null states by calling the removeNull function.

removeNull :: [MyState] -> [MyState]:

Takes a list of states as input and removes all Null states from the list using the built-in *filter* function

isGoal :: MyState -> Bool:

Receives a state as an input, confirms if the robot has reached its goal by checking if the list of mine positions is empty. If it is, returns True; If not, returns False

search :: [MyState] -> MyState:

Receives a list of states and checks the head. If isGoal is true, it returns the head, otherwise it views the tail and the next states resulting from the head until one returns isGoal equal to true. Returns the first state resulting from reaching the goal which is the shortest path to collect all mines on the grid.

constructSolution :: MyState -> [String]:

Receives as input the current state of the robot and returns the moves as a string needed to perform to collect all mines by calling constructSolutionHelper on the state returned by passing the current state to the search function.

constructSolutionHelper :: MyState -> [String]:

Takes the state which results in collecting all mines as an input and returns the moves needed to do so as a list of strings.

solve :: Cell -> [Cell] -> [String]:

Receives starting position of robot and locations of the mines as inputs then calls constructSolution with the initial state

```
S startingPos minePos "" Null
```

to find the set of actions needed to be performed to solve the grid.

Bonus Explanation:

Modifications (highlighted):

data MyState = Null | S Cell Cell [Cell] [Cell] String MyState deriving (Show,Eq):

Creates a new data type called MyState which is either null or the constructor S followed by:

- Current location
- Max board bounds
- List of mine positions
- List of already visited positions
- String containing the last action performed
- Previous state

(up, down, left, right) :: MyState -> MyState:

Takes the robot's current state as input and returns the state after any change in location. Returns Null if the next movement would result in leaving grid bounds (now determined by the new Cell added to MyState). Returns Null if the next movement would move the robot to a cell which was already visited (element of the already visited positions list)

solve :: Cell -> [Cell] -> [String]:

Receives starting position of robot and locations of the mines as inputs then calls constructSolution with the initial state

```
S startingPos (maxFinder(minePos++[startingPos])) minePos
[startingPos] "" Null
```

to find the set of actions needed to be performed to solve the grid.

New Functions:

maxFinder :: [Cell] -> Cell:

Takes a list of cells as input (the initial robot position and the mine positions) and returns a cell which consists of (maxX, maxY) where maxX is the row number of the downmost object on the grid and maxY is the column number of the rightmost object on the grid. The resulting cell is used to determine grid bounds.

ScreenShots:

Normal Runs:

Run 1:

```
Main> solve (0,3)[(1,2), (2,1), (3,1)]
["down","left","collect","down","left","collect","down","collect"]
```

			R
		X	
	X		
	X		

Run 2:

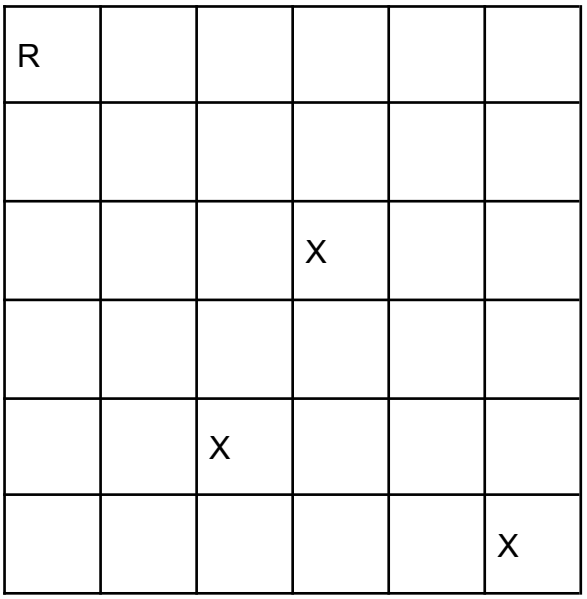
```
Main> solve (0,3)[(3,0), (1,2)]
["down","left","collect","down","down","left","left","collect"]
```

			R
		X	
X			

Bonus Runs:

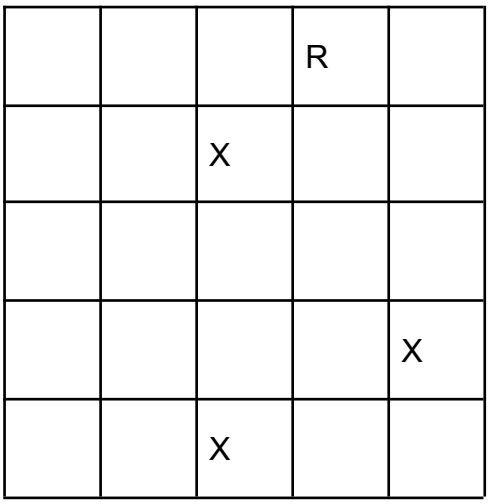
Run 1:

```
Ok, one module loaded.
*Main> solve (0,0) [(5,5), (4,2), (2,3)]
["down","down","right","right","right","collect","down","down","left","collect","down","right","right","right","collect"]
*Main>
```



Run 2:

```
*Main> solve (0,3) [(1,2), (3,4), (4,2)]
["down","left","collect","down","down","down","collect","right","up","right","collect"]
*Main>
```



Run 3:

```
Ok, one module loaded.
*Main> solve (0,3) [(4,3), (4,4), (4,2), (3,3)]
["down","down","down","collect","left","down","collect","right","collect","right","collect"]
*Main>
```

			R	
			X	
		X	X	X