The University of Hong Kong

COMP3258: Functional Programming

# Final Project (Kodable Game)

**Deadline: 23:55, Dec 21, 2020 (HKT)**

---

## 1 Kodable

In this project the goal is to code a clone of a children game called *Kodable*. With Kodable kids learn core programming concepts such as sequencing, conditionals, functions, and loops. In essence the game encourages logical thinking. Here is the website of Kodable:

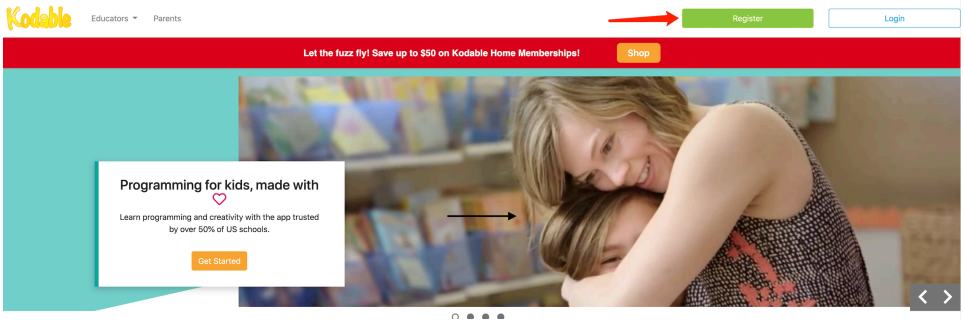
https://www.kodable.com

If you have an apple device (IPad or IPhone) you can try to download the application via the app store. You can also try the game directly in the web browser otherwise (after registering). The game is paid, but there is a free trial for a few days. So you can try it out for free for 7 days.

## 2 Game Introduction

We will explain the game next. Thus you do not have to install it to understand how the game works. However installing it may give you some better feeling for the game and may help you to think about new features for your clone.

In the website version you can just click register, if you want to try it.




In the iphone version you also have the option to login or register.

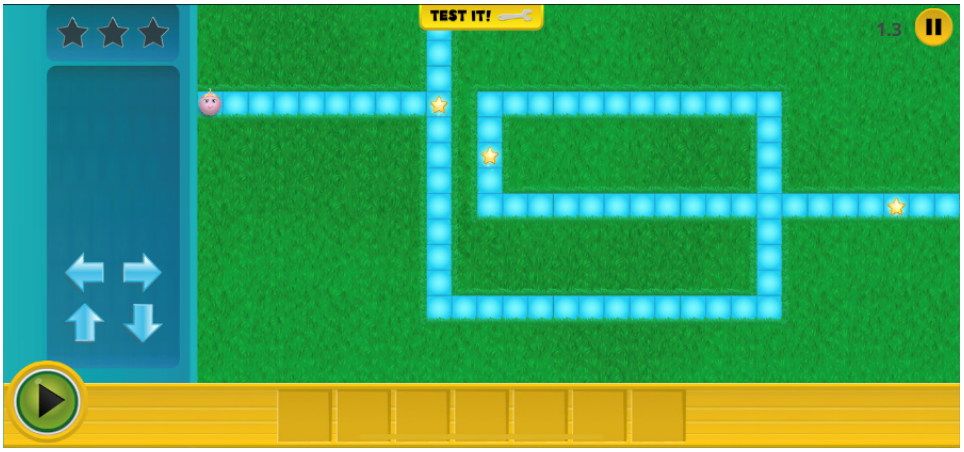

Let's start our Kodable journey!

There are many small games in the Kodable, let's play the Smeeborg game:



**Simple map**

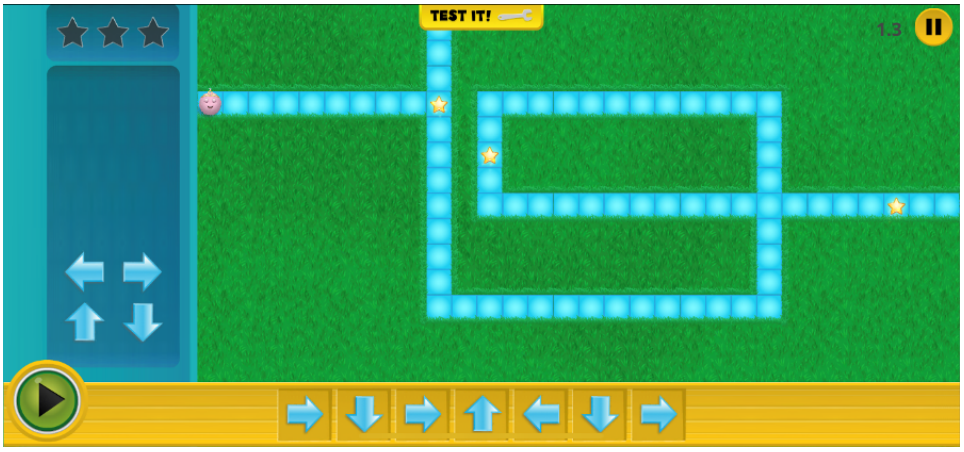
At first, you will start with a simple puzzle. The aim of this puzzle is to teach about the notion of sequencing to kids. That is executing commands in sequence. There is a small ball (on the left, at the start of the maze), some empty blue boxes, three stars and some arrows as shown in the picture:




The goal is to get the ball rolling to the target point (at the right-side of the screen). You have to choose the directions the ball is going to roll. Once you choose the direction, the ball will roll towards that direction and only stop and change to the direction that you choose, when it encounters an obstacle (the grass). From start point to the target point, the directions which the ball could change are limited. The number of the boxes at the bottom of the screen (7 boxes in this case) indicate how many directions can be given to the ball. The three stars are bonuses, if the ball rolls through the stars, you could get the bonus.

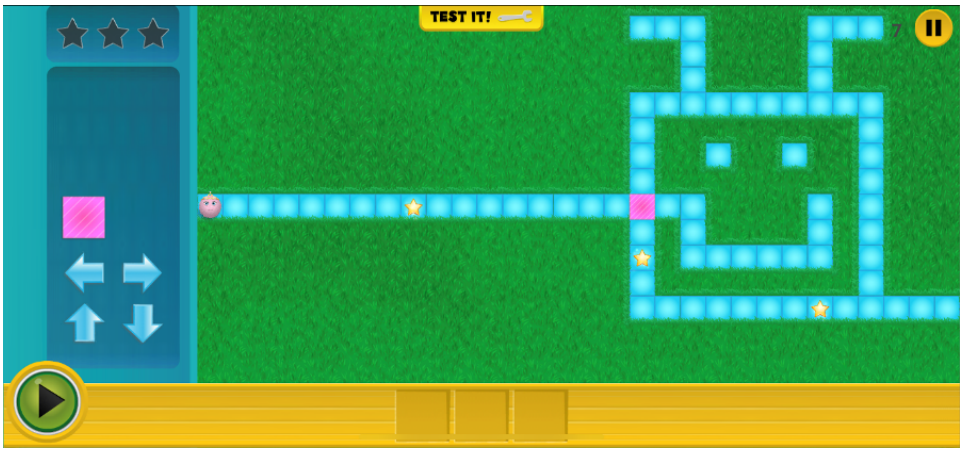
You should drag the arrows (on the left side of the screen) to the boxes. After you put the directions (arrows) into the boxes and click on the play button, the game will test the correctness of your directions choice and how many bonuses you get.

The next image shows a solution to the puzzle. The boxes at the bottom have been filled with arrows (right, down, right, up, left, down, right) which, when executed, will lead the ball to roll to the final target at the right side.



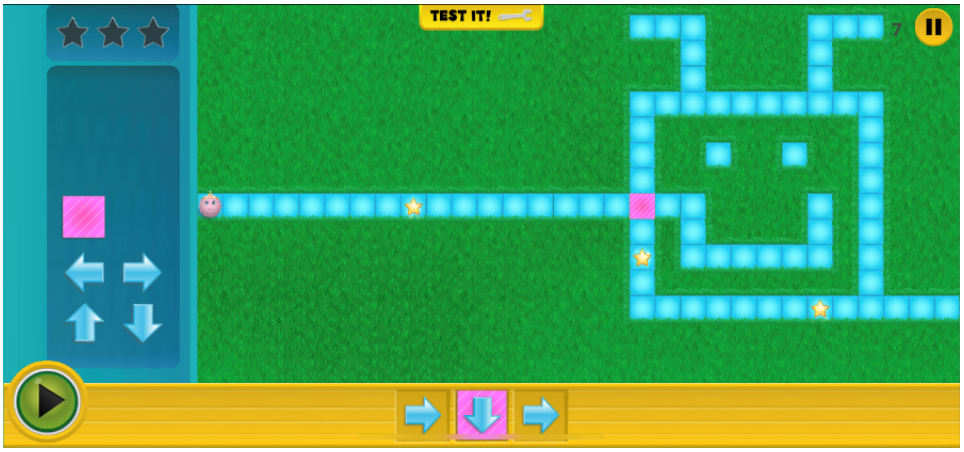

**Color blocks (conditionals)**

Sequencing is the first and most basic concept that is presented to children. At the next stage children learn about the notion of conditionals. The game will add color blocks to your map afer you have passed through the simple maps. The color block act as the condition. As we know above, the ball will stop and change to the direction that you choose, only when it encounters an obstacle. But if the ball roll to the color, it will change to the direction which you label in the color block.



