COMP105: Programming Paradigms
Lab Sheet 10

Filename: Lab10.hs

This is a challenge lab on IO, where we will implement a maze game.

Maze files. Before starting, Windows users should download `lab10windows.zip` from Canvas, while Mac and Linux users should download `lab10unix.zip`. Both files contain the same data:

- `Code.hs` contains some code that we saw in the lectures that we will use in this lab.
- `maze2.txt` through `maze6.txt` are small mazes that you can use for testing in Part A, ranging from 2×2 through 6×6 in size.
- `maze-big-1.txt` through `maze-big-4.txt` are larger mazes that you can use for testing in Part B.

The mazes. The mazes are represented by ASCII art, and can be seen by opening one of the files. For example `maze6.txt` contains:

```
###########
#         #
##### ### #
# # # # # #
# # # # # #
#   # # # #
# # # # # #
# #     # #
# ##### # #
#     # # #
###########
```

The maze is composed of square tiles, and each tile is either a wall or a corridor. The `#` character represents the wall tiles, while the space character represents the corridor tiles. You can assume that the outer edge of the maze is surrounded by walls, as in the example above. You can also assume that all corridors are exactly one tile wide.

The game will allow the player to explore the maze. The player will be represented by the `@` (at) character, and will be able to move around the maze using the keys `wasd`, as described in Question 5. The player can walk through corridors, but not through walls. The game will end either when the player finds

the exit, which will be represented by the `>` character as described in Question 7, or when the player types `quit`.

Restrictions. Unlike the previous two challenge labs, there are no restrictions at all in this lab. Use whatever techniques you like.

Part A

In part A we will build some functions for loading the maze, printing the maze, and manipulating the maze.

Question 1. We will represent mazes as a list of strings. So, the maze

```
#####
#   #
# # #
# # #
#####
```

will be represented as

```
["#####",
 "#   #",
 "# # #",
 "# # #",
 "#####"]
```

Write an IO action `get_maze :: String -> IO [String]` that takes a string containing a file path for a maze, and returns the representation of that maze as a list of strings. For example, if `"maze2.txt"` is saved as `M:\maze2.txt`, then:

```
ghci> get_maze "M:\\maze2.txt"
["#####","#   #","# # #","# # #","#####"]
```

Note that on Windows, you must use two backslashes in your file paths, so `C:\Documents\Haskell\maze2.txt` must be written as

`"C:\\Documents\\Haskell\\maze2.txt"`.

This is because \ is the escape character in Haskell. Linux and Mac users do not need to do this. Hints:

- You can do this using the `readFile` and `lines` functions.
- Remember that IO types need to be boxed and unboxed. So before you use the output of `readFile`, you will need to unbox the string. Likewise, `get_maze` is required to return type `IO [String]`, so you will need to use the `return` function to box the value that you want to return.

Note: for testing purposes it is reccomended that you create a constant named `maze_path` in your file like so

```
maze_path = "M:\\path\\to\\my\\maze\\dir\\maze2.txt"
```

so that you can run `get_maze maze_path` instead of having to type out the long file path every time. All further examples in this document assume that you have done this.

Question 2. Write an IO action `print_maze :: [String] -> IO ()` that takes a maze, and prints it out to the screen. For example:

```
ghci> m <- get_maze maze_path

ghci> print_maze m
#####
#   #
# # #
# # #
#####

ghci>
```

Hints:

- You can use the `putStrLn` and `unlines` functions to do this.
- Using these two functions together will produce an extra newline at the bottom of the maze, as shown above. This is the expected (and required) behaviour.

Question 3. Write a function `is_wall :: [String] -> (Int, Int) -> Bool` that takes a maze and a pair of integers, and returns `True` if the tile at that position is a wall (represented by `'#'`) and `False` if it is a corridor (represented by `' '`). Note that the coordinates are given as pairs `(x, y)` where `x` represents the horizontal coordinate, and `y` represents the vertical coordinate. Coordinates are zero indexed, starting from the top left of the maze. So, in the following diagram:

```
#####
#  b#
# # #
#a# #
#####
```

`a` is at position `(1, 3)` while `b` is at position `(3, 1)`. For example:

```
ghci> m <- get_maze maze_path

ghci> is_wall m (0, 0)
True

ghci> is_wall m (3, 1)
False
```

Hint: The `get` function that is provided in `Code.hs` will get the tile at a specified pair of coordinates. So you just need to determine whether it is a wall or not.

Question 4. Write a function

```
place_object :: [String] -> (Int, Int) -> Char -> [String]
```

that takes a maze, a pair of coordinates, and a character `c`, and returns a new maze with `c` at those coordinates. You can assume that the given coordinates are not inside a wall. For example:

```
ghci> m <- get_maze maze_path

ghci> let x = place_object m (1, 1) '@'

ghci> print_maze x
#####
#@  #
# # #
# # #
#####
```

Hint: You can use the `set` function that is provided `Code.hs` to do this.

