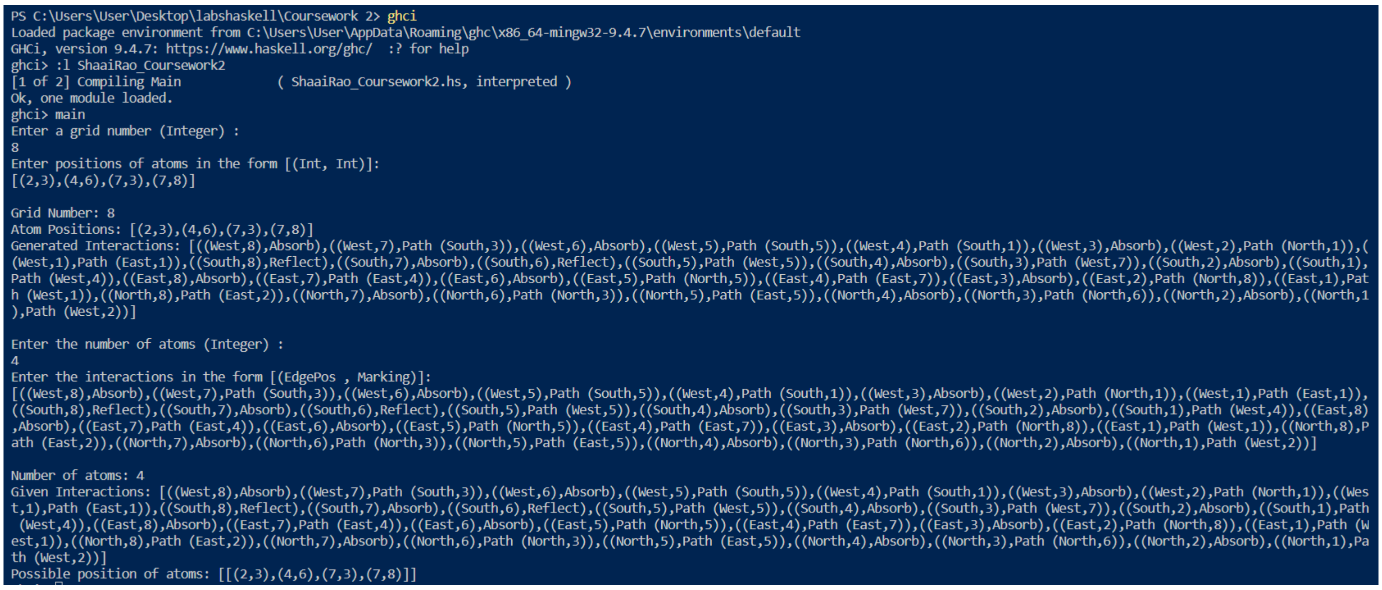
Prepared by: Shaai Rao

Table of contents

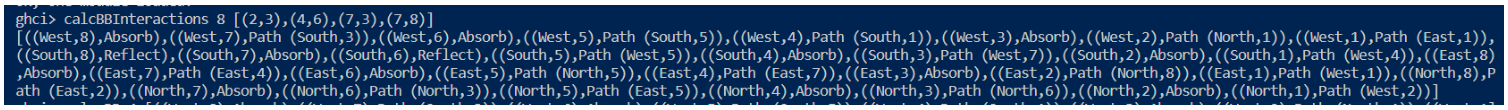
Ways to run the code

1) Open GHCI, then load the file, and type main. Fill in the prompt and answer will be returned.

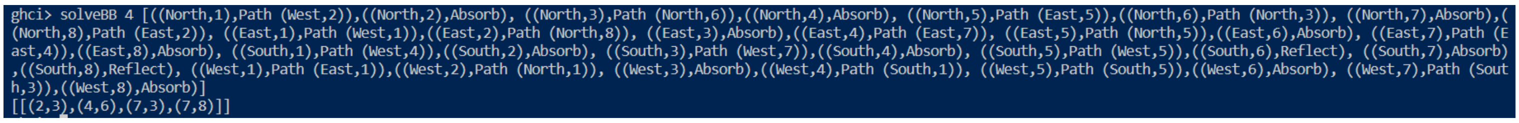


```
PS C:\Users\User\Desktop\labshaskell\Coursework 2> ghci
Loaded package environment from C:\Users\User\AppData\Roaming\ghc\x86_64-mingw32-9.4.7\environments\default
GHCi, version 9.4.7: https://www.haskell.org/ghc/  :? for help
ghci> :l ShaaiRao_Coursework2
[1 of 2] Compiling Main             ( ShaaiRao_Coursework2.hs, interpreted )
Ok, one module loaded.
ghci> main
Enter a grid number (Integer) :
8
Enter positions of atoms in the form [(Int, Int)]:
[(2,3),(4,6),(7,3),(7,8)]

Grid Number: 8
Atom Positions: [(2,3),(4,6),(7,3),(7,8)]
Generated Interactions: [((West,8),Absorb),((West,7),Path (South,3)),((West,6),Absorb),((West,5),Path (South,5)),((West,4),Path (South,1)),((West,3),Absorb),((West,2),Path (North,1)),(
(West,1),Path (East,1)),((South,8),Reflect),((South,7),Absorb),((South,6),Reflect),((South,5),Path (West,5)),((South,4),Absorb),((South,3),Path (West,7)),((South,2),Absorb),((South,1),
Path (West,4)),((East,8),Absorb),((East,7),Path (East,4)),((East,6),Absorb),((East,5),Path (North,5)),((East,4),Path (East,7)),((East,3),Absorb),((East,2),Path (North,8)),((East,1),Pat
h (West,1)),((North,8),Path (East,2)),((North,7),Absorb),((North,6),Path (North,3)),((North,5),Path (East,5)),((North,4),Absorb),((North,3),Path (North,6)),((North,2),Absorb),((North,1
),Path (West,2))]

Enter the number of atoms (Integer) :
4
Enter the interactions in the form [(EdgePos , Marking)]:
[((West,8),Absorb),((West,7),Path (South,3)),((West,6),Absorb),((West,5),Path (South,5)),((West,4),Path (South,1)),((West,3),Absorb),((West,2),Path (North,1)),((West,1),Path (East,1)),
((South,8),Reflect),((South,7),Absorb),((South,6),Reflect),((South,5),Path (West,5)),((South,4),Absorb),((South,3),Path (West,7)),((South,2),Absorb),((South,1),Path (West,4)),((East,8)
,Absorb),((East,7),Path (East,4)),((East,6),Absorb),((East,5),Path (North,5)),((East,4),Path (East,7)),((East,3),Absorb),((East,2),Path (North,8)),((East,1),Path (West,1)),((North,8),P
ath (East,2)),((North,7),Absorb),((North,6),Path (North,3)),((North,5),Path (East,5)),((North,4),Absorb),((North,3),Path (North,6)),((North,2),Absorb),((North,1),Path (West,2))]

Number of atoms: 4
Given Interactions: [((West,8),Absorb),((West,7),Path (South,3)),((West,6),Absorb),((West,5),Path (South,5)),((West,4),Path (South,1)),((West,3),Absorb),((West,2),Path (North,1)),((Wes
t,1),Path (East,1)),((South,8),Reflect),((South,7),Absorb),((South,6),Reflect),((South,5),Path (West,5)),((South,4),Absorb),((South,3),Path (West,7)),((South,2),Absorb),((South,1),Path
 (West,4)),((East,8),Absorb),((East,7),Path (East,4)),((East,6),Absorb),((East,5),Path (North,5)),((East,4),Path (East,7)),((East,3),Absorb),((East,2),Path (North,8)),((East,1),Path (W
est,1)),((North,8),Path (East,2)),((North,7),Absorb),((North,6),Path (North,3)),((North,5),Path (East,5)),((North,4),Absorb),((North,3),Path (North,6)),((North,2),Absorb),((North,1),Pa
th (West,2))]
Possible position of atoms: [[(2,3),(4,6),(7,3),(7,8)]]
```

2) Open GHCI, then load the file and test each main function.



```
ghci> calcBBInteractions 8 [(2,3),(4,6),(7,3),(7,8)]
[((West,8),Absorb),((West,7),Path (South,3)),((West,6),Absorb),((West,5),Path (South,5)),((West,4),Path (South,1)),((West,3),Absorb),((West,2),Path (North,1)),((West,1),Path (East,1)),
((South,8),Reflect),((South,7),Absorb),((South,6),Reflect),((South,5),Path (West,5)),((South,4),Absorb),((South,3),Path (West,7)),((South,2),Absorb),((South,1),Path (West,4)),((East,8)
,Absorb),((East,7),Path (East,4)),((East,6),Absorb),((East,5),Path (North,5)),((East,4),Path (East,7)),((East,3),Absorb),((East,2),Path (North,8)),((East,1),Path (West,1)),((North,8),P
ath (East,2)),((North,7),Absorb),((North,6),Path (North,3)),((North,5),Path (East,5)),((North,4),Absorb),((North,3),Path (North,6)),((North,2),Absorb),((North,1),Path (West,2))]
```



```
ghci> solveBB 4 [((North,1),Path (West,2)),((North,2),Absorb), ((North,3),Path (North,6)),((North,4),Absorb), ((North,5),Path (East,5)),((North,6),Path (North,3)), ((North,7),Absorb),(
(North,8),Path (East,2)), ((East,1),Path (West,1)),((East,2),Path (North,8)), ((East,3),Absorb),((East,4),Path (East,7)), ((East,5),Path (North,5)),((East,6),Absorb), ((East,7),Path (E
ast,4)),((East,8),Absorb), ((South,1),Path (West,4)),((South,2),Absorb), ((South,3),Path (West,7)),((South,4),Absorb), ((South,5),Path (West,5)),((South,6),Reflect), ((South,7),Absorb)
,((South,8),Reflect), ((West,1),Path (East,1)),((West,2),Path (North,1)), ((West,3),Absorb),((West,4),Path (South,1)), ((West,5),Path (South,5)),((West,6),Absorb), ((West,7),Path (Sout
h,3)),((West,8),Absorb)]
[[(2,3),(4,6),(7,3),(7,8)]]
```

- YELLOW: atoms. RED: deflection points. In explanation section, functions are bolded.

Challenge 1

The intuition behind this approach is to take multiple steps each time if certain conditions is met instead of continuously taking one step at a time to reach the destination. For better understanding about the efficiency of this algorithm, let's consider some examples below.

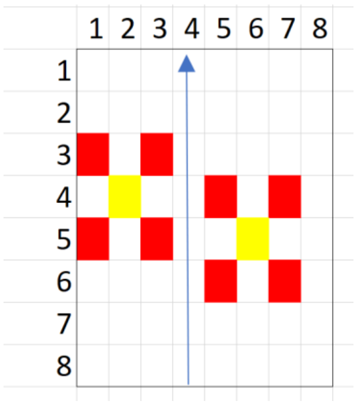


Figure 1

What I mean by the statement "to take multiple steps each time if certain conditions is met" is, let's say the ray in Figure 1 is entering from (South,4). If there's no atoms or deflection points on that path **(conditions met)**, we can surely say that the final destination would be (North,4). Therefore, the output would be ((South,4), Path (North,4)). In this case, we are taking multiple steps at a time which is from (4,8) to (4,1) instantly instead of taking one step at a time such as (4,7),(4,6) up to (4,1). Imagine the same case for a grid with size of 1000x1000, we can move directly to the output with less computation and less recursive call.

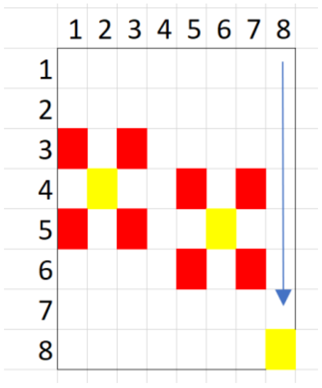


Figure 2

In Figure 2, if there's an atom and no deflection points **(conditions met)** on the path of the ray coming from (North,8), we can directly return an output which is ((North ,8), Absorb). In this case, we saved multiple steps by not moving one step at a time to check condition for each position such as (8,2),(8,3) and so on. This will speed up the process of obtaining the output, especially in larger grids.

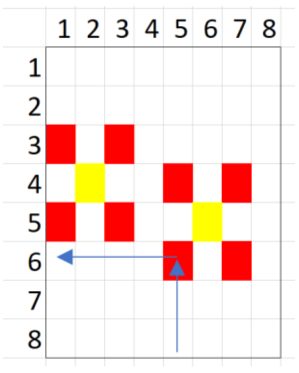


Figure 3

In figure 3, the ray is coming from (South,5) and the initial starting point for the ray will be interpreted as (5,8). The algorithm will be able to identify the nearest deflection point which is on (5,6), hence the ray will directly move to (5,6) and check some condition before getting deflected. At this point, we saved a step as we directly move to the nearest deflection point instead of checking for (5,7). Once get deflected, the algorithm will check again if there's any atoms or deflection points on the deflected path **(conditions met)**. If there's none, it will directly return the output without checking in between points. In this scenario, the output would be ((South,5),Path (West,6)).

