香 港 大 學
THE UNIVERSITY OF HONG KONG

COMP3258 Final Project

Kodable Haskell

Rajat Jain - 3035453684

Department of Computer Science

December 21, 2020

Contents

1 Setup **1**

2 Basic Game Play **1**

2.1 Loading a File 1

2.2 Checking if map is solvable 2

2.3 Finding optimal solution for a map 2

2.4 Play 3

3 Extra Features **4**

3.1 Hints 4

3.2 Random Map Generation 5

3.3 Animation & Graphics 7

4 Error Handling **8**

4.1 Map Loading 8

4.2 Invalid Options in Menu 8

4.3 Invalid input in Play 8

4.4 Invalid moves in Play 9

5 Data Structures and Design **10**

5.1 Board's Data Structure 10

5.2 Action's Data Type 10

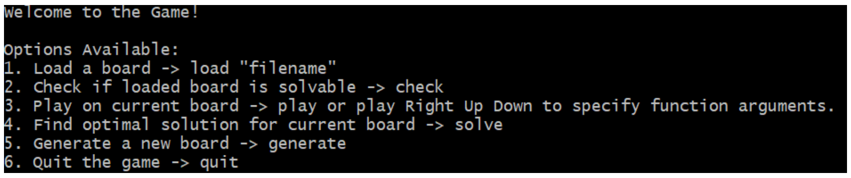
6 Finding optimal solution **11**

6.1 Idea 11

6.2 Algorithm 11

1 Setup

Before the building the project, please ensure that the package manager stack is installed on your local machine. Stack will be required to install the packages random ansi-terminal. The instructions for installation of stack depending on your operating system can be found here: https://docs.haskellstack.org/en/stable/install_and_upgrade/. Once stack has been installed, the project can be built and executed using the script ./start.sh. You should be able to see the message in Figure 1.



```
Welcome to the Game!

Options Available:
1. Load a board -> load "filename"
2. Check if loaded board is solvable -> check
3. Play on current board -> play or play Right Up Down to specify function arguments.
4. Find optimal solution for current board -> solve
5. Generate a new board -> generate
6. Quit the game -> quit
```

Figure 1: Welcome Screen

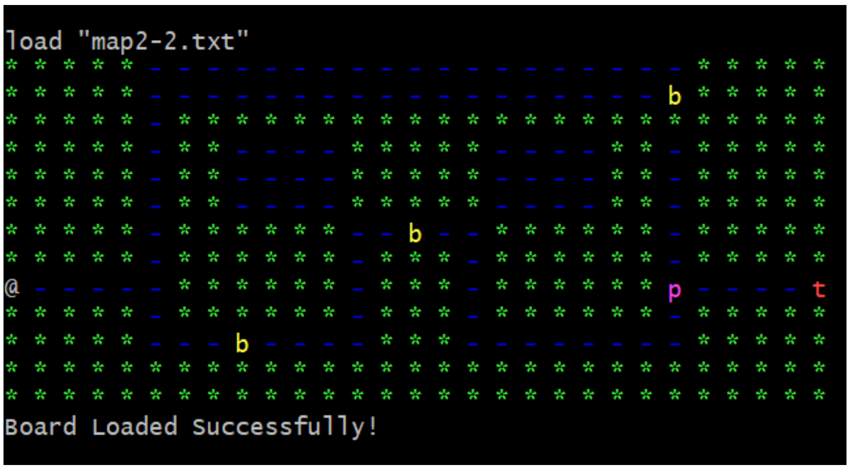
2 Basic Game Play

2.1 Loading a File

A new file can be loaded in the game by issuing the command:

```
load "map1-2.txt"
```