Question 5. We will now write a function to process the player's inputs. The player will control the game using the following keys:

- `'w'` to move up
- `'s'` to move down
- `'a'` to move left
- `'d'` to move right

Write a function `move :: (Int, Int) -> Char -> (Int, Int)` that takes a pair of coordinates, and a character, and returns a pair of coordinates moved in the appropriate direction. If the character is not in `"wasd"`, then the coordinates should be left unchanged. For example:

```
ghci> move (1, 1) 'w'
(1,0)
ghci> move (1, 1) 's'
(1,2)
ghci> move (1, 1) 'a'
(0,1)
ghci> move (1, 1) 'd'
(2,1)
ghci> move (1, 1) 'q'
(1,1)
```

Question 6. If the player tries to walk into a wall, then we should not allow them to do this. Write a function

```
can_move :: [String] -> (Int, Int) -> Char -> Bool
```

that takes a maze, the current position of the player, a direction character, and returns `True` if the player can walk in that direction, and `False` otherwise. For example:

```
ghci> m <- get_maze maze_path

ghci> can_move m (1, 1) 's'
True

ghci> can_move m (1, 1) 'w'
False
```

This shows that from the coordinates `(1, 1)` in `"maze2.txt"`, you can move down, but you cannot move up because there is a wall blocking the way.

Question 7. Write an IO action

```
game_loop :: [String] -> (Int, Int) -> (Int, Int) -> IO ()
```

that implements the maze game. The arguments to `game_loop` are the maze, the current location of the player, and the location of the exit. Your function needs to do the following things:

- Print out the maze with the player, represented by the `'@'` character, in their current position, and with the exit, represented by the `>` character in its position.
- Ask the user for input using `getLine`, which is a function in prelude that gets a line of text from the user. The first character of the string returned by `getLine` should be treated as the input from the player. You can assume that the user inputs a non-empty string.
- Check whether the player can move in the direction specified, and create a new set of coordinates where:
  - If the player can move, then the new coordinates are the new location of the player.
  - If the player cannot move, then the new coordinates are the same as the current coordinates.
- If the new coordinates are the exit coordinates, then the game should terminate, and the string `You win!` should be printed.
- If the player typed the string `quit` when they were asked for input then the game should terminate.
- Otherwise, the function should recursively call itself with the new coordinates to continue the game.

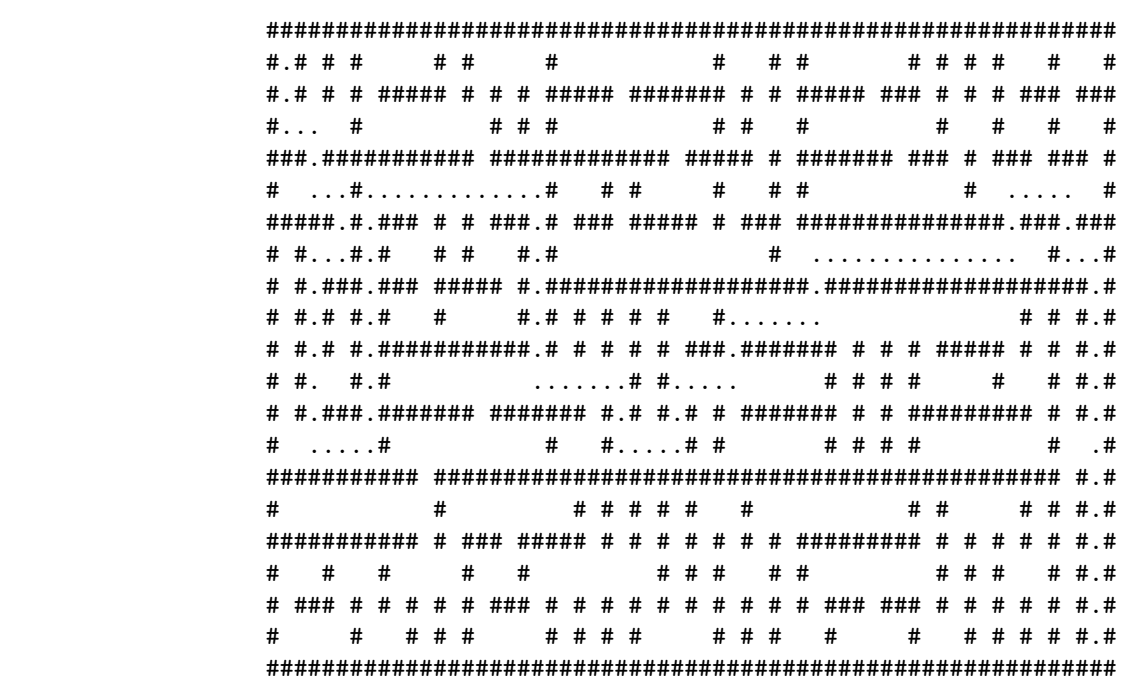
An example interaction is:

```
ghci> m <- get_maze maze_path

ghci> game_loop m (2, 1) (3, 3)

#####
# @ #
# # #
```

```
# #>#
#####

d
#####
#  @#
# # #
# #>#
#####

s
#####
#   #
# #@#
# #>#
#####

s
You win!
```