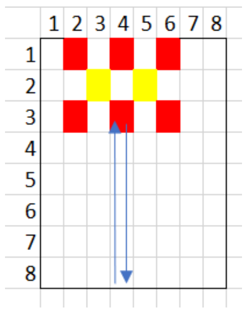


Figure 4

In figure 4, the ray is coming from (South,4). After checking the path, the algorithm will be able to identify the closest deflection point which is on (4,3). The ray will directly move to the deflection point and check some condition before getting deflected. One of it is, the algorithm will check if the there's more than one deflection point at a single point. In this case, there's two deflection points located at the point (4,3) which is bottom right for first atom and bottom left for second atom. Since this is a double deflection, the ray will be deflected to the opposite direction. Again, if there's no atoms or deflection points on that path **(conditions met)**, we can directly return the output instead of checking multiple points.

Explanations for the functions available in Challenge 1.

All the outputs for the below explanations are based on the grid in Figure 5.

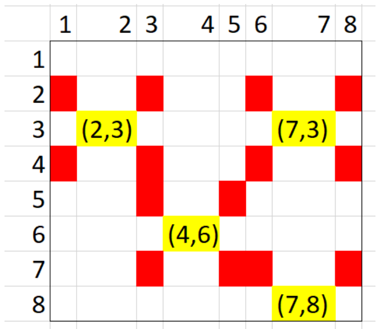
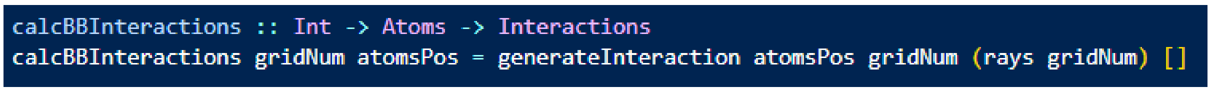


Figure 5 : The representation is the same as given in the coursework

1) calcBBInteractions

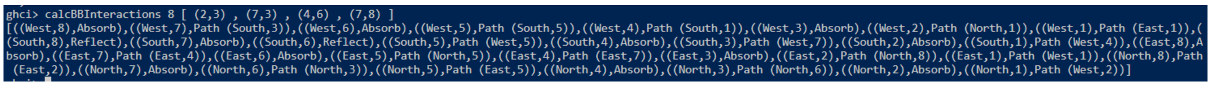


```haskell
calcBBInteractions :: Int -> Atoms -> Interactions
calcBBInteractions gridNum atomsPos = generateInteraction atomsPos gridNum (rays gridNum) []
```

Explanation:

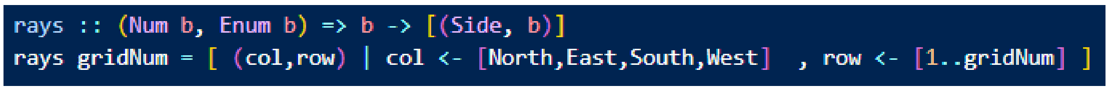
- **calcBBInteractions** function takes two arguments which is number of grid and position of atoms and returns the list of interactions.
- It will call another function known as **generateInteraction** to produce all the possible interactions.
- **rays** function will be called to produce a list of all possible combinations of side and value. For example, (North,1),(North,2) and so on.

Output:



```
ghci> calcBBInteractions 8 [ (2,3) , (7,3) , (4,6) , (7,8) ]
[((West,8),Absorb),((West,7),Path (South,3)),((West,6),Absorb),((West,5),Path (South,5)),((West,4),Path (South,1)),((West,3),Absorb),((West,2),Path (North,1)),((West,1),Path (East,1)),(
(South,8),Reflect),((South,7),Absorb),((South,6),Reflect),((South,5),Path (West,5)),((South,4),Absorb),((South,3),Path (West,7)),((South,2),Absorb),((South,1),Path (West,4)),((East,8),A
bsorb),((East,7),Path (East,4)),((East,6),Absorb),((East,5),Path (North,5)),((East,4),Path (East,7)),((East,3),Absorb),((East,2),Path (North,8)),((East,1),Path (West,1)),((North,8),Path
 (East,2)),((North,7),Absorb),((North,6),Path (North,3)),((North,5),Path (East,5)),((North,4),Absorb),((North,3),Path (North,6)),((North,2),Absorb),((North,1),Path (West,2))]
```

2) rays

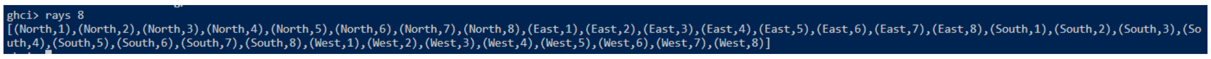


```
rays :: (Num b, Enum b) => b -> [(Side, b)]
rays gridNum = [ (col,row) | col <- [North,East,South,West]  , row <- [1..gridNum] ]
```

Explanation:

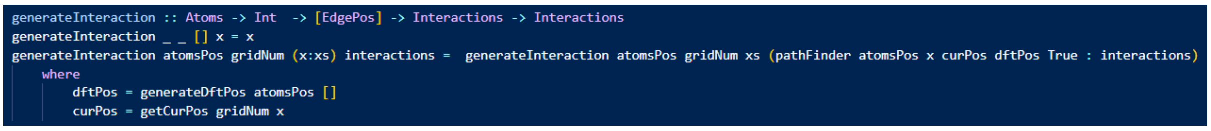
- **rays** function takes the number of grid as the argument and returns a list of all possible combinations of side and value such as (North,1),(North,2) and so on.

Output:



```
ghci> rays 8
[(North,1),(North,2),(North,3),(North,4),(North,5),(North,6),(North,7),(North,8),(East,1),(East,2),(East,3),(East,4),(East,5),(East,6),(East,7),(East,8),(South,1),(South,2),(South,3),(So
uth,4),(South,5),(South,6),(South,7),(South,8),(West,1),(West,2),(West,3),(West,4),(West,5),(West,6),(West,7),(West,8)]
```

3) generateInteraction

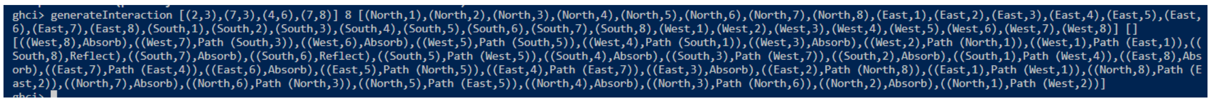


```
generateInteraction :: Atoms -> Int  -> [EdgePos] -> Interactions -> Interactions
generateInteraction _ _ [] x = x
generateInteraction atomsPos gridNum (x:xs) interactions =  generateInteraction atomsPos gridNum xs (pathFinder atomsPos x curPos dftPos True : interactions)
    where
        dftPos = generateDftPos atomsPos []
        curPos = getCurPos gridNum x
```

Explanation:

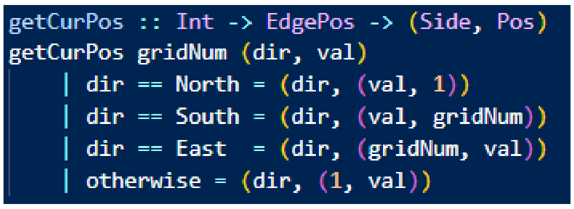
- **generateInteraction** function takes four arguments which are the list of position of atoms, grid number, list of EdgePos which is [(Side,Int)] and an initial empty list.
- The initial empty list will be used to store the computed interactions. Once the function reaches the base case which is when the list of EdgePos is empty, the list of computed interactions will be returned.
- The function **pathfinder** is used to obtain the interaction for each Edgepos.
- In the 'where' clause, curPos is the current position of the ray on the grid which will be calculated using the **getCurPos** function whereas dftPos is the list of deflection points on the grid which will be calculated using **generateDftPos** function.

Output:



```
ghci> generateInteraction [(2,3),(7,3),(4,6),(7,8)] 8 [(North,1),(North,2),(North,3),(North,4),(North,5),(North,6),(North,7),(North,8),(East,1),(East,2),(East,3),(East,4),(East,5),(East,
6),(East,7),(East,8),(South,1),(South,2),(South,3),(South,4),(South,5),(South,6),(South,7),(South,8),(West,1),(West,2),(West,3),(West,4),(West,5),(West,6),(West,7),(West,8)] []
[((West,8),Absorb),((West,7),Path (South,3)),((West,6),Absorb),((West,5),Path (South,5)),((West,4),Path (South,1)),((West,3),Absorb),((West,2),Path (North,1)),((West,1),Path (East,1)),((
South,8),Reflect),((South,7),Absorb),((South,6),Reflect),((South,5),Path (West,5)),((South,4),Absorb),((South,3),Path (West,7)),((South,2),Absorb),((South,1),Path (West,4)),((East,8),Abs
orb),((East,7),Path (East,4)),((East,6),Absorb),((East,5),Path (North,5)),((East,4),Path (East,7)),((East,3),Absorb),((East,2),Path (North,8)),((East,1),Path (West,1)),((North,8),Path (E
ast,2)),((North,7),Absorb),((North,6),Path (North,3)),((North,5),Path (East,5)),((North,4),Absorb),((North,3),Path (North,6)),((North,2),Absorb),((North,1),Path (West,2))]
```

4) getCurPos

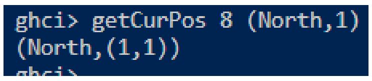


```haskell
getCurPos :: Int -> EdgePos -> (Side, Pos)
getCurPos gridNum (dir, val)
    | dir == North = (dir, (val, 1))
    | dir == South = (dir, (val, gridNum))
    | dir == East  = (dir, (gridNum, val))
    | otherwise = (dir, (1, val))
```

Explanation:

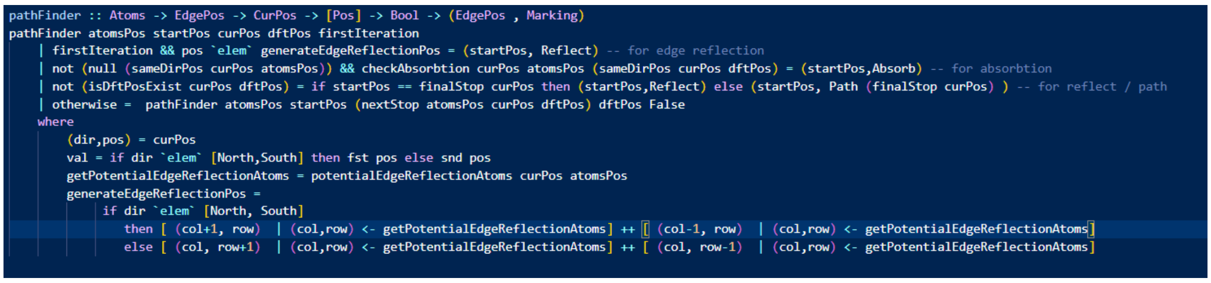
- **getCurPos** is used to obtain the starting position in the grid even before the computation starts.

Output:



```
ghci> getCurPos 8 (North,1)
(North,(1,1))
```

5) pathFinder



```haskell
pathFinder :: Atoms -> EdgePos -> CurPos -> [Pos] -> Bool -> (EdgePos , Marking)
pathFinder atomsPos startPos curPos dftPos firstIteration
    | firstIteration && pos `elem` generateEdgeReflectionPos = (startPos, Reflect) -- for edge reflection
    | not (null (sameDirPos curPos atomsPos)) && checkAbsorbtion curPos atomsPos (sameDirPos curPos dftPos) = (startPos,Absorb) -- for absorbtion
    | not (isDftPosExist curPos dftPos) = if startPos == finalStop curPos then (startPos,Reflect) else (startPos, Path (finalStop curPos) ) -- for reflect / path
    | otherwise =  pathFinder atomsPos startPos (nextStop atomsPos curPos dftPos) dftPos False
    where
       (dir,pos) = curPos
       val = if dir `elem` [North,South] then fst pos else snd pos
       getPotentialEdgeReflectionAtoms = potentialEdgeReflectionAtoms curPos atomsPos
       generateEdgeReflectionPos =
           if dir `elem` [North, South]
              then [ (col+1, row)  | (col,row) <- getPotentialEdgeReflectionAtoms] ++ [ (col-1, row)  | (col,row) <- getPotentialEdgeReflectionAtoms]
              else [ (col, row+1)  | (col,row) <- getPotentialEdgeReflectionAtoms] ++ [ (col, row-1)  | (col,row) <- getPotentialEdgeReflectionAtoms]
```

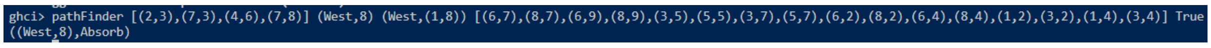
Explanation:

- **pathFinder** function takes five arguments and returns the interaction for each EdgePos. atomsPos is the list of position of atoms on the grid. startPos is the starting position of the ray. curPos is the current position of the ray. dftPos is the list of deflection points on the grid. firstIteration is a boolean value which is used to check if the **pathFinder** is currently at the first iteration. This boolean value is helpful for identifying edge reflection as this scenario can only happen at the first iteration.
- *First guard:* Used to check for edge reflection. If the firstIteration is True and the coordinate of current position is among the generated edge reflection positions, it returns a tuple with startPos and Reflect. Due to lazy evaluation, expressions are not evaluated until their values are actually needed. If firstIteration is False in the pathfinder function, the values of generateEdgeReflectionPos and getPotentialEdgeReflectionAtoms will not be calculated until they are used in the subsequent code.
- *Second guard:* If there are atoms in the same direction as the current position of the ray, and the absorption condition is met, it returns a tuple with startPos and Absorb.
- *Third guard:* If deflection points do not exist in the same direction as the current position of the ray, this indicates that the ray can directly reach the final position. It checks if the final position is the same as the starting position. If so, it returns a tuple

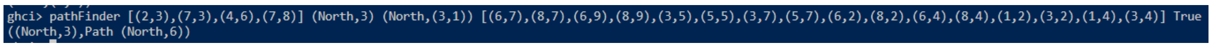
with startPos and Reflect; otherwise, it returns a tuple with startPos and a Path to the final stopping position. **finalStop** function is used to obtain the final position.

