Report

> For COMP3258 Functional Programming – Final project
> By Leung Chun Yin (3035437939)
> Compiled & tested on GHC 8.8.4, macOS Big Sur

Build Instruction

> Required software: make, ghc, any shell

1. Go to the project folder by using a shell
2. Run command 'make'
3. Run the game using the binary executable `Game`

Game instruction

Lobby screen

To play the game, you will first need to have a map stored in the directory `map`.

Once you started the game, you will be greeted by the intro message and a list of commands you can use. Here will explain how each command behaves.

`load <path>`

It is used to load a map file located in the map folder. If you try to load an invalid or not existing file, you will receive an error.

`check`

It is used to check if a loaded map solvable, i.e. At least one path that brings the ball `@` to the target `t`.

`solve`

It is used to generate a path that tried to get the most bonuses with least steps if a map is loaded.

`play <function>`

Bring you to interactive mode if you have a solvable map loaded. You can provide a function with up to three directions to use a function in interactive mode. Please check the next section on how to play in the interactive mode.

`quit`

Quit and exit the game.

Interactive mode

In interactive mode, you can start solving the game using the commands below. The game will end once the ball reaches the target. There will be a short message evaluating your result and then bring you back to the lobby screen.

`<command(s)>`

Enter a chain of command such as `Left Cond{p}{Right}`. The commands will only be executed when all commands are entered correctly.

`undo`

Undo a command by this command. Note that the commands are undone one by one so that you might need to undo multiple time for a loop or a chain of commands.

`hint`

Get a hint on getting to the target `t` by this command.

`quit`

Quit the interactive play and return to the lobby.

Basic functionalities

- ### Loading a map in `map` directory

  It only supports format using 'symbols', where

  ```
  * = wall
  - = path
  @ = ball (only 1)
  t = target (only 1)
  b = bonus
  p = pink tile
  g = green tile
  y = yellow tile
  ```

If the map contains any other symbols, it will be rejected and prompt an error.

- ### Command supported

  It currently supports these commands.

  ```
  Up, Down, Left, Right -- Directions
  Cond{<color>}{<direction>} -- Conditional directions
  Function -- Function that supports up to 3 predefined (cond)directions
  Loop{<number of iteration>}{<direction 1><direction 2>} -- Loop that repeats 2 (co
  ```

- ### Check a map is solvable

  It can check if a map is solvable which means there is an at least one path lead the ball to target

- ### Solve a map

  It can provide a path that can get the most bonuses with the least steps. Or an error if it is unsolvable. It will make use of all commands with priority of Loop > Function > Condition = Direction

- ### Interactive play

  It provides an interactive game mode to let a user enter command one by one. If some command fails, it will show the last good step and report to the user which step failed.

Additional functionalities

- ### Colored map

  It has a colour text map that is more interesting to look at.

- ### Undo in interactive play

  It has a undo command that let the user undo their move when wanted.

- ### Hint in interactive play

  It provides a hint that leads the user to the target.

- ### End game performance evaluation in interactive play

  It has a summary of the gameplay to help the user evaluate their performance

Module explanations

Main.hs

It is the entry point of the program where it handles io in the lobby screen. A simple data `LoadState` to simplify the lobby loop `mainLoop`. The indefinite loop will repeat until the user enters `quit`. I also tried my best to separate IO and other things as much as possible to reduce wrapping and unwrapping monads.

GameMap.hs

It is where I define the map structure. The map itself is saved in the form of `[[String]]`. And the coloured tiles is defined by `ColorTile`. With these two structures and `Maybe`, it is a convenience to access and compare.

Command.hs

It is where the command string parsed into `[Command]`. There is a parser to read and validate the string and convert them into command.

I defined three levels of commands type.

- Direction - The most basic building block, Up, Down, Left, Right
- Action - The atom of an action that is either pure direction / conditional direction
- Command - Command that can be broken down into actions : Pure Action/ Function/ Loop

With this structure, Commands can be broken down to a list of actions and execute by a runner in Game.hs. It also support condensing a list of action to a list of command that leverage the power of loop and function. For loop it detect for consecutive repeating sequence by doing comparison in pair recursively. While for function, it looks for repeating sized-3 actions.

Game.hs

It is where I define the loop that represents the interactive game. It is a loop of the updating the `GameState` and `BallState`, where `GameState` represent the overall state of the game, such as the target coordinate and `BallState` represent the current state of the ball.

To run the commands, it will first parse it into a `[Command]` using a parser. If it failed, it would prompt an error. Then, it expands to a list of command and sends to the runner queue. Then runner will update the state recursively until either of case happens:

1. It failed to process further (walking to an invalid place) It will return Nothing, and an error is prompted and return last good state to the game loop.
2. All action is executed It will return the final state to the game loop.

The `undo` function is implemented by saving every iteration of the BallState into a stack and recover them one by one when called.

The `hint` function is implemented by running the solver using the current map and get the first instruction of the path.

Solver.hs

It is where I create the 'AI' that find the path. The following steps implement it:

1. Start at position (Initially at the original `@`)
2. Try to walk in all direction and return only walkable one
3. Record bonus collected and update the map accordingly (remove the bonus)
4. Recursively run again with the new map and new position Or
5. Return if it reaches target `t`. The cost is computed by `steps + remaining bonus * 100`
6. Get the best path by selecting the path with the least cost

Util.hs

It is where the utility function located. I coloured the text using ANSI escape code.

Error handling

The following tree shows the error handling.

```
\-- Lobby
|- load map
|- If map file exist
|- If the map valid
|-> Return the map and filename
|- Else
|-x [Error] Invalid map
| Else
|-x [Error] File do not exist
|- check
|- If there is map loaded
|-> Check the map
|- Else
|-x [Error] no map loaded
|- solve
|- If there is map loaded
|-> Solve the map
|- Else
|-x [Error] no map loaded
|- play
|- Parse the function input
```

```
  |- Success but longer than 3
  |-x [Error] Function too long
  |- Success
  |-> Load it to interactive game
  |- Failed
  |-x [Error] Invalid function
  |- Failed to match any command
  |-x [Error] Bad command

  \-- Interactive play
  |- undo
  |- If no more element in state stack
  |-x [Error] Unable to undo
  |-> Return to last state
  |- parse command
  |- if it failed
  |-x [Error] Invalid command string
  |- Success
  |- If the runner return Nothing (failed to run)
  |-x [Error] Cannot move to <Direction>
  |- If it return Just ballState
  |-> Loop with new state
  |- If it failed to match anything
  |-x [Error] Invalid command
```

Acknowledge

- I used the parser module provide in assignment 2 for my command.
- I am also get inspired by many examples in haskell wiki