If the file has been loaded successfully, the user will able to see the screen as shown in Figure 2.



```
load "map2-2.txt"
* * * * * - - - - - - - - - - - - - - - - - - - * * * * *
* * * * * - - - - - - - - - - - - - - - - - - b * * * * *
* * * * * - * * * * * * * * * * * * * * * * * * * * * * *
* * * * * - * * - - - - * * * * * - - - - * * - * * * * *
* * * * * - * * - - - - * * * * * - - - - * * - * * * * *
* * * * * - * * - - - - * * * * * - - - - * * - * * * * *
* * * * * - * * * * * * - - b - - * * * * * * - * * * * *
* * * * * - * * * * * * - * * * - * * * * * * - * * * * *
@ - - - - - * * * * * * - * * * - * * * * * * p - - - - t
* * * * * - * * * * * * - * * * - * * * * * * - * * * * *
* * * * * - - - b - - - - * * * - - - - - - - - * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Board Loaded Successfully!
```

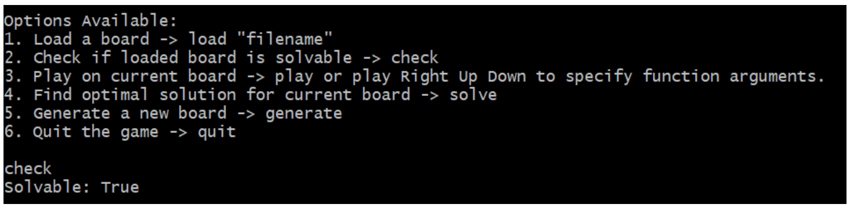
Figure 2: Map Loaded

2.2 Checking if map is solvable

After a map has been loaded, we can check if the map is solvable by issuing the following command:

```
check
```

For the map loaded in Figure 2, we can see that it is solvable, as illustrated in Figure 3.



```
Options Available:
1. Load a board -> load "filename"
2. Check if loaded board is solvable -> check
3. Play on current board -> play or play Right Up Down to specify function arguments.
4. Find optimal solution for current board -> solve
5. Generate a new board -> generate
6. Quit the game -> quit

check
Solvable: True
```

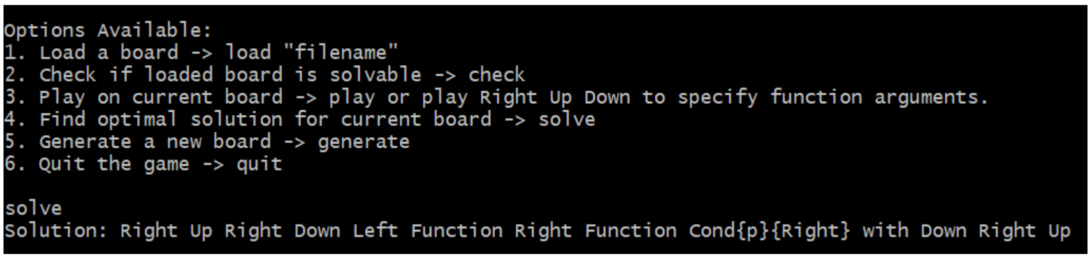
Figure 3: Map Loaded

2.3 Finding optimal solution for a map

The optimal solution for a map can be generated, provided that it is a solvable map. The solution can be generated by issuing the following command after the map has been loaded:

```
solve
```

For the map loaded in Figure 2, The generated solution is illustrated in Figure 4.



```
Options Available:
1. Load a board -> load "filename"
2. Check if loaded board is solvable -> check
3. Play on current board -> play or play Right Up Down to specify function arguments.
4. Find optimal solution for current board -> solve
5. Generate a new board -> generate
6. Quit the game -> quit

solve
Solution: Right Up Right Down Left Function Right Function Cond{p}{Right} with Down Right Up
```

Figure 4: Map Loaded

2.4 Play

The user can play on a loaded map by issuing either of the following commands. The former should be used if the user doesn't intend to use a function, while the later should be issued if the user intends to use a function and must specify the 3 actions which form the function.

```
play
```