- *Fourth guard (Otherwise):* If none of the above conditions are met, it recursively calls pathFinder with the updated current position obtained from **nextStop** and sets firstIteration to False.

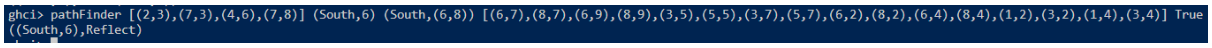
Output:



```
ghci> pathFinder [(2,3),(7,3),(4,6),(7,8)] (West,8) (West,(1,8)) [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)] True
((West,8),Absorb)
```



```
ghci> pathFinder [(2,3),(7,3),(4,6),(7,8)] (North,3) (North,(3,1)) [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)] True
((North,3),Path (North,6))
```



```
ghci> pathFinder [(2,3),(7,3),(4,6),(7,8)] (South,6) (South,(6,8)) [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)] True
((South,6),Reflect)
```

Visual explanation of how the guards (except otherwise) in **pathFinder** works:

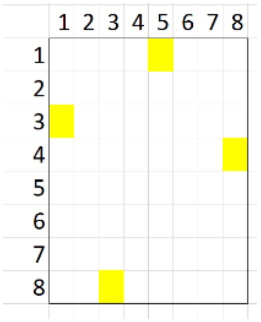


Figure 6

First guard (Edge Reflections):

getPotentialEdgeReflectionAtoms:

If the ray is from North or South , we only return atoms with same row as the starting position of ray. For instance, if startPos is (North,4) which the curPos is (North,(4,1)) , we return [(5,1)]. If the ray is from East or West , we only return atoms with the same column as the starting position of the ray. For instance, if startPos is (East,5) which the curPos is (East,(8,5)) , we return [(8,4)].

generateEdgeReflectionPos:

Let's consider (East,(8,5)) as the curPos and the returned potential atoms is [(8,4)]. In this case, the generated edge reflection positions would be [(8,3),(8,5)].

In first guard, for the same ray above, it will be True since, (8,5) is an element of [(8,3),(8,5)].

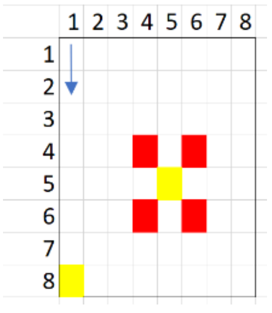


Figure 7

Second guard (Absorption):

not (null (sameDirPos curPos atomsPos)):

Check if there's any atoms in the same direction as the current position of the ray. Only if this returns true, the condition below will be evaluated.

checkAbsorbtion curPos atomsPos (sameDirPos curPos dftPos):

checkAbsorbtion functions evaluates all possible conditions for absorption to occur. If this function returns true, then the outcome will be considered as Absorb.

For example, in Figure 7, the ray is from (North,1) which the curPos would be (North,(1,1)). In this case, we can return the output as ((North,1), Absorb) in just one **pathFinder** function call because we are not going to evaluate each position such as (1,2), (1,3) … (1,7).

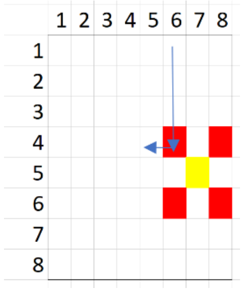


Figure 8

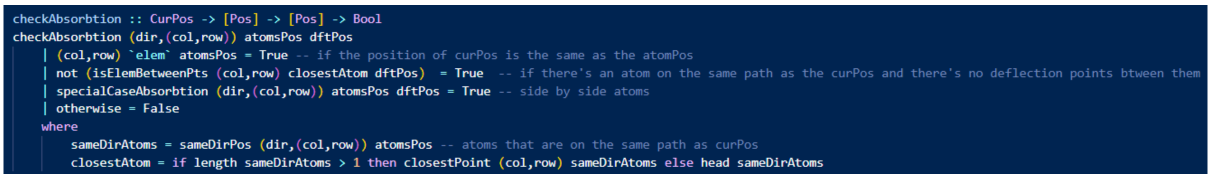
Third guard (Path or Reflect):

not (isDftPosExist curPos dftPos):

This checks if there's any deflection points in the same direction as the current position of the ray. If there's none, then we can return the final output as either reflect or in terms of path depending on the conditions.

For instance, in Figure 8, after the ray being deflected, the coordinate of the ray would be at (5,4). The algorithm will check if there's any deflection points from (1,4) to (5,4). Since there's none, it will directly return the output as ((North,6), Path (West,4)). In this case, we saved several steps since we did not evaluate position (1,4), (2,4), (3,4) and (4,4). The algorithm is able to find the final output in just three pathFinder function call.

6) checkAbsorbtion



```
checkAbsorbtion :: CurPos -> [Pos] -> [Pos] -> Bool
checkAbsorbtion (dir,(col,row)) atomsPos dftPos
    | (col,row) `elem` atomsPos = True -- if the position of curPos is the same as the atomPos
    | not (isElemBetweenPts (col,row) closestAtom dftPos)  = True  -- if there's an atom on the same path as the curPos and there's no deflection points btween them
    | specialCaseAbsorbtion (dir,(col,row)) atomsPos dftPos = True -- side by side atoms
    | otherwise = False
   where
      sameDirAtoms = sameDirPos (dir,(col,row)) atomsPos -- atoms that are on the same path as curPos
      closestAtom = if length sameDirAtoms > 1 then closestPoint (col,row) sameDirAtoms else head sameDirAtoms
```

Explanation:

- **checkAbsorbtion** function takes 3 arguments which is the current position of the ray, list of atoms' position and list of deflection points and returns True if the ray will be absorbed, else False.
- *First guard*: If the current position of the ray is also the same as the position of any of the atom in the atoms' list, then it will be absorbed.
- *Second guard*: If there's no deflection points between the closest atom and the current position of ray, then it will be absorbed.
- *Third guard*: If the current position of the ray belongs to the special case absorption (side by side atoms), it will be absorbed.
- *Fourth guard (Otherwise):* It will not be absorbed.

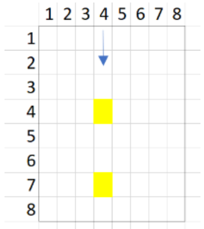


Figure 9

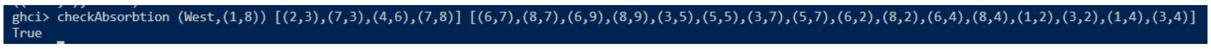
sameDirAtoms:

This return atoms with same direction as the current position of the ray. For instance, according to Figure 6, it will return [(4,4),(4,7)]. It uses **sameDirPos** function.

closestAtom:

If there's more than one atom, then the closest atom will be returned. Based on Figure 6, it will return (4,4)

Output:



```
ghci> checkAbsorbtion (West,(1,8)) [(2,3),(7,3),(4,6),(7,8)] [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)]
True
```

Visual explanation of how the guard (except otherwise) in **checkAbsorbtion** works:

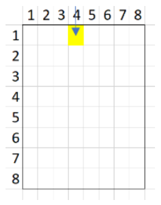


Figure 13

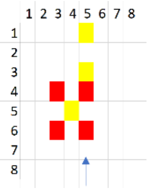


Figure 12

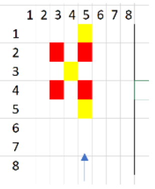


Figure 11

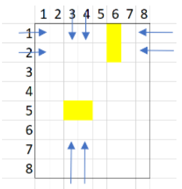


Figure 10

First guard:

This is True since both positions are the same.

Second guard:

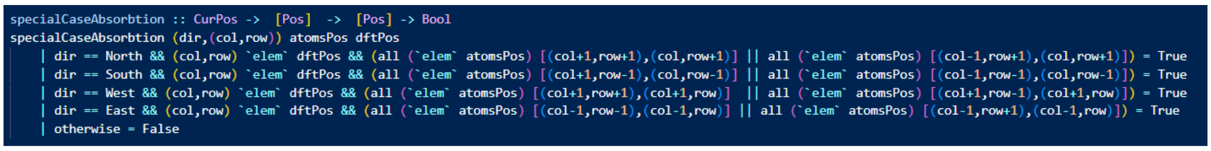
For Figure 11, it would True.

For Figure 12, it would be False since there's deflection point between the position of ray and the closest atom.

Third guard:

If the atoms are side by side, it will be absorbed. All blue rays in Figure 10 will be absorbed.

7) specialCaseAbsorbtion

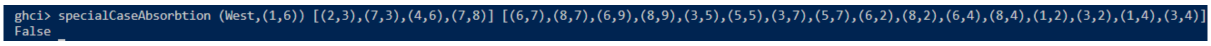


```
specialCaseAbsorbtion :: CurPos ->  [Pos]  ->  [Pos] -> Bool
specialCaseAbsorbtion (dir,(col,row)) atomsPos dftPos
    | dir == North && (col,row) `elem` dftPos && (all (`elem` atomsPos) [(col+1,row+1),(col,row+1)] || all (`elem` atomsPos) [(col-1,row+1),(col,row+1)]) = True
    | dir == South && (col,row) `elem` dftPos && (all (`elem` atomsPos) [(col+1,row-1),(col,row-1)] || all (`elem` atomsPos) [(col-1,row-1),(col,row-1)]) = True
    | dir == West && (col,row) `elem` dftPos && (all (`elem` atomsPos) [(col+1,row+1),(col+1,row)]  || all (`elem` atomsPos) [(col+1,row-1),(col+1,row)]) = True
    | dir == East && (col,row) `elem` dftPos && (all (`elem` atomsPos) [(col-1,row-1),(col-1,row)] || all (`elem` atomsPos) [(col-1,row+1),(col-1,row)]) = True
    | otherwise = False
```

Explanation:

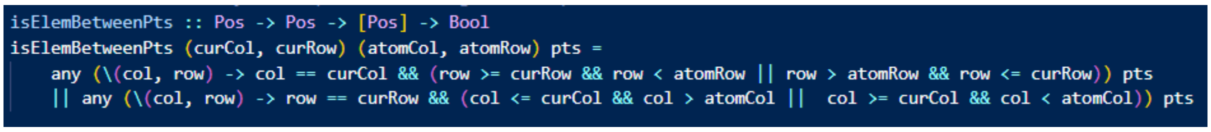
- **specialCaseAbsorbtion** is used to check for absorption where there's side by side atoms. This function takes three arguments such as the current position of the ray, the list of position of atoms and list of position of deflection points and returns True if condition is met, otherwise False. For visualization, please refer to Figure 10.

Output:



```
ghci> specialCaseAbsorbtion (West,(1,6)) [(2,3),(7,3),(4,6),(7,8)] [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)]
False
```

8) isElemBetweenPts

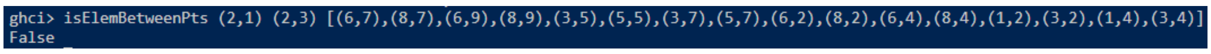


```
isElemBetweenPts :: Pos -> Pos -> [Pos] -> Bool
isElemBetweenPts (curCol, curRow) (atomCol, atomRow) pts =
    any (\(col, row) -> col == curCol && (row >= curRow && row < atomRow || row > atomRow && row <= curRow)) pts
    || any (\(col, row) -> row == curRow && (col <= curCol && col > atomCol ||  col >= curCol && col < atomCol)) pts
```

Explanation:

- This function takes three arguments which is the position of ray and the position of the atom and a list of deflection points. It will return True if there's any deflection points between the current position of ray and atom, otherwise False. For visualization, please refer to Figure 11 and 12.

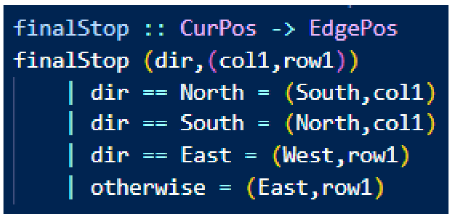
Output:



```
ghci> isElemBetweenPts (2,1) (2,3) [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)]
False _
```

As we can see in Figure 5, there's no deflection points between atom (2,3) and the initial position of ray which is at (2,1).

