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

Concepts of Programming Languages, Spring Term 2024
Project 2: Peg Reversal

Due: 20^{th} *May 2024*

1. Project Description

In this project, you are going to implement a simple single-player board game of flipping the color of pieces. The game is played on a board in the shape of a '+' where the middle position is (0,0) and all other positions on the board are relative to it. You can see exactly how the positions of all the pegs on the board in the figure below.

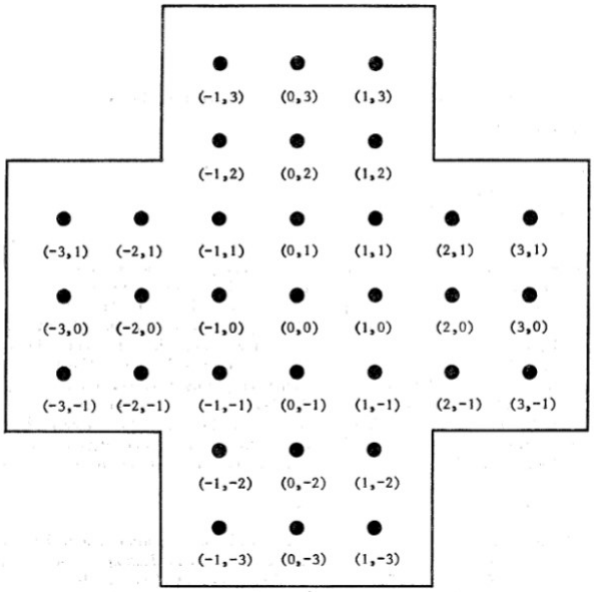

Figure 1: Peg Position on Board Configuration

When the game starts, All the pegs are black except one which will be white. The position of the first white peg can be anywhere on the board. The player can then make a move of flipping a black peg to make it a white one. This can be done if:

a) The peg itself is black (white pegs cannot be flipped).
b) There is at least one peg around it that is white.

When checking the pegs around some positions, we do not look at the diagonals. So, we only look at the pegs in the 4 directions around it (up, down, left, right). The player wins if they are able to flip all the pegs on the board to white.

Your implementation for the game should have functions that create the initial state of the board, output the possible states reachable from a certain state, check the validity of certain moves and check if the goal is reached or not

2. Implementation Description. Your implementation must contain the following type definitions.

```
type Position = (Int,Int)
data Color = W | B deriving (Eq, Show)
data Peg = Peg Position Color deriving (Eq, Show)
data Move = M Position deriving (Eq, Show)
type Board = [Peg]
data State = S Move Board deriving (Eq, Show)
```

a) `Position` represents the position of pegs on the board with a column and row numbers.

b) `Color` is the color of the pegs either white or black.

c) `Peg` represents a piece on the board with the data constructor `Peg`, followed by the position of the peg and it's color. e.g. Peg (0,1) B means that the peg in column 0, row 1 has the color Black

d) `Move` is represented by one of the data constructor `M`, followed by the peg's position on the board that we want to change to white.

e) `Board` is represented by a list of Pegs.

f) `State` is represented by 2 items: the move and the updated board after the move is executed.

Your implementation must also contain the following functions. You are not allowed to change the given type definitions of the functions, but you can add any other helper functions you need.

a) `createBoard:: Position -> Board`
The function takes the position of the first white peg as input and returns a board representing the initial configuration where all pegs are on the board are black except the given position. The function should display an error message if the position is not on the board Example:

```
> createBoard (-2,2)
Program error: The position is not valid.

> createBoard (1,1)
[Peg (-3,-1) B,Peg (-3,0) B,Peg (-3,1) B,Peg (-2,-1) B,
Peg (-2,0) B,Peg (-2,1) B,Peg (-1,-3) B,Peg (-1,-2) B,
Peg (-1,-1) B,Peg (-1,0) B,Peg (-1,1) B,Peg (-1,2) B,
Peg (-1,3) B,Peg (0,-3) B,Peg (0,-2) B,Peg (0,-1) B,
Peg (0,0) B,Peg (0,1) B,Peg (0,2) B,Peg (0,3) B,
Peg (1,-3) B,Peg (1,-2) B,Peg (1,-1) B,Peg (1,0) B,
Peg (1,1) W,Peg (1,2) B,Peg (1,3) B,Peg (2,-1) B,
Peg (2,0) B,Peg (2,1) B,Peg (3,-1) B,Peg (3,0) B,Peg (3,1) B]
```

```
> createBoard (-2,1)
[Peg (-3,-1) B,Peg (-3,0) B,Peg (-3,1) B,Peg (-2,-1) B,
Peg (-2,0) B,Peg (-2,1) W,Peg (-1,-3) B,Peg (-1,-2) B,
Peg (-1,-1) B,Peg (-1,0) B,Peg (-1,1) B,Peg (-1,2) B,
Peg (-1,3) B,Peg (0,-3) B,Peg (0,-2) B,Peg (0,-1) B,
Peg (0,0) B,Peg (0,1) B,Peg (0,2) B,Peg (0,3) B,
Peg (1,-3) B,Peg (1,-2) B,Peg (1,-1) B,Peg (1,0) B,
Peg (1,1) B,Peg (1,2) B,Peg (1,3) B,Peg (2,-1) B,
Peg (2,0) B,Peg (2,1) B,Peg (3,-1) B,Peg (3,0) B,Peg (3,1) B]
```

