Functional Programming Project : Kodable Game

Marco Brian Widjaja , UID: 3035493024

December 2020

Contents

1 How to build project **1**

2 Game play walk-through **2**

3 Game Logic and Data Structures **8**

4 Error Handling **10**

5 Extra features **12**

1 How to build project

I am building this project on a **Windows 10** machine with **GHC version 8.10.2**.

There are two haskell files that will be used for building the project. All the code for the kodable game logic is found in **project.hs** and I am re-using the **Parser.hs** file from the previous assignments for creating my own custom parsers which is defined in **project.hs** .

To build the project simply type the command

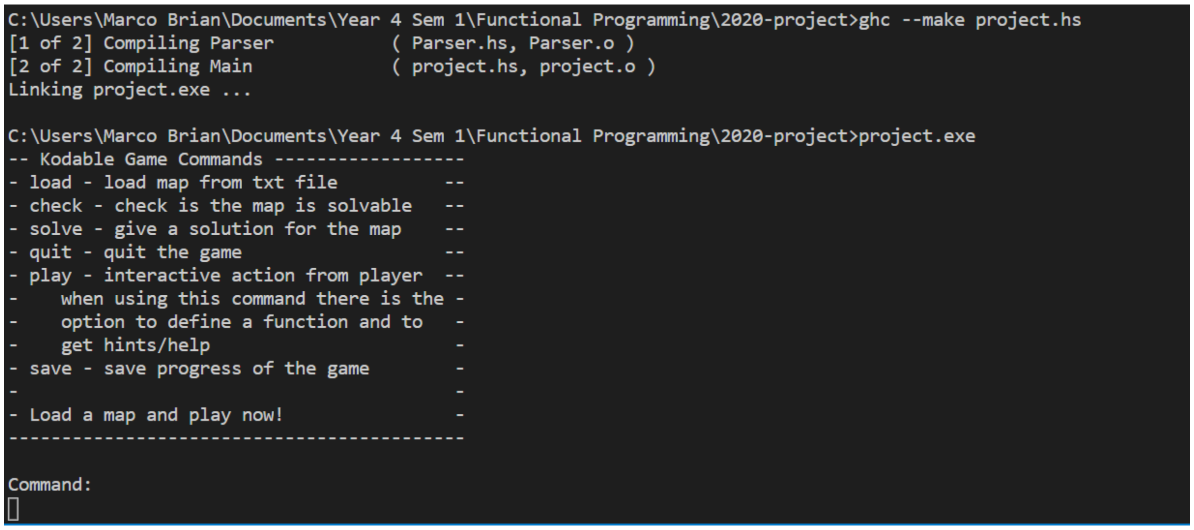
```
ghc --make project.hs
```

then there will object files created along with an executable called project.exe To start the game on the terminal type:

```
project.exe
```

or if on UNIX based machine

```
./project
```



```
C:\Users\Marco Brian\Documents\Year 4 Sem 1\Functional Programming\2020-project>ghc --make project.hs
[1 of 2] Compiling Parser           ( Parser.hs, Parser.o )
[2 of 2] Compiling Main             ( project.hs, project.o )
Linking project.exe ...

C:\Users\Marco Brian\Documents\Year 4 Sem 1\Functional Programming\2020-project>project.exe
-- Kodable Game Commands ------------------
- load - load map from txt file          --
- check - check is the map is solvable   --
- solve - give a solution for the map    --
- quit - quit the game                   --
- play - interactive action from player  --
-    when using this command there is the -
-    option to define a function and to   -
-    get hints/help                       -
- save - save progress of the game        -
-                                         -
- Load a map and play now!                -
-------------------------------------------

Command:
```

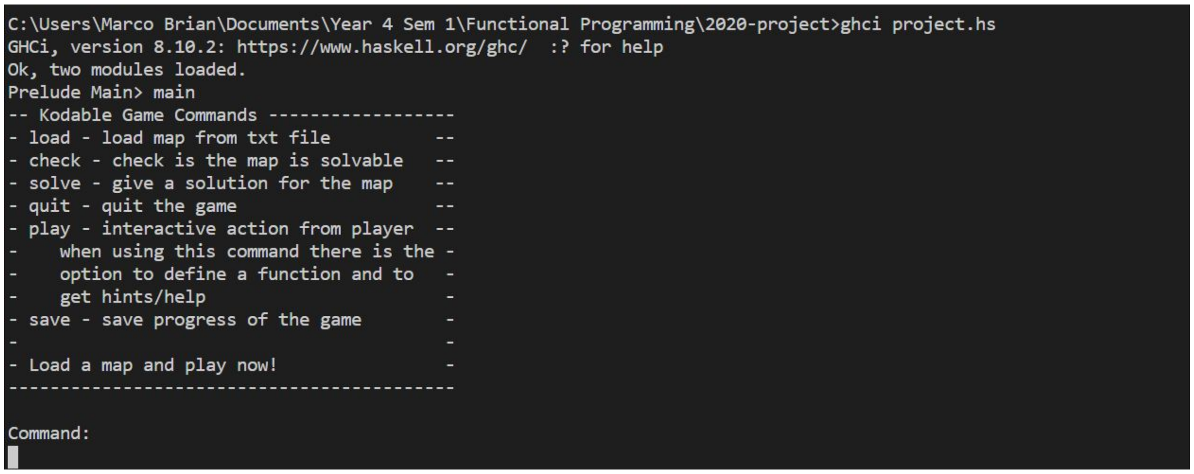
Figure 1: Compilation successful and starting the executable

Shall at the event that compilation fails, you can still run the game by going to terminal and use ghci.

```
ghci project.hs
```

When the ghci starts, start the game by typing

```
main
```



```
C:\Users\Marco Brian\Documents\Year 4 Sem 1\Functional Programming\2020-project>ghci project.hs
GHCi, version 8.10.2: https://www.haskell.org/ghc/  :? for help
Ok, two modules loaded.
Prelude Main> main
-- Kodable Game Commands ------------------
- load - load map from txt file          --
- check - check is the map is solvable   --
- solve - give a solution for the map    --
- quit - quit the game                   --
- play - interactive action from player  --
-    when using this command there is the -
-    option to define a function and to   -
-    get hints/help                       -
- save - save progress of the game        -
-                                         -
- Load a map and play now!                -
-------------------------------------------

Command:
```

Figure 2: Using ghci

2 Game play walk-through

The first thing to do once the game starts is to load a valid map. Without loading a text file for the map, the user will not be able to play the game. The other basic commands such as *solve* , *check, and play* will not work if the map has not been loaded.

