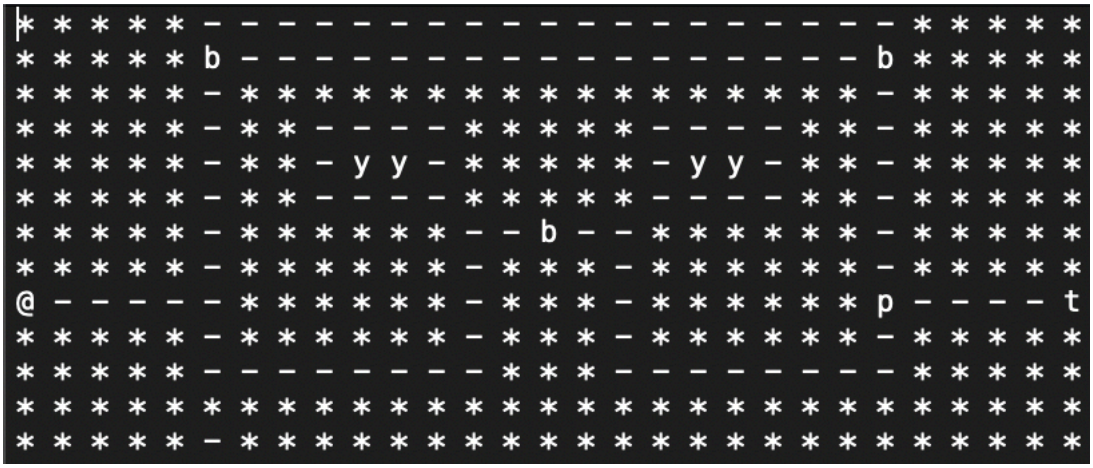
COMP3258

Functional Programming Final Project – Kodable

Name: Harsh Nagra
UID: 3035437707

Map File Assumptions:

1. Naming convention – suffix “-2.txt”. (example - map1-2.txt)
2. File convention: “ “, “@”, “*”, “-“, “p”, “o”, “y”, “t”, “b” are the only characters that should be used.
3. Horizontally every character should have a space between them, the first character in every line should be any of the characters mentioned above excluding “ “ and the last character of every line should be a “ “.
4. There should be no empty line in between.
5. The length and breadth of the map should be consistent as well.
6. Every map should have ‘@’ and ‘t’ representing the player and the target.



Example of a typical map file

Get Started:

- *ghci kodable.hs*
- *kodable*

These commands will start the interactive kodable game.

```
harshnagra@Harshs-MacBook-Pro 2020-project % ghci kodable.hs
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
[1 of 6] Compiling Common           ( Common.hs, interpreted )
[2 of 6] Compiling Solution         ( Solution.hs, interpreted )
[3 of 6] Compiling Solvable         ( Solvable.hs, interpreted )
[4 of 6] Compiling Hint             ( Hint.hs, interpreted )
[5 of 6] Compiling Play             ( Play.hs, interpreted )
[6 of 6] Compiling Kodable          ( kodable.hs, interpreted )
Ok, six modules loaded.
*Kodable> kodable
Select File: map5-2.txt map3-2.txt map1-2.txt map4-2.txt map2-2.txt
Please load a map:
```

The program will display all the files that the user can potentially use from the current directory i.e. "./" to ensure the maps with kind "-2.txt" are used.

```
*Kodable> kodable
Select File: map5-2.txt map3-2.txt map1-2.txt map4-2.txt map2-2.txt
Please load a map: map5.txt
Invalid File.
Select File: map5-2.txt map3-2.txt map1-2.txt map4-2.txt map2-2.txt
Please load a map:
```

If user enters a map that does not exist in the list, the user gets the error "Invalid File" and is asked to a file name again.

```
Select File: map5-2.txt map3-2.txt map1-2.txt map4-2.txt map2-2.txt
Please load a map: map5-2.txt
* * * * * - - - - - - - - - - - - - - - - - * * * *
* * * * * - - - - - - - - - - - - - - - - - b * * * *
* * * * * - * * * * * * * * * * * * * * * - * * * * *
* * * * * * * * - - - - - - - - * * - - - - * * - * * * *
* * * * * - * * - - - - * * - * * - - - - * * * * * *
* * * * * * * * - - - - * * - * - - - - - * * - * * * *
* * * * * - - - - * * * - - b - - * * * * * * - * * * *
* * * * * - * * * * * * * - * * * - * * * * * * - * * * *
@ - - - - - * * * * * * - * * * - * * * * * * y - - - t
* * * * * * * * * * * * - * * * - * * * * * * - * * * *
* * * * * - - - b - - - - * * * - - - - - - - - - * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
This map is SOLVABLE!

Please Choose:
play          -> Play without function.
play _ _ _    -> Play with function.
solvable      -> Checks if the map is solvable.
bonus         -> Checks if player -> (all bonus) reachable AND (all bonus) -> target reachable.
solution      -> Get an optimal solution to the loaded map.
load          -> Load new map.
quit          -> To Quit the game.
Your choice:
```

Once the user enters the correct file name, it will display the map read from that file. Right after reading and displaying the map, the program will check if the map is solvable or not to inform the user if they are working on a Solvable/Not-Solvable map.

