COMP3258 Functional Programming

Final Project Report Kodable

Kush Baheti (3035436583)

Introduction

The objective of the final project of the course was to create a clone of the game *Kodable* (https://www.kodable.com/). Kodable helps kids learn core programming concepts such as sequencing, conditionals, loops, and functions.

Kodable presents the users with “maps” of increasing complexity. The goal is to help a ball traverse this map and reach the target location, while also collecting all reachable bonus(es). This is done by sequentially executing commands that direct where and how the ball should move.

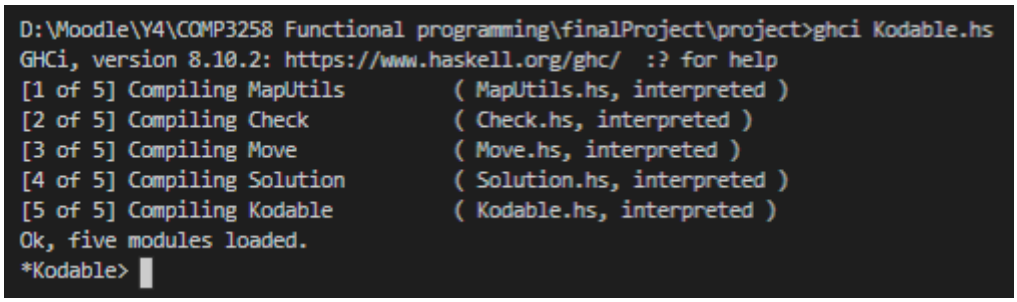
The implementation of the game is in Haskell, and it uses a command line interface to enable game play. The following symbols are used to represent maps on the command line:

@ -> ball
t -> target
b -> bonus
- -> path
* -> wall/barrier
p -> pink
o -> orange
y -> yellow

Building the Game

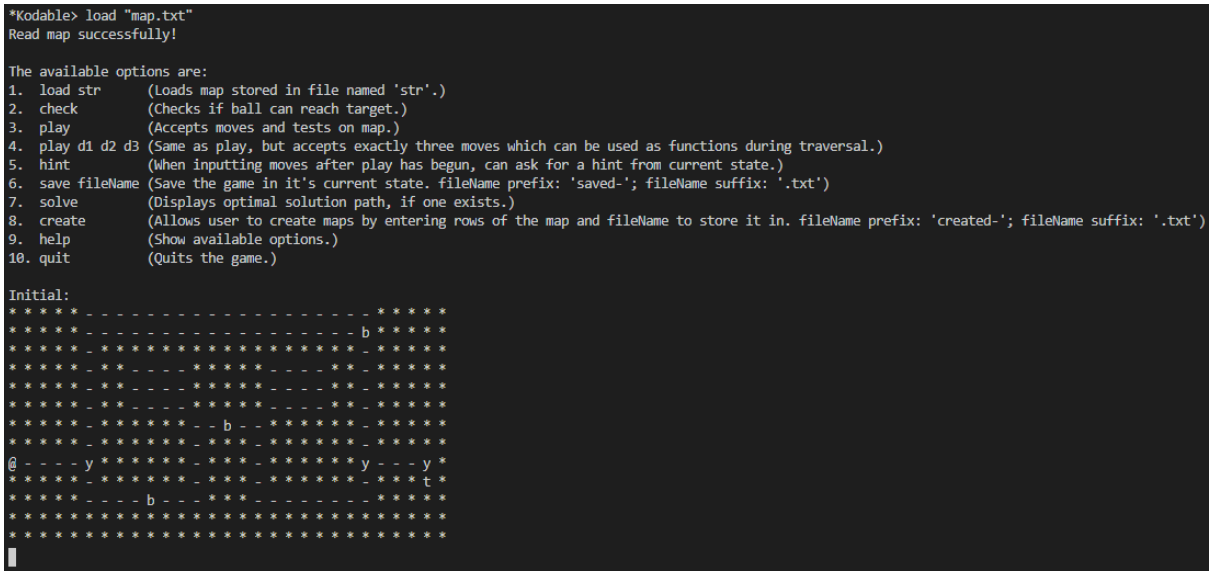
Please follow the steps below to build the game:

1. Ensure the system can run Haskell code. An quick and easy way to get started is to download install from here: https://www.haskell.org/platform/. This includes the **Glasgow Haskell Compiler** and some other tools.
2. Download the project and unzip it. Open up a terminal and navigate to the project directory.
3. Type `ghci Kodable.hs` to build the Haskell files.



Game Interface

Once the game is built, the user can start playing by loading a map. The map must be a *.txt* file that ideally contains symbols described above. A map called *map.txt* can be loaded using the command `load "map.txt"` .



This is what the terminal would look once a valid text file representing a map is loaded successfully. All the game functionality, options and caveats are discussed in the following sections.

Basic Functionality

There are 10 options available to the user, each one of these is discussed in the following section.

Load

Syntax: `load fileName`
Note: when the load command is first used to begin the game, the filename must be wrapped in double quotes, i.e., `load "map.txt"`. If a map is loaded while the game is running, there is no need to use double quotes, i.e., `load map.txt`.

Once the load command is given, the following output is presented to the user (refer to image above):

On reading the file successfully, *"Read map successfully!"* is printed.
Then the user is presented with the 10 available options, and a short description to inform the user what the functionality of each option is (The first of these is `load` itself).

