COMP105 Assignment 3: Writing a program

Assessment Number	3 (of 4)
Weighting	25%
Assignment Date Circulated	Wednesday the 1st of December (week 10)
Deadline	Friday the 17th of December (week 12) at 12:00 mid-day
Submission Mode	Electronic only
Learning outcome assessed	• Write programs using a functional programming language.
Marking criteria	Correctness: 100%
Submission necessary in order to satisfy Module requirements?	No
Late Submission Penalty	UoL standard penalty applies

Important: Please read all of the instructions carefully before starting the assignment.

Introduction. In this assignment, you will implement a maze game. Before starting, Windows users should download `a3windows.zip` from the module webpage, while Mac and Linux users should download `a3unix.zip`. Both files contain the same data:

- `Main.hs`. This is a template file with stub code for each question, and with some helper code from Lecture 25 that you may use in the assignment.
- `maze2.txt` through `maze6.txt` are small mazes that you can use for testing, 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.

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 assignment. In parts A and B you will implement the maze game, as described above. Unlike Assignments 1 and 2, **there are no restrictions on the functions you can use**. Use whatever techniques you think appropriate.

Parts A and B combined are worth 65% of the marks for the assignment, which is equivalent to a 2:1 mark. Part C should be attempted by ambitious students who want to push for 100%. The task in part C is to write a program that finds and displays the path from the top-left corner of the maze to the bottom-right corner of the maze.

Marking. Marks will be awarded for correct function implementations only. No marks will be gained or lost for coding style. If you have a non-working function that you would like to be looked at by a human marker, then include the following comment above the function.

```
-- PLEASE READ
```

Please make sure that the comment is *exactly* as shown above including capitalization. Functions that do not include this comment will be marked by the automated marker only. If you have multiple non-working functions that you would like to be looked at, then please include the comment above each one. Do not include the comment above working functions – no extra feedback will be given for those other than what is produced by the automated marker.

Submission. You should complete the assignment by filling out the template `"Main.hs"` (please do not change the name). Submit this file to the assessment task submission system:

`https://sam.csc.liv.ac.uk/COMP/Submissions.pl`

Since your functions will be called from the automated marker, please make sure that you follow these guidelines:

- Your code should compile and load into ghci without errors. (Warnings are fine).
- You should not alter the type signatures in the template file in any way.
- You should leave the stub error-message implementation in the template file if you have not answered a question. If you remove this implementation, and then later decide to comment out your code because it does not compile, then please reinstate the original stub implementation.

There will be a **5% penalty** for any submission that does not follow these guidelines. You can use the checker file (see the section at the end of this document) to check whether your submission will get this penalty.

If you have some code that does not compile, but you would like the marker to see it, then comment out the code, and explain in another comment what the code is supposed to do. Remember that you will need to reinstate the stub implementation after you comment out your code, or the file will still fail to compile.

Academic Integrity. The work that you submit for this assessment must be your own. The University takes plagiarism very seriously, so you should not collude with another student, copy anyone else's work, or plagiarise from other sources.

Deadline. The deadline for this task is:

Friday the 17th of December (week 12) at 12:00 midday

The standard university late penalty will be applied: 5% of the *total marks available* for the assessment shall be deducted from the assessment mark for each 24 hour period after the submission deadline. Late submissions will no longer be accepted after five calendar days have passed.

Part A (worth 30%)

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 may be convenient to modify the `maze_path` line of the template to a line like

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

in your file, 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 the template 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_player :: [String] -> (Int, Int) -> [String]` that takes a maze and a pair of coordinates, and returns a new maze with the `@` symbol at those coordinates. For example:

```
ghci> m <- get_maze maze_path

ghci> let x = place_player m (1, 1)

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

Hint: The set function that is provided in the template will place a character at a particular point in the maze.

Part B (worth 35%)

In part B, we will build the logic for our game, and produce a working program.

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

- Print out the maze with the player in the current 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. All other characters in the string should be ignored. 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.
- In either case, the function should then recursively call itself with the new coordinates to continue the game.

The game will continue until the user presses control-c. An example interaction is:

```
ghci> m <- get_maze maze_path

ghci> game_loop m (1, 1)

#####
#@  #
# # #
# # #
#####

d
#####
# @ #
# # #
# # #
#####

d
#####
#  @#
# # #
# # #
#####
```

Part C (worth 35%)

In part C, 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
```

The Checker

You can download `Checker3.hs` from the COMP105 website. This file is intended to help you check whether your code will run successfully with the automated marker. To run the checker, follow these instructions.