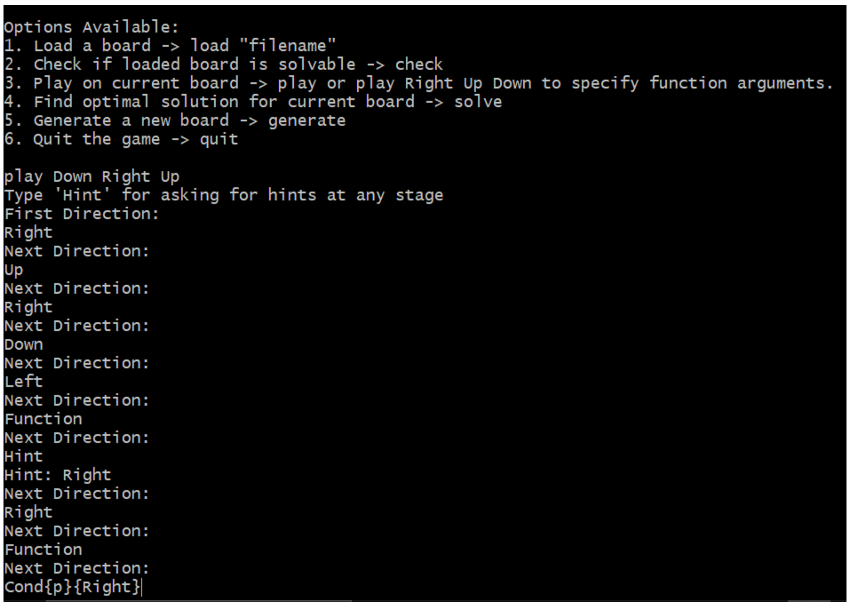
or

```
play Down Up Right
```

The user must input all the directions that they intend one after the other and finally hit enter to execute those sequence of commands. While playing the user can also ask for hints by typing 'Hint' when prompted for the next direction.

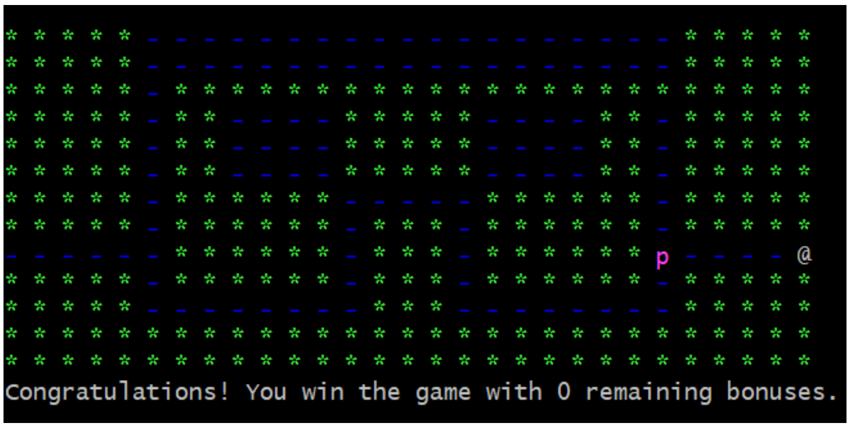
```
Next Direction: Hint
```

Example game play has been illustrated in Figure 5, while the end result has been illustrated Figure 6. The intermediate steps have been omitted as they are animated in the orignal game.



```
Options Available:
1. Load a board -> load "filename"
2. Check if loaded board is solvable -> check
3. Play on current board -> play or play Right Up Down to specify function arguments.
4. Find optimal solution for current board -> solve
5. Generate a new board -> generate
6. Quit the game -> quit

play Down Right Up
Type 'Hint' for asking for hints at any stage
First Direction:
Right
Next Direction:
Up
Next Direction:
Right
Next Direction:
Down
Next Direction:
Left
Next Direction:
Function
Next Direction:
Hint
Hint: Right
Next Direction:
Right
Next Direction:
Function
Next Direction:
Cond{p}{Right}
```

Figure 5: Example Game Play



```
* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * * * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * * * * * _ _ _ _ _ * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
_ _ _ _ _ _ * * * * * * _ * * * _ * * * * * * p _ _ _ _ @
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
* * * * * _ _ _ _ _ _ _ _ * * * _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Congratulations! You win the game with 0 remaining bonuses.
```

Figure 6: Game Ending

3 Extra Features

3.1 Hints

The user can ask for hints while using play in order to find the optimal next step for reaching the target. The usage of Hints have been illustrated in Section 2.4: Play.