(*) Part B

In part B, we will build a program that *solves* the maze. That is, we want to find the path from the start of the maze, which is always the top left tile, to the end of the maze, which is always the bottom right tile. For example, the solution to `"maze-big-1.txt"` is shown by the dotted line in the following diagram:



To make this task simpler, you may assume that the following property always holds: **The maze is a tree.** This means that there are never any cycles (loops) in the maze, and that there is exactly one path from the start to the finish. All example mazes have this property.

Question 8. Write a function

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

that takes a maze, a start coordinate, and a target coordinate, and returns the *path* from the start coordinate to the target coordinate. The path should include the target and start coordinates. For example:

```
ghci> m <- get_maze maze_path

ghci> get_path m (1, 1) (3, 3)
[(1,1),(2,1),(3,1),(3,2),(3,3)]
```

Hints:

- The fact that the maze has a tree structure is very important here, because it makes the question much simpler than it otherwise would be. While there are path-finding algorithms that can deal with loopy mazes, a simpler recursive approach can be used here.
- You may have heard of algorithms like depth-first search and breadth-first search. One of these algorithms has a particularly simple recursive implementation for trees.
- We saw some example code for finding things in trees in Lecture 22. A similar approach can be used here. Bear in mind that the code from Lecture 22 only works for binary trees, where every node has exactly two children. Here, the nodes in our tree can have up to four children.
- It is possible to solve this question by creating your own tree data structure, but it is not recommended, because it will overcomplicate things. You can use the maze itself as the data structure while exploring it, so there is no need to define your own data type.
- One thing that you need to bear in mind is that you should never end up walking in circles. So if you are at `(1, 1)`, and you decide to recursively explore `(2, 1)`, then you need to make sure that the recursive call then doesn't decide to recursively explore `(1, 1)` again. Otherwise you will end up looping infinitely between the two coordinates.
- You can assume that both the start and target coordinates are corridors. This means that there definitely is a path from the start to the target.

Question 9. Now we will turn our maze solver into a program. Write a function `main :: IO ()` that does the following:

- The program takes one command line argument, which is the path to a maze file. You can assume that this command line argument is correct, and it does not matter what your code does if it is incorrect.

- The program loads the maze from the file, solves it, and then prints out the path from top-left to bottom-right, using the `.` character to indicate the path.

For example, assuming that `"maze-big-2.txt"` is in the current working directory, and that the program has been compiled, and is named `Main`, then entering

```
Main maze-big-2.txt
```

on the command line in Windows (that is, in `cmd`) should produce:

```
################################################################
#...#   # #.....  #         #             #     # #       # #
# #.### # #.# #.### # # # # # ### ##### # # ##### ### ##### #
# #.........# #.    # # # #   #     #   #         # #...# # #
###############.######################################## #.#.# # #
# #     #      .#   #   # # #   #   #             #  .#...# #
# ### ### # ###.# ### ### # ### ### ### ### ### # # #.# #.# #
# #     # #   #...# #                     #   # # # #.# #.# #
# # ### ####### #.# ####### ############################.# #.# #
#   #           #...........#...#.....  # #.....  #...# #...#
### ###################### #.#.#.#.###.### #.# #.###.# # # #.#
# #     #     # # # # #   #...#...#  .  # #.# #.# #.# # # #.#
# # # # # # ### # # # ### ####### ###.### #.###.# #.# # # #.#
#   # #   #     #   #     #     # #  ...# #.# #.#...# # # #.#
# ### # # # ##### # ### ### # ### ### #.# #.# #.#.# # # # #.#
#   # # # #     # # #     # #     #   #.....  #...# # # # #.#
### ############### ######### ####################### # # #.#
#     # #           #   #         #   # #         #   # # #.#
# ### # # ### # # # # ### ##### # # # # # # # # # # # # # #.#
# #         # # # # #         # #   #     # # # # # # # # #.#
################################################################
```

Linux or Mac users would instead type:

```
./Main maze-big-2.txt
```