Then *"Initial:"* is printed, which is followed by the map itself.
After this, the flow of control is passed to the start function, and the terminal is ready to take in more commands from the user.

If the user tries to load a non-text file is, then an appropriate error message is shown as follows:

```
*Kodable> load "final.pdf"
Invalid file type. Please load a '.txt' file.
```

Once the map is loaded, the load function checks validity of the map. A map is deemed valid if it contains exactly one ball, exactly one target, and exactly three bonuses.
If these validity constraints are not met, informative error messages are printed, prompting the user to modify the map and reload.

```
*Kodable> load "map.txt"
Read map successfully!
Invalid map. There must be only one ball ('@') on the map.
Quitting game. Please load a new/updated map.
*Kodable>
```

```
*Kodable> load "map.txt"
Read map successfully!
Invalid map. There must be only one target ('t') on the map.
Quitting game. Please load a new/updated map.
*Kodable>
```

```
*Kodable> load "map.txt"
Read map successfully!
Invalid map. Number of bonuses must be exactly 3.
Quitting game. Please load a new/updated map.
*Kodable>
```

If a file is given, and this file does not exist, Haskell throws an exception error as follows:

```
*Kodable> load "non-existent.txt"
*** Exception: non-existent.txt: openFile: does not exist (No such file or directory)
*Kodable>
```

Check

The second available option is `check`. This command checks whether the given map is solvable. A map is deemed solvable if there is a path connecting the ball to the target. The

number of bonuses that can be reached are not used to determine whether map is solvable, i.e., it is assumed that maps where all three bonuses cannot be reached are also solvable.

The following represents the sunny case:

```
check
Checking if solvable, this may take a few seconds...
The map is solvable (ball can reach target). Enter play to begin!
```

The following represents the rainy case:

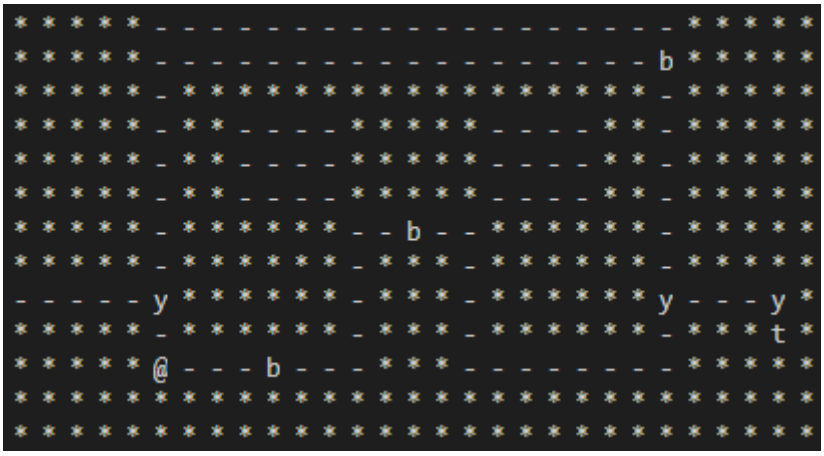
```
check
Checking if solvable, this may take a few seconds...
The map is not solvable. Please try again with a new/updated map.
```

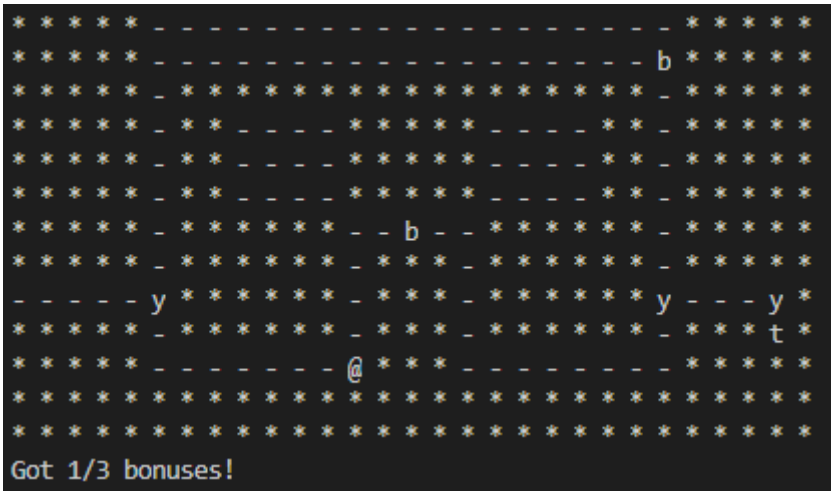
Play

`play` is the most important command in terms of gameplay, this command allows user to start entering directions and to try and accomplish the goal of moving the ball to the target.

The user is prompted to enter the directions one by one, and once the user has entered all directions, the user can simply hit enter to start testing the directions entered on the map.