It will further display many options that the user can choose from to proceed in the game.

```
Please Choose:
play          -> Play without function.
play _ _ _    -> Play with function.
solvable      -> Checks if the map is solvable.
bonus         -> Checks if player -> (all bonus) reachable AND (all bonus) -> target reachable.
solution      -> Get an optimal solution to the loaded map.
load          -> Load new map.
quit          -> To Quit the game.
Your choice: solvable
```

The command *"load"* can be used to load another map into our interactive application. It will display all the available files and ask the user to input the file name again.

The command *"solvable"* can be used in the future as well to recheck if the map is solvable or not. This is achieved by check that a player "@" can reach the target "t".

The command *"bonus"* can be used to check if all the bonuses mentioned in the file are reachable by the player "@" or not. In addition, this command also checks if the player reaches "b" then will it be able to reach "t" from there.

```
Your choice: solvable

This map is solvable!

Please Choose:
play          -> Play without function.
play _ _ _    -> Play with function.
solvable      -> Checks if the map is solvable.
bonus         -> Checks if player -> (all bonus) reachable AND (all bonus) -> target reachable.
solution      -> Get an optimal solution to the loaded map.
load          -> Load new map.
quit          -> To Quit the game.
Your choice: bonus

All bonus reachable!
Target not reachable from 1 bonus.
```

For the currently loaded map, the player can reach all the bonuses but if the player goes to one of the bonuses it will not be able to reach the target. This will help the user understand the map better.

To display an optimal solution for the given map the user can choose the option "*solution*".

```
All bonus reachable!
Target not reachable from 1 bonus.

Please Choose:
play          -> Play without function.
play _ _ _    -> Play with function.
solvable      -> Checks if the map is solvable.
bonus         -> Checks if player -> (all bonus) reachable AND (all bonus) -> target reachable.
solution      -> Get an optimal solution to the loaded map.
load          -> Load new map.
quit          -> To Quit the game.
Your choice: solution

Solution without Function and Loops: Right Up Right Up Right Down Left Down Left Right Up Right Down Right Up Cond{y}{Right}
Compressed: Loop{2}{Right,Up} Right Down Left Down Left Right Function Right Up Cond{y}{Right} with Up Right Down
```

"solution" displays the solution with least number of direction changes and then further compresses this solution to give the same directions but with Loops & Functions.

Now, the user can also play the game themselves by choosing the option play or play with three parameters (which are only allowed to be valid directions or conditionals).

Any kind of typos and extra spacing will generate error.

Option *"play"* will define a function with empty parameters, therefore, if a function is called after choosing the option *"play"* it will be ineffective and not make any move.

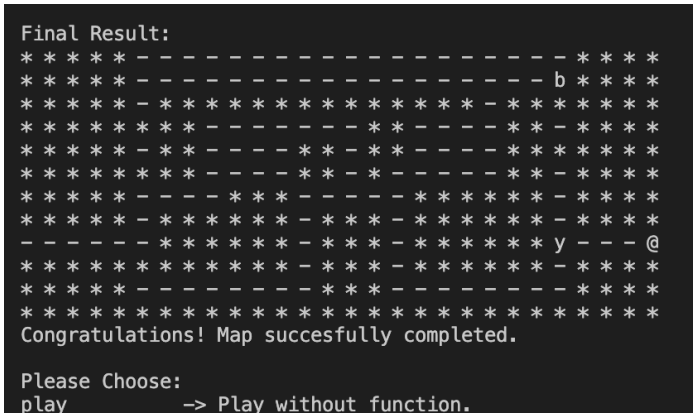
```
quit          -> To quit the game.
Your choice: play Right Up Right

Valid Directions: Left, Right, Up, Down
Valid Block: p, o, y
Valid Moves: Direction, Cond{Block}{Direction}, Loop{iterations}{Direction, Direction}, Function

First Move: Function
Next Move: Up
Next Move:
```

Above, the user chooses play with three arguments. Makes his first move and second move. Now, if the user presses enter without typing anything, the program will compute the final location of the user.