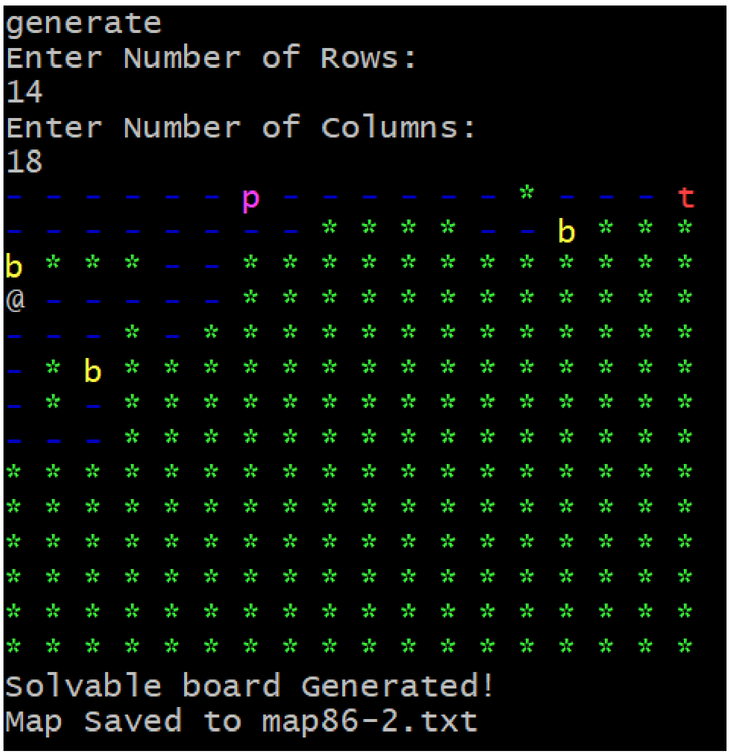
The implementation of play involves finding the current location of the user by accumulating all the actions that the user has entered so far. Following this, the solve function is used to find the optimal path from the current location to the target. Only the first step from the optimal path is displayed to the user as a hint.

3.2 Random Map Generation

The user can generate a new random map by issuing the command:

```
generate
```

After the user issues this command, the user will be prompted for the number of rows and columns in the new map. The algorithm will keep generating new boards till a solvable board is generated, which will be automatically saved to a new file. All the intermediate boards generated can be seen while the user waits for a solvable board.



```
generate
Enter Number of Rows:
14
Enter Number of Columns:
18
_ _ _ _ _ _ p _ _ _ _ _ _ * _ _ _ t
_ _ _ _ _ _ _ _ * * * * _ _ b * * *
b * * * _ _ * * * * * * * * * * * *
@ _ _ _ _ _ * * * * * * * * * * * *
_ _ _ * _ * * * * * * * * * * * * *
_ * b * * * * * * * * * * * * * * *
_ * _ * * * * * * * * * * * * * * *
_ _ _ * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *
Solvable board Generated!
Map Saved to map86-2.txt
```

Figure 7: Random Board Generation

A map is generated assuming that the ball will always be in the first column and the target will always be in the last column.