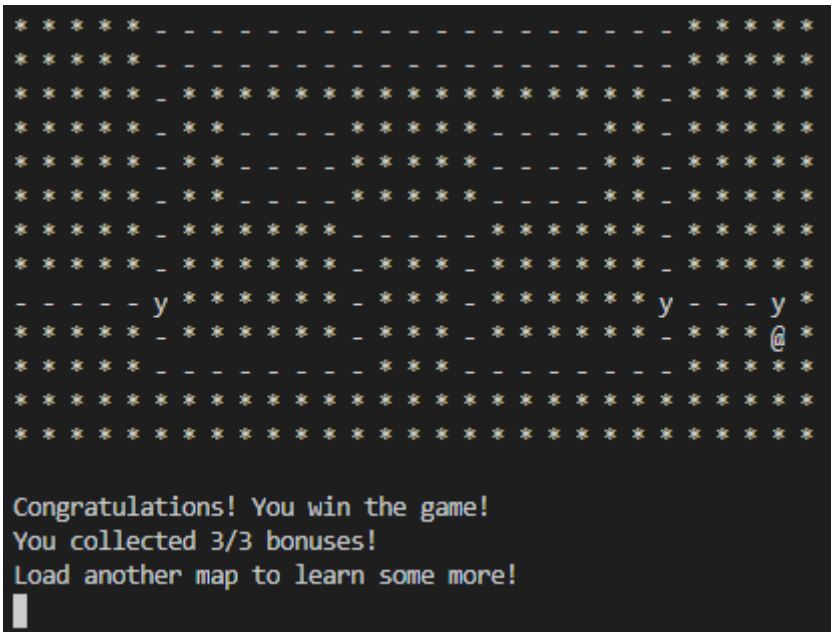
In the image above, we can see that the terminal has started testing and also the new map after that ball has executed the first command, i.e., *Right*. A new map is printed for every direction executed.





The two images above are the states of the map after the second command (*Down*) and third command (*Right*). Note that on execution of the third command, the first bonus is capture/collected, and accordingly, a message is displayed.

After skipping ahead to the final state, where the ball has reached the target, we have the following output:



Here, the user is congratulated on winning the game, provided some basic statistics on how many stars were collected and given the option to enter more commands and continue playing.

The directions entered must strictly adhere to certain guidelines, any violation will lead to an error message being displayed, and the user will have to reload the map and start again.

There are also some special directions the user can enter, such as conditionals, loops and functions. These rules and special directions are discussed below.

The rules the directions must conform to are as follows:

Basic directions

Syntax: (“*Right*”, “*Left*”, “*Up*”, “*Down*”)

Rule: Must be entered in title case, i.e., first letter is capital, the rest are not.

Conditionals

Syntax: Cond{color}{Direction}

Rule: color belongs to (“*p*”, “*o*”, “*y*”) and Direction belongs to (“*Right*”, “*Left*”, “*Up*”, “*Down*”)

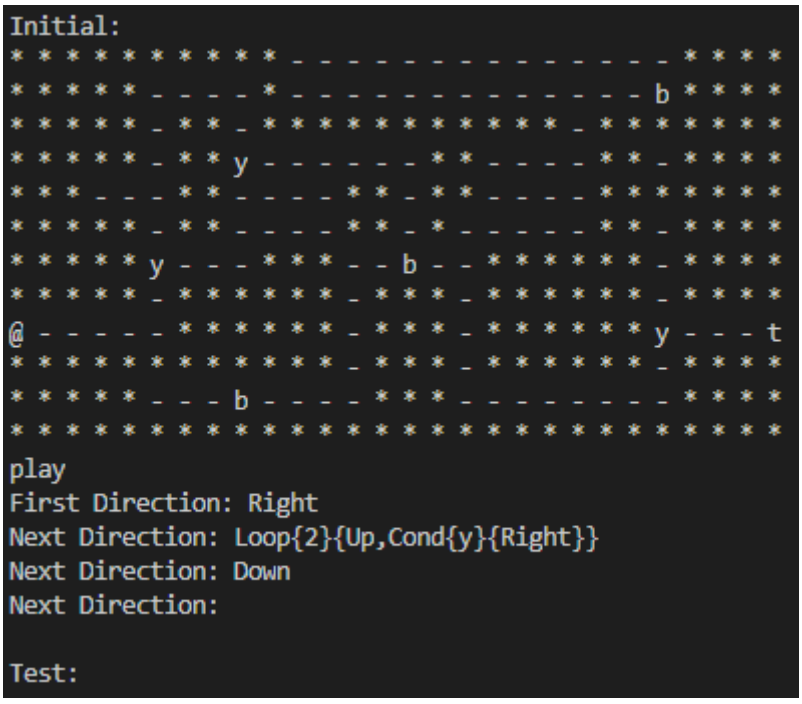
Conditionals allow users to switch directions when they land on a coloured tile of the map.

Loops

Syntax: Loop{itr}{Direction,Direction}

Rule: itr belongs to [0, 5] and Direction belongs to (“*Right*”, “*Left*”, “*Up*”, “*Down*”) and and Loops cannot contain Functions.

Loops allow the user to carry out a pair of directions multiple times.



Example of Loop with Up and Cond{y}{Right}, where execution of all moves collects one bonus and does not reach the target.

`play` can also be used with a slightly different syntax. If three directions are provided along with play, such as, `play direction1 direction2 direction3`, then these three directions are treated as a function.