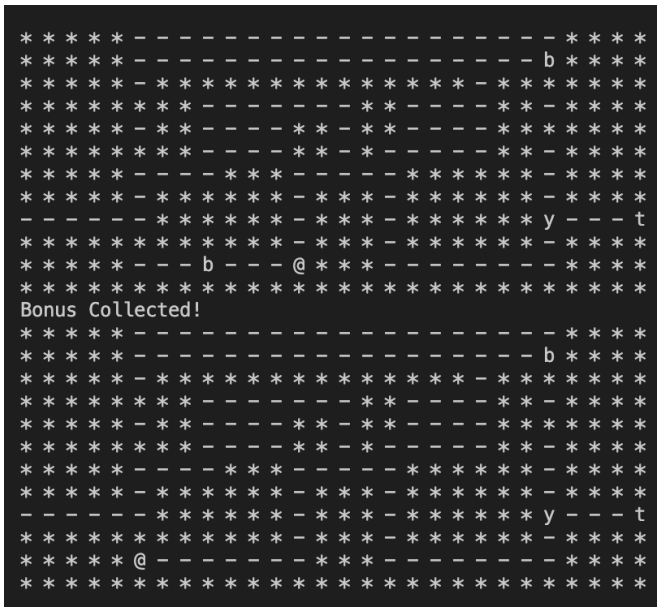
```
Final Result:
* * * * * - - - - - - - - - - - - - - - - - * * * *
* * * * * - - - - - - - - - - - - - - - - - b * * * *
* * * * * - * * * * * * * * * * * * * * - * * * * * *
* * * * * * * @ - - - - - - * * - - - - * * - * * * *
* * * * * - * * - - - - * * - * * - - - - * * * * * *
* * * * * * * - - - - * * - * - - - - - * * - * * * *
* * * * * - - - - * * * - - b - - * * * * * * - * * * *
* * * * * - * * * * * * - * * * - * * * * * * - * * * *
- - - - - - * * * * * * - * * * - * * * * * * y - - - t
* * * * * * * * * * * - * * * - * * * * * * - * * * *
* * * * * - - - b - - - - * * * _ - - - - - - - * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
Sorry! You did not reach the target.
Do you need a hint? (Yes/No)
Yes
Hint: Right Down
Next Move: Right
Next Move:
```

As the user did not reach the target location, the program informs them that they have not reached the target and asks if they need a hint to reach there. If the user enters anything except Yes, the program will assume that they would like to stop playing. Hint provides the user with suggestion of next two moves.

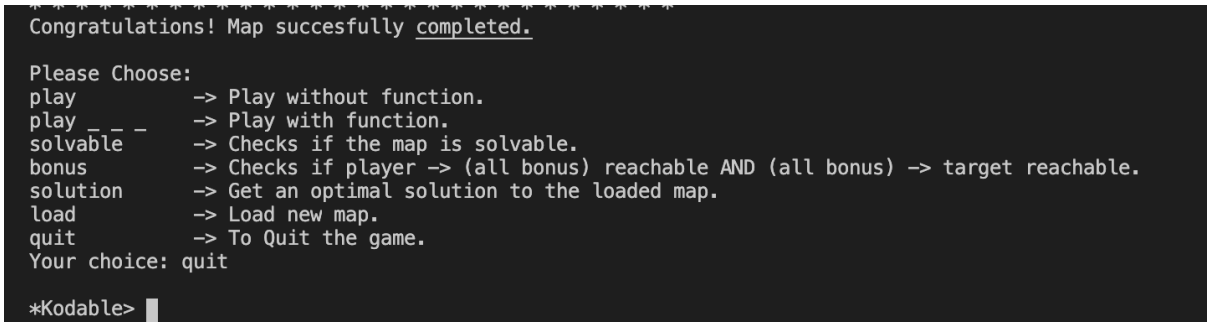
If the user keeps on inputting values until which finally lands the player at the target, then a congratulations message is posted.



Every map after every move is actually displayed before the final result. I have not added the screenshots of those maps due to size constraints of the document.



It also notifies between the two moves a bonus is collected. In this map the user does not go to the third bonus as they won't be able to reach back as mentioned earlier.



Finally, the player can quit kodable by using the *"quit"* option.

Data Structures and other logical details:

1. *Reading text file*

The text file is read using “readFile” and then saved in the variable map throughout the program.

The type of map is [String]. Every string represents a line of the file, and whenever the map needs to be printed it can be done using printMap which is a self-defined function.