Loading a map

To load a map the user must type in the command with format

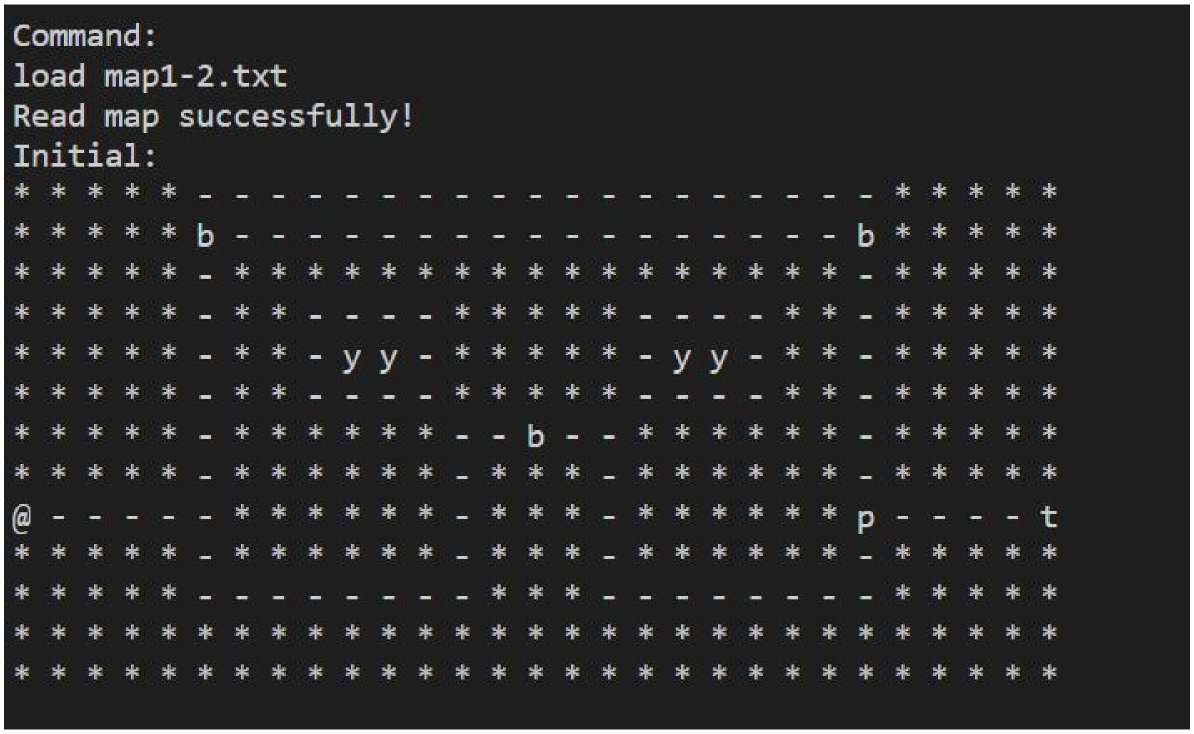
```
load filename
```

where filename is the name of the .txt file which contains the map

So for example if the user has a file named “map.txt”. The user must type in

```
load map.txt
```

Notice that there is no need for double quotation marks around the file name when trying to load the file. The file loaded needs to exist in the current directory of the program or else the program will raise an exception. Once the user has loaded a map, the user will then be able to use the basic commands, such as *solve, check and play* on the loaded game map.



```
Command:
load map1-2.txt
Read map successfully!
Initial:
* * * * * - - - - - - - - - - - - - - - - - - - * * * * *
* * * * * b - - - - - - - - - - - - - - - - - b * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * _ * * * * *
* * * * * _ * * - - - - * * * * * - - - - * * _ * * * * *
* * * * * _ * * - y y - * * * * * - y y - * * _ * * * * *
* * * * * _ * * - - - - * * * * * - - - - * * _ * * * * *
* * * * * _ * * * * * * - - b - - * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
@ - - - - - * * * * * * _ * * * _ * * * * * * p - - - - t
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
* * * * * - - - - - - - - * * * - - - - - - - - * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Figure 3: Game map loaded

Check

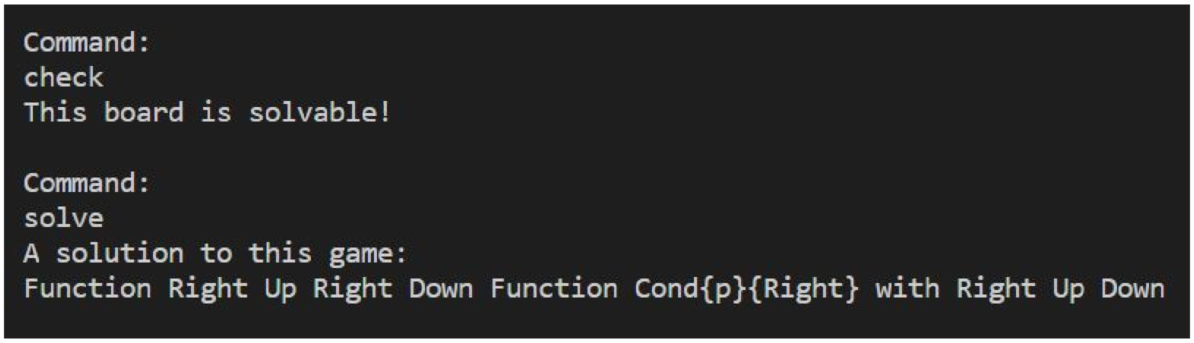
The check command will allow the user to check if this map is solvable or not.

```
check
```

Solve

The solve command will give the user the a set of optimal moves (least directions) in order to complete the game. The user may sometimes notice that the solve optimal move may initially go further than the end target but this is probably due to trying to achieve the bonus. It is not because the program is not optimal.

```
solve
```



```
Command:
check
This board is solvable!

Command:
solve
A solution to this game:
Function Right Up Right Down Function Cond{p}{Right} with Right Up Down
```

Figure 4: Check and Solve command

Interactive Mode

The play command allows the user to enter the interactive mode to play the game.

```
play
```

When using the play command, the user can additionally put in **3 moves** from [**Left, Right, Up, Down**] to define the **Function** command ex :

```
play Right Down Right
```

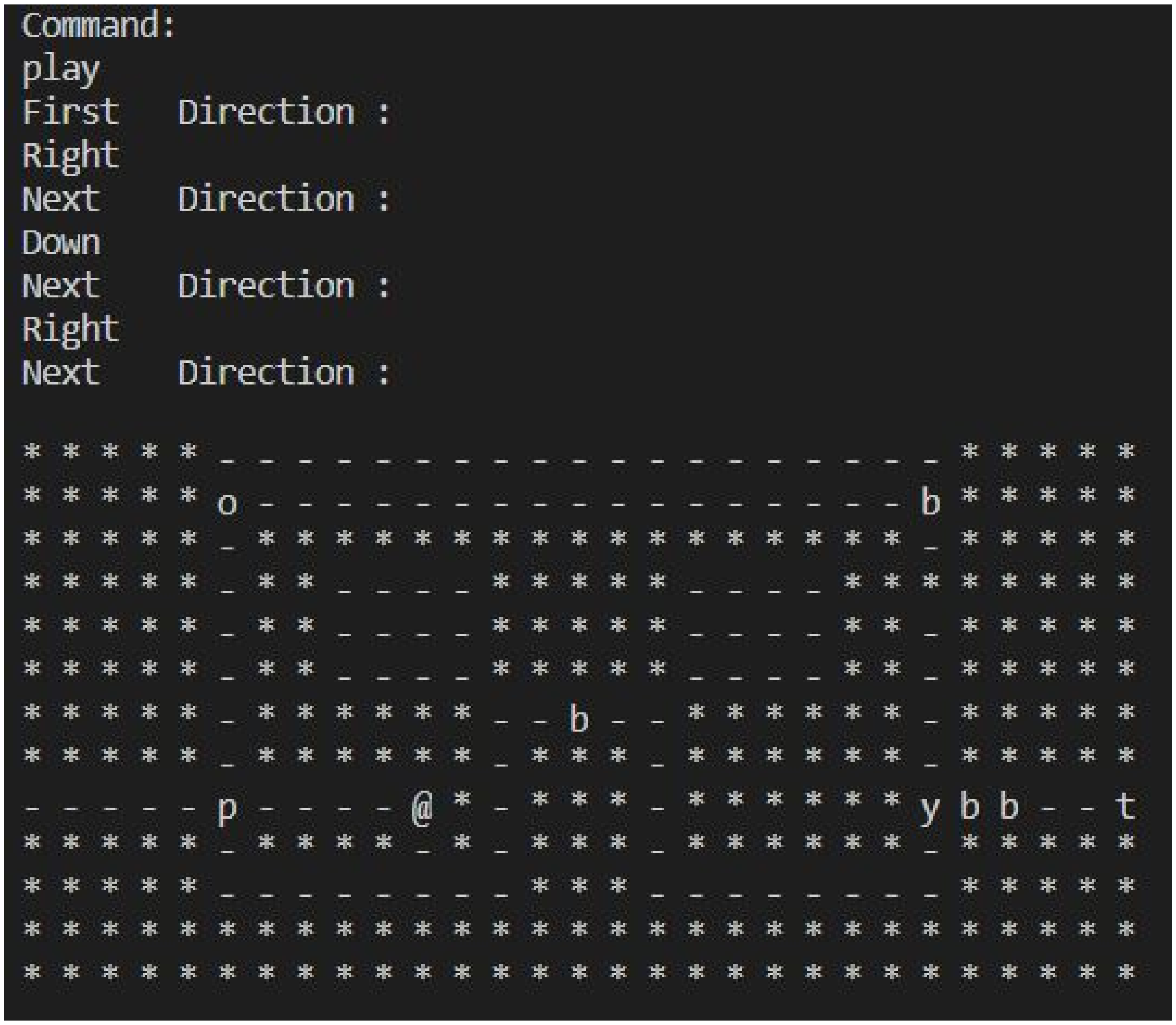
During the interactive mode the user can interact by inputting the set of moves from:

```
Function, Loop{n}{Direction}, Cond{y}{Direction}, Left, Right, Up, Down
```

A user can keep inputting moves and the moves will be executed once the user press the **enter key**. Pressing an invalid command will make the program execute valid inputs that were previously inputted. If the user did not initialize the **Function** definition, then if the user enters **Function** as one of the commands, it will simply be ignored and it will not have any effect on the ball movement.

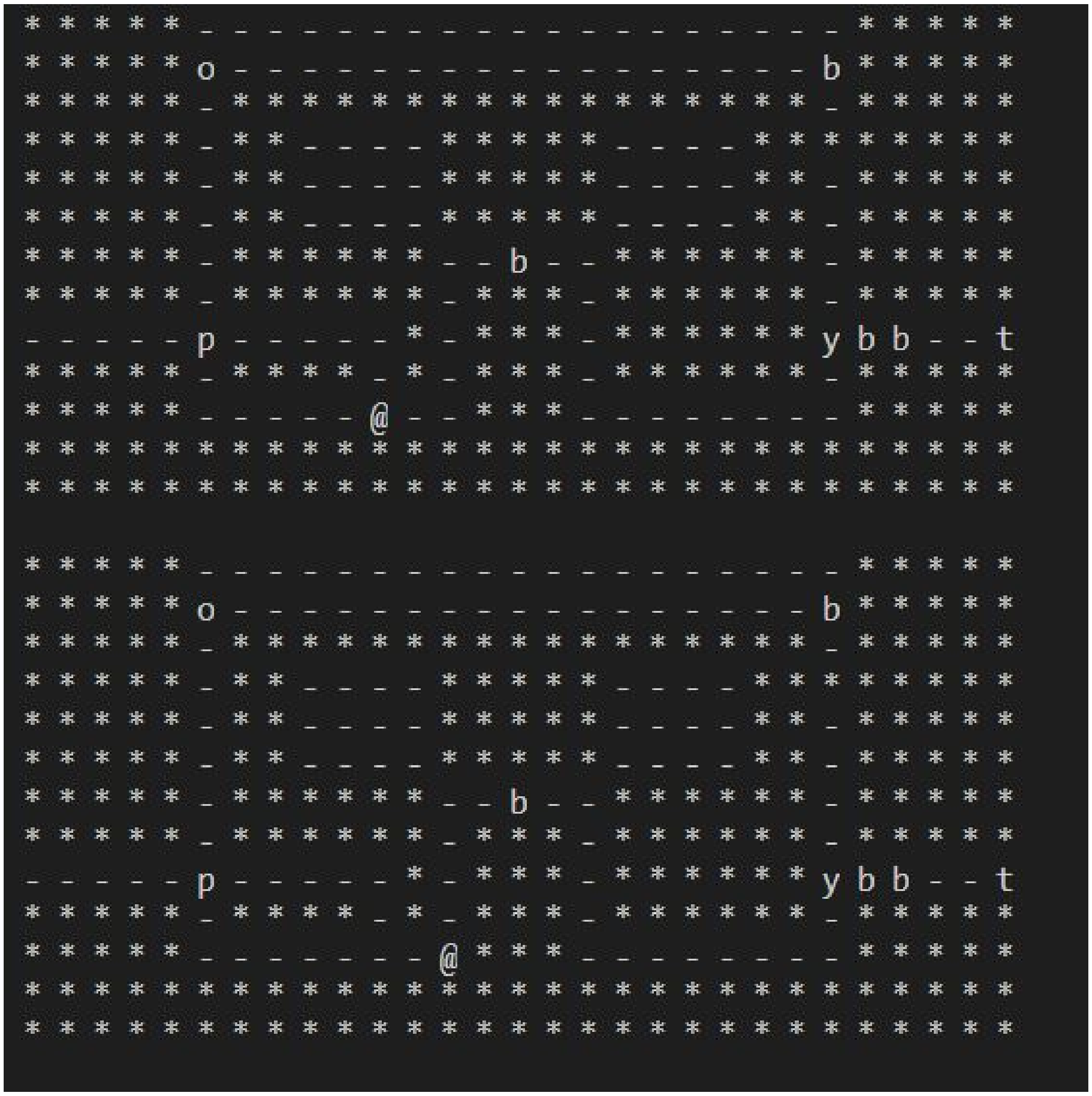
Gameplay Example

The game will display all the positions of the ball for every move it made.



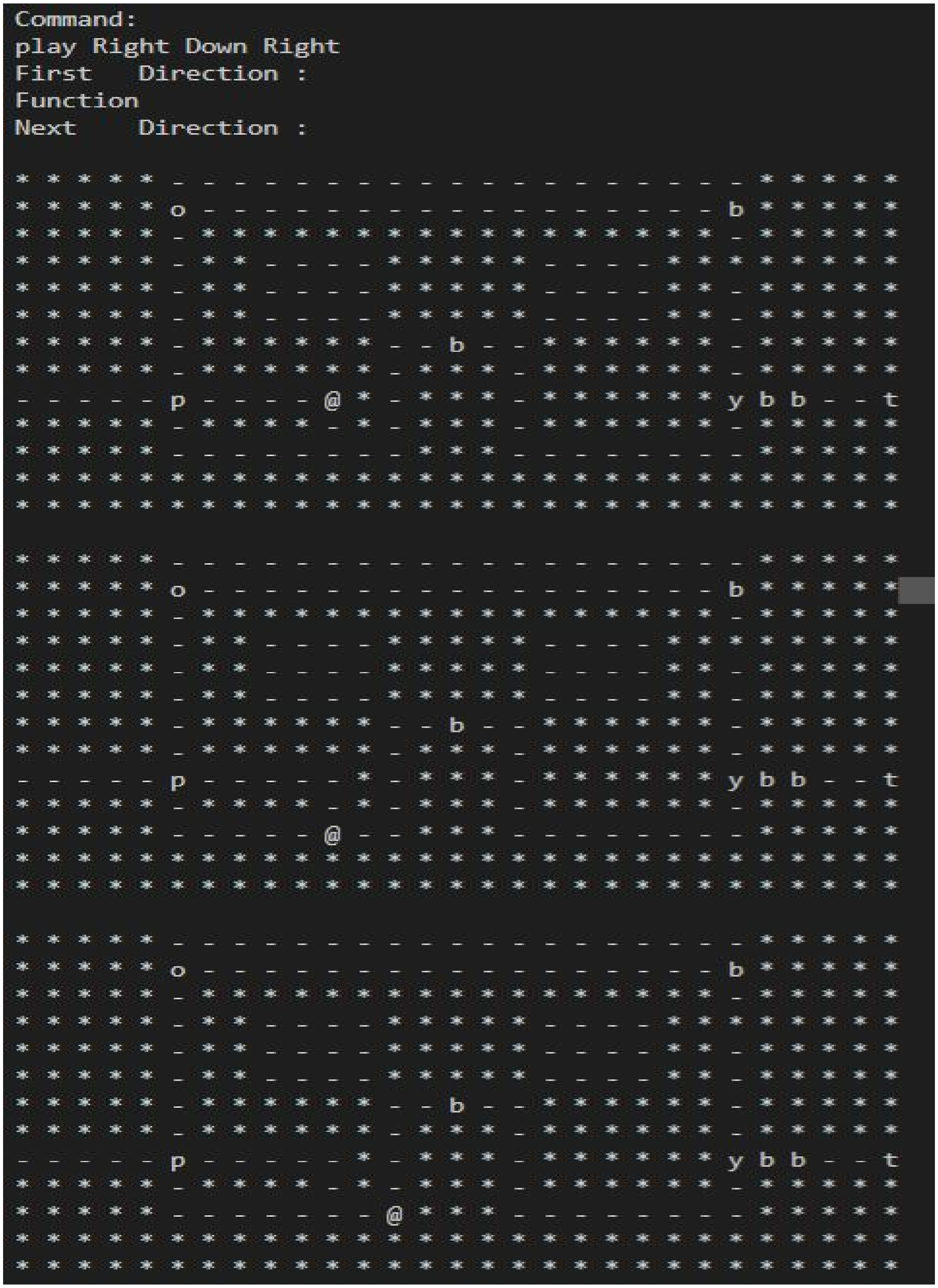
```
Command:
play
First   Direction :
Right
Next    Direction :
Down
Next    Direction :
Right
Next    Direction :

* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * o - - - - - - - - - - - - - - - - - b * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * * * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * * * * * - - b - - * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
- - - - - p - - - - @ * - * * * - * * * * * * y b b - - t
* * * * * _ * * * * _ * _ * * * _ * * * * * * _ * * * * *
* * * * * _ _ _ _ _ _ _ _ * * * _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Figure 5: Play command



```
* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * o _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ b * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * * * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * * * * * _ _ b _ _ * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
_ _ _ _ _ p _ _ _ _ _ * _ * * * _ * * * * * * y b b _ _ t
* * * * * _ * * * * _ * _ * * * _ * * * * * * _ * * * * *
* * * * * _ _ _ _ _ @ _ _ * * * _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * o _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ b * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * * * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * * * * * _ _ b _ _ * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
_ _ _ _ _ p _ _ _ _ _ * _ * * * _ * * * * * * y b b _ _ t
* * * * * _ * * * * _ * _ * * * _ * * * * * * _ * * * * *
* * * * * _ _ _ _ _ _ _ @ * * * _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

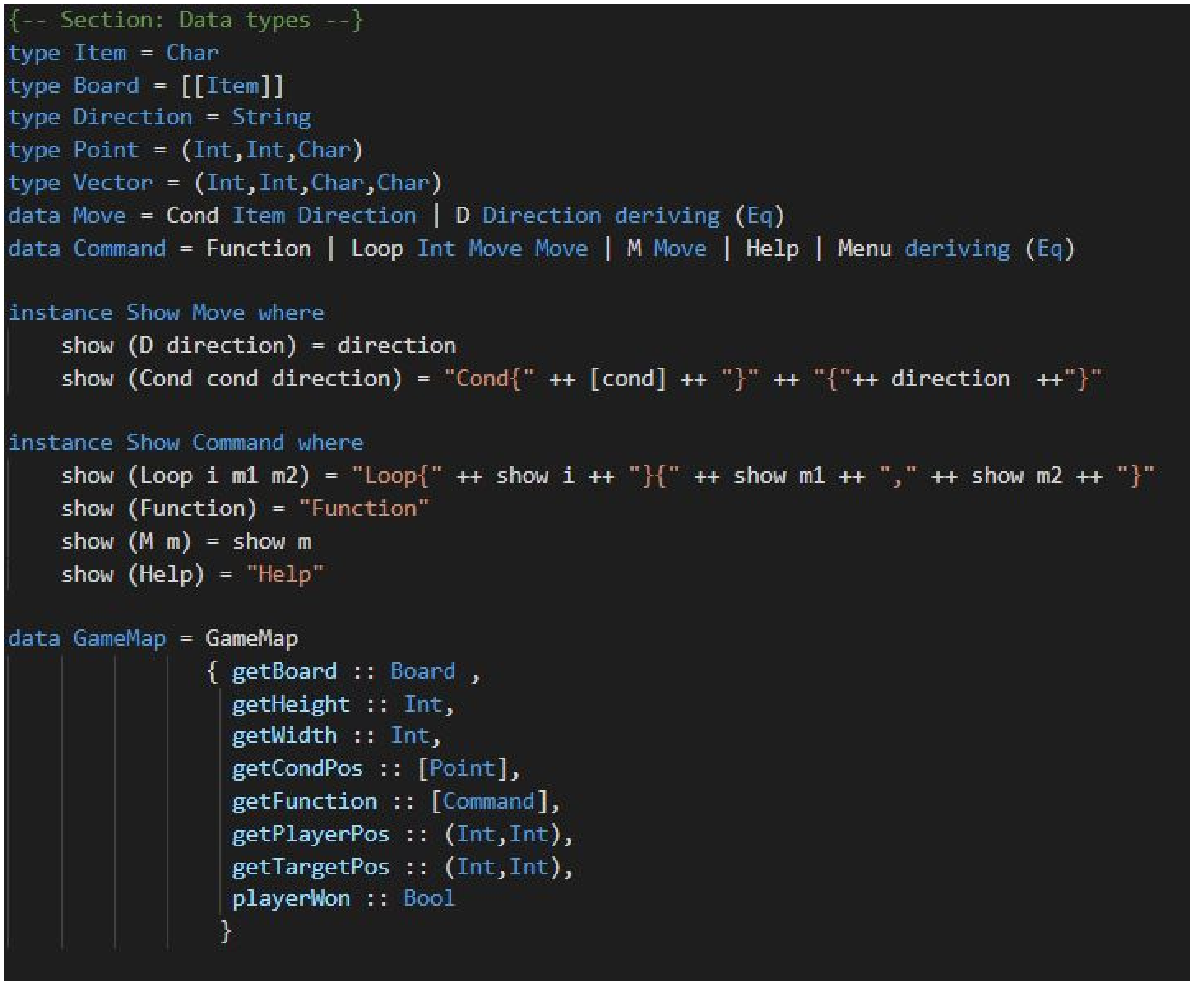
Figure 6: Play command part 2



```
Command:
play Right Down Right
First   Direction :
Function
Next    Direction :

* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * o - - - - - - - - - - - - - - - - - b * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * * * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * * * * * - - b - - * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
- - - - - p - - - - @ * _ * * * _ * * * * * * y b b - - t
* * * * * _ * * * * _ * _ * * * _ * * * * * * _ * * * * *
* * * * * _ _ _ _ _ _ _ _ * * * _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * o - - - - - - - - - - - - - - - - - b * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * * * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * * * * * - - b - - * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
- - - - - p - - - - - * _ * * * _ * * * * * * y b b - - t
* * * * * _ * * * * _ * _ * * * _ * * * * * * _ * * * * *
* * * * * _ _ _ _ _ @ _ _ * * * _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * o - - - - - - - - - - - - - - - - - b * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * * * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * * * * * - - b - - * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
- - - - - p - - - - - * _ * * * _ * * * * * * y b b - - t
* * * * * _ * * * * _ * _ * * * _ * * * * * * _ * * * * *
* * * * * _ _ _ _ _ _ _ @ * * * _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Figure 7: Play Function defined

3 Game Logic and Data Structures



```haskell
{-- Section: Data types --}
type Item = Char
type Board = [[Item]]
type Direction = String
type Point = (Int,Int,Char)
type Vector = (Int,Int,Char,Char)
data Move = Cond Item Direction | D Direction deriving (Eq)
data Command = Function | Loop Int Move Move | M Move | Help | Menu deriving (Eq)

instance Show Move where
    show (D direction) = direction
    show (Cond cond direction) = "Cond{" ++ [cond] ++ "}" ++ "{"++ direction  ++"}"

instance Show Command where
    show (Loop i m1 m2) = "Loop{" ++ show i ++ "}{" ++ show m1 ++ "," ++ show m2 ++ "}"
    show (Function) = "Function"
    show (M m) = show m
    show (Help) = "Help"

data GameMap = GameMap
               { getBoard :: Board ,
                 getHeight :: Int,
                 getWidth :: Int,
                 getCondPos :: [Point],
                 getFunction :: [Command],
                 getPlayerPos :: (Int,Int),
                 getTargetPos :: (Int,Int),
                 playerWon :: Bool
                }
```

Figure 8: Data structures

Data Structures

At the lowest level data structure, we have the **Item** type and the **Board** type as well as the **Direction** type.

The **Item** type is used to refer to the individual items of the board. I am using the textual representation instead of the digit representation for this implementation.

The **Board** type is actually a 2D array of **Item** used to store and define the actual map of the game

The **Direction** is just another alias for the **String** representation of the moves like **Left,Right,Up,Down**.

The **Point** data type and the **Vector** data type is used during optimal path construction. **Point** has **(Int,Int,Char)** format which is representation of the x-y coordinate and the direction for where the ball is going. **Vector** is similar to **Point** but it has an additional **Char** which tells us the element in that coordinate.

The **Move** data type is used to represent the most basic commands such as **Cond{y}{Direction}** and basic **Left,Right,Down,Up**. Using string representation for Conditionals is much more tricky therefore i decided to use a new data type to represent this.

The **Command** data type is the umbrella data type covering all commands. Up to **Function** and **Loops**. The reason i create a new data type for this is because i noticed and understood that all the commands can be broken down into the most basic moves. Therefore it would be much easier for me to execute the moves in the GUI logic just using the most basic moves. Therefore i created a function that would be able to convert the **Command** type into the more basic **Move** type. This way the GUI logic only deals with the most simple moves.

I created a **GameMap** data structure, which allows us to keep track of the current state (important details such as the current player position, board state, etc) of the game and makes it much easier to work with. I also had the idea to do this because i thought that it might be better to store some information rather than calculating them all the time, such as knowing the player position.

Implementation Details

At the very top level. We have an IO action that is used to gather input from the user and redirect the user to a different IO function for every choice that they make.

I am going to try and explain from a top level approach how each of these commands work and how i implemented them.

Load

This command was probably the simplest one to make. At the very core, it is just reading in the text file and extracting the data which we convert to the data structures I defined above. It will create a GameMap data structure and this will contain all the useful information such as the game board, player position, height, width of board, etc. This command also perform some format check for the board. For instance it will not allow to load a map that is not rectangular in nature. It also has some error handling for reading non existing files as well.

You can see the logic in the **loadFile** function.

Check

To implement this, i tweaked the Breadth first search algorithm a bit so that it follows the rules of the game (Not being able to traverse the other directions until it reaches a grass or conditional). Afterwards i started performing a search from the starting position of the ball and continue searching all the paths until i could reach the target position. If there are no paths to the target then the algorithm would eventually terminate and tell us that the board is not solvable.

I implemented this in the **isBoardSolvable** function.

Solve

The logic for this function is to find all the possible paths to the target. Afterwards I would then find the paths which has the most bonus and also the shortest paths. This is probably the most complex feature to build. Because I started by finding the paths with the **basic moves** and **Conditions**, I also had to create a function that would be able to take in a series of those basic moves and convert them to include **Loops** and **Function** so that the final set of commands would be much more efficient and has the least amount of instructions. The way that i find the **Loops** and the **Function** is that i pick up all the possible contiguous subset combinations from the basic commands that could make up a **Function** or **Loop**. Afterwards I try to find which combination will end up with the shortest number of instructions, and thus the most optimal.

defined in **optimalPath** function

Play

For this part, I defined custom parsers to parse the input string from users and turn them into the data types I defined. This will help to easily filter out invalid commands and also makes it much more easier to work with because we can now work with more robust data type instead of string inputs.

After parsing through user inputs, we iteratively execute those commands.

Executing the commands means we have to somehow be able to change the state of the board accordingly. Therefore i have defined low level functions that will allow me to make the ball move up, move left, move right, move down. And then building on top of that I build the game GUI logic that takes in the possibility of meeting a conditional, and the normal basic moves. Because Functions and Loop are basically made out of the same fundamental moves, I was able to reuse the same code to perform the GUI for those commands too.

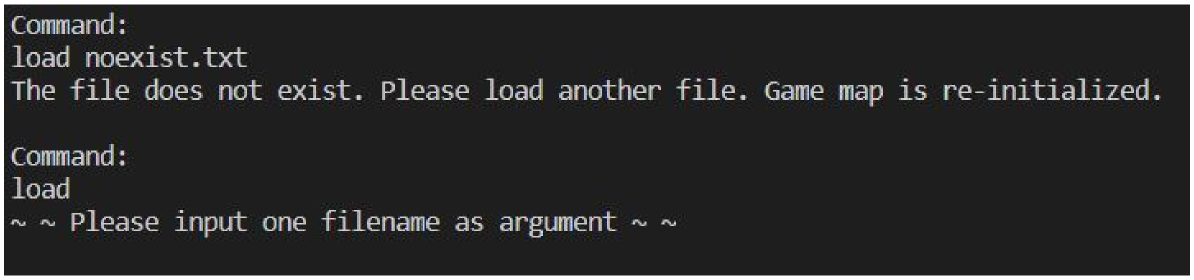
I hope that high level overview gave a brief understanding of how these commands work. Of course in the very minute details, I have created much much more smaller utility functions. For example such as checking whether an element is a grass or a path, or checking whether a particular coordinate is still in range of the board, etc.

4 Error Handling

Loading error handling

When loading a file that does not exist , the program will tell the user that the file does not exist and therefore will reload the game with an empty game map. The program also tells the user to try and reload another file.

The load command also forces the user to take in one argument for the file name.