1. Place `Checker3.hs` and your completed `Main.hs` into the *same folder*. The checker will not work if it is not in the same folder as your submission.
2. You have two options to run the checker.
   (a) Navigate to the folder in Windows explorer, and double-click on `Checker3.hs` to open the file into ghci.
   (b) In a ghci instance, first use the `:cd` command to change to the directory containing the checker, and then run `:load Checker3.hs`

   You **cannot** load the checker directly with something like `:l M:\Checker3.hs`, because you must first change the directory so that ghci can also find `Main.hs`. Both of the methods above will do this.

If the checker compiles successfully, then you are guaranteed to not receive the 5% penalty for non-compilation. You can run the `compileCheck` in ghci to confirm that the checker has been compiled successfully.

You can also run the `test` function to automatically test your code using some simple test cases. **These tests will only check the pure functions in the assignment.** The checker verifies that IO actions have the correct type, but it does not run test cases on them. Passing these test cases does not guarantee any marks in the assessment, since different test cases will be used to test your code after submission. You are encouraged to test your functions on a wide variety of inputs to ensure that they are correct.

FAQ

In Questions 4 and 7, what should happen if the player is placed inside a wall?

You can assume that the inputs to these two functions will always place the player in a corridor.

In Question 7, wouldn't it be better to use `getChar` rather than `getLine`?

It would, but unfortunately there is a long-standing bug in the Windows version of Haskell regarding the input-buffering behaviour of `getChar`, which would lead to students on different platforms getting different outputs. To ensure that the assignment is consistent for all students you are *required* to use `getLine` in Question 7.

What should `maze_path` contain when I submit my code?

It doesn't matter – the auto-marker will not use your `math_path` definition. Make sure that your functions work for *any* paths, not just the path that you put in `maze_path`.

Are there any guarantees about the mazes?

You can assume that all mazes have walls entirely around the outside, and that the top-left and bottom-right corners are corridors.

Can I use this library function, or write this helper function?

Yes. You can do anything that you like in Assignment 3.

Should the file be called `Main.hs` or `Assignment3.hs`?

It should be called `Main.hs`. The change in file name is due to three reasons:

- Question 9 asks you to compile and run a program. This will only work in modules that are called `Main`, or files that have no module declaration.
- To load your code in, the checker requires that your code has a module name.
- Haskell requires that the file name and the module name in the module declaration must be the same.

So in Assignment 3, the file you will work in will be called `Main.hs`. Please make sure that you submit it with this filename.

The `test` function in the checker reports that some of my functions are incorrect, throw exceptions, or timeout. Will I receive the penalty?

No. The 5% penalty is only for code that *doesn't compile* when the checker is loaded into ghci. If you can successfully load the checker into ghci, then you will not receive a penalty. You are likely to receive fewer marks for a question if your function doesn't work, however.

The checker doesn't compile with an error like `Could not find module 'Main'`

There are three things to check:

- Are you following the instructions on how to run the checker? Note that just doing something like `:load M:\Checker3.hs` or similar **will not work**, because you need to change the directory first.
- Is your file name *exactly* `Main.hs`? If it is misnamed, the compiler won't be able to find it.
- Did you alter or remove the `module Main ...` line at the top of the template? This line needs to be in place unaltered for the checker to load it in.

If you are still having difficulty, then send an email to get help.

My `Main.hs` file compiles, but the checker does not compile and shows one of the assignment functions in the error. What should I do?

The most likely culprits are:

- You removed or altered the `module Main ...` line at the top of the template.
- You removed or altered one of the type annotations in the template file.

Make sure that these are all present. If they are, then your source file should also fail to compile, so you can fix that error. If you are still having difficulty, then send an email to get help.