Assignment 1

CSCI 3136: Principles of Programming Languages

Due Jan 29, 2018

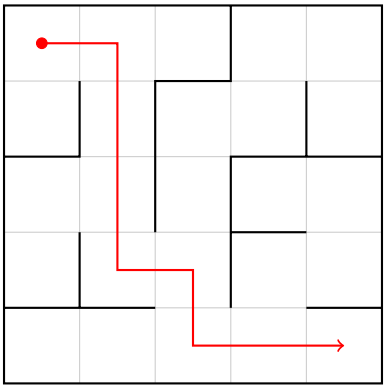
Banner ID: _____ Name: _____

Banner ID: _____ Name: _____

Banner ID: _____ Name: _____

Problem Description

In this assignment, you will write two simple programs in Haskell and Prolog. Both solve the following problem: You are given a maze of $m \times n$ cells and you are to find a path from the top-left cell to the bottom-right cell. The input specifies whether there's a wall between adjacent cells. The path must not cross any walls and must not visit any cell more than once. The following is an example:



In general, the maze is not necessarily square. However, the maze is connected and contains no cycles. In particular, there always exists a path from the start cell to the end cell, so depth-first search or breadth-first search is an appropriate search strategy.

The input to your program is given as a file storing the maze row by row. Each row contains one character per cell. The character is a hexadecimal digit obtained by taking the bitwise OR of values representing the walls of the cell, where

- 1 = wall above the cell
- 2 = wall to the left of the cell
- 4 = wall to the right of the cell
- 8 = wall below the cell

The input file for the maze above thus looks like this:

```
31d35
e63ce
346b5
ea43c
b988d
```

The output must list the cells visited by the path in order from start to finish. Each cell occupies a line of its own and is to be represented by its row and its cell, separated by a space. The output for the maze above thus looks like this:

```
1 1
1 2
2 2
3 2
4 2
4 3
5 3
5 4
5 5
```

Your Task

(a) Write a Haskell program to be run from the command line. It takes two arguments: the name of the maze file to be read and the name of the file to write the solution to. I provide a skeleton file `SolveMaze.hs` in the zip file for this assignment. This skeleton takes care of reading the input file and writing the result and provides a function stub `findPath :: Maze -> Path` that you need to implement. A `Maze` is a list of lists of `Cells`. A `Cell` stores four `Bools` indicating whether the top, left, right, and bottom walls of this cell are present. A `Path` is a list of pairs of `Ints` representing cell positions, row first, column second. Rows and columns are counted from 1.

Your implementation of `findPath` must be purely functional, that is, must not use the `IO` or `ST` monad, but you are allowed to use the `State` monad or the monad instances of `Maybe` or lists if this helps you. If you implement a plain depth-first or breadth-first search, `Data.Map` provides a dictionary implementation that you can use to store for each cell whether you have already visited it. Since there are no cycles in the maze, you are effectively exploring a tree of cells. This can be done *without* keeping track of previously visited cells, but there is no penalty if you do keep track of them. Your implementation should take $O(n \lg n)$ time.

(b) Write a Prolog program to be run from the command line. It takes two arguments: the name of the maze file to be read and the name of the file to write the solution to. Before we start the Prolog introduction in class, I will add a skeleton file `SolveMaze.pl` to the zip file for this assignment. This skeleton takes care of reading the input file and writing the result. Reading the maze file populates your database with facts representing the size of the maze and the presence of walls. For a 20×10 maze, for example, the following two facts are added to the database:

```
rows(20).
columns(10).
```

A wall above the cell $(5, 4)$ is represented as:

`wall(5,4,top).`

Since this is also the bottom wall of the cell $(4,4)$, the fact

`wall(4,4,bottom).`

is also in the database. Other possible wall positions are `left` and `right`.

Your task is to implement a predicate `path(Path)` that holds exactly if `Path` is the unique path from the start cell to the finish cell in the maze described by the above facts. if `Path` is given as a variable, the predicate must infer `Path`.

Tools to Help You

The zip file for this assignment includes a number of Haskell files and a Python3 file. The Haskell files can be compiled to obtain three programs:

- A data generator.

  Compile with

  ```
  stack ghc -- --make GenMaze.hs -o genMaze
  ```

  if you are using Stack or

  ```
  ghc --make GenMaze.hs -o genMaze
  ```

  using vanilla GHC. You run this as `genMaze <rows> <cols> <file>` to generate a maze with `<rows>` rows and `<cols>` columns and write it to the file `<file>`.

- A program for printing the generated maze in a more easy-to-look-at fashion.

  Compile with

  ```
  stack ghc -- --make PrintMaze.hs -o printMaze
  ```

  if you are using Stack or

  ```
  ghc --make PrintMaze.hs -o printMaze
  ```

  using vanilla GHC. You run this as `printMaze <file>` to print the maze in the file `<file>`.

- A program that checks your solution.

  Compile with

```
stack ghc -- --make CheckSolution.hs -o checkSolution
```

if you are using Stack or

```
ghc --make CheckSolution.hs -o checkSolution
```

using vanilla GHC.

You run this as `checkSolution <maze> <solution>`, where `<maze>` is the maze file and `<solution>` is the solution file. The output is either `OK` if the solution is correct or a message that gives a reason why your solution is incorrect.

The python file can be run as:

```
python3 solve_maze.py [-p] <maze> <solution>
```

This takes the maze file `<maze>`, finds the correct path in it, and writes the solution to `<solution>`. If you give the optional flag `-p`, it also prints the maze with the path drawn in on stdout. This python implementation is provided so you can (a) see what the solution to any given maze looks like and (b) take the imperative solution to the problem as a starting point for your Haskell and Prolog solutions.

Submission Instructions

Submit your assignment by email to `YaserAlkayale@dal.ca`. Your submission has to have exactly one attachment, a zip file containing your Haskell code in a single `.hs` file and your Prolog code in a single `.pl` file. These files must contain all the code for the complete program, including the I/O code to read the maze and write the solution. Thus, it is advisable that you start with the skeleton files I provide and add your implementations of `findPath` and `path/1`, respectively. (You are free to re-implement the I/O code yourself if you feel like it.)

Your email submission must clearly state the list of students submitting this assignment, including banner numbers!