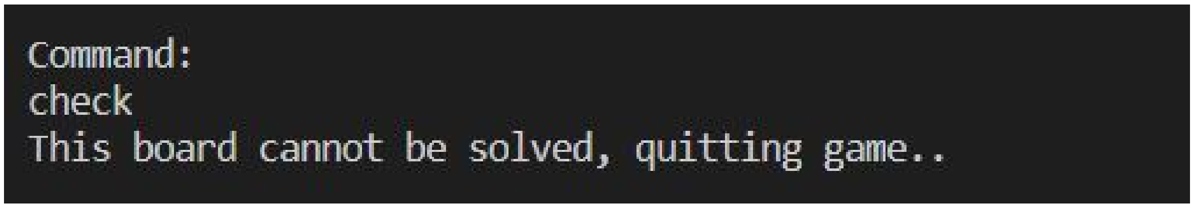


```
Command:
load noexist.txt
The file does not exist. Please load another file. Game map is re-initialized.

Command:
load
~ ~ Please input one filename as argument ~ ~
```

Figure 9: Load command error handling

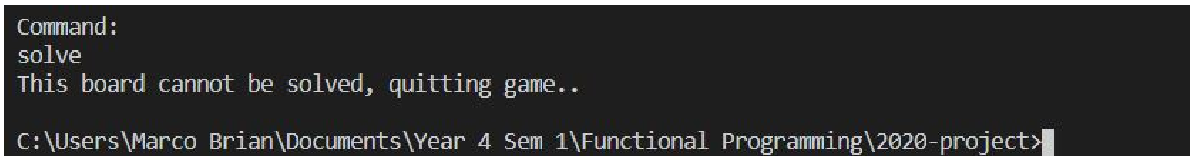
Check and solve error handling

When using the check command, If the game map is not solvable the game will quit. When using the solve command, it also will quit the game if the solution does not exist (meaning the board cannot be solved).



```
Command:
check
This board cannot be solved, quitting game..
```

Figure 10: checking on empty map cause quit



```
Command:
solve
This board cannot be solved, quitting game..

C:\Users\Marco Brian\Documents\Year 4 Sem 1\Functional Programming\2020-project>
```

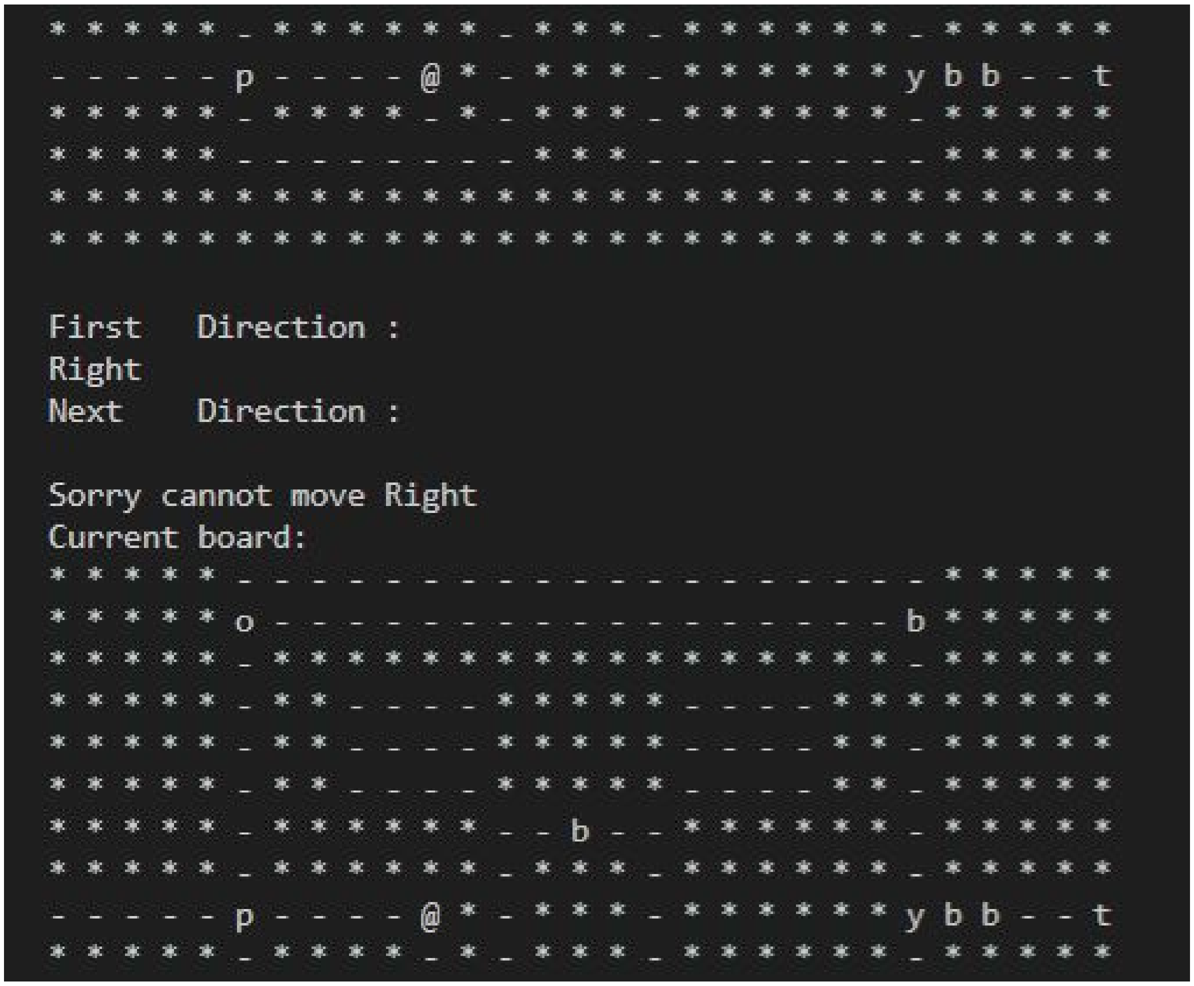
Figure 11: Unsolvable game to make program quit

Play error handling and winning game

When the user enters an invalid move, the game will not read the move, simply ignoring the wrong command. But if the user enters a valid move like **Left** but it hits a grass, the game will not make the ball move and will give a warning to the user that the ball cannot move to the left.

If the user defines a **Function** during play, the user will be able to use the **Function** he defined. But if the user did not do so, the user won't be able to use the **Function** command when inside the interactive mode and if the user types in Function command, the command will just be ignored when the user **presses enter**.

When the ball reaches the target, the game congratulates the user and terminates the program. We know the user has won the game by checking if the ball's position is the same as the target position. If it is the same then we know we have reached the target and the game is won! The game will not terminate if the target has not been reached, but the user can go back to main menu again & save the game (explained more in extra features section).



```
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
- - - - - p - - - - @ * - * * * - * * * * * * y b b - - t
* * * * * _ * * * * _ * _ * * * _ * * * * * * _ * * * * *
* * * * * _ _ _ _ _ _ _ _ * * * _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *


First   Direction :
Right
Next    Direction :

Sorry cannot move Right
Current board:
* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * o - - - - - - - - - - - - - - - - - b * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * * * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * * * * * - - b - - * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
- - - - - p - - - - @ * - * * * - * * * * * * y b b - - t
* * * * * _ * * * * _ * _ * * * _ * * * * * * _ * * * * *
```