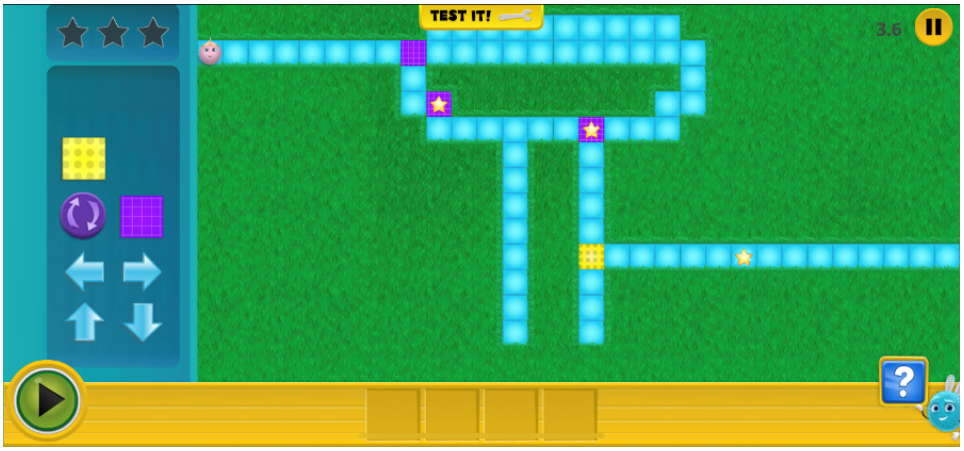

You should drag the arrows to the boxes. For the color block, you need to drag the color block to the box first and the drag the direction in the color block. The system will test the correctness of your directions choice and how many bonuses you get.

A solution for the puzzle with the conditional box is shown in the next image. In this case, we first instruct the ball to go left and when it encounters the pink block (the conditional), turn down, and finally move right.



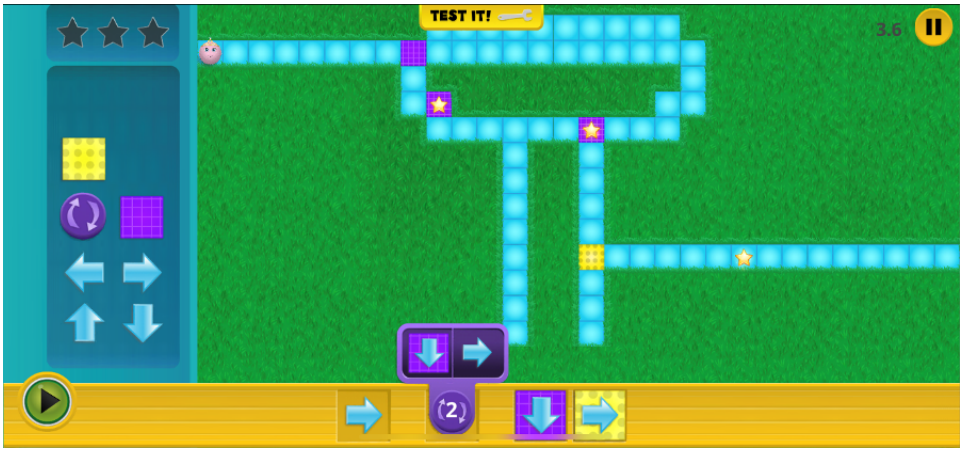

**Loops**

After you pass through the color block maps, the next feature are loops. Loops are added to reduce repetition. A loop in Kodable should just contain two directions. And one loop cannot iterate more than 5 times.



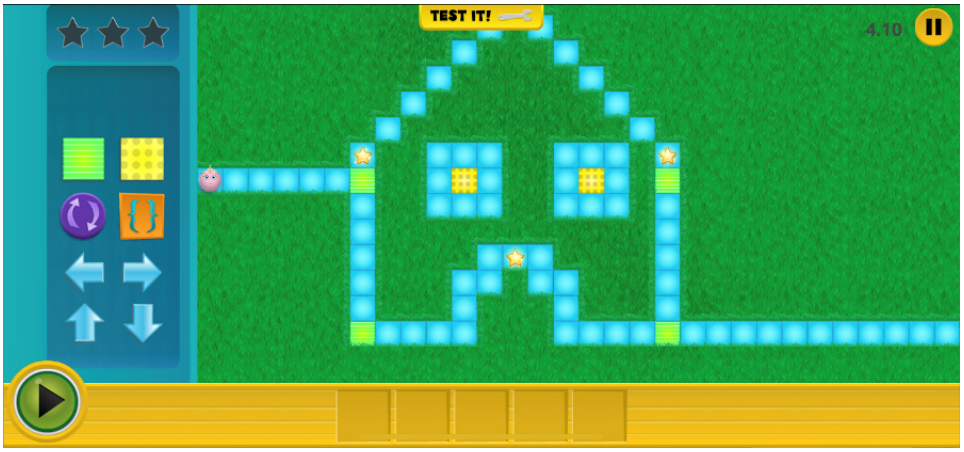

The way to use a loop is just to drag the loop block to the box and the loop will generate two blank boxes and you should drag two directions you want to these two boxes. The default loop counter is 0, but it can be increased or decreased (upto 5). You can click the number to increase or decrease. The system will test the correctness of your directions choice and how many bonuses you get.

The next picture illustrates the loop feature in Kodable.