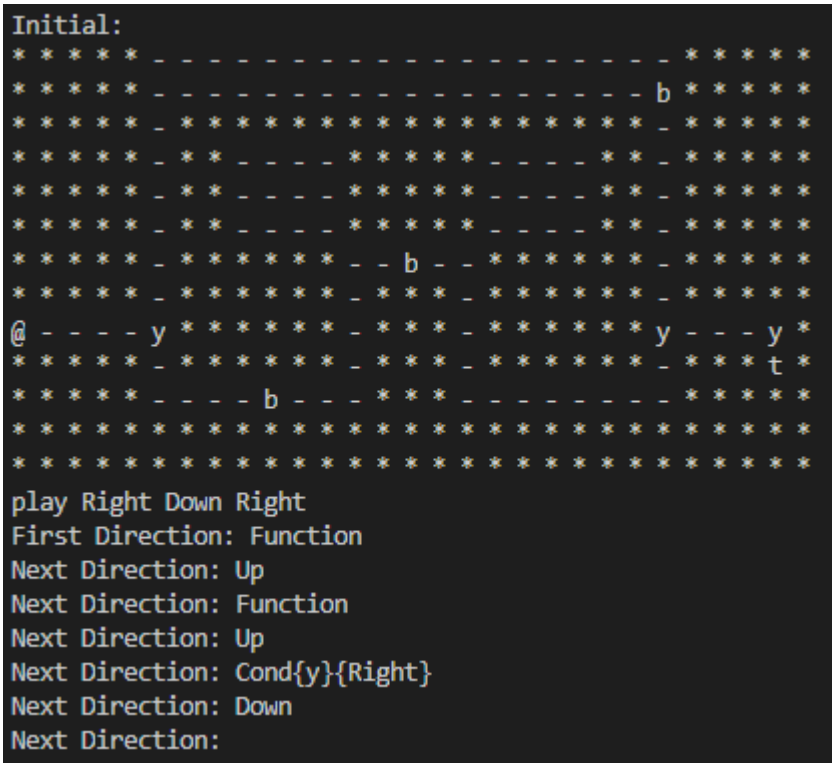
Functions

Syntax: play DirectionCond DirectionCond DirectionCond

Rule: DirectionCond belongs (“*Right*”, “*Left*”, “*Up*”, “*Down*”, “*Cond{itr}{Direction}*”) and Direction belongs to (“*Right*”, “*Left*”, “*Up*”, “*Down*”) and Functions cannot contain Loops.

Functions allow users to carry out a fixed set of three directions multiple times during the game, using a single command.

When these rules are not followed, error messages are displayed:



Example of functon application that where execution of all move collects two bonuses and reaches the target.

```
play
First Direction:
Invalid direction.
Please reload map and start again.
```

Direction cannot be empty.

```
play
First Direction: RIGht
Invalid direction.
Please reload map and start again.
```

Direction did not follow title case lettering.

```
play
First Direction: Cond{x}{Right}
Invalid direction.
Please reload map and start again.
```

Invalid colour given to Conditional.

```
play
First Direction: Loop{6}{Right,Down}
Invalid direction.
Please reload map and start again.
```

Number of Loops can be between 0 and 5 (both inclusive) only.

```
play Right Down Right
First Direction: Loop{2}{Function,Up}
Invalid direction.
Please reload map and start again.
```

Loops cannot have a Function inside them.

```
play Right Left
Invalid command. Please try again.
```

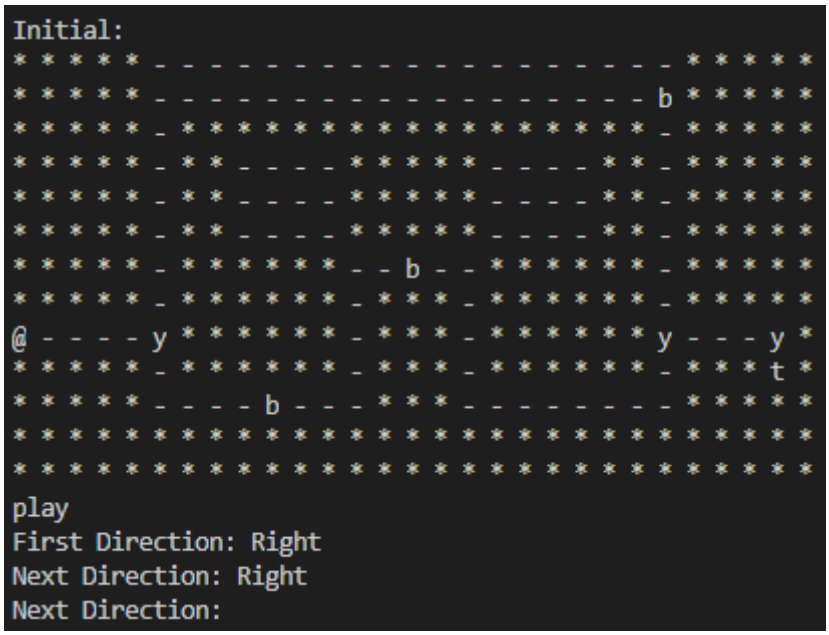
```
play Right Down Left Up
Invalid command. Please try again.
```

Functions need to have exactly three directions.