9) finalStop

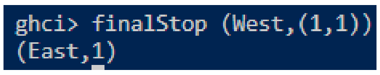


```
finalStop :: CurPos -> EdgePos
finalStop (dir,(col1,row1))
    | dir == North = (South,col1)
    | dir == South = (North,col1)
    | dir == East = (West,row1)
    | otherwise = (East,row1)
```

Explanation:

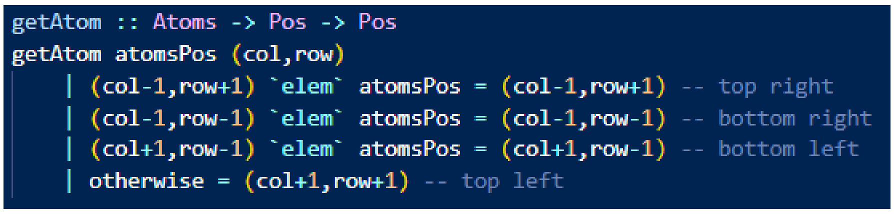
- **finalStop** function takes in one argument which is the current position of the ray and returns the final path.

Output:



```
ghci> finalStop (West,(1,1))
(East,1)
```

10) getAtom

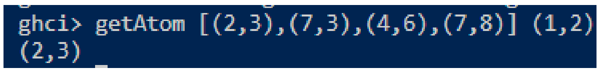


```
getAtom :: Atoms -> Pos -> Pos
getAtom atomsPos (col,row)
    | (col-1,row+1) `elem` atomsPos = (col-1,row+1) -- top right
    | (col-1,row-1) `elem` atomsPos = (col-1,row-1) -- bottom right
    | (col+1,row-1) `elem` atomsPos = (col+1,row-1) -- bottom left
    | otherwise = (col+1,row+1) -- top left
```

Explanation:

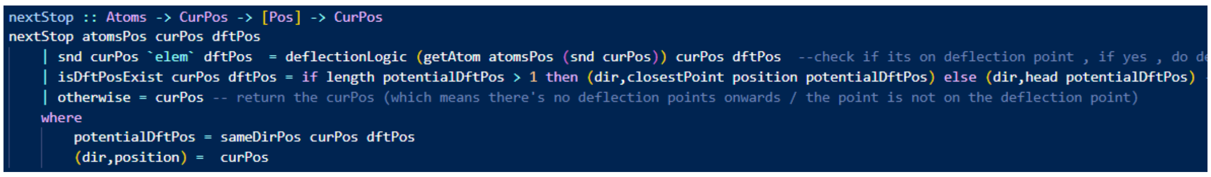
- **getAtom** function takes a coordinate as an argument and returns the position of an atom.

Output:



```
ghci> getAtom [(2,3),(7,3),(4,6),(7,8)] (1,2)
(2,3) _
```

11) nextStop

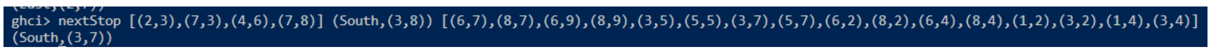


```
nextStop :: Atoms -> CurPos -> [Pos] -> CurPos
nextStop atomsPos curPos dftPos
    | snd curPos `elem` dftPos  = deflectionLogic (getAtom atomsPos (snd curPos)) curPos dftPos  --check if its on deflection point , if yes , do de
    | isDftPosExist curPos dftPos = if length potentialDftPos > 1 then (dir,closestPoint position potentialDftPos) else (dir,head potentialDftPos)
    | otherwise = curPos -- return the curPos (which means there's no deflection points onwards / the point is not on the deflection point)
    where
        potentialDftPos = sameDirPos curPos dftPos
        (dir,position) =  curPos
```

Explanation:

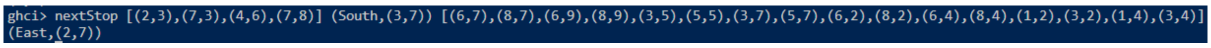
- **nextStop** function takes three arguments which are list of position of atoms, current position of ray and list of deflection points.
- *First guard*: If the current position of the ray is on a deflection point, deflection logic will be applied, and new position of the ray will be returned.
- *Second guard*: If the current position is not on a deflection point but there are deflection points in the same direction as the current position, it calculates the closest deflection point using **closestPoint** function and move the ray to the deflection point.
- *Third guard (otherwise):* If the current position is not on a deflection point, and there are no more deflection points in the same direction, it simply returns the current position without any changes.

Output:



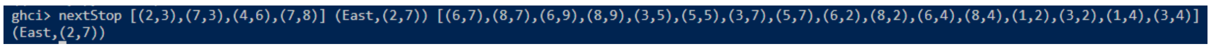
```
ghci> nextStop [(2,3),(7,3),(4,6),(7,8)] (South,(3,8)) [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)]
(South,(3,7))
```

If the current position of the ray is not on deflection point but there are deflection points as in the same direction as the current position of the ray, the ray will be moved to the closest deflection point, which in this case is (3,7).



```
ghci> nextStop [(2,3),(7,3),(4,6),(7,8)] (South,(3,7)) [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)]
(East,(2,7))
```

Since the ray's current position is at (South, (3,7)) which is also a deflection point, the first guard will be executed. Once it gets deflected through **deflectionLogic** function, the new current position of the ray would be (East, (2,7)) as it will be assumed as if it's coming from East.



```
ghci> nextStop [(2,3),(7,3),(4,6),(7,8)] (East,(2,7)) [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)]
(East,(2,7))
```

Since there's no deflection points as in the same direction as the current position of the ray, it simply returns the current position without any changes.

All of this will happen in a series of recursive calls.

Visual explanation of how the guard (except otherwise) in **nextStop** works:

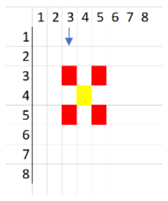


Figure 15

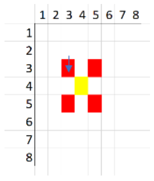


Figure 16

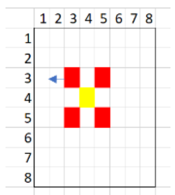


Figure 14

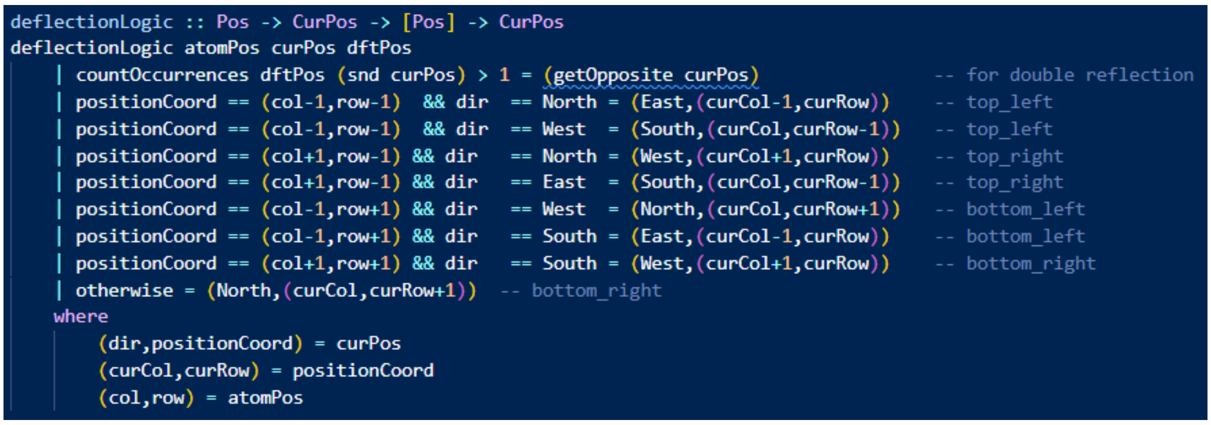
Guard 2:

From Figure 15, current position of ray is (North,(3,1)), the ray will be moved to the closest deflection point as in Figure 16 where the current position would be (North,(3,3)).

Guard 1:

From Figure 16, since the ray is on deflection point, deflection logic will be applied on it and new position will be returned as in Figure 14 which is (East,(2,3))

12) deflectionLogic

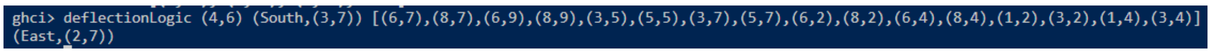


```haskell
deflectionLogic :: Pos -> CurPos -> [Pos] -> CurPos
deflectionLogic atomPos curPos dftPos
    | countOccurrences dftPos (snd curPos) > 1 = (getOpposite curPos)          -- for double reflection
    | positionCoord == (col-1,row-1)  && dir  == North = (East,(curCol-1,curRow))    -- top_left
    | positionCoord == (col-1,row-1)  && dir  == West  = (South,(curCol,curRow-1))   -- top_left
    | positionCoord == (col+1,row-1) && dir   == North = (West,(curCol+1,curRow))    -- top_right
    | positionCoord == (col+1,row-1) && dir   == East  = (South,(curCol,curRow-1))   -- top_right
    | positionCoord == (col-1,row+1) && dir   == West  = (North,(curCol,curRow+1))   -- bottom_left
    | positionCoord == (col-1,row+1) && dir   == South = (East,(curCol-1,curRow))    -- bottom_left
    | positionCoord == (col+1,row+1) && dir   == South = (West,(curCol+1,curRow))    -- bottom_right
    | otherwise = (North,(curCol,curRow+1))  -- bottom_right
   where
      (dir,positionCoord) = curPos
      (curCol,curRow) = positionCoord
      (col,row) = atomPos
```

Explanation:

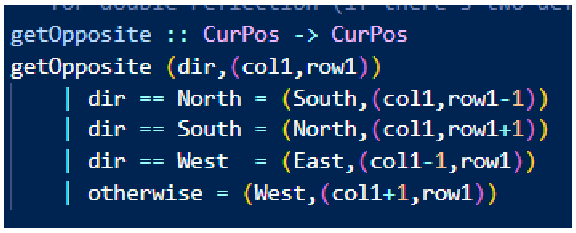
- **deflectionLogic** takes three arguments which is an atom point, the current position of the ray and a list of deflection points and returns the deflected position of the ray.
- *First guard*: If the position of ray is on more than one deflection points, it will be considered as double deflection and the opposite of current position of ray will be returned as an output.
- *Remaining guards*: Performs all kinds of deflection logic based on the position and direction.

Output:



```
ghci> deflectionLogic (4,6) (South,(3,7)) [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)]
(East,(2,7))
```

13) getOpposite



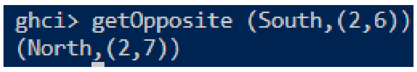
```
getOpposite :: CurPos -> CurPos
getOpposite (dir,(col1,row1))
    | dir == North = (South,(col1,row1-1))
    | dir == South = (North,(col1,row1+1))
    | dir == West  = (East,(col1-1,row1))
    | otherwise = (West,(col1+1,row1))
```

Explanation:

- This function takes the current position of the ray and return the opposite for the provided ray.

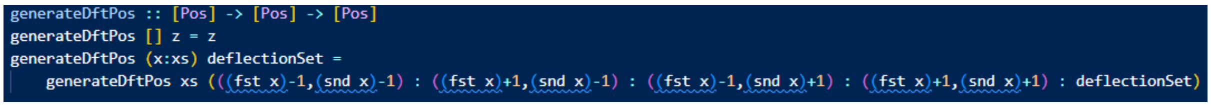
Output:

This case doesn't apply for the grid in Figure 5. This is just an example of how this function works.



```
ghci> getOpposite (South,(2,6))
(North,(2,7))
```

14) generateDftPos

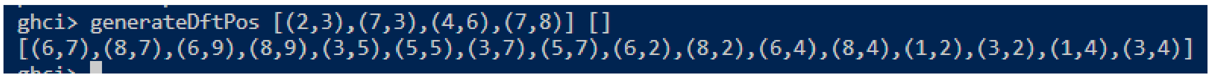


```
generateDftPos :: [Pos] -> [Pos] -> [Pos]
generateDftPos [] z = z
generateDftPos (x:xs) deflectionSet =
    generateDftPos xs (((fst x)-1,(snd x)-1) : ((fst x)+1,(snd x)-1) : ((fst x)-1,(snd x)+1) : ((fst x)+1,(snd x)+1) : deflectionSet)
```

Explanation:

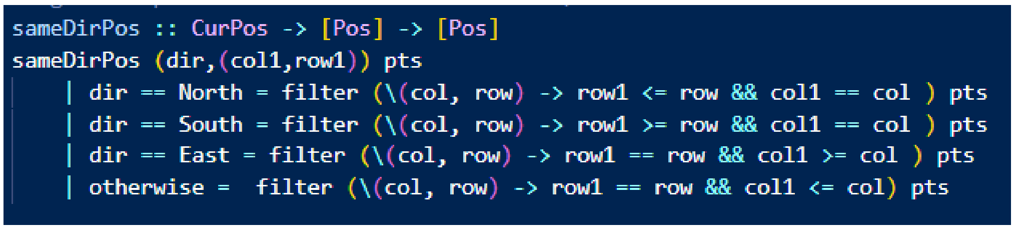
- This function takes two arguments which are the list of atoms and an initial empty list and returns a list of deflection points once the base case is reached. For each atom, the four sides of deflection points will be calculated and inserted into the initial empty list. Once the list of atoms becomes empty, the deflection set will be returned.