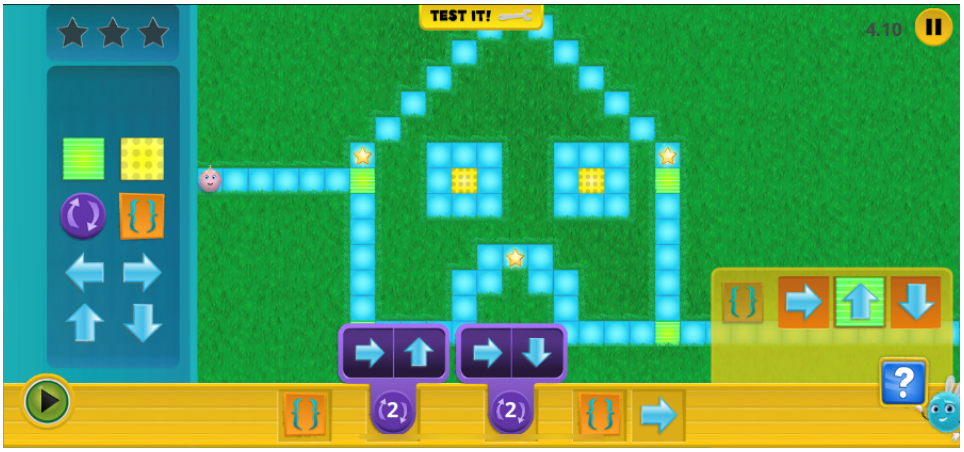

**Functions**

Finally, Kodable supports function blocks. For function blocks, you can include three directions and you cannot use loops in the function.




The way to use function is that you drag the function block to the box and the function will generate three blank boxes and you should drag three directions you want to these three boxes. The system will test the correctness of your directions choice and how many bonuses you get.

The next image illustrates functions in Kodable (see the right side of the screen, near the bottom).




## 3 Description of the Project

In this project, you are required to **design and implement** an **interactive** Kodable game. Obviously the GUI aspects of the actual Kodable game would involve alot of work, so the goal of the project is to focus on the core game logic. We will focus on a text version of Kodable (but, if students wish to, they can develop a more sophisticated GUI).

Our Kodable clone game has a map with the path which means the ball could roll on the path and some of the tiles of the path have special color, like pink, orange, and yellow.

Maps are represented with text. For example, a map might look like:

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

**Meaning of the textual characters**

‘@’ represents the ball.

‘-’ represents a path block that the ball could roll on.

’*’ represents the grass (obstacles).

‘p’ represents the tile of the path’s block color is the special color : pink.

‘o’ represents the tile of the path’s block color is the special color : orange.

‘y’ represents the tile of the path’s block color is the special color : yellow.

‘b’ represents the bonus (the stars in Kodable).

‘t’ represents the target point.

Another way to represent the map (like the above) is to use digits to represent blocks:

```
11111000000000000000000011111
11111500000000000000000511111
11111011111111111111111011111
11111011000011111000011011111
11111011044011111044011011111
11111011000011111000011011111
11111011111100500111111011111
11111011111101110111111011111
60000011111101110111111200007
11111011111101110111111011111
11111000000001110000000011111
11111111111111111111111111111
11111111111111111111111111111
```

**Meaning of the characters with the alternative representation**

‘0’ represent the path that the ball could roll.

‘1’ represent the grass.

‘2’ represent the tile of the path’s color is the special color : pink.

‘3’ represent the tile of the path’s color is the special color : orange.

‘4’ represent the tile of the path’s color is the special color : yellow.

'5' represent the bonus (the stars in Kodable).

'6' represent the start point.

'7' represent the target point.

Both representations are fine and you can choose either one of them. Note, however that representing maps using single-digit numbers has the drawback that there are only 10 single digit numbers. If you wish to had more features to the game this can be limiting or you may need to start using non-digit characters.

Note that we will do some automated testing to check your implementation. Thus, it is quite important that you use one of the two representations above. If you add new features you should strive to be backwards compatible and still accept maps with only the features described above.

**Kodable Clone Game Description**

Our Kodable clone game will mimick much of the functionality of the original Kodable game:

1) The ball will from the start point initially.
2) The ball needs to roll from the start point to the end point. And it should find the best path to get all the three bonus points (the stars).
3) Once it has decided the direction, the ball will stop only when it comes across obstacles or conditional (colored) blocks (if a conditional instruction is given).
4) If the ball comes across a color tile and the next instruction is a conditional instruction with the same color, the ball will turn to the direction specified in the conditional instruction.
5) The player can use functions, and functions should include only three directions and without loop.
6) The player can use loops. A loop should include only two directions and one loop can be used up to 5 iterations.
7) Maps must be rectangular, but they can otherwise be of arbitrary size.
8) Unlike Kodable we do not impose a restriction on the number of instructions that can be used to solve a maze.

# 4 Basic game functionality

You're required to implement five basic functionalities for this game:

1) **Loading a game from a text map.**: You should implement a load operation that loads a map from a given text file. We provide two four files (two in each format) with the maps in this document as a starting point for you. You can also design more text files yourself. For advanced levels and to test your program, the map shape should be complicated enough to use functions, loop and conditions (color tiles). Thus you may want to design a few more maps on your own.