This format makes the access easier to every element as it can now be treated as a 2D array. (map !! x) !! y where (x,y) are the coordinated.

2. *Checking if the Map is solvable*

solvable takes the map, the current position and the target position and finally returns if it is solvable or not.

```
solvable :: [String] -> (Int, Int) -> (Int, Int) -> Bool
solvable map current target = findPath map current target [] (-1)
```

To check if a map is solvable, we do depth first search based on three cases.

a. Base case, that is if the player is at the target already then return True.
b. If the player is not at target and can’t move in any of the directions, then False.
c. Finally, if there is a direction to move in the player moves in that direction. To ensure that the player can’t change direction without being on a block or next to the grass we also keep checking the last direction of the player.

While, moving into other directions the player checks if they have already visited a node by checking a list of coordinates [(Int, Int)] to ensure that it does not keep iterating between any points. Every new visited by the player is added to this list. If anyone path is found the function will return True.

The file solvable.hs contains the solution for this part. The same program is used to check the number of reachable bonuses and then reaching the target from these bonuses. This further helps in Hint and Solution.

3. *Computing an Optimal Solution*

A very similar approach to solvable is taken in finding the optimal solution. The important heuristic of finding the optimal solution is to make sure that the player collects as many as possible bonuses.

solution requires the map, current position and returns the most optimal solution with the least number of directions.

```
solution :: [String] -> (Int, Int) -> Int -> [String]
```

The code for this part is in solution.hs

solutions returns the list of list of moves that carry all the possible solutions.

```
solutions :: [String] -> (Int, Int) ->  [String] -> [(Int, Int)] -> [((Int, Int),
Int)] -> Int -> [[String]]
```

The data structures required in this part were

a. [String] for the map
b. (Int, Int) for current position
c. [String] for building a new solution i.e. incomplete current solution
d. [(Int,Int)] a list of bonuses that have been collected.
e. [((Int, Int), Int)] a list carrying the visited nodes and these visited nodes are now based on the number of bonuses that been reached. Therefore, we can revisit a node if we have now collected a greater number of bonuses.
f. Finally, the total number of reachable bonuses i.e. (total '@' to 'b' reachable minus total 'b' to 't' not reachable). This is calculated using solvable.

Now, the three main cases are

- If player has reached target and has collected all the possible bonuses, then add solution to the list of solutions.
- If player has reached target and has not collected all the possible bonuses, then do not add solution to the list of solutions.
- If the player has not reached the target and he can still visit new nodes after checking possible movements (up, down, left, right) and visited nodes, then move in all possible directions from here and add this node with the current bonus count of this node to the visited list.
    - Also check if on conditional block or not, add the direction accordingly. (Cond{p}{Right} or Right)

Only the final reachable nodes are added to visited list and nodes which come on the way but is not possible to stop at isn't added.

The compression of the most optimal solution is also done in this file by first finding any loops in the solution and then finally adding a function with the most repeated values 3 values (one after another).

4. Play functionalities

Play has simple but lengthy logical implementation.

The user is initially asked to enter all moves and then these moves are checked for validity.

a. These moves are then decompressed by the algorithm to remove Loops and Functions to create a normal list of moves.
b. After the decompression is completed, the algorithm starts applying these moves one by one to the map and in return receiving the new position of the player.
c. After all the positions are collected of the player movement, all the maps regarding this movement are printed by the algorithm while also showing which all bonuses are collected as well. If the player reaches the target, they are congratulated.
d. Otherwise, they are asked if they need a hint. This hint is calculated using a complex mixture of Solvable and Solution. Every **hint** considers how many bonuses are reachable from the current position and also if it is possible to reach back to target from these bonuses. This is accomplished using solvable. After we have the right number of reachable bonuses the algorithm calls solution with the bonus count and the algorithm returns the next two moves of the most optimal solution.

```
movePlayer :: [String] -> (Int, Int) -> String -> Char -> (Int, Int)
movePlayer map position move nextMove
```

```
expandList :: [String] -> [String] -> [String]
```