Output:



```
ghci> generateDftPos [(2,3),(7,3),(4,6),(7,8)] []
[(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)]
```

15) sameDirPos



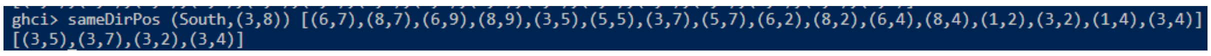
```
sameDirPos :: CurPos -> [Pos] -> [Pos]
sameDirPos (dir,(col1,row1)) pts
    | dir == North = filter (\(col, row) -> row1 <= row && col1 == col ) pts
    | dir == South = filter (\(col, row) -> row1 >= row && col1 == col ) pts
    | dir == East = filter (\(col, row) -> row1 == row && col1 >= col ) pts
    | otherwise =  filter (\(col, row) -> row1 == row && col1 <= col) pts
```

Explanation:

- **sameDirPos** function takes two arguments which is the current position of the ray and a list of points and finally returns a new list of points which are filtered from the given list. The new list of points is on the same direction as the ray. Left behind points will not be considered.

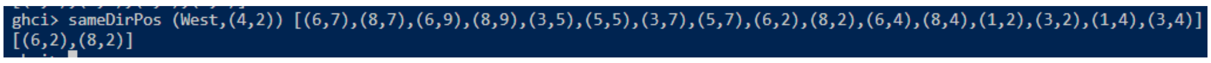
Output:

In this case, it checks if there's any deflection points as in the same direction as the current position of the ray. This is because the list of points that we passed as argument is list of deflection points. In some cases, we also use list of atoms to check if there's any atoms in the same direction as well.



```
ghci> sameDirPos (South,(3,8)) [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)]
[(3,5),(3,7),(3,2),(3,4)]
```

For the scenario below (Figure 17), assuming out curPos is (West,(4,2)) we can notice that the algorithm does not consider the left behind points. In this case, the left behind points would be (3,2) and (1,2).



```
ghci> sameDirPos (West,(4,2)) [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)]
[(6,2),(8,2)]
```

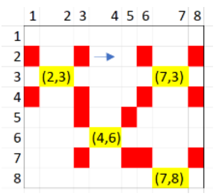
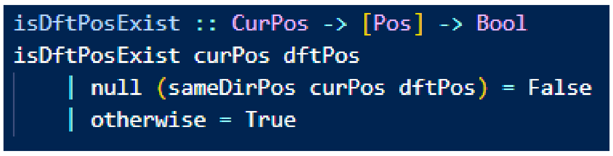


Figure 17

16) isDftPosExist

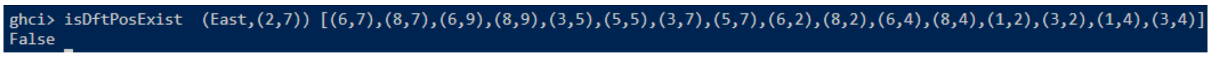


```
isDftPosExist :: CurPos -> [Pos] -> Bool
isDftPosExist curPos dftPos
    | null (sameDirPos curPos dftPos) = False
    | otherwise = True
```

Explanation:

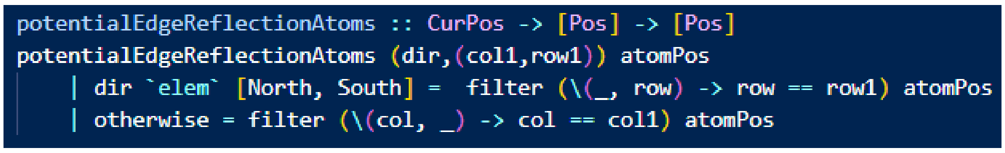
- This function takes two arguments which is the current position of the ray and a list of deflection points and returns True if there's any deflection point on the same direction as the ray, otherwise False.

Output:



```
ghci> isDftPosExist  (East,(2,7)) [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)]
False
```

17) potentialEdgeReflectionAtoms

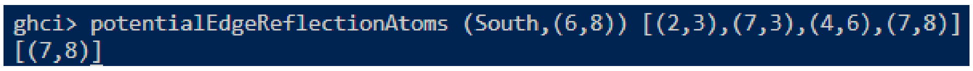


```haskell
potentialEdgeReflectionAtoms :: CurPos -> [Pos] -> [Pos]
potentialEdgeReflectionAtoms (dir,(col1,row1)) atomPos
    | dir `elem` [North, South] =  filter (\(_, row) -> row == row1) atomPos
    | otherwise = filter (\(col, _) -> col == col1) atomPos
```

Explanation:

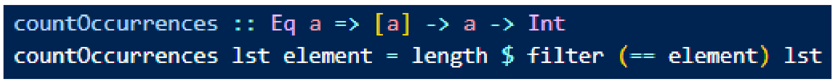
- **potentialEdgeReflectionAtoms** functions takes two arguments which is the current position of the ray and list of atoms. It returns all the potential atoms for edge reflection. It has the same explanation as “getPotentialEdgeReflectionAtoms” in Figure 6.

Output:



```
ghci> potentialEdgeReflectionAtoms (South,(6,8)) [(2,3),(7,3),(4,6),(7,8)]
[(7,8)]
```

18) countOccurences



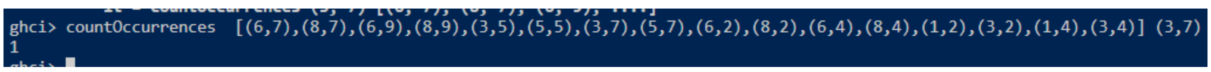
```haskell
countOccurrences :: Eq a => [a] -> a -> Int
countOccurrences lst element = length $ filter (== element) lst
```

Explanation:

- This function takes two arguments which is a list of points and a target point and returns the number of occurrences of the target point in the list. It is used to check for double deflection cases.

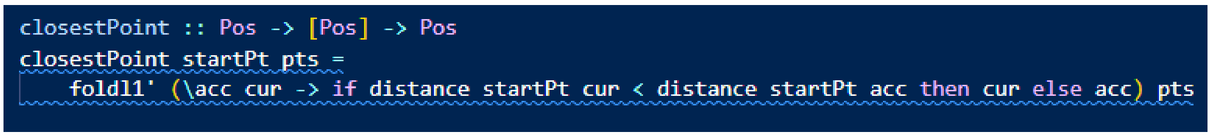
Output:

Since there’s only one point in the list, which is similar to the target point, 1 is returned. This is an indication that it is not a double deflection.



```
ghci> countOccurrences  [(6,7),(8,7),(6,9),(8,9),(3,5),(5,5),(3,7),(5,7),(6,2),(8,2),(6,4),(8,4),(1,2),(3,2),(1,4),(3,4)] (3,7)
1
```

19) closestPoint



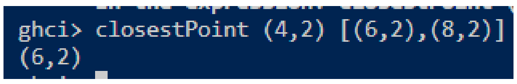
```haskell
closestPoint :: Pos -> [Pos] -> Pos
closestPoint startPt pts =
    foldl1' (\acc cur -> if distance startPt cur < distance startPt acc then cur else acc) pts
```

Explanation:

- **closestPoint** function takes two arguments which is a target point and a list of points and returns a point from the list that is the closest to the target point.

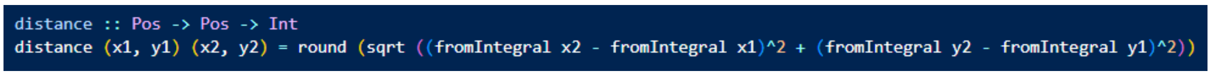
Output:

If the current position of the ray is at (West, (4,2)), then the closest deflection point is (6,2). Please refer to Figure 17 for visualization.



```
ghci> closestPoint (4,2) [(6,2),(8,2)]
(6,2)
```

20) distance

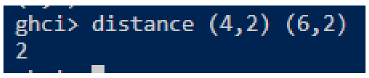


```haskell
distance :: Pos -> Pos -> Int
distance (x1, y1) (x2, y2) = round (sqrt ((fromIntegral x2 - fromIntegral x1)^2 + (fromIntegral y2 - fromIntegral y1)^2))
```

Explanation:

- This function takes two points and calculates the distance between them.

Output:



```
ghci> distance (4,2) (6,2)
2
```

Challenge 2

The approach that has been taken for this challenge is by generating all possible combinations of N atoms from a generated list of points. Grid size which can be obtained by analysing interactions that passed as argument will be used to generate list of points. Then, iterations occurs where each possible combination and each interaction in interactions will be passed to the **pathFinder in Question 1**. For a specific combination, if each interaction in interactions is the same as the interaction generated by **pathFinder**, then the combination of atoms is considered valid. The output will be returned once all the possible combinations has been evaluated. For better understanding about the efficiency of this algorithm, let's consider some examples below.

Let's use the same interactions as given in the coursework, which is:

```
[((North,1),Path (West,2)),((North,2),Absorb), ((North,3),Path (North,6)),((North,4),Absorb),
((North,5),Path (East,5)),((North,6),Path (North,3)), ((North,7),Absorb),((North,8),Path (East,2)),
((East,1),Path (West,1)),((East,2),Path (North,8)), ((East,3),Absorb),((East,4),Path (East,7)),
((East,5),Path (North,5)),((East,6),Absorb), ((East,7),Path (East,4)),((East,8),Absorb),
((South,1),Path (West,4)),((South,2),Absorb), ((South,3),Path (West,7)),((South,4),Absorb),
((South,5),Path (West,5)),((South,6),Reflect), ((South,7),Absorb),((South,8),Reflect),
((West,1),Path (East,1)),((West,2),Path (North,1)), ((West,3),Absorb),((West,4),Path (South,1)),
((West,5),Path (South,5)),((West,6),Absorb), ((West,7),Path (South,3)),((West,8),Absorb)]
```

Figure 18

1) **pathFinder** is used instead of **calcBBInteractions** for the evaluation. This is because, **calcBBInteractions** will return a fully evaluated interactions whereas **pathFinder** will return an interaction for each call. If the interactions that has been passed as an argument is a long list and the algorithm found out that the second interaction in the list itself is invalid for a specific combination of atoms, the algorithm can proceed with the next combination instead of evaluating the remaining interaction in the given list. [managed by **generatePossibleAtomsComb**, **isInteractionValid** and **isValidAtomComb** functions]

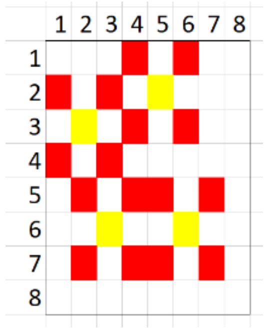


Figure 19

For example, let's say N=4. The generated possible combinations based on Figure 19 would be […. , [(2,3),(3,6),(5,2),(6,6)] , [(2,3),(3,6),(5,2),(6,7)] ….] and the current evaluation is at [(2,3),(3,6),(5,2),(6,6)]. Same interactions as Figure 18 will be used for this example.

For each combination, the algorithm starts by evaluating each interaction in interactions.

((North,1),Path (West,2)) = True

((North,2),Absorb) = True

((North,3),Path (North,6)) = False

At third interaction, the algorithm identifies that the current combination is invalid. Therefore, instead of continuing with the evaluation of fourth interaction and so on, the algorithm will stop the evaluation and proceed with the next combination which is [(2,3),(3,6),(5,2),(6,7)]. We saved numerous steps of evaluation by using **pathFinder** instead of **calcBBInteractions**.

2) Invalid positions of atom will be considered before generating the possible combinations of N atoms. By taking this approach, the number of possible combinations to be evaluated can be largely reduced. [managed by **getInvalidAtomPos** and **atomCombination** functions]

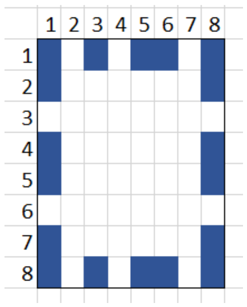


Figure 20

Invalid position of atoms refers to positions where atoms cannot exist. A pattern that can be identified by analyzing the interactions is that, if the final destination of the ray is not absorbed, that means no atom can exist at the start position of the ray. For an example, let's take interaction ((North,1), Path (West,2)). Since the final destination is not Absorb, that means, no atom can exist at (1,1) which is also the start position of the ray in the example.

The blue region in Figure 20 is the invalid positions of atoms which was obtained by analysing interactions in Figure 18. By ignoring the blue regions during the generation of possible combinations, the number of combinations to be evaluated can be largely reduced.