2) **Checking that a map is solvable**: You should implement a function that checks whether a map can be solved or not. Not all maps are solvable of course. For instance there may be no connection from the start point to the end point. Thus there should be a validation function that checks whether maps are solvable or not.

3) **Computing an (ideally optimal) solution**: Given that a map is loaded, your game should support a function that automatically computes a solution to the map if a solution exists. A solution in this case is a set of directions (including conditionals, loops and functions). While there are many different popssible solutions for a map, you should strive to compute a minimal solution. That is a solution that requires less instructions and directions.

For question 3) we suggest that if you cannot figure out how to compute the minimal solution or to deal with all features, you still try to present a solution. If you do this you will still get some marks (though not full marks for this question). For instance you may present a solution that only supports basic directions (Left, Right, Up, Down), but does not try to use loops or functions to make the set of instructions shorter. In your report you should be clear about the limitations if any.

4) **Interactive Play**: You should allow a player to play the game and propose a solution by himself. We will illustrate a potential interactive interface for this later in this document.

5) **Error Handling**: When a player proposes a solution, your program should check whether the solution is valid or not and whether it solves the game. If the solution solves the game then a winning message should be presented to the player. If the solution has errors or does not solve the game some useful error messages should be reported, that give feedback on the problems for the current solution.

One way to develop the user-interface of the program is to think of it as consisting of a few different commands. For example, for the basic functionality above, we could have the following commands:

1) **load file**: Loads a map.
2) **check**: Checks if the currently loaded map is solvable.
3) **solve**: Solves the map and presents the solution as a set of instructions.
4) **quit**: Quits the game.
5) **play**: Allows the player to propose a solution as a set of instructions. If the proposed solution solves the problem a winning message is displayed. If the proposed solution had problems, errors should be reported and the player should be asked to try again.

# 5 Modelling Paths: Instructions for the solutions.

For the basic functionality of questions 3 and 4 it is necessary to express a solution to the map/maze. Todo this the game will have a little "programming language" that can express the path that the ball should take. The syntax for this little language is as follows:

- *Basic directions.* There are four basic directions: `Left`, `Right`, `Up` and `Down`
- *Conditionals.* Conditionals can be expressed with a syntax of the form: `Cond{color}{Direction}`. For example `Cond{p}{Right}` expresses that the ball should turn to the right when it encounters a pink block.
- *Loops.* Loops can be expressed with a syntax of the form: `Loop{n}{Direction,Direction}`. For example `Loop{3}{Right,Up}` expresses that the ball should move right, then up 3 times. The iteration of the loop can range from 0 to 5. Thus loops with larger numbers are errors.
- *Functions.* We will assume that only one function can be defined (like in Kodable). The syntax of functions involves two constructs. The function call is done by writing the keyword `Function`. The definition of the function is done by appending `with Direction Direction Direction` to the set of instructions. See example 4 next.

A "program" that represents the path that the ball should take is just a sequence of the intructions above separated by spaces. We illustrate with some examples next.

1) The solution to the first map (in page 4) would be:

```
Right Down Right Up Left Down Right
```

2) The solution to the map with conditionals (in page 5 and 6) would be:

```
Right Cond{p}{Down} Right
```

3) The solution to the map with loops and conditionals (in page 7) would be:

```
Right Loop{2}{Cond{p}{Down},Right} Cond{p}{Down} Cond{y}{Right}
```

Note that here we use the color pink (`p`) instead of purple (since we only support pink, yellow and orange).

4) The solution to the map with functions assumming the color yellow instead of green (i.e. the map in page 8) would be:

```
Function Loop{2}{Right,Up} Loop{2}{Right,Down} Function Right
  with Right Cond{y}{Down} Down
```

For the question 3 the computed solution should be presented in the format above. That is if we were asking for the solutions of the 4 mazes, then what should be printed would be the textual representations of the paths/programs similar to the ones we illustrated above.

## 6 Example of the game play

Although we do not require you necessarily follow the example for the game play, your program should have the basic functionality pointed out previously.

For the game play you have a few different options. When asking the player for moves/instructions to solve the maze, one option could be for example, just entering a program/path like the one we have shown in the previous section. Another option is to make the interface more interactive and asking for one move at a time. Both options are fine and will be accepted, and you're even allowed to propose different ways to play/interact with the game (as long as you explain it in your report).

Here we are going to illustrate a possible game play using the more interactive option.

**Loading file** When the program loads it should allow the option to read a map from a text file (we provide an example called *map.txt*).

Format of the text file: The two formats mentioned earlier in the document will be accepted (you should pick one of the two and you do not need to accept both formats in your program).