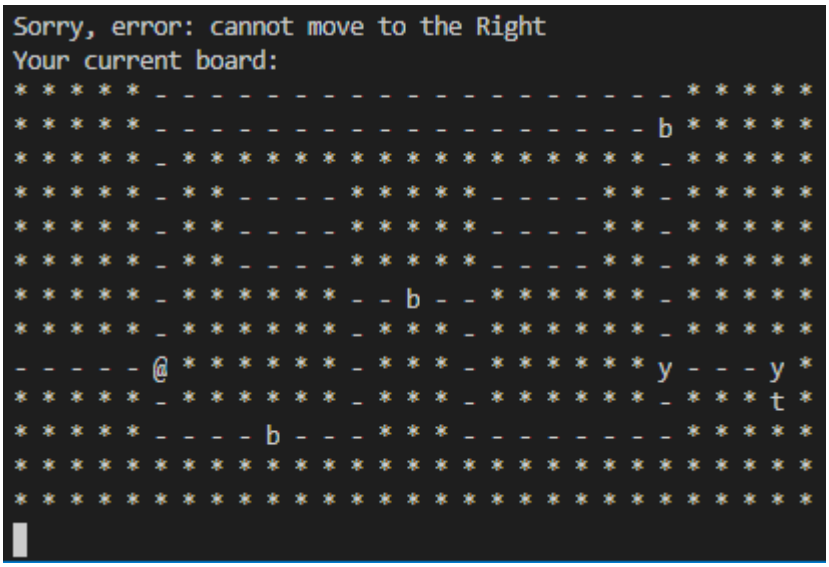
```
play Right Down Loop{2}{Left,Up}
Invalid direction.
Please reload map and start again.
```

Functions cannot have Loops.

Now let us discuss the cases which lead to premature termination of the play command because the user is unable to solve the map.



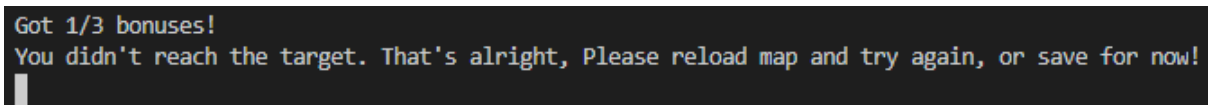
If the user enters a direction but the ball cannot move in that direction because of some obstacle, the following error message is displayed:



Similarly, when the user inputs a Conditional direction for a color that is not present at the current ball location, the following error message is displayed:

Note: this uses the same map as the previous error message example.

Finally, if the user enters valid commands but is unable to reach the target, i.e., loses the game, the following is displayed:



Hint

`hint` is the fifth available option. Since it is a bonus/additional feature, it is discussed in the Additional Features section.

Save

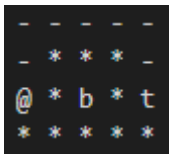
`save` is the sixth available option. Since it is a bonus/additional feature, it is discussed in the Additional Features section.

Solve

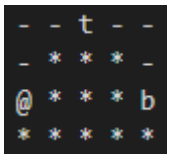
The user is given an option, solve, which prints the most optimal solution to the screen.

The most optimal solution is defined as a solution path that has minimal changes in direction. The optimal path collects all reachable bonuses that can be collected while ensuring the target can still be reached after collecting/capturing these bonuses.

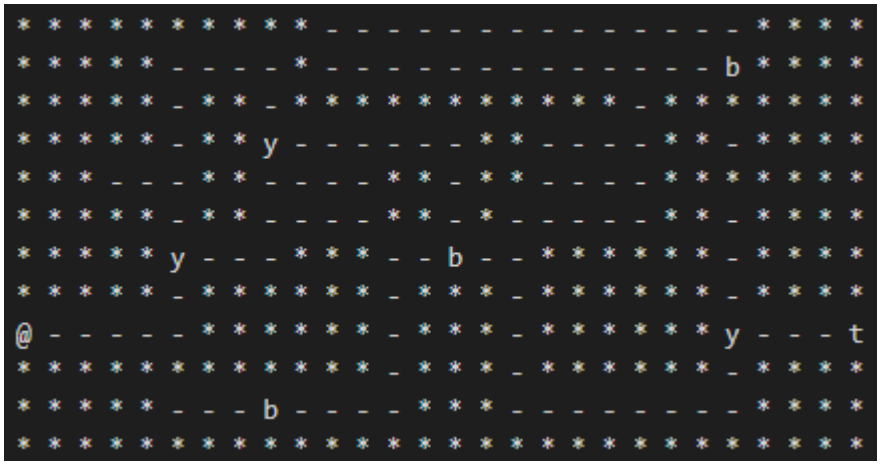
Let us discuss the concept of reachable bonuses. A bonus could be unreachable in the following map scenarios (note: these are example maps, which may be invalid):



No path from ball to bonus.



Path from ball to bonus blocked by target



Ball gets trapped in a certain section of the map in the attempt of capturing/collecting a bonus.

Along with the optimal solution, the solve command also returns a condensed optimal solution. The condensed optimal solution is simply the optimal solution path made even shorted using loops and functions (if possible).

The following represents the sunny case:

```
Initial:
* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * *
* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ b * * * *
* * * * * _ * * * * * * * * * * * * * * * * * _ * * * * *
* * * * * * * * _ _ _ _ _ _ _ * * _ _ _ _ * * _ * * * *
* * * * * _ * * _ _ _ _ * * _ * * _ _ _ _ * * * * * * *
* * * * * * * * _ _ _ _ * * _ * _ _ _ _ _ * * _ * * * *
* * * * * _ _ _ _ * * * _ _ b _ _ * * * * * * _ * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * *
@ _ _ _ _ _ * * * * * * _ * * * _ * * * * * * y _ _ _ t
* * * * * * * * * * * * _ * * * _ * * * * * * _ * * * *
* * * * * _ _ _ b _ _ _ _ * * * _ _ _ _ _ _ _ _ _ * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
solve
Finding optimal solution, this may take a few seconds...
Optimal solution which collects all reachable bonuses and has least change in directions is:
Right Up Right Up Right Down Left Down Left Right Up Right Down Right Up Cond{y}{Right}
Compressed optimal solution:
Loop{2}{Right,Up} Right Loop{2}{Down,Left} Right Up Right Down Function
```