Steps for Map Generation:

- Two random numbers x, y are selected between 0 and (number of rows - 1) to identify the ball and target location, which will be (x, 0) and (y, columns-1).
- Using the ball location as the starting point and the target location as the end point, a search algorithm is initiated.
- One of the actions left, right, up and down is randomly chosen. Following this, we choose whether we want to apply this action with probability 1/3 and the previous action with probability 2/3. This ensures higher probability of applying previous action and discards very sparse maps.
- If the chosen action leads to a valid neighbour, we again start the search algorithm with the newly found neighbour as the starting point and adding it to the path. Otherwise we call this function again with the same starting point.
- The search continues till the starting point and target are the same. At this point, the list of all the points that have been explored so far is returned which will constitute the '-' entries in the map.
- After the list containing all the path entries is returned, we randomly choose 3 values and make them bonuses. One element from this list is also randomly chosen to be a conditional block. Furthermore, among all elements in the last column, another block is randomly chosen to be a conditional block, to ensure higher probability of a solvable map. Finally, all indices of the rows x columns matrix not belonging to this list are converted to Grass.
- The algorithm repeatedly generated new boards till a solvable board is generated.

```
genPath (x, y) target path row col last
  | (x, y) == target = do
    return path
  | otherwise =
    do
      index <- randomRIO (0, 3)
      let (newXDif, newYDif) = offsets !! index
      prob <- randomRIO (0, 2)
      let (xDif, yDif) = [(newXDif, newYDif), last, last] !! prob
      let newX = x + xDif
      let newY = y + yDif
      let finalPath = path ++ [(newX, newY)]
      if isValid (newX, newY) row col
        then genPath (newX, newY) target finalPath row col (xDif, yDif)
        else genPath (x, y) target path row col last
```

Note that sometimes the board may be generated instantly while sometimes it may take very long for the algorithm to find a solvable board. It is recommended that the user restarts the program if it is taking too long to generate a solvable board.

3.3 Animation & Graphics

The different colours in the printed board seen while using the load, play and generate function can be attributed to the ansi-terminal library. Before each element of the board is printed, it's corresponding type is checked according to which the color is adjusted prior to printing the character.

One can also see an animation when the actions are being executed in the play function. The Control.Concurrent() package's timedelay function has been used to introduce a delay before each time the board is printed after an action in the play function. In addition to this, sufficient space is added to the screen prior to printing the new board and hence the new board is printed at the same location. This creates an illusion that the ball is moving inside the map as each action is executed.

4 Error Handling

4.1 Map Loading

The file loading will fail if the file is not in the correct format. Criteria for correctness of the map:

- Number of elements in each row must be the same.
- Every two elements must be separated by a space.
- Only the characters ’@’, ’t’, ’-’, ’*’, ’p’, ’o’, ’y’ and ’b’ are accepted as valid characters.

4.2 Invalid Options in Menu

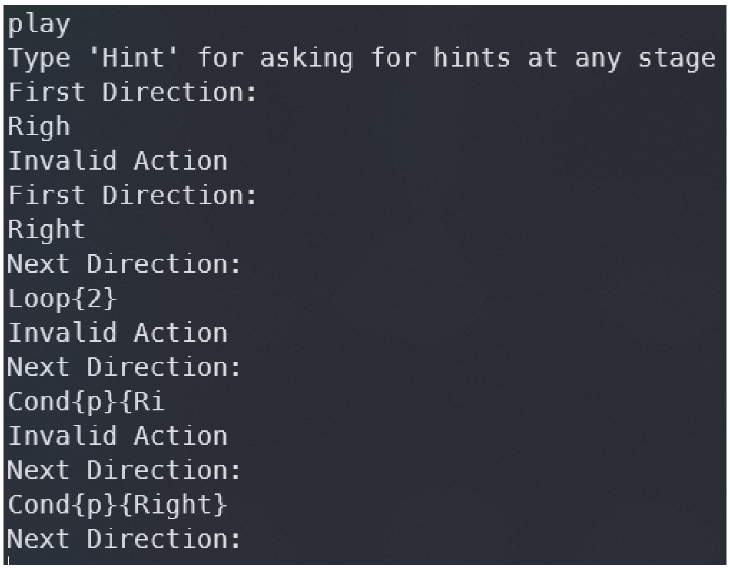
The following commands are acceptable in the game menu:

- load ”filename”
- check
- solve
- play
- generate
- quit

Any other command is considered invalid and the user is prompted to enter a valid command. Furthermore, a board must be loaded before the play, solve or check command is triggered, otherwise user is prompted to load the board first.