For example, the content of *map.txt* file might be:

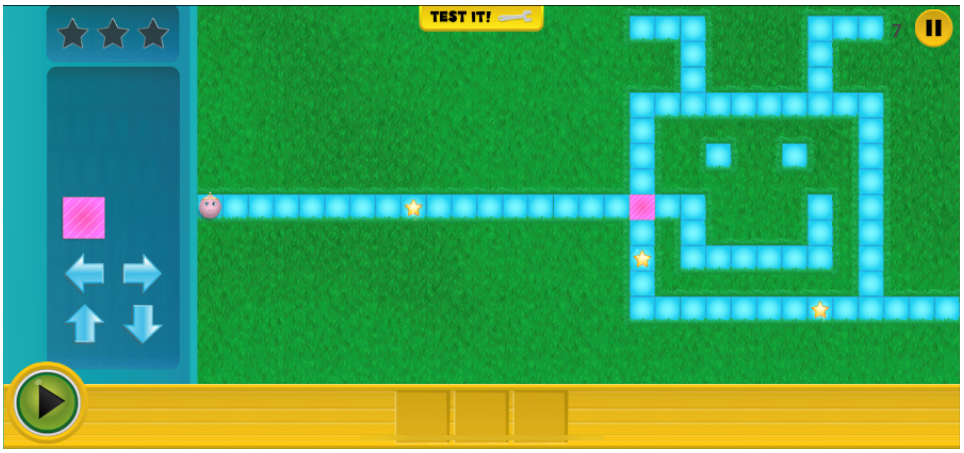
```
* * * * * - - - - - - - - - - - - - - - - - - - - - * * * * *
* * * * * b - - - - - - - - - - - - - - - - - - - - b * * * * *
* * * * * - * * * * * * * * * * * * * * * * * * * - * * * * *
* * * * * - * * - - - - * * * * * - - - - * * - * * * * * *
* * * * * - * * * - y y - * * * * * - y y - * * -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

To load a file the player may type `load "map.txt"`, and we would expect something like the following to happen:

```
> load "map.txt"
Read map successfully!
Initial:
* * * * * - - - - - - - - - - - - - - - - - - - - - * * * * *
* * * * * b - - - - - - - - - - - - - - - - - - - - b * * * * *
* * * * * - * * * * * * * * * * * * * * * * * * * - * * * * *
* * * * * - * * * - - - - * * * * * - - - - * * - * * * * * *
* * * * * - * * * - y y - * * * * * - y y - * * - * * * * * *
* * * * * - * * * - - - - * * * * * - - - - * * - * * * * * *
* * * * * - * * * * * * * - - b - - * * * * * * - * * * * * *
* * * * * - * * * * * * * - * * * - * * * * * * - * * * * * *
@ - - - - - * * * * * * * - * * * - * * * * * * p - - - - e
* * * * * - * * * * * * * - * * * - * * * * * * - * * * * * *
* * * * * - - - - - - - - * * * - - - - - - - - - * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

**Interactive Play** At first, we would like your program to take command step by step as the player's instructions but there could be a difficult part because of the color block (conditional). Let me take an example to illustrate a little bit.

Suppose we have a map given as the picture below, the right directions could be `Right` and then `Cond{p}{Down}` but after the first instruction of the player, the ball would just roll through the pink block without changing the direction. Obviously, it is not resonable to take commands one-by-one. So we would like you to collect all instructions first and only afterwards executed.