The following represents the rainy case:

```
Initial:
* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
* * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ b * * * * *
* * * * * _ * * * * * * * * * * * * * * * * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * _ y y _ * * * * * _ y y _ * * _ * * * * *
* * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
* * * * * _ * * * * * * _ _ b _ _ * * * * * * _ * * * * *
* * * * * _ * * * * * * _ * * * _ * * * * * * _ * * * * *
@ _ _ _ _ y * * * * * * _ * * * _ * * * * * * p _ _ _ * t
* * * * * _ * * * * * * _ * x * _ * * * * * * _ * * * * *
* * * * * _ _ _ _ _ b _ _ * * * _ _ _ _ _ _ _ _ _ * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
solve
Finding optimal solution, this may take a few seconds...
Optimal solution which collects all reachable bonuses and has least change in directions is:
Not solvable.
Compressed optimal solution:
Not solvable.
```

The logic for finding **optimalSolution** is as follows:

The optimalSolutionUtil function takes a map, the x and y coordinates of the ball on the map, the visited list, which consists of tuples of three, representing the visited x coordinate, y coordinate and the number of bonuses captured when visiting this location, i.e., (x, y, bonusCount), a current solution path, the number of bonuses captured and the target number of bonuses to capture.

The function recursively checks whether the current cell is the target cell and also whether the number of bonuses captured is equal to the number of target bonuses to capture. If this is true, the current solution path is returned as a list of string representing the directions.

If the ball has reached the target but failed to collect the target number of bonuses, then return an empty list as this is a useless path.

If we are on a coloured cell, then recursively find all outcomes if the ball moves in the four directions from this point, and then return these four (or possibly lesser) solution paths.

Similarly, in the otherwise case, the function finds all solution paths if the ball moves in any of the four directions from this point and returns these four (or possibly lesser) paths.

The optimalSolution function is a wrapper function that takes in only a map and returns a single path. This function first checks if all three bonuses are reachable, i.e., if there exists a solution when target bonus count is three. If not, it checks for the condition that target bonus count is two, then one and zero. Since this function receives a list of multiple paths, it uses shortestSolution function to find the shortest and return that.

The logic for **compressOptimalSolution** the path is as follows:

This function takes a solution path and compresses it by adding loops and functions where appropriate. Loops are only added if a pair of directions are executed consecutively two or more times. A function is added when there is a triplet of moves that occurs repeatedly in the solution path or simply at the end to turn the last three moves into a function and reduce length of path by two.

The idea behind this function can be imagined by thinking about the various states and options every direction has at any given state. Given a list of directions, each direction can choose to do one of three things. The first is to abstain from pairing with another direction (to form a loop or function), the second is to form a pair (to form a loop), and the third is to form a triplet (to form a function).

Using this logic, at every point, i.e., for every direction in the solution path, the function creates three branches, corresponding to the three conditions described above. This returns three compressed solution paths, and the shortest amongst these is found using shortestSolution and returned.

Thus, both these algorithms carry out an exhaustive search of all possible permutations using recursion and return the best.

Create

`create` is the eighth available option. Since it is a bonus/additional feature, it is discussed in the Additional Features section.

Help

`help` is the penultimate available option. Since it is a bonus/additional feature, it is discussed in the Additional Features section.

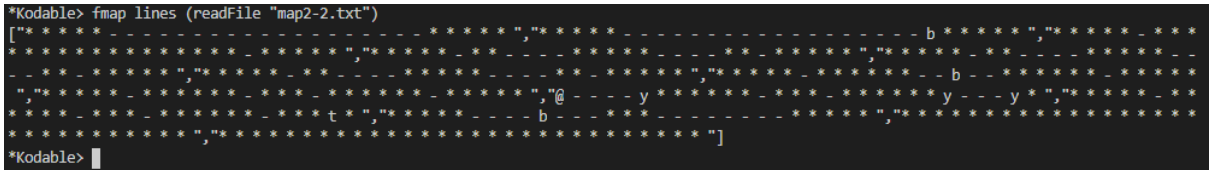
Quit

The final available option is `quit`. This command quits the game.

```
quit
Thank you for playing Kodable, come back soon!
*Kodable>
```

Data Structure

The Kodable map is represented as a list of string, i.e., `[String]`. Each element of this list represents one row of the map. Each row itself is a string that holds the specified symbols (separated by whitespace) to represent the map. An example of the map data structure is as below:



Since a Haskell string is considered a list of characters, representing the map as a list of strings allowed me to treat the map as a two-dimensional list. This is turned out to be a highly convenient and easy to visualise format, which aided in development of algorithms to carry out required manipulations. Using lists also enabled efficient retrieval of rows and columns using indexing (*!!*) and various other useful in-built functions such as *take*, *drop*, *head* etc. Furthermore, apart from retrieval, it also ensured we could easily update the state of the map when the ball moves around and captures bonuses.

Representation as a list also enabled easy read (and write, as discussed in Additional Features section) of files using commands like *readFile* & *lines*, and *writeFile* & *unlines*.

Thus, the data structure of a list of strings was chosen for its simplicity and robustness.

Error Handling

I have attempted to handle various causes of potential errors and exceptions and have described them in the Basic Functionality (above) and Additional Features (below) sections through screen clippings of the terminal. I attempted to create a smooth and friendly user experience by providing as much detail and information as I could in the error messages, to ensure the user is not confused and knows what actions are available and can be taken from this point.