5. Error Checking

```
*Kodable> kodable
Select File: map5-2.txt map3-2.txt map1-2.txt map4-2.txt map2-2.txt
Please load a map: map2.txt
Invalid File.
Select File: map5-2.txt map3-2.txt map1-2.txt map4-2.txt map2-2.txt
Please load a map:
```

```
*Kodable> kodable
Select File: map5-2.txt map3-2.txt map1-2.txt map4-2.txt map2-2.txt
Please load a map: map1-2.txt
* * * * * - - - - - - - - - - - - - - - - - - - * * * * *
* * * * * b - - - - - - - - - - - - - - - - - - b * * * * *
* * * * * - * * * * * * * * * * * * * * * * * - * * * * *
* * * * * - * * - - - - * * * * * - - - - * * - * * * * *
* * * * * - * * - y y - * * * * * - y y - * * - * * * * *
* * * * * - * * - - - - * * * * * - - - - * * - * * * * *
* * * * * - * * * * * * - - b - - * * * * * - * * * * *
* * * * * - * * * * * * - * * * - * * * * * - * * * * *
@ - - - - - * * * * * * - * * * - * * * * * p - - * - t
* * * * * - * * * * * * - * * * - * * * * * - * * * * *
* * * * * - - - - - - - - * * * - - - - - - - - * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * - * * * * * * * * * * * * * * * * * * * * * *
This map is NOT SOLVABLE. I suggest you to load a new map.
```

```
* * * * * - * * * * * * * * * * * * * * * * * * * * *
This map is SOLVABLE!

Please Choose:
play          -> Play without function.
play _ _ _    -> Play with function.
solvable      -> Checks if the map is solvable.
bonus         -> Checks if player -> (all bonus) reachable AND (all bonus) -> target reachable.
solution      -> Get an optimal solution to the loaded map.
load          -> Load new map.
quit          -> To Quit the game.
Your choice: player

Invalid Choice

Please Choose:
play          -> Play without function.
```

```
Please Choose:
play          -> Play without function.
play _ _ _    -> Play with function.
solvable      -> Checks if the map is solvable.
bonus         -> Checks if player -> (all bonus) reachable AND (all bonus) -> target reachable.
solution      -> Get an optimal solution to the loaded map.
load          -> Load new map.
quit          -> To Quit the game.
Your choice: play Left Left

Invalid defination of Play Function! Rules: No Loops & 3 Directions with or without Conditions.
```

```
Please Choose:
play          -> Play without function.
play _ _ _    -> Play with function.
solvable      -> Checks if the map is solvable.
bonus         -> Checks if player -> (all bonus) reachable AND (all bonus) -> target reachable.
solution      -> Get an optimal solution to the loaded map.
load          -> Load new map.
quit          -> To Quit the game.
Your choice: solution

Solution without Function and Loops: No Solution!
Compressed: No Solution!
quit          -> To Quit the game.
Your choice: play

Valid Directions: Left, Right, Up, Down
Valid Block: p, o, y
Valid Moves: Direction, Cond{Block}{Direction}, Loop{iterations}{Direction, Direction}, Function

First Move: Right
Next Move: Cond{x}{Right}
Next Move:
Invalid Input
```

```
quit          -> To Quit the game.
Your choice: play

Valid Directions: Left, Right, Up, Down
Valid Block: p, o, y
Valid Moves: Direction, Cond{Block}{Direction}, Loop{iterations}{Direction, Direction}, Function

First Move: Right
Next Move: Right
Next Move:
Can't make move: Right
```

```
load          -> Load new map.
quit          -> To Quit the game.
Your choice: play

Valid Directions: Left, Right, Up, Down
Valid Block: p, o, y
Valid Moves: Direction, Cond{Block}{Direction}, Loop{iterations}{Direction, Direction}, Function

First Move: Right
Next Move: Down
Next Move: Cond{p}{Right}
Next Move:
Can't make move: Right or blockp not there.
```

```
Please Choose:
play          -> Play without function.
play _ _ _    -> Play with function.
solvable      -> Checks if the map is solvable.
bonus         -> Checks if player -> (all bonus) reachable AND (all bonus) -> target reachable.
solution      -> Get an optimal solution to the loaded map.
load          -> Load new map.
quit          -> To Quit the game.
Your choice: play Loop{2}{Right,Down} Right Up

Invalid defination of Play Function! Rules: No Loops & 3 Directions with or without Conditions.
```

6. Additional Features

i. Check available maps and check if wrong map file is entered. This is done by using System.Directory library which helped finding all the all files in current directory and making a list of the files with the suffix “-2.txt”.

ii. Hints – Every **hint** considers how many bonuses are reachable from the current position and also if it is possible to reach back to target from these bonuses. This is accomplished using solvable. After we have the right number of reachable bonuses the algorithm calls solution with the bonus count and the algorithm returns the next two moves of the most optimal solution. Hint ensures the most optimal solution and if will always reach all the possible bonuses.

iii. Check reachable bonus from player and check if target is reachable from all bonuses. This is done using solvable from ‘@’ to ‘b’ and all ‘b’ to ‘t’.