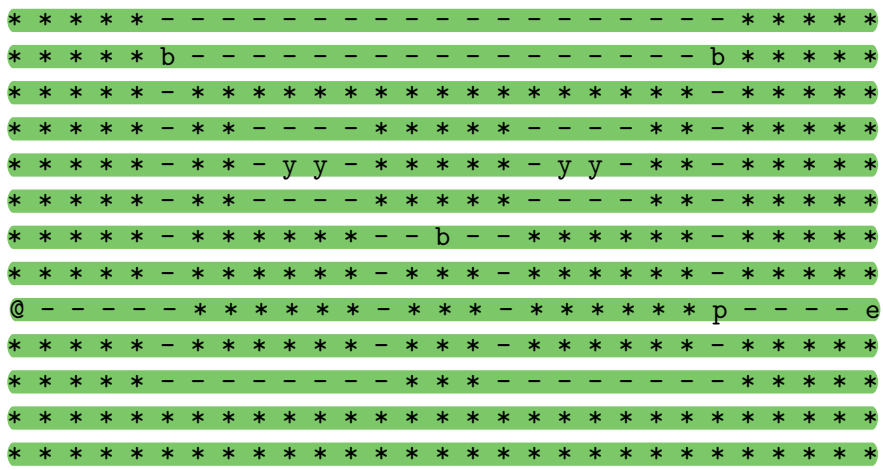


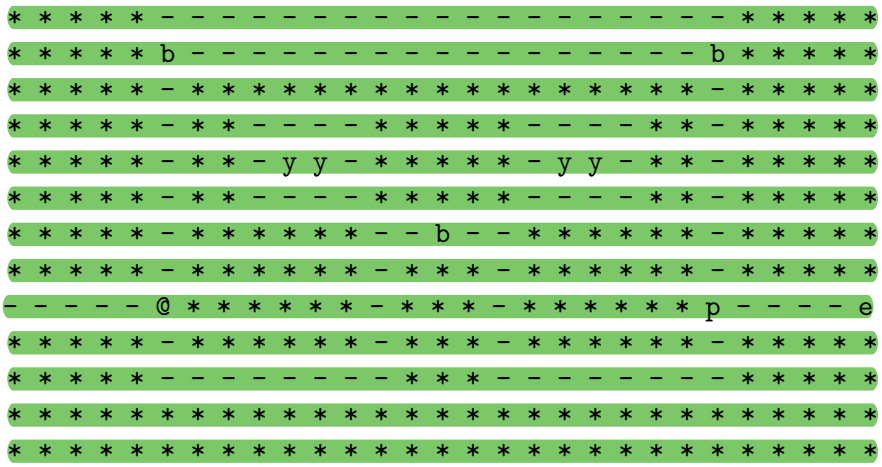

To start the game, after loading the map, the player can type `play`. After that, the player could give instructions one-by-one. But only when they press enter, the ball go through as the instructions.

We would expect something like the following to happen:

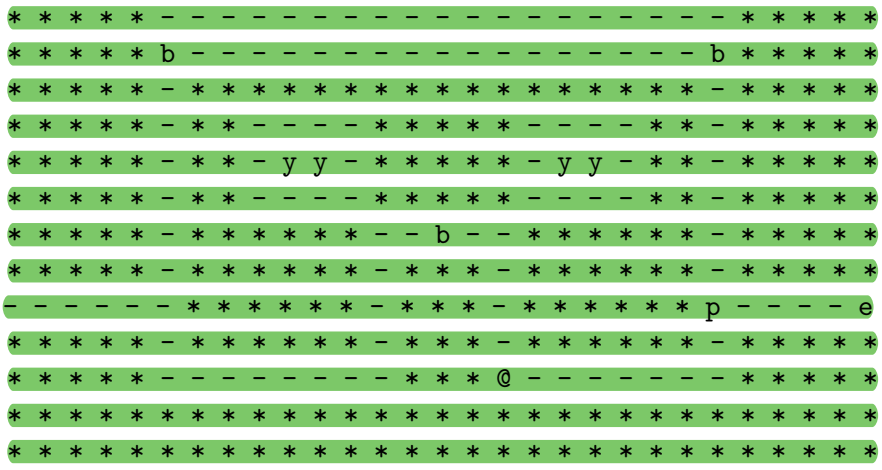
```
> play
First direction: Right
Next direction: Down
Next direction: Right
Next direction: Up
Next direction: Right
Next direction: Down
....
Next direction: <Enter>    ("here the player simply presses enter")

test:
```





...



**Functions** In this more interactive version of the game, if a player wants to use a function, the directions to be used in the function should be set before the game start. One way to do this is to have `play` take 3 directions optionally as arguments. For example:

```
> play Right Up Down
First direction: Function
Next  direction: Right
Next  direction: <Enter>   ("here the player press enter")

test:

* * * * * - - - - - - - - - - - - - - - - - - - - - * * * * *
* * * * * - - - - - - - - - - - - - - - - - - - - - b * * * * *
* * * * * - * * * * * * * * * * * * * * * * * * * - * * * * *
* * * * * - * * - - - - * * * * * - - - - * * - * * * * * *
* * * * * - * * * - y y - * * * * * - y y - * * * - * * * * *
* * * * * - * * - - - - * * * * * - - - - * * - * * * * * *
* * * * * - * * * * * * - - b - - * * * * * * - * * * * * *
* * * * * - * * * * * * * - * * * - * * * * * * - * * * * *
- - - - - - * * * * * * * - * * * - * * * * * * p - - - - e
* * * * * - * * * * * * * - * * * - * * * * * * - * * * * *
* * * * * @ - - - - - - - * * * - - - - - - - - - * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

got the first bonus!

* * * * * - - - - - - - - - - - - - - - - - - - - - * * * * *
* * * * * - - - - - - - - - - - - - - - - - - - - - b * * * * *
* * * * * - * * * * * * * * * * * * * * * * * * * - * * * * *
* * * * * - * * * - - - - * * * * * - - - - * * - * * * * * *
* * * * * - * * * - y y - * * * * * - y y - * * * - * * * * *
* * * * * - * * * - - - - * * * * * - - - - * * - * * * * * *
* * * * * - * * * * * * - - b - - * * * * * * - * * * * * *
* * * * * - * * * * * * * - * * * - * * * * * * - * * * * *
- - - - - - * * * * * * * - * * * - * * * * * * p - - - - e
* * * * * - * * * * * * * - * * * - * * * * * * - * * * * *
* * * * * - - - - - - - - @ * * * - - - - - - - - - * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

**Error handling** Your program should handle some error cases.

