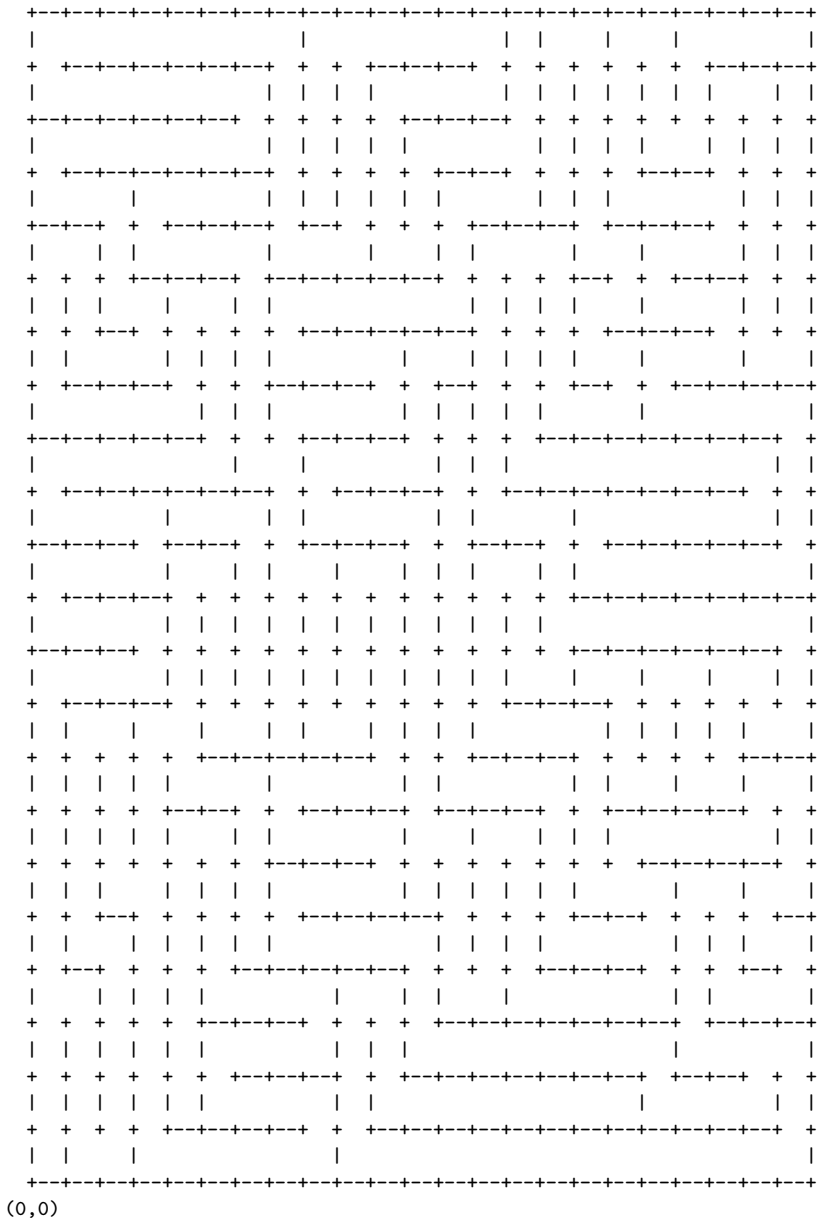
FUNCTIONAL PROGRAMMING MT2022

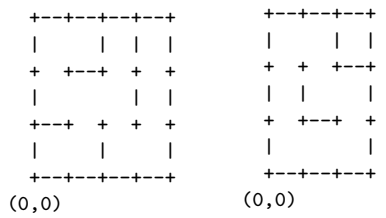
Practical 2: Solving Mazes

Deadline: sessions after the end of term, or beginning of next term (as organised by the demonatrators).

The aim of this practical is to represent, draw and attempt to solve mazes, like this large one:



and these:



one of which is small, and easy; the other of which is small but impossible.

Your report for this practical should be in the form of a commented Haskell script, answering the exercises below. You should paste into your script the results of a few tests, showing your programs working. When you have completed the exercises, you should show them to a demonstrator during a practical session. Your work should be completed, and signed-off by a demonstrator, by the end of your practical session in Week 8.

Getting started

Copy the files `Geography.lhs`, `Maze.lhs`, and `Main.lhs` from the course web page at `https://www.cs.ox.ac.uk/teaching/materials22-23/fp/` or from the course practicals directory `/usr/local/practicals/fp/prac2` into your own directory.

The file `Geography.lhs` defines some basic datatypes and functions, to do with positions and directions. Study the script as you read the following description.

We represent a place (i.e. a square) in the maze by its x and y coordinates:

```
> type Place = (Int, Int)
```

In Ordnance Survey terms you can think of the coordinates as the Eastings followed by the Northings.

We will represent a direction by one of the four cardinal compass points:

```
> data Direction = N | S | E | W  deriving (Eq, Show)
```

The **deriving** keyword has the effect of producing default instances of the type classes: in the case of *Eq*, two *Direction* are equal if and only if they are identical; and in the case of *Show* a *Direction* is printed using the strings containing the name of the value as written in the program.