b) `isValidMove:: Move -> Board -> Bool`
The function takes as input a move and a board. It returns `True` if the move of on the given board is legal, and `False` otherwise.
Example:

```
> isValidMove (M (1,0))
[Peg (-3,-1) B,Peg (-3,0) B,Peg (-3,1) B,Peg (-2,-1) B,
Peg (-2,0) B,Peg (-2,1) B,Peg (-1,-3) B,Peg (-1,-2) B,
Peg (-1,-1) B,Peg (-1,0) B,Peg (-1,1) B,Peg (-1,2) B,
Peg (-1,3) B,Peg (0,-3) B,Peg (0,-2) B,Peg (0,-1) B,
Peg (0,0) B,Peg (0,1) B,Peg (0,2) B,Peg (0,3) B,
Peg (1,-3) B,Peg (1,-2) B,Peg (1,-1) B,Peg (1,0) B,
Peg (1,1) B,Peg (1,2) B,Peg (1,3) B,Peg (2,-1) B,
Peg (2,0) W,Peg (2,1) B,Peg (3,-1) B,Peg (3,0) B,Peg (3,1) B]

True

> isValidMove (M (-2,0))
[Peg (-3,-1) B,Peg (-3,0) B,Peg (-3,1) B,Peg (-2,-1) B,
Peg (-2,0) B,Peg (-2,1) B,Peg (-1,-3) B,Peg (-1,-2) B,
Peg (-1,-1) B,Peg (-1,0) B,Peg (-1,1) B,Peg (-1,2) B,
Peg (-1,3) B,Peg (0,-3) B,Peg (0,-2) B,Peg (0,-1) B,
Peg (0,0) B,Peg (0,1) B,Peg (0,2) B,Peg (0,3) B,
Peg (1,-3) B,Peg (1,-2) B,Peg (1,-1) B,Peg (1,0) B,
Peg (1,1) B,Peg (1,2) B,Peg (1,3) B,Peg (2,-1) B,
Peg (2,0) W,Peg (2,1) B,Peg (3,-1) B,Peg (3,0) B,Peg (3,1) B]

False
```

c) `isGoal:: Board -> Bool`
The function takes as input a board. It returns `True` if the board is a winning one; meaning that all pegs have been turned white on the board, and `False` otherwise. Example:

```
> isGoal
[Peg (-3,-1) W,Peg (-3,0) W,Peg (-3,1) W,Peg (-2,-1) W,
Peg (-2,0) W,Peg (-2,1) W,Peg (-1,-3) W,Peg (-1,-2) W,
Peg (-1,-1) W,Peg (-1,0) W,Peg (-1,1) W,Peg (-1,2) W,
Peg (-1,3) W,Peg (0,-3) W,Peg (0,-2) W,Peg (0,-1) W,
Peg (0,0) W,Peg (0,1) W,Peg (0,2) W,Peg (0,3) W,
Peg (1,-3) W,Peg (1,-2) W,Peg (1,-1) W,Peg (1,0) W,
Peg (1,1) W,Peg (1,2) W,Peg (1,3) W,Peg (2,-1) W,
Peg (2,0) W,Peg (2,1) W,Peg (3,-1) W,Peg (3,0) W,Peg (3,1) W]

True
```

```
> isGoal
[Peg (-3,-1) W,Peg (-3,0) W,Peg (-3,1) W,Peg (-2,-1) W,
Peg (-2,0) W,Peg (-2,1) W,Peg (-1,-3) W,Peg (-1,-2) W,
Peg (-1,-1) W,Peg (-1,0) W,Peg (-1,1) W,Peg (-1,2) W,
Peg (-1,3) W,Peg (0,-3) W,Peg (0,-2) W,Peg (0,-1) W,
Peg (0,0) W,Peg (0,1) B,Peg (0,2) W,Peg (0,3) W,
Peg (1,-3) W,Peg (1,-2) W,Peg (1,-1) W,Peg (1,0) W,
Peg (1,1) W,Peg (1,2) W,Peg (1,3) W,Peg (2,-1) W,
Peg (2,0) W,Peg (2,1) W,Peg (3,-1) W,Peg (3,0) W,Peg (3,1) W]

False
```