4.3 Invalid input in Play

The parser library from Assignment 2 A Parser has been used to parse the actions entered by the user while using play. The use of parser over regular string manipulation ensures robustness and is less prone to run-time errors. In case the parser fails to parse the string, the ”Invalid” action type is returned, which prompts the user to re-input the action.

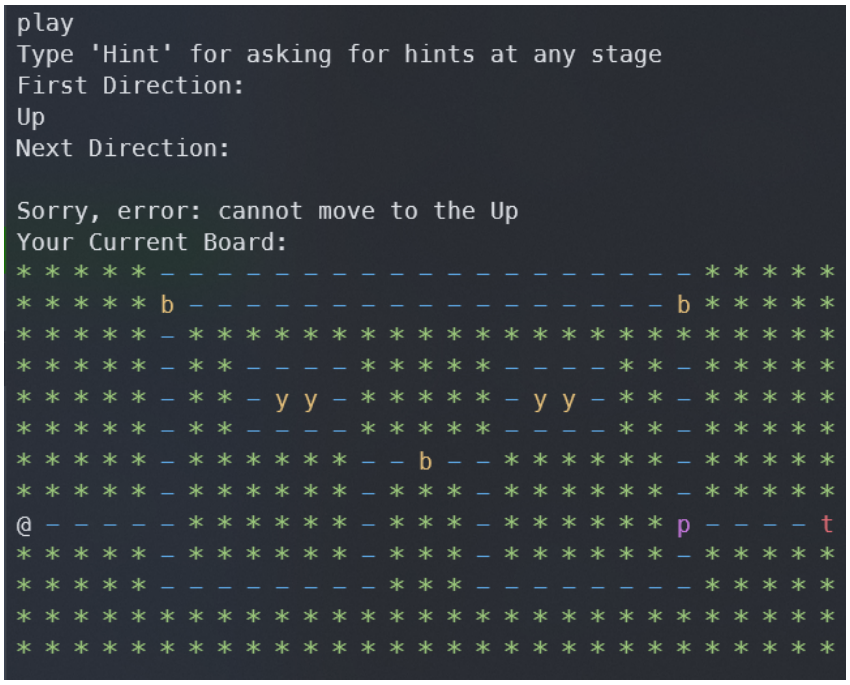


```
play
Type 'Hint' for asking for hints at any stage
First Direction:
Righ
Invalid Action
First Direction:
Right
Next Direction:
Loop{2}
Invalid Action
Next Direction:
Cond{p}{Ri
Invalid Action
Next Direction:
Cond{p}{Right}
Next Direction:
```

Figure 8: Parsing User Action

4.4 Invalid moves in Play

Whenever a player tries to make a move which is not feasible in the play function, only the moves up-to that move will be executed and the program will terminate at that point, telling the user that the move is not possible.