We will need two functions, for which only partial definitions are given. The function *opposite* reverses a direction; for example, the opposite of North is South.

```
> opposite :: Direction -> Direction
> opposite N = S
```

The function *move* calculates the effect of moving in a particular direction from a particular place; for example, moving North from square (i, j) will take you to $(i, j + 1)$.

```
> move :: Direction -> Place -> Place
> move N (i,j) = (i,j+1)
```

Exercise 1 Complete the definitions of *opposite* and *move*. ♢

We will represent a piece of wall by a place and a direction. For example, $((3, 4), N)$ will represent that there is a wall to the North of $(3, 4)$; this means that there will also be a wall to the South of $(3, 5)$.

```
> type Wall = (Place, Direction)
```

Finally, we need a type to record the size of mazes:

```
> type Size = (Int, Int)
```

A size of (x, y) meant that the squares of the maze are numbered (i, j) where i is $0 \leqslant i < x$ and $0 \leqslant j < y$.

Representing mazes

The file `Maze.lhs` defines a type *Maze* for representing mazes, together with three functions:

- $makeMaze :: Size \to [Wall] \to Maze$ which creates a maze given its size and a list of internal walls. The function completes the maze by adding the boundary walls, and ensuring that the list of walls is complete, in the sense that if the list contains $((3, 4), N)$ then the maze will automatically also contain a wall to the south of $(3, 5)$.
- $hasWall :: Maze \to Place \to Direction \to Bool$ which tests whether the maze contains a wall in a particular direction from a particular place.
- $sizeOf :: Maze \to Size$ which returns the size of the maze.

Drawing mazes

The file `Main.lhs` contains definitions of the three mazes at the top of this document. In each case, the goal is *to find a path from the bottom-left corner to the top-right corner.*

Exercise 2 Use

```
putStr :: String -> IO ()
```

to define a function

```
> drawMaze :: Maze -> IO()
```

that draws a picture of a maze like those in this document.

You will probably want to define two subsidiary functions:

- one that draws one row of east-west oriented walls; and
- one that draws one row of north-south oriented walls, and the spaces between them.

Use the function *hasWall* to test for the presence of walls.

The main function should call these two functions with appropriate arguments to draw each row of the picture in turn. Test your function out on the small maze, and when you think you have got it right test it on the large maze. (It is normally a good idea to develop code on small examples, to make testing faster.) ♢

Solving mazes

We will now develop a function to solve a maze. The function will return a result of type

```
> type Path = [Direction]
```

giving the list of directions that must be followed to get from the start to the target. We will develop a function

```
> solveMaze :: Maze -> Place -> Place -> Path
```

The second and third arguments give the starting and target locations, respectively.

We will solve the maze using a *breadth-first search*. In effect, we will first consider all paths of length 1, then all paths of length 2, then all paths of length 3, etc., until we find a path to the target place.

The function will make use of a subsidiary function:

```
> solveMazeIter :: Maze -> Place -> [(Place, Path)] -> Path
```

The first argument is the maze; the second argument is the target place; the third argument is a list of partial solutions found so far, that is, a list of places in the maze together with a path that will lead there from the starting place.

We will deal with each partial solution in turn. If the place is the target place, then we are done. Otherwise, we can consider each of the four adjacent places, and for those for which there is not an intervening wall, form a new path to that place; these new places and paths can be appended onto the end of the list; we can then move on to the next partial solution.

Exercise 3 Define the function *solveMazeIter* following the above description, and use it to define the function *solveMaze*.

Test your function on the small maze; you will probably not be able to solve the large maze yet. ♢

The function above is not particularly efficient: if it finds a path that leads to a place that it has considered before then it will still look for more onward paths, thus repeating work; for example, in the large maze the function will consider paths such as $[N, S, N, S, N, S, \ldots]$. This also means that if the maze is impossible, the function will never terminate.

Exercise 4 Adapt your solution to the previous question by replacing your *solveMazeIter* with a new function *fastSolveMazeIter*. The new function should have an additional argument *visited* that records those places we have already considered (i.e. to which we have found paths), so as to avoid considering such places again.

Test the function on the small and large mazes (solving the large maze should take less than a second). Also, check that your new function works correctly on the impossible maze (it should give a suitable error message) and on empty mazes (without any internal walls). ♢

Changing the representation

We will now consider the representation of mazes given in `Maze.lhs`. That representation uses a single list of all the walls:

```
> data Maze = AMaze Size [Wall]
```

This is not very efficient, because searching will take a long time. Study the representation, and make sure you understand how each function works.

A better representation would be to store information about all walls corresponding to a particular direction together. We can represent a maze using the following representation:

```
> data Maze = AMaze Size [Place] [Place] [Place] [Place]
```

The first list of places represents all those places that have a wall to the North; the second represents all those places that have a wall to the South; the third represents all those places that have a wall to the East; and the fourth represents all those places that have a wall to the West.

Exercise 5 Create a new module *MyMaze*, in a file `MyMaze.lhs`, that uses this representation.

Change the *import* command in `Main.lhs` to use this new module.

Recall that if you type the command "`:set +s`" into GHCi, it will indicate the approximate time and memory space taken to evaluate each subsequent expression. Use this to compare the approximate time and memory space needed to solve the large maze for this new representation against those for the old representation; paste some test results into your report. ♢

Optional exercises

Exercise 6 Create some mazes of your own devising. Paste drawings of your mazes into your file. (The mazes do not have to be particularly difficult. Some visually attractive mazes created using elegant programs are ideal.) ♢

Exercise 7 A further enhancement to the representation would be to replace each list of places, from the representation in Exercise 5, with a binary

search tree storing the places. Adapt the representation to make use of binary search trees. *Write this new representation in a different file, so that you can include solutions to all the exercises into your report.*

If you find that your tree-based solution is not significantly faster than the MyMaze one, it may be that it generates highly unbalanced trees. If this is the case, adapt your solution so that the trees are balanced. Hint: *sort the list of places initially.* ♢

Geraint Jones, October 2022