Calculation for the reduction in number of combinations (based on Figure 20):

Possible combinations that can be generated for (grid of size = 8) and (N = 4) = 64 C 4 = 635,376 combinations

Number of points that will not be evaluated (blue region) = 18

Number of points that will be evaluated = 64 – 18 = 46

New possible combinations that can be generated = 46 C 4 = 163,185 combinations

Total reduction in combinations = 635,376 - 163,185 = 472,191 combinations.

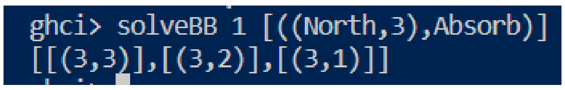
In this case, we have reduced the possible combinations to be evaluated from 635,376 to 163,185 which is a reduction of 472,191 combinations. By only evaluating 163,185 combinations, the output can be obtained even faster and more efficiently.

3) This algorithm also works if the number of interactions entered is incomplete. For instance, let's take an example.

Example: solveBB 1 [((North,3), Absorb)]

- In this case, by analysing the interactions given, the algorithm will be able to identify that the grid size will be 3.
- Then, by going through the same operation as explained, the final output will be returned.

Sample output:

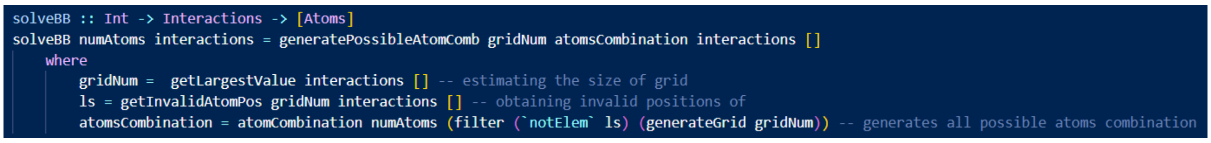


```
ghci> solveBB 1 [((North,3),Absorb)]
[[(3,3)],[(3,2)],[(3,1)]]
```

Explanations for the functions available in Challenge 2.

All the outputs for the below explanations are based on the grid in Figure 5.

1) solveBB

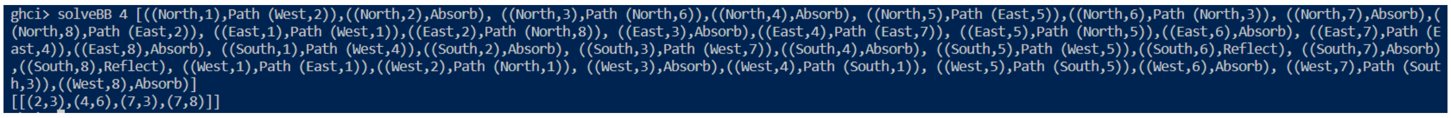


```haskell
solveBB :: Int -> Interactions -> [Atoms]
solveBB numAtoms interactions = generatePossibleAtomComb gridNum atomsCombination interactions []
    where
        gridNum =  getLargestValue interactions [] -- estimating the size of grid
        ls = getInvalidAtomPos gridNum interactions [] -- obtaining invalid positions of
        atomsCombination = atomCombination numAtoms (filter (`notElem` ls) (generateGrid gridNum)) -- generates all possible atoms combination
```

Explanation:

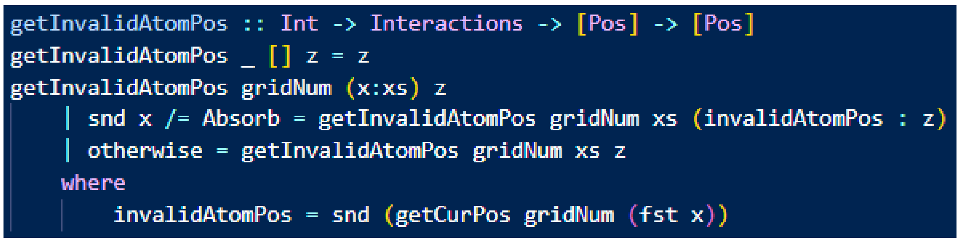
- **solveBB** function takes two arguments which is the number of atoms and interactions and returns a list of possible combinations of atoms that satisfies the interactions. In coursework sheet, Atoms was used as the return type whereas in this code, [Atoms] was used. This is because by returning a list containing the combinations of possible atoms such as [[combination 1] , [combination 2]] is more readable than [combination1,combination2].
- In the where clause, gridNum stores the size of the grid which is obtained from the computation in **getLargestValue** function.
- ls stores the list of invalid positions of atoms which is obtained from the computation in **getInvalidAtomPos** function.
- atomsCombination stores all possible combination of atoms based on the specified number of atoms. Before **atomCombination** function is computed, filtration for removing invalid position of atoms will occur so that only valid atom positions will be considered for the combinations.

Output:



```
ghci> solveBB 4 [((North,1),Path (West,2)),((North,2),Absorb), ((North,3),Path (North,6)),((North,4),Absorb), ((North,5),Path (East,5)),((North,6),Path (North,3)), ((North,7),Absorb),(
(North,8),Path (East,2)), ((East,1),Path (West,1)),((East,2),Path (North,8)), ((East,3),Absorb),((East,4),Path (East,7)), ((East,5),Path (North,5)),((East,6),Absorb), ((East,7),Path (E
ast,4)),((East,8),Absorb), ((South,1),Path (West,4)),((South,2),Absorb), ((South,3),Path (West,7)),((South,4),Absorb), ((South,5),Path (West,5)),((South,6),Reflect), ((South,7),Absorb)
,((South,8),Reflect), ((West,1),Path (East,1)),((West,2),Path (North,1)), ((West,3),Absorb),((West,4),Path (South,1)), ((West,5),Path (South,5)),((West,6),Absorb), ((West,7),Path (Sout
h,3)),((West,8),Absorb)]
[[(2,3),(4,6),(7,3),(7,8)]]
```

2) getInvalidAtomPos



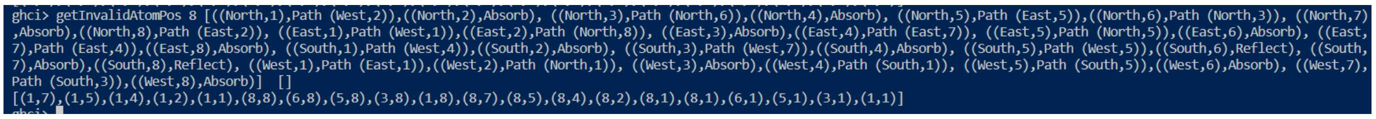
```haskell
getInvalidAtomPos :: Int -> Interactions -> [Pos] -> [Pos]
getInvalidAtomPos _ [] z = z
getInvalidAtomPos gridNum (x:xs) z
    | snd x /= Absorb = getInvalidAtomPos gridNum xs (invalidAtomPos : z)
    | otherwise = getInvalidAtomPos gridNum xs z
    where
      invalidAtomPos = snd (getCurPos gridNum (fst x))
```

Explanation:

- This function takes three arguments which is the number of grid, interactions and an initial empty list. When the base case is achieved, the function will return a list of invalid atom positions. **getCurPos from Question 1** is used to obtain the position of invalid atom.

Output:

Same Output as in Figure 20.



```
ghci> getInvalidAtomPos 8 [((North,1),Path (West,2)),((North,2),Absorb), ((North,3),Path (North,6)),((North,4),Absorb), ((North,5),Path (East,5)),((North,6),Path (North,3)), ((North,7)
,Absorb),((North,8),Path (East,2)), ((East,1),Path (West,1)),((East,2),Path (North,8)), ((East,3),Absorb),((East,4),Path (East,7)), ((East,5),Path (North,5)),((East,6),Absorb), ((East,
7),Path (East,4)),((East,8),Absorb), ((South,1),Path (West,4)),((South,2),Absorb), ((South,3),Path (West,7)),((South,4),Absorb), ((South,5),Path (West,5)),((South,6),Reflect), ((South,
7),Absorb),((South,8),Reflect), ((West,1),Path (East,1)),((West,2),Path (North,1)), ((West,3),Absorb),((West,4),Path (South,1)), ((West,5),Path (South,5)),((West,6),Absorb), ((West,7),
Path (South,3)),((West,8),Absorb)]  []
[(1,7),(1,5),(1,4),(1,2),(1,1),(8,8),(6,8),(5,8),(3,8),(1,8),(8,7),(8,5),(8,4),(8,2),(8,1),(8,1),(6,1),(5,1),(3,1),(1,1)]
```

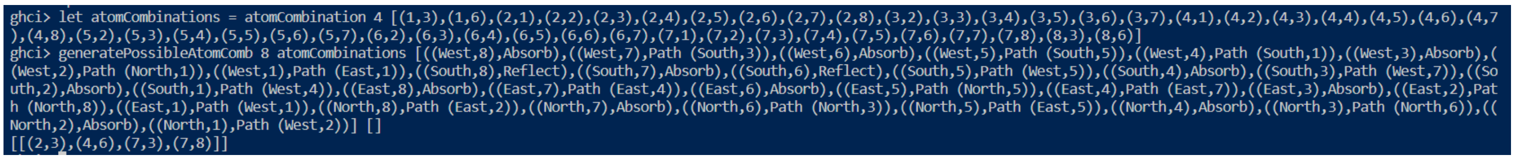
3) generatePossibleAtomComb

```
generatePossibleAtomComb ::  Int -> [Atoms] -> Interactions -> [Atoms] -> [Atoms]
generatePossibleAtomComb _ [] _ output = output
generatePossibleAtomComb gridNum (x:xs) interactions output
    | isValidAtomPos gridNum x dftPos interactions = generatePossibleAtomComb gridNum xs interactions (x : output)
    | otherwise =  generatePossibleAtomComb gridNum xs interactions output
    where
        dftPos = generateDftPos x []
```

Explanation:

- This function takes four arguments which are grid size, list of atom combinations, interactions and an initial empty list to store the potential atom combinations. Once the base case is reached which is when all the atom combinations has been evaluated, the list of potential atom combinations will be returned.
- **generateDftPos from Question 1** is used to generate deflection points.
- *First guard:* If the combination is valid, then combination will be added to the output list which store the potential atom combinations.
- *Second guard (otherwise):* Proceeds with the next combinations without any update on the output list.

Output:



```
ghci> let atomCombinations = atomCombination 4 [(1,3),(1,6),(2,1),(2,2),(2,3),(2,4),(2,5),(2,6),(2,7),(2,8),(3,2),(3,3),(3,4),(3,5),(3,6),(3,7),(4,1),(4,2),(4,3),(4,4),(4,5),(4,6),(4,7
),(4,8),(5,2),(5,3),(5,4),(5,5),(5,6),(5,7),(6,2),(6,3),(6,4),(6,5),(6,6),(6,7),(7,1),(7,2),(7,3),(7,4),(7,5),(7,6),(7,7),(7,8),(8,3),(8,6)]
ghci> generatePossibleAtomComb 8 atomCombinations [((West,8),Absorb),((West,7),Path (South,3)),((West,6),Absorb),((West,5),Path (South,5)),((West,4),Path (South,1)),((West,3),Absorb),(
(West,2),Path (North,1)),((West,1),Path (East,1)),((South,8),Reflect),((South,7),Absorb),((South,6),Reflect),((South,5),Path (West,5)),((South,4),Absorb),((South,3),Path (West,7)),((So
uth,2),Absorb),((South,1),Path (West,4)),((East,8),Absorb),((East,7),Path (East,4)),((East,6),Absorb),((East,5),Path (North,5)),((East,4),Path (East,7)),((East,3),Absorb),((East,2),Pat
h (North,8)),((East,1),Path (West,1)),((North,8),Path (East,2)),((North,7),Absorb),((North,6),Path (North,3)),((North,5),Path (East,5)),((North,4),Absorb),((North,3),Path (North,6)),((
North,2),Absorb),((North,1),Path (West,2))] []
[[(2,3),(4,6),(7,3),(7,8)]]
```

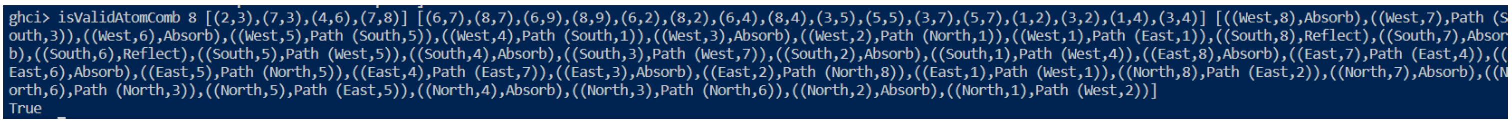
4) isValidAtomComb

```
isValidAtomComb :: Int -> Atoms -> [Pos] -> Interactions -> Bool
isValidAtomComb _ _ _ [] = True
isValidAtomComb gridNum atomsPos dftPos (x:xs)
    | isInteractionValid atomsPos x curPos dftPos = isValidAtomComb gridNum atomsPos dftPos xs
    | otherwise = False
    where
        curPos = getCurPos gridNum (fst x)
```

Explanation:

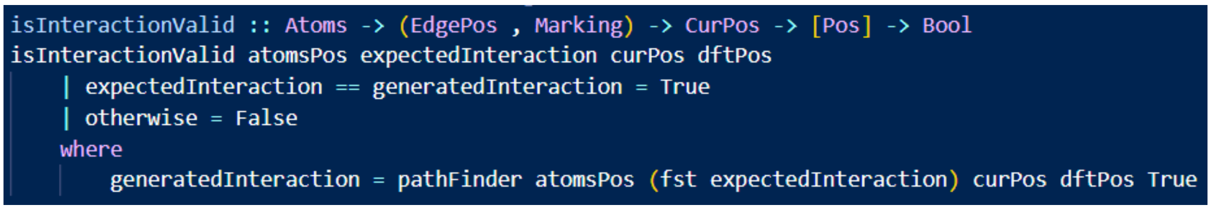
- **isValidAtomComb** function takes four arguments which is the grid size, list of position of atoms, list of deflection points and interactions. It returns a Bool indicating whether the given combination of atoms is valid.
- The base case of the recursion is when the list of interactions (Interactions) is empty. In this case, the combination is considered valid, and the function returns True.
- *First guard*: For each interaction (x), it checks whether the interaction formed by the atoms is valid using the **isInteractionValid** function. If the interaction is valid, the function proceeds to check the validity of the remaining interactions (xs).
- *Second guard (otherwise)*: If the interaction is invalid, the function immediately returns False.
- The curPos variable is defined using **getCurPos function in Question 1**.