Figure 12: Cannot Move Right

5 Extra features

Help/Hint Feature

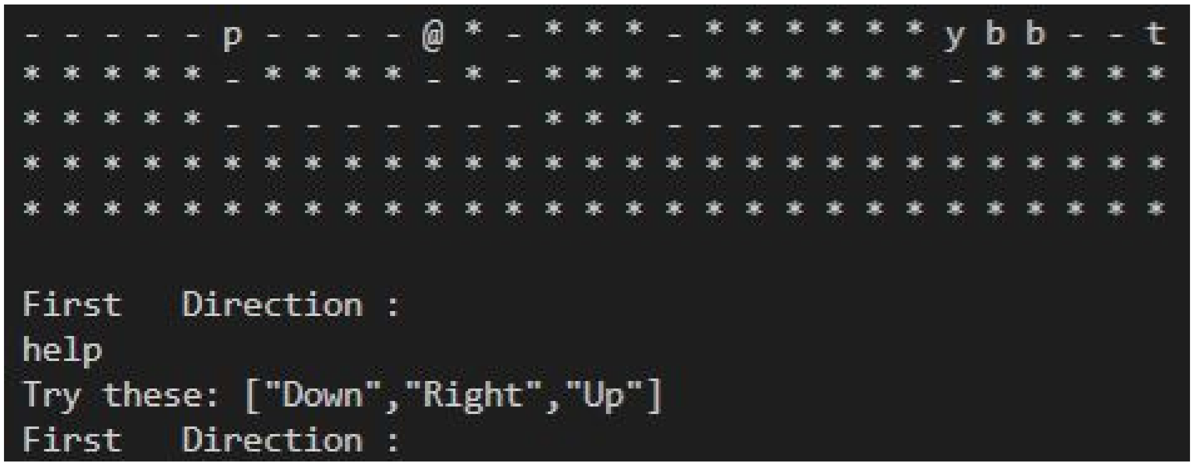
When the user is in the middle of the interactive play and is stuck on how to achieve the goal. The user can enter ”help” or ”Help”, this will make the program generate at most 3 moves that will help guide the user to complete the game. The moves are generated from the optimal search path function and therefore will always guide the user to the best path.

Back to main menu and save

Maybe comes a point where the user is tired of playing but he wants to save the current game state. What he can do is he can have the option to go to the main menu. In the interactive mode, user can press the option ”menu” or ”Menu”. This will bring the user back to the main menu. Then in the main menu, the user then can type in the ”save” command which will ask the user for a new file name. The current map will then be saved in with that file name and next time the user can load the file again to continue where he/she left off.

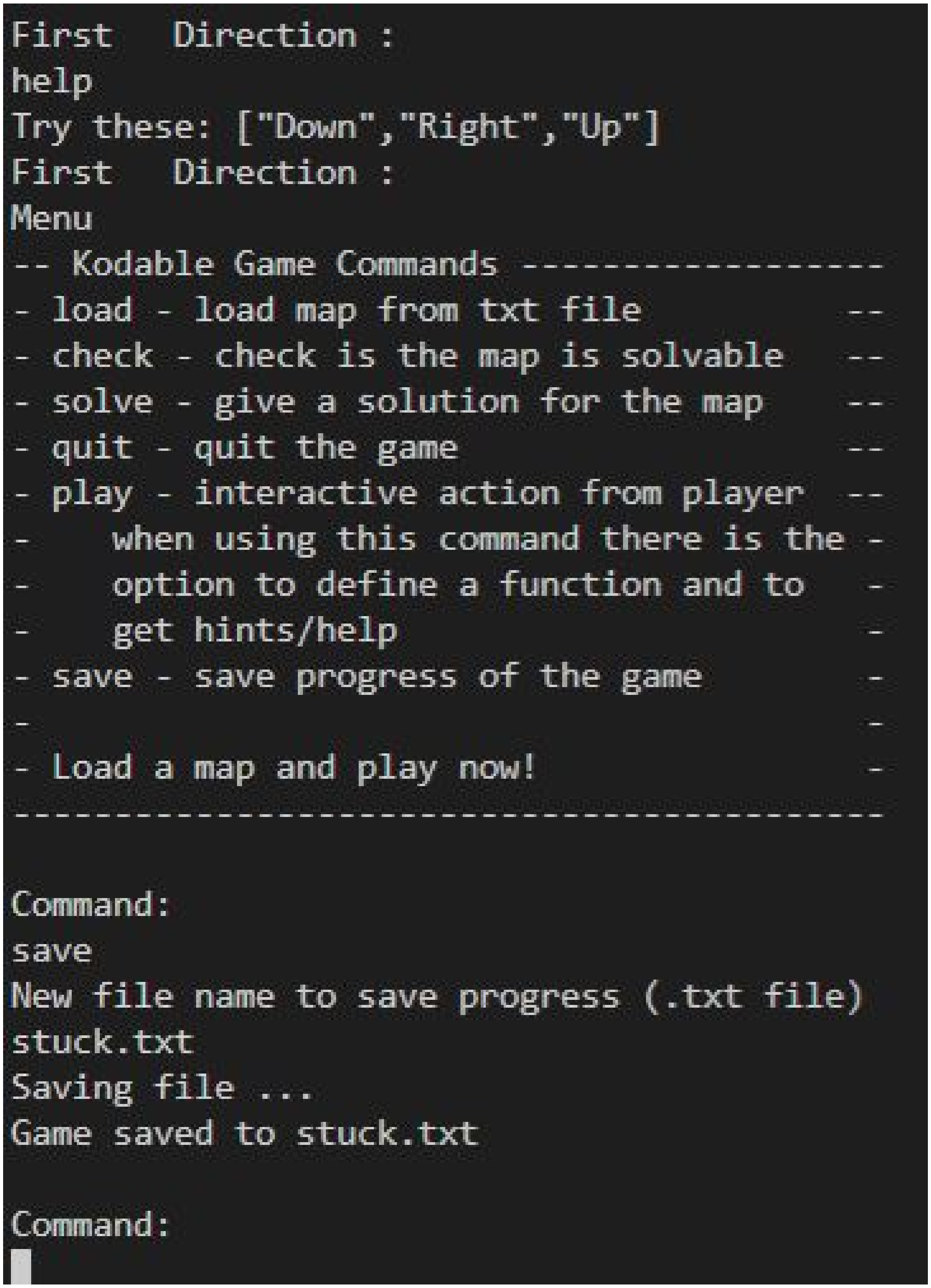
Shortest Path Solution

This option allows user to find the shortest path of the ball in terms of **distance** to the target instead of **the number of direction**. Implements the **bread-first search** algorithm. User can try the command ”bfs” on the main menu.