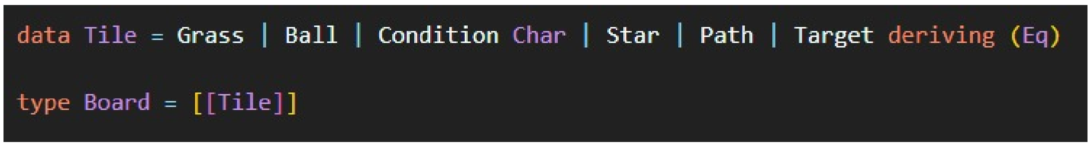
```
play
Type 'Hint' for asking for hints at any stage
First Direction:
Up
Next Direction:

Sorry, error: cannot move to the Up
Your Current Board:
* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * b _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ b * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * * * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ y y _ * * * * * _ y y _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * * * * * _ _ b _ _ * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
@ _ _ _ _ _ * * * * * * _ * * * _ * * * * * * p _ _ _ _ t
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
* * * * * _ _ _ _ _ _ _ _ * * * _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Figure 9: Invalid moves in play

5 Data Structures and Design

5.1 Board's Data Structure

The board in the text files can be thought of as a 2D Matrix. Hence, a list of lists is an ideal choice of a data structure for the board. A new data type called 'Tile' has been defined which is used to represent the different elements on the board such as Grass, Ball, Target, Conditional Blocks and Bonuses. The Conditional blocks take a parameter of type char to specify the color of the conditional block. Furthermore, a new type called Board is defined which a list of list of tiles, in order to improve readability.

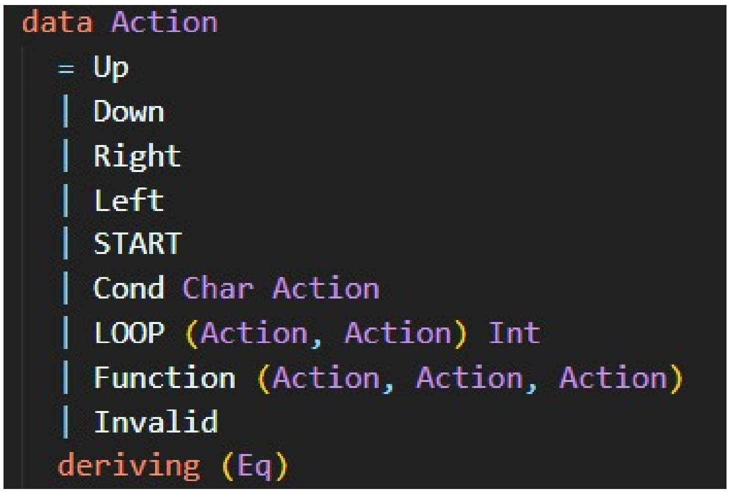


```
data Tile = Grass | Ball | Condition Char | Star | Path | Target deriving (Eq)

type Board = [[Tile]]
```

Figure 10: Board Data Structure

5.2 Action's Data Type

A new data type has been constructed to represent an action. A data type has been created in order to leverage the ability of data types to supported nested structures. Apart from trivial action types such as Up, Down, Left, Right, complex action types for Loops, Function and Conditionals have been defined which encapsulate other actions inside them apart from other parameters such as block colour or loop frequency.



```
data Action
  = Up
  | Down
  | Right
  | Left
  | START
  | Cond Char Action
  | LOOP (Action, Action) Int
  | Function (Action, Action, Action)
  | Invalid
  deriving (Eq)
```

Figure 11: Action Data Type

6 Finding optimal solution

6.1 Idea

A comparison of this problem with Pacman was used to devise a search strategy and find the optimal solution. The problem of collecting all bonuses and reaching the target can be considered similar to Pacman but differs in various aspects. Both search problems have an agent i.e the ball or pacman which must maximize the rewards, the food dots or the bonuses. A greedy approach with a heuristic being a function of the Manhattan distance of the closest or all food dots can find a solution for Pacman. However, as the board in this problem is not as spare as compared to Pacman, Manhattan distance is not an effective heuristic. Furthermore, as the board in this problem is not as sparse, the number of possible neighbours at each step is much lesser. Therefore, the search space in this problem is much smaller and we can explore each valid neighbour for a node and try to build all the possible paths to the target, many of which will contain all the bonuses.

6.2 Algorithm

An approach similar to Breadth First Search (BFS) was used to explore all the neighbours for each node in the frontier. However, instead of terminating the algorithm when the target is reached, we generate all possible paths to the target. The paths which contain the maximum number of bonuses are filtered out from these paths. Following that, all paths are parsed to contain Loops, Conditional Actions and Functions instead of just the trivial actions (this can be done in O(n) time where n is the length of the path). Among these parsed paths, the path of the shortest length is selected, which is the optimal solution.