Output:



```
ghci> isValidAtomComb 8 [(2,3),(7,3),(4,6),(7,8)] [(6,7),(8,7),(6,9),(8,9),(6,2),(8,2),(6,4),(8,4),(3,5),(5,5),(3,7),(5,7),(1,2),(3,2),(1,4),(3,4)] [((West,8),Absorb),((West,7),Path (S
outh,3)),((West,6),Absorb),((West,5),Path (South,5)),((West,4),Path (South,1)),((West,3),Absorb),((West,2),Path (North,1)),((West,1),Path (East,1)),((South,8),Reflect),((South,7),Absor
b),((South,6),Reflect),((South,5),Path (West,5)),((South,4),Absorb),((South,3),Path (West,7)),((South,2),Absorb),((South,1),Path (West,4)),((East,8),Absorb),((East,7),Path (East,4)),((
East,6),Absorb),((East,5),Path (North,5)),((East,4),Path (East,7)),((East,3),Absorb),((East,2),Path (North,8)),((East,1),Path (West,1)),((North,8),Path (East,2)),((North,7),Absorb),((N
orth,6),Path (North,3)),((North,5),Path (East,5)),((North,4),Absorb),((North,3),Path (North,6)),((North,2),Absorb),((North,1),Path (West,2))]
True
```

5) isInteractionValid

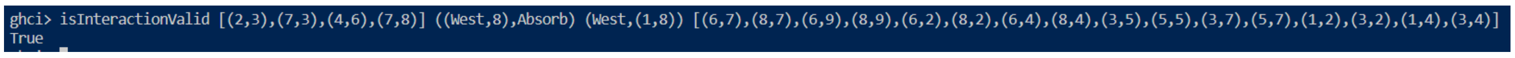


```haskell
isInteractionValid :: Atoms -> (EdgePos , Marking) -> CurPos -> [Pos] -> Bool
isInteractionValid atomsPos expectedInteraction curPos dftPos
    | expectedInteraction == generatedInteraction = True
    | otherwise = False
    where
      generatedInteraction = pathFinder atomsPos (fst expectedInteraction) curPos dftPos True
```

Explanation:

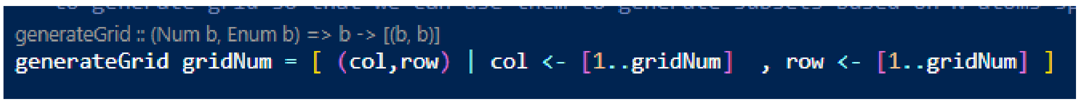
- This function takes four arguments, which is the list of atoms, an interaction, current position of atom and list of deflection points.
- *First guard:* If the generated interaction is the same as expected interaction, then the function returns True, indicating the interaction is valid.
- *Second guard (otherwise):* If they are not equal, then the function returns False.
- generatedInteraction in ‘where’ clause is evaluated using **pathFinder in Question 1**.

Output:



```
ghci> isInteractionValid [(2,3),(7,3),(4,6),(7,8)] ((West,8),Absorb) (West,(1,8)) [(6,7),(8,7),(6,9),(8,9),(6,2),(8,2),(6,4),(8,4),(3,5),(5,5),(3,7),(5,7),(1,2),(3,2),(1,4),(3,4)]
True
```

6) generateGrid

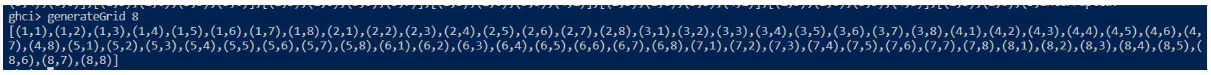


```haskell
generateGrid :: (Num b, Enum b) => b -> [(b, b)]
generateGrid gridNum = [ (col,row) | col <- [1..gridNum]  , row <- [1..gridNum] ]
```

Explanation:

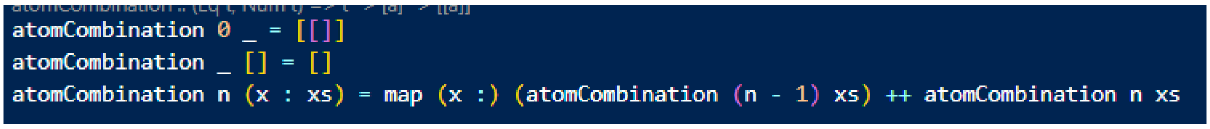
- This function takes grid size as the argument and returns all the points in a grid.

Output:



```
ghci> generateGrid 8
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(2,1),(2,2),(2,3),(2,4),(2,5),(2,6),(2,7),(2,8),(3,1),(3,2),(3,3),(3,4),(3,5),(3,6),(3,7),(3,8),(4,1),(4,2),(4,3),(4,4),(4,5),(4,6),(4,
7),(4,8),(5,1),(5,2),(5,3),(5,4),(5,5),(5,6),(5,7),(5,8),(6,1),(6,2),(6,3),(6,4),(6,5),(6,6),(6,7),(6,8),(7,1),(7,2),(7,3),(7,4),(7,5),(7,6),(7,7),(7,8),(8,1),(8,2),(8,3),(8,4),(8,5),(
8,6),(8,7),(8,8)]
```

7) atomCombination

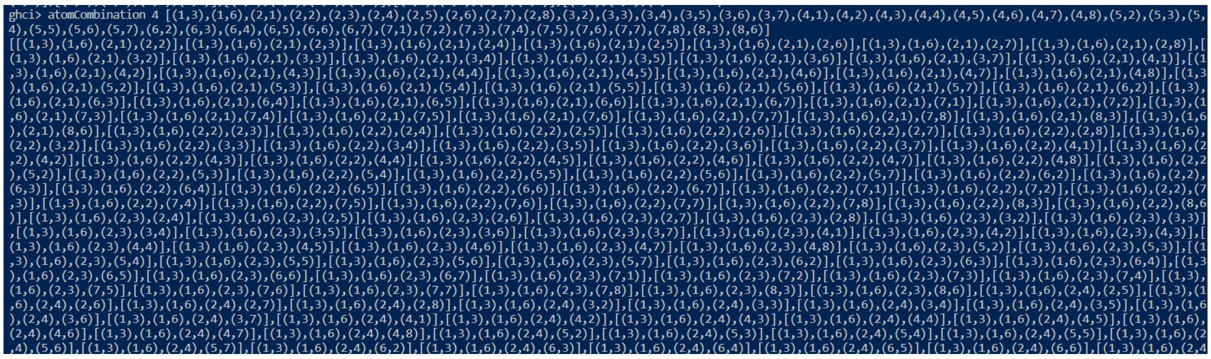


```haskell
atomCombination 0 _ = [[]]
atomCombination _ [] = []
atomCombination n (x : xs) = map (x :) (atomCombination (n - 1) xs) ++ atomCombination n xs
```

Explanation:

- atomCombination function takes two arguments which is number of elements in each combination and a list of points and it will return a list containing all the combinations of points with the specific number of elements. For instance, if number of element is 2 and list of [(1,1),(1,2),(1,3)] is passed as argument, the output would be [[(1,1),(1,2)],[(1,1),(1,3)],[(1,2),(1,3)]].