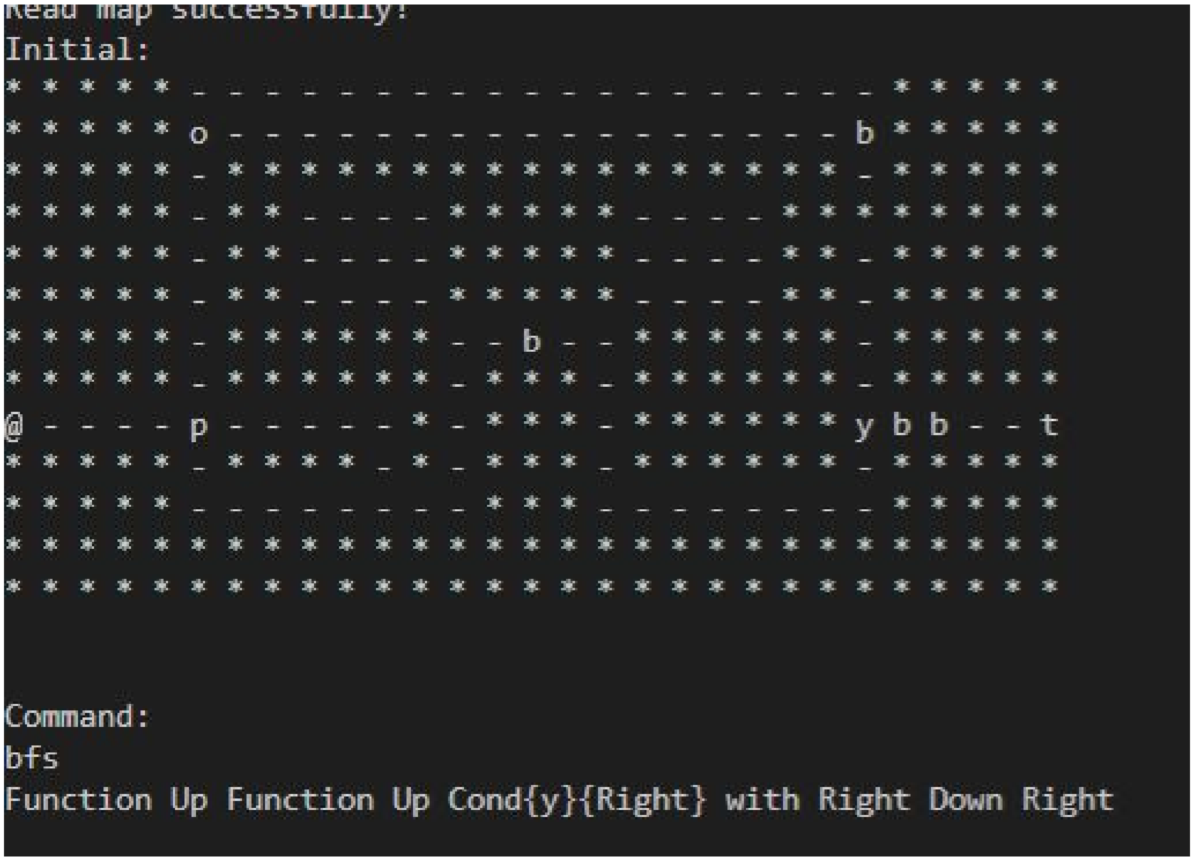
```
- - - - - p - - - - @ * _ * * * _ * * * * * * y b b - - t
* * * * * _ * * * * _ * _ * * * _ * * * * * * _ * * * * *
* * * * * _ _ _ _ _ _ _ _ * * * _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *


First   Direction :
help
Try these: ["Down","Right","Up"]
First   Direction :
```

Figure 13: Help command in interactive mode



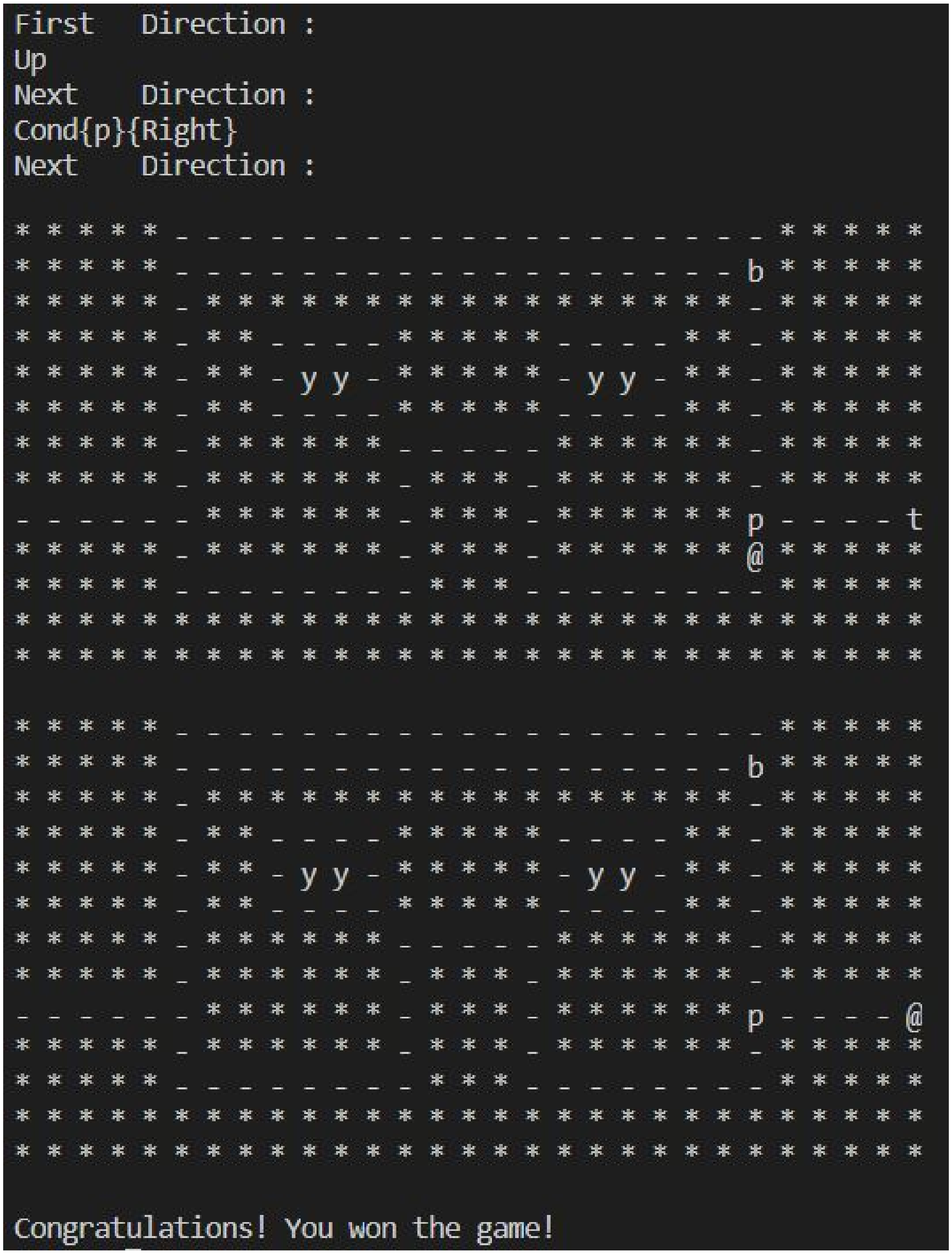
```
First   Direction :
help
Try these: ["Down","Right","Up"]
First   Direction :
Menu
-- Kodable Game Commands -----------------
- load - load map from txt file         --
- check - check is the map is solvable   --
- solve - give a solution for the map    --
- quit - quit the game                  --
- play - interactive action from player  --
-    when using this command there is the -
-    option to define a function and to   -
-    get hints/help                       -
- save - save progress of the game        -
-                                         -
- Load a map and play now!                -
-------------------------------------------


Command:
save
New file name to save progress (.txt file)
stuck.txt
Saving file ...
Game saved to stuck.txt

Command:
```

Figure 14: Go back to menu and save game



```
Read map successfully!
Initial:
* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * o - - - - - - - - - - - - - - - - - b * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * * * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * * * * * _ _ b - - * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
@ - - - - p - - - - - * _ * * * _ * * * * * * y b b - - t
* * * * * _ * * * * _ * _ * * * _ * * * * * * _ * * * * *
* * * * * _ _ _ _ _ _ _ _ * * * _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *



Command:
bfs
Function Up Function Up Cond{y}{Right} with Right Down Right
```

Figure 15: BFS command



```
First   Direction :
Up
Next    Direction :
Cond{p}{Right}
Next    Direction :

* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ b * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ y y _ * * * * * _ y y _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * * * * * _ _ _ _ _ * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
_ _ _ _ _ _ * * * * * * _ * * * _ * * * * * * p _ _ _ _ t
* * * * * _ * * * * * * _ * * * _ * * * * * * @ * * * * *
* * * * * _ _ _ _ _ _ _ _ * * * _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ b * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ y y _ * * * * * _ y y _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * * * * * _ _ _ _ _ * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
_ _ _ _ _ _ * * * * * * _ * * * _ * * * * * * p _ _ _ _ @
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
* * * * * _ _ _ _ _ _ _ _ * * * _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *


Congratulations! You won the game!
```

Figure 16: Winning the game