```
<we omit some previous steps>
Next direction: Left

Sorry, error: cannot move to the Left.
Your current board:
```

```
* * * * * - - - - - - - - - - - - - - - - - - - - - * * * * *
* * * * * - - - - - - - - - - - - - - - - - - - - - b * * * * *
* * * * * - * * * * * * * * * * * * * * * * * * * - * * * * *
* * * * * - * * - - - - * * * * * - - - - * * - * * * * * *
* * * * * - * * * y y - * * * * * - y y - * * * - * * * * *
* * * * * - * * - - - - * * * * * - - - - * * - * * * * * *
* * * * * - * * * * * * - - b - - * * * * * * - * * * * * *
* * * * * - * * * * * * * - * * * - * * * * * * - * * * * *
- - - - - - * * * * * * * - * * * - * * * * * * p - - - - e
* * * * * - * * * * * * * - * * * - * * * * * * - * * * * *
* * * * * @ - - - - - - - - * * * - - - - - - - - - * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

**Ending** Determine if a game ends.

For example, if the player reach the end and get all the three bonus, you should terminate the game if the player wins.

```
<we omit some previous steps>
New map:
```

```
* * * * * - - - - - - - - - - - - - - - - - - - - - * * * * *
* * * * * - - - - - - - - - - - - - - - - - - - - - * * * * *
* * * * * - * * * * * * * * * * * * * * * * * * * - * * * * *
* * * * * - * * - - - - * * * * * - - - - * * - * * * * * *
* * * * * - * * * y y - * * * * * - y y - * * * - * * * * *
* * * * * - * * - - - - * * * * * - - - - * * - * * * * * *
* * * * * - * * * * * * - - - - - * * * * * * - * * * * * *
* * * * * - * * * * * * * - * * * - * * * * * * - * * * * *
- - - - - - * * * * * * * - * * * - * * * * * * p - - - - @
```

```
* * * * * - * * * * * * * - * * * - * * * * * * - * * * * *
* * * * * - - - - - - - - * * * - - - - - - - - * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
Congratulations! You win the game!
```

# 7 Extra Features

Besides the basic features, your program can provide more functionality. The actual Kodable game should provide several ideas for extra features that are not part of the basic functionality. So that, on its own, is a huge source of ideas.

Nonetheless, here are some simple ideas for additional features:

- **Hints**: You could support hints, allowing the user to ask for a hint. The hint could be in the form of a next valid move that is closer to the final target. The hint should of course always be valid (i.e. you should not give invalid hints or hints that move you away from the target).
- **map generation**: You could support a functionality that allows the automatic generation of new map. Your generator should only generate **solvable** maps.
- **Good-looking user interface**: you could support a better user interface. For example, instead of a command-line interpreter, you could develop a graphical interface using one of the various graphic libraries for widgets available in Haskell. You could also try to support some animation that shows the ball rolling (this could be done even in text mode).
- **Other features**: There are several other features that you could implement. We will value your creativity and non-trivial features that you design and implement.

Most of the extra features above (except the UI) can be added via new commands to the user interface. For example, you could have (some) of the following commands:

1) **new**: random map generation

2) **generate**: for automaticaly generating a board
3) **hint**: to ask for a hint

If you develop a graphic user interface (or some other more elaborated form of interface) you can adapt the commands into other things (like items in a menu). Your report should then have a description of how to use the game.

# 8 Final Report

You should write a short final report (in pdf format) that:

1. Describes how to build your project. It is highly recommended that your project builds with 1 or 2 commands (for example by employing a Makefile or some other build scripts). If your project cannot be easily build, you may be penalized.
2. Describes the functionality of your program (how to play a game, how to load files, etc).
3. Explains your choice of data structures for representings maps, programs and other parts of the game.
4. Explains how your code deals with error cases and ending.
5. Explains the additional features that you implement. You should start by listing **all** the additional features that you have implemented, and then explain those features and how their implementation works.

# 9 Grading

Grading criteria:

1. Correctness (50%): your program should implement the 4 basic requirements (and optional extra features) and give the correct feedback to user.
2. Lecture understanding (30%): We will evaluate good use of functional programming techniques (such as recursion, list comprehensions, higher-order functions and data and type declaration).

3. Code specification (20%): your code should have good naming convention/code reuse/comments. See https://wiki.haskell.org/Programming_guidelines for details.

4. Extra features: If you only implement the basic functionality (no extra features) the maximum grade that you can have is capped at 75 points. Therefore a perfect project solution scoring 100 points should implement the basic functionality correctly, with elegant well-document Haskell code, some interesting extra features and a well-written report that describes the whole project in a good manner.

# 10 Advice

The main advice is to get the basic functionality for the project correct, together with a good report. This alone will probably guarantee a positive (pass) grade for the course. If time permits you can try to design some extra features for extra marks.

Also do not forget that we have plenty of Haskell code from the Lectures and Tutorials that can be helpful to get you started with the project. If use existing code available online somehow in the project, please make sure to acknowledge the source in your comments and your final report to avoid plagiarism!

---

**Submission**

Please submit your solution on Moodle before the deadline. All the files should be compressed into a Zip file. **Please note that the deadline is strict and set by the University. The Lecturer and TA's have no power to extend the deadline!** Failure to submit on the deadline will mean that you'll score 0.