d) `showPossibleNextStates:: Board -> [State]`
The function takes as input a board and returns a list of ***all* unique** possible states reachable from that board. a board is reachable if there is a valid move that can be made on the input list that will give us that board. if the board is already in a winning state, it should display an error message.
Example:

```
> showPossibleNextStates
[Peg (-3,-1) W,Peg (-3,0) W,Peg (-3,1) W,Peg (-2,-1) W,
Peg (-2,0) W,Peg (-2,1) W,Peg (-1,-3) W,Peg (-1,-2) W,
Peg (-1,-1) W,Peg (-1,0) W,Peg (-1,1) W,Peg (-1,2) W,
Peg (-1,3) W,Peg (0,-3) W,Peg (0,-2) W,Peg (0,-1) W,
Peg (0,0) W,Peg (0,1) B,Peg (0,2) W,Peg (0,3) W,
Peg (1,-3) W,Peg (1,-2) W,Peg (1,-1) W,Peg (1,0) W,
Peg (1,1) W,Peg (1,2) W,Peg (1,3) W,Peg (2,-1) W,
Peg (2,0) W,Peg (2,1) W,Peg (3,-1) W,Peg (3,0) W,Peg (3,1) W]
```

```
[S (M (0,1)) [Peg (-3,-1) W,Peg (-3,0) W,
Peg (-3,1) W,Peg (-2,-1) W,Peg (-2,0) W,Peg (-2,1) W,
Peg (-1,-3) W,Peg (-1,-2) W,Peg (-1,-1) W,Peg (-1,0) W,
Peg (-1,1) W,Peg (-1,2) W,Peg (-1,3) W,Peg (0,-3) W,
Peg (0,-2) W,Peg (0,-1) W,Peg (0,0) W,Peg (0,1) W,
Peg (0,2) W,Peg (0,3) W,Peg (1,-3) W,Peg (1,-2) W,
Peg (1,-1) W,Peg (1,0) W,Peg (1,1) W,Peg (1,2) W,
Peg (1,3) W,Peg (2,-1) W,Peg (2,0) W,Peg (2,1) W,
Peg (3,-1) W,Peg (3,0) W,Peg (3,1) W]]
```

The state shown in this example is the only possible one in this case because the only valid possible move is to change the piece at (0,1) to be white.

```
> showPossibleNextStates
[Peg (-3,-1) W,Peg (-3,0) W,Peg (-3,1) W,Peg (-2,-1) W,
Peg (-2,0) W,Peg (-2,1) W,Peg (-1,-3) W,Peg (-1,-2) W,
Peg (-1,-1) W,Peg (-1,0) W,Peg (-1,1) W,Peg (-1,2) W,
Peg (-1,3) W,Peg (0,-3) W,Peg (0,-2) W,Peg (0,-1) W,
Peg (0,0) W,Peg (0,1) W,Peg (0,2) W,Peg (0,3) W,
Peg (1,-3) W,Peg (1,-2) W,Peg (1,-1) W,Peg (1,0) W,
Peg (1,1) W,Peg (1,2) W,Peg (1,3) W,Peg (2,-1) W,
Peg (2,0) W,Peg (2,1) W,Peg (3,-1) W,Peg (3,0) W,Peg (3,1) W]

Program error: No Possible States Exist.
```

There are no possible states since there are no valid moves to be made.

Please note that the error messages in your implementation should be the exact same messages as shown in the examples above.

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. **Deliverables.** You should submit a single `.hs` file named with your team ID containing your implementation. The submission link will be posted on the CMS prior to the submission deadline.