Additional Features

List of all Additional Features

1. **Map Validity**
2. **Save**
3. **Create**
4. **Help**

Map Validity

I have implemented the isValid function to check validity of the map. As described in the section that discussed the `load` command, a map is deemed valid if it contains exactly one ball, exactly one target, and exactly three bonuses. This enforces some degree of sanitation in the user provided input. For example, the ball position is retrieved from the function ballPos. ballPos in turn calls on the function coords to find the ball coordinates. This is returned as a list containing a tuple. The list is assumed to contain a single tuple only (since there is only one ball on the map) and this is destructured using *head*. If this validity check were not in place, then there could be cases where there exist multiple balls, or no ball on the map. In the latter scenario, *head* of the empty list would throw an exception.

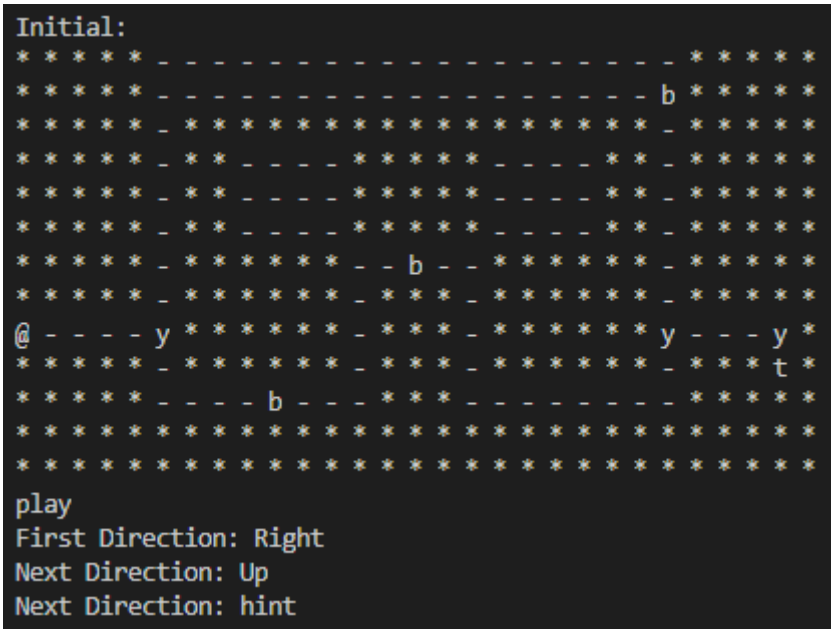
Hint

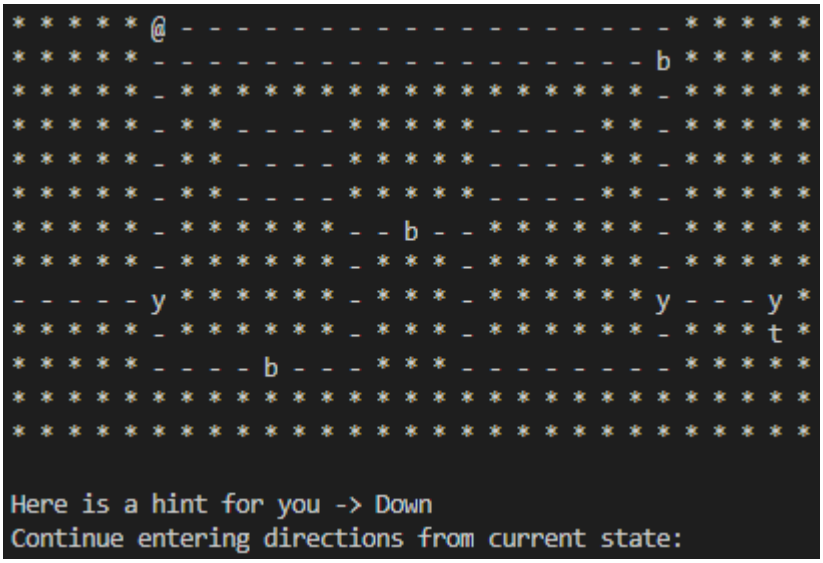
`hint` is one of the Kodable command line options and can be input when the user is entering directions for the ball to move in (after the `play` command). This command finds a move/direction that would help the user optimally solve the board from this map state, i.e.,

the hint would either lead the user towards a reachable bonus that is yet to be captured or the target.

The hint feature was implemented using the optimalSolution function (used for `solve`). The parameters that optimalSolution takes is simply a map, and it returns a list of directions, which is the optimal solution path – one that collects all reachable bonuses and then moves to the target. When the user has input directions after the `play` command, I simulate the movement of the balls according to the directions given. I noticed that this process ensured I have every intermediate map state, i.e., the state of the map after one move is completely executed. This meant that if the input direction happened to be the string "*hint*", rather than an actual direction, I would have everything I needed to find the optimal solution from this point on, the current state of the map. Thus, I was able to call on the optimalSolution function and find the optimal solution from this point. I then extract the very first direction from the solution path and display this to the user.

I felt it would be appropriate to allow the user to ask for a hint during the process of direction input, as it would be highly inconvenient if the user had to exit and restart the inputs from the very beginning. This also allows the user to visualise the ball position after the few moves the user has already inputted (since the updated map is printed after every move), helping give a better understanding of the hint and also sequencing. This also allowed for reusability of code.





