Programming Paradigms 2022-2023: Joint Lab

Introduction

Please find the following files and download them from Moodle:

maze__sample_java.zip	// the sample Java file
String.pdf	// Java String API
ArrayList.pdf	// Java ArrayList API
maze_sample.hs	// the sample Haskell file

In this joint lab, we will explore how to construct a maze and how to solve it using both Java and Haskell. A maze is defined as an x * y grid, in which each cell is represented as a character. The meaning of the characters are given as follows:

- '.' = empty space;
- '#' = wall;
- 'A' = start point;
- 'B' = goal point;

Following is a sample maze which consists of 3 * 3 characters:

```
A#.
...
#.B
```

As we can see, the start point of this maze is at its top-left corner. Let's assume there is an agent whose current location is cell A, and its goal is to reach goal point B which is at the bottom-right corner. The agent can move from its current cell to one of its adjacent cells that does not contain a wall. In the above example, the agent can move down, but cannot move right as its right adjacent cell is a wall. Moreover, the agent also cannot move up or left, as it already reaches the up and left border of the maze. The agent cannot move diagonally either.

For this lab, please work in a group of 3-4 students on the following tasks.

Initial Tasks

1. Your first task here is to get familiar with the sample Haskell and Java code.
    a. In Haskell, a maze is defined as a list of `String` (i.e., `[String]`). We provide a function called `showMaze` which takes a maze as its input and then outputs the maze to the console window. This function uses some Haskell features we have not covered yet, so please take it on trust for now. You can run it at the GHCi prompt by passing it a maze: e.g. `showMaze maze1`. All other Haskell code is using techniques you should be familiar with from lectures, so please make sure you fully understand the remainder of the sample Haskell code.
    b. In Java, a maze is defined as an array of `String` (i.e., `String[]`). In order to solve a maze, a class called `MazeSolver` is given. Each `MazeSolver` object is associated with a maze which can be given when the object is initialised (i.e., through its constructor) or updated by calling the `initMaze` method. In the sample Java code, you are also given a class called `Position` which specifies a tuple of integer numbers that represent the x-axis and y-axis values respectively. Please read the sample Java code carefully and make sure you fully understand all the methods provided in both the `MazeSolver` and the `Position` class. (For now, please ignore the `drawStatAt` method in `MazeSolver` and the `equals` method in Position, as they use some Java features that we haven't covered yet. You only need to know that the method `drawStarAt` is used to draw '`*`' in a given maze at a specified location, and the method `equals` is used to compare if a given Position object is the same as the current one.)

2. You are asked to write the following functions/methods in Haskell and Java, which might be useful for solving the maze problem. For Haskell, ensure you provide the type declaration for your functions. For Java, you may find the build-in method for String useful (e.g., `charAt(int n)` returns a character at the specified position in a string). Please refer to Java String API for more details.
    a. Write a Haskell function `mazeInBounds` which takes a maze and a pair of integers as input, and returns `True` if the given pair of integer numbers is within the bounds of the maze. Otherwise, the function returns `False`.
    b. Write a similar Java method `mazeInBounds` in `MazeSolver`, which takes

a position object as input and returns true if the given position is within the bounds of the maze. Otherwise, the method returns `false`.

c. Write a Haskell function `mazeAt` which takes a maze and a pair (i.e. a 2-tuple) of integer values as input, and returns the character at the specified location.

d. Write a similar Java method `mazeAt` in `MazeSolver`, which takes a position object as input and returns the character at the specified position in the maze.

e. Write a Haskell function `mazeFind` which takes a maze and a character as its input and returns the position of the first occurrence of the specified character in a maze.

f. Write a similar Java method `mazeFind` in `MazeSolver`, which takes a character as its input, and returns the position of the first occurrence of the given character in a maze. If the character cannot be found, return `(-1,-1)`.

g. Rewrite the `initMaze` method in `MazeSolver` such that it updates not only the whole maze, but also the positions of the start point and goal point.

Solving the Maze

3. Now you need to solve the maze problem by searching the maze space to find out a path from start point A to goal point B. To do this, depth-first search is implemented using the given function in Haskell, `searchDF`, and the given method `solveDFS` in Java. These will make use of the code you have written in Q2 above. Your next task is to ensure you get these working right. In particular, your solution should meet the following requirements:

   Your solution should show a path from A to B as a trail of asterisks `'*'` in the original maze. For example, if the maze is:

```
A#.
...
#.B
```

   Then your solution should appear as:

```
*#.
***
#.B
```

For Java, you may find the build-in class ArrayList useful which can be seen as an array but with non-fixed size, i.e., you could add/remove items to/from an ArrayList. Please refer to the Java ArrayList API for more details.

4. (optional) If you have time, or as homework, can you write a `searchBF` function in Haskell, and a `solveBFS` method in Java to search for a solution using breadth-first search?

5. Compare and contrast the solutions in Haskell and in Java. What are the key differences, and pros and cons? Discuss within your lab group.