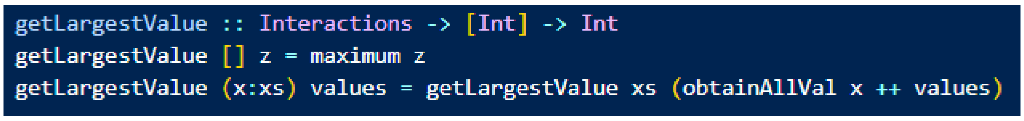
Output:



```
ghci> atomCombination 4 [(1,3),(1,6),(2,1),(2,2),(2,3),(2,4),(2,5),(2,6),(2,7),(2,8),(3,2),(3,3),(3,4),(3,5),(3,6),(3,7),(4,1),(4,2),(4,3),(4,4),(4,5),(4,6),(4,7),(4,8),(5,2),(5,3),(5,
4),(5,5),(5,6),(5,7),(6,2),(6,3),(6,4),(6,5),(6,6),(6,7),(7,1),(7,2),(7,3),(7,4),(7,5),(7,6),(7,7),(7,8),(8,3),(8,6)]
[[(1,3),(1,6),(2,1),(2,2)],[(1,3),(1,6),(2,1),(2,3)],[(1,3),(1,6),(2,1),(2,4)],[(1,3),(1,6),(2,1),(2,5)],[(1,3),(1,6),(2,1),(2,6)],[(1,3),(1,6),(2,1),(2,7)],[(1,3),(1,6),(2,1),(2,8)],[
(1,3),(1,6),(2,1),(3,2)],[(1,3),(1,6),(2,1),(3,3)],[(1,3),(1,6),(2,1),(3,4)],[(1,3),(1,6),(2,1),(3,5)],[(1,3),(1,6),(2,1),(3,6)],[(1,3),(1,6),(2,1),(3,7)],[(1,3),(1,6),(2,1),(4,1)],[(1
,3),(1,6),(2,1),(4,2)],[(1,3),(1,6),(2,1),(4,3)],[(1,3),(1,6),(2,1),(4,4)],[(1,3),(1,6),(2,1),(4,5)],[(1,3),(1,6),(2,1),(4,6)],[(1,3),(1,6),(2,1),(4,7)],[(1,3),(1,6),(2,1),(4,8)],[(1,3
),(1,6),(2,1),(5,2)],[(1,3),(1,6),(2,1),(5,3)],[(1,3),(1,6),(2,1),(5,4)],[(1,3),(1,6),(2,1),(5,5)],[(1,3),(1,6),(2,1),(5,6)],[(1,3),(1,6),(2,1),(5,7)],[(1,3),(1,6),(2,1),(6,2)],[(1,3),
(1,6),(2,1),(6,3)],[(1,3),(1,6),(2,1),(6,4)],[(1,3),(1,6),(2,1),(6,5)],[(1,3),(1,6),(2,1),(6,6)],[(1,3),(1,6),(2,1),(6,7)],[(1,3),(1,6),(2,1),(7,1)],[(1,3),(1,6),(2,1),(7,2)],[(1,3),(1
,6),(2,1),(7,3)],[(1,3),(1,6),(2,1),(7,4)],[(1,3),(1,6),(2,1),(7,5)],[(1,3),(1,6),(2,1),(7,6)],[(1,3),(1,6),(2,1),(7,7)],[(1,3),(1,6),(2,1),(7,8)],[(1,3),(1,6),(2,1),(8,3)],[(1,3),(1,6
),(2,1),(8,6)],[(1,3),(1,6),(2,2),(2,3)],[(1,3),(1,6),(2,2),(2,4)],[(1,3),(1,6),(2,2),(2,5)],[(1,3),(1,6),(2,2),(2,6)],[(1,3),(1,6),(2,2),(2,7)],[(1,3),(1,6),(2,2),(2,8)],[(1,3),(1,6),
(2,2),(3,2)],[(1,3),(1,6),(2,2),(3,3)],[(1,3),(1,6),(2,2),(3,4)],[(1,3),(1,6),(2,2),(3,5)],[(1,3),(1,6),(2,2),(3,6)],[(1,3),(1,6),(2,2),(3,7)],[(1,3),(1,6),(2,2),(4,1)],[(1,3),(1,6),(2
,2),(4,2)],[(1,3),(1,6),(2,2),(4,3)],[(1,3),(1,6),(2,2),(4,4)],[(1,3),(1,6),(2,2),(4,5)],[(1,3),(1,6),(2,2),(4,6)],[(1,3),(1,6),(2,2),(4,7)],[(1,3),(1,6),(2,2),(4,8)],[(1,3),(1,6),(2,2
),(5,2)],[(1,3),(1,6),(2,2),(5,3)],[(1,3),(1,6),(2,2),(5,4)],[(1,3),(1,6),(2,2),(5,5)],[(1,3),(1,6),(2,2),(5,6)],[(1,3),(1,6),(2,2),(5,7)],[(1,3),(1,6),(2,2),(6,2)],[(1,3),(1,6),(2,2),
(6,3)],[(1,3),(1,6),(2,2),(6,4)],[(1,3),(1,6),(2,2),(6,5)],[(1,3),(1,6),(2,2),(6,6)],[(1,3),(1,6),(2,2),(6,7)],[(1,3),(1,6),(2,2),(7,1)],[(1,3),(1,6),(2,2),(7,2)],[(1,3),(1,6),(2,2),(7
,3)],[(1,3),(1,6),(2,2),(7,4)],[(1,3),(1,6),(2,2),(7,5)],[(1,3),(1,6),(2,2),(7,6)],[(1,3),(1,6),(2,2),(7,7)],[(1,3),(1,6),(2,2),(7,8)],[(1,3),(1,6),(2,2),(8,3)],[(1,3),(1,6),(2,2),(8,6
)],[(1,3),(1,6),(2,3),(2,4)],[(1,3),(1,6),(2,3),(2,5)],[(1,3),(1,6),(2,3),(2,6)],[(1,3),(1,6),(2,3),(2,7)],[(1,3),(1,6),(2,3),(2,8)],[(1,3),(1,6),(2,3),(3,2)],[(1,3),(1,6),(2,3),(3,3)]
,[(1,3),(1,6),(2,3),(3,4)],[(1,3),(1,6),(2,3),(3,5)],[(1,3),(1,6),(2,3),(3,6)],[(1,3),(1,6),(2,3),(3,7)],[(1,3),(1,6),(2,3),(4,1)],[(1,3),(1,6),(2,3),(4,2)],[(1,3),(1,6),(2,3),(4,3)],[
(1,3),(1,6),(2,3),(4,4)],[(1,3),(1,6),(2,3),(4,5)],[(1,3),(1,6),(2,3),(4,6)],[(1,3),(1,6),(2,3),(4,7)],[(1,3),(1,6),(2,3),(4,8)],[(1,3),(1,6),(2,3),(5,2)],[(1,3),(1,6),(2,3),(5,3)],[(1
,3),(1,6),(2,3),(5,4)],[(1,3),(1,6),(2,3),(5,5)],[(1,3),(1,6),(2,3),(5,6)],[(1,3),(1,6),(2,3),(5,7)],[(1,3),(1,6),(2,3),(6,2)],[(1,3),(1,6),(2,3),(6,3)],[(1,3),(1,6),(2,3),(6,4)],[(1,3
),(1,6),(2,3),(6,5)],[(1,3),(1,6),(2,3),(6,6)],[(1,3),(1,6),(2,3),(6,7)],[(1,3),(1,6),(2,3),(7,1)],[(1,3),(1,6),(2,3),(7,2)],[(1,3),(1,6),(2,3),(7,3)],[(1,3),(1,6),(2,3),(7,4)],[(1,3),
(1,6),(2,3),(7,5)],[(1,3),(1,6),(2,3),(7,6)],[(1,3),(1,6),(2,3),(7,7)],[(1,3),(1,6),(2,3),(7,8)],[(1,3),(1,6),(2,3),(8,3)],[(1,3),(1,6),(2,3),(8,6)],[(1,3),(1,6),(2,4),(2,5)],[(1,3),(1
,6),(2,4),(2,6)],[(1,3),(1,6),(2,4),(2,7)],[(1,3),(1,6),(2,4),(2,8)],[(1,3),(1,6),(2,4),(3,2)],[(1,3),(1,6),(2,4),(3,3)],[(1,3),(1,6),(2,4),(3,4)],[(1,3),(1,6),(2,4),(3,5)],[(1,3),(1,6
),(2,4),(3,6)],[(1,3),(1,6),(2,4),(3,7)],[(1,3),(1,6),(2,4),(4,1)],[(1,3),(1,6),(2,4),(4,2)],[(1,3),(1,6),(2,4),(4,3)],[(1,3),(1,6),(2,4),(4,4)],[(1,3),(1,6),(2,4),(4,5)],[(1,3),(1,6),
(2,4),(4,6)],[(1,3),(1,6),(2,4),(4,7)],[(1,3),(1,6),(2,4),(4,8)],[(1,3),(1,6),(2,4),(5,2)],[(1,3),(1,6),(2,4),(5,3)],[(1,3),(1,6),(2,4),(5,4)],[(1,3),(1,6),(2,4),(5,5)],[(1,3),(1,6),(2
,4),(5,6)],[(1,3),(1,6),(2,4),(5,7)],[(1,3),(1,6),(2,4),(6,2)],[(1,3),(1,6),(2,4),(6,3)],[(1,3),(1,6),(2,4),(6,4)],[(1,3),(1,6),(2,4),(6,5)],[(1,3),(1,6),(2,4),(6,6)],[(1,3),(1,6),(2,4
```

The full output will have 163,185 combinations.

8) getLargestValue

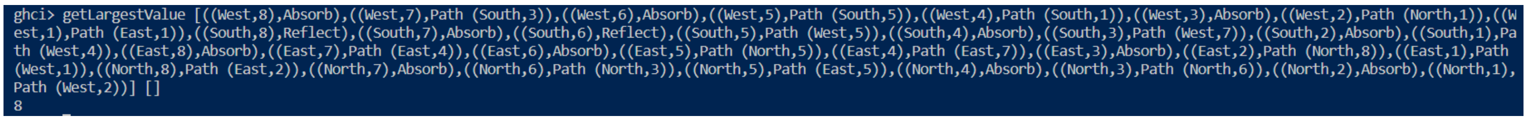


```haskell
getLargestValue :: Interactions -> [Int] -> Int
getLargestValue [] z = maximum z
getLargestValue (x:xs) values = getLargestValue xs (obtainAllVal x ++ values)
```

Explanation:

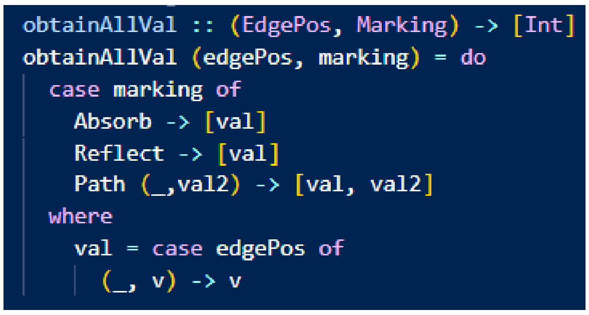
- This function takes two arguments which is the interactions and an initial empty list which will be used to store the values from each interaction. For instance, if the interaction is ((South,2),Path(North,1)), the value 1 and 2 will be returned from **obtainAllVal** function and stored in the list. When the interactions is empty, this indicates that all the value has been collected. Hence, the maximum value in the list will be returned as an output. This value will then be used as the grid size.

Output:



```
ghci> getLargestValue [((West,8),Absorb),((West,7),Path (South,3)),((West,6),Absorb),((West,5),Path (South,5)),((West,4),Path (South,1)),((West,3),Absorb),((West,2),Path (North,1)),((W
est,1),Path (East,1)),((South,8),Reflect),((South,7),Absorb),((South,6),Reflect),((South,5),Path (West,5)),((South,4),Absorb),((South,3),Path (West,7)),((South,2),Absorb),((South,1),Pa
th (West,4)),((East,8),Absorb),((East,7),Path (East,4)),((East,6),Absorb),((East,5),Path (North,5)),((East,4),Path (East,7)),((East,3),Absorb),((East,2),Path (North,8)),((East,1),Path
(West,1)),((North,8),Path (East,2)),((North,7),Absorb),((North,6),Path (North,3)),((North,5),Path (East,5)),((North,4),Absorb),((North,3),Path (North,6)),((North,2),Absorb),((North,1),
Path (West,2))] []
8
```

9) obtainAllVal

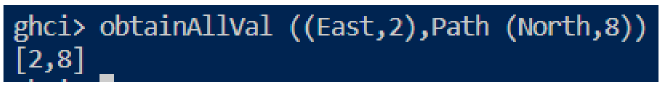


```haskell
obtainAllVal :: (EdgePos, Marking) -> [Int]
obtainAllVal (edgePos, marking) = do
  case marking of
    Absorb -> [val]
    Reflect -> [val]
    Path (_,val2) -> [val, val2]
  where
    val = case edgePos of
      (_, v) -> v
```

Explanation:

- obtainAllVal takes an argument which is an interaction and return a list of values. This function is used to extract value from the given interaction. For instance, if the interaction is ((South,2),Path(North,1)), a list containing value 1 and 2 will be returned.

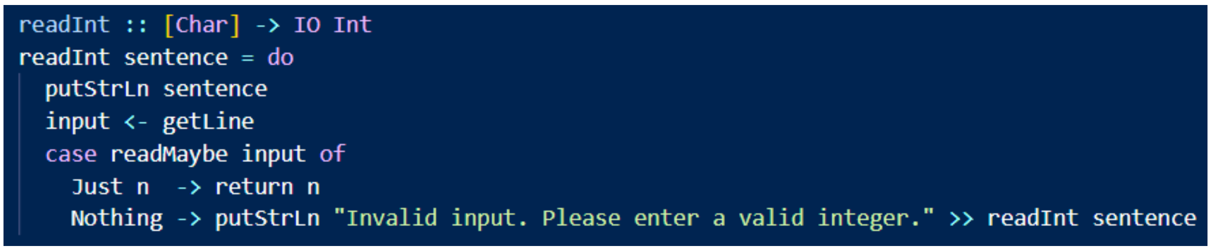
Output:



```
ghci> obtainAllVal ((East,2),Path (North,8))
[2,8]
```

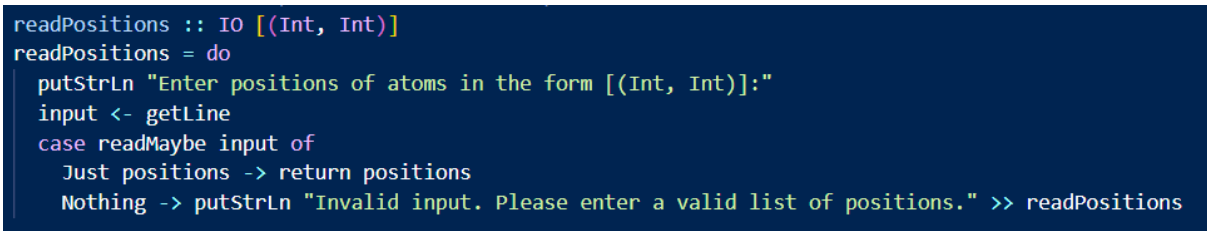
<u>IO operations</u>

These functions are mainly used for error handling when receiving input from the user.



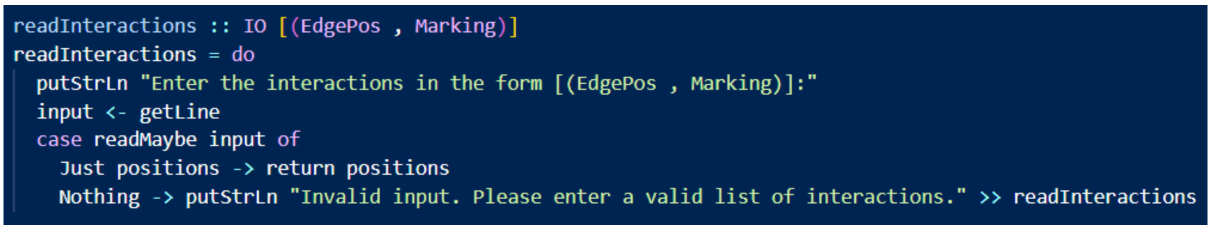
```haskell
readInt :: [Char] -> IO Int
readInt sentence = do
  putStrLn sentence
  input <- getLine
  case readMaybe input of
    Just n  -> return n
    Nothing -> putStrLn "Invalid input. Please enter a valid integer." >> readInt sentence
```

- Used to read integer values as input from the user



```haskell
readPositions :: IO [(Int, Int)]
readPositions = do
  putStrLn "Enter positions of atoms in the form [(Int, Int)]:"
  input <- getLine
  case readMaybe input of
    Just positions -> return positions
    Nothing -> putStrLn "Invalid input. Please enter a valid list of positions." >> readPositions
```

- Used to read input of type [(Int,Int)] from the user.



```haskell
readInteractions :: IO [(EdgePos , Marking)]
readInteractions = do
  putStrLn "Enter the interactions in the form [(EdgePos , Marking)]:"
  input <- getLine
  case readMaybe input of
    Just positions -> return positions
    Nothing -> putStrLn "Invalid input. Please enter a valid list of interactions." >> readInteractions
```

- Used to read input of type [(EdgePos,Marking)] from the user.