Save

I implemented `save` as another additional feature. It is yet another Kodable command line option. The thought process behind this was that Kodable is a game, and one of the basic functions of any game is the ability to save your progress and come back and start from where you left off instead of restarting (which would be highly inconvenient, and a bad user experience). `save` allows the user to save the map in its modified state, i.e., after the user has played and made some moves on the map. This map can then be loaded at any point in the future and the user can continue playing from the saved stage.

This feature was implemented using a function that is a mirror image of the load function of sorts. The map, which is a list of strings, is converted into file contents using the function *unlines*. This content is then saved into a file using the *writeFile* method.

One caveat to this feature is that if the user has played and made some moves, it is possible that the map now has less than three bonuses. Thus, when the user tries to load this map in the future, the isValid function would reject it. To avoid this, I have imposed a condition that all maps being saved by the user using the save command must have a prefix "*saved-*" and must also be a text file, i.e., have a suffix of "*.txt*". In the load function, I simply check whether the map being loaded is a saved map, and if true, the load function skips the validation step. (One drawback to this implementation is that there is no safeguard if a saved file is manually changed and made invalid.)

```
Got 1/3 bonuses!
You didn't reach the target. That's alright, Please reload map and try again, or save for now!
save wrongname.txt
Invalid file name/type. File name should begin with 'saved-' and end with '.txt'
Enter new file name below:
saved-map.txt
Game saved succesfully!
```

saved-map.txt

```
1   * * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
2   * * * * * _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ * * * * *
3   * * * * * _ * * * * * * * * * * * * * * * * * * * * * _ * * * * *
4   * * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
5   * * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
6   * * * * * _ * * _ _ _ _ * * * * * _ _ _ _ * * _ * * * * *
7   * * * * * _ * * * * * * * _ _ b _ _ * * * * * * _ * * * * *
8   * * * * * _ * * * * * * * _ * * * _ * * * * * * _ * * * * *
9   _ _ _ _ _ y * * * * * * * _ * * * _ * * * * * * y _ _ _ y *
10  * * * * * _ * * * * * * * _ * * * _ * * * * * * _ * * * t *
11  * * * * * _ _ _ _ b _ _ _ * * * _ _ _ _ _ _ _ @ * * * * *
12  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
13  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
14
```

Create

I felt it would be very interesting if the user were allowed to create their very own maps. This would allow them to satiate their curiosity and create wonderful maps that allow them to explore various special conditions and edge cases, and deepen their knowledge and understanding, apart from being a fun activity. I accomplished this functionality with the `create` command, which is one of the last Kodable command line tools I will discuss.

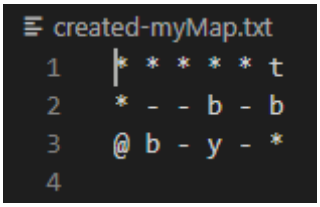
To implement this, I created a function that initially takes and empty list and the integer (-1). This function then checks the value of the integer and if (-1), if prompts the user to enter the rows one at a time. The row input by the user is received using *getLine* and the function is recursively called until the user has input all rows and also the file name to save the created map. The last value is extracted as the file name, some simple validation is carried out and the map along with the file name is sent to the save function.

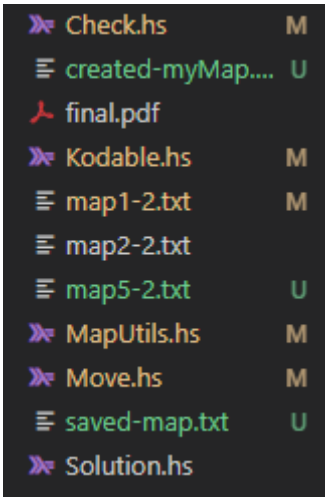
Similar to save, the create function also imposes a restriction on the file name. the name must be suffixed with "*created-*" in order to differentiate from saved files, so that they are validated on loading. They also have the second restriction of being a text file.

```
create
Enter rows of map one at a time, and finally, enter the file name.
* * * * * t
* - - b - b
@ b - y - *
myMap.txt

Invalid file name. Created map files must have suffix 'created-'.
created-myMap.txt

Game saved succesfully!
```





Help

Finally, I will discuss the `help` command, that simply displays the options panel for the suer so that the user knows what actions can be taken form this point onwards.

```
help

The available options are:
1.  load str      (Loads map stored in file named 'str'.)
2.  check         (Checks if ball can reach target.)
3.  play          (Accepts moves and tests on map.)
4.  play d1 d2 d3 (Same as play, but accepts exactly three moves which can be used as functions during traversal.)
5.  hint          (When inputting moves after play has begun, can ask for a hint from current state.)
6.  save fileName (Save the game in it's current state. fileName prefix: 'saved-'; fileName suffix: '.txt')
7.  solve         (Displays optimal solution path, if one exists.)
8.  create        (Allows user to create maps by entering rows of the map and fileName to store it in. fileName prefix: 'created-'; fileName suffix: '.txt')
9.  help          (Show available options.)
10. quit          (Quits the game.)
```

Submission

The following files are included in my zip file for submission

1. Kodable.hs
2. MapUtil.hs
3. Move.hs
4. Check.hs
5. Solution.hs
6. map1-2.txt
7. map2-2.txt
8. map3-2.txt
9. saved-map.txt
10. created-map.txt