Project 4: Mazes

Due: November 30th

The goal of this project is to write a program that generates mazes, solves mazes, and provides a GUI for people to interactively solve them.

1 Generating Mazes

Design a data definition for a maze that represents the connectivity information for a maze that exists on a grid. Imagine a maze as a piece of graph paper where some of the lines between the squares have been darkened (and thus do not allow passage) and others have not (and thus do allow passage).

```
;; a maze is a ...
```

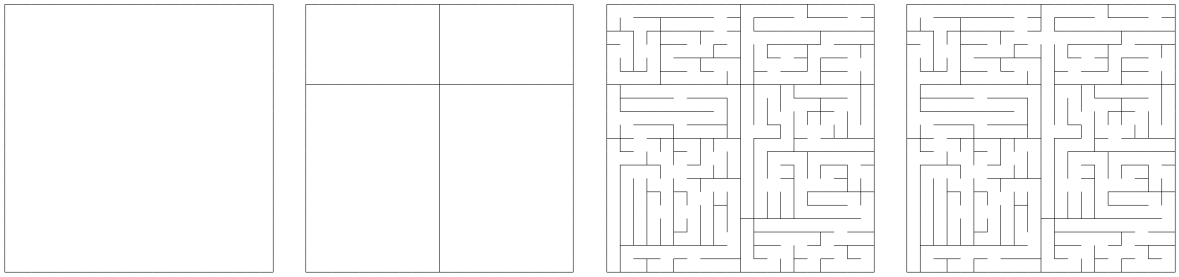
Design the function:

```
;; generate-maze : number number -> maze
;; to generate a maze of width 'w' and height 'h'
(define (generate-maze w h) ...)
```

following this algorithm:

- If `w` or `h` is `1`, generate a corridor that runs the length of the maze (either vertically or horizontally).
- Otherwise, divide the maze into four quadrants, picking two dividing lines randomly. Then, make four separate mazes in those four quadrants. Finally, connect all but one pair of the quadrants by randomly opening a hole somewhere in the line connecting them.

  These pictures show the steps in action for a particular input. On the left is a blank maze. The next step shows where the dividing lines for the quadrants were picked. In the third step, the sub-mazes have been created, and the fourth step shows the completed maze after the quadrants have been connected (the bottom left and bottom right quadrants were not connected, but the other three were).



These mazes can have beginnings and endings anywhere, but for the purposes of this assignment, take the bottom right corner as the starting position of the maze, and the top-left corner as the ending position.

HINT: You will probably have many functions that operate on coordinates in tricky ways and also call `random`. The best way to test these functions is to write helper functions that do the tricky arithmetic but do not call `random`. One way to do this is to abstract over the random numbers, leaving them as arguments to a helper function. Then, test the tricky arithmetic carefully (being sure to at least test the largest and smallest possible random values).

HINT: There are only 4 possible 2x2 mazes, so that size makes a good test for the combination of your helper functions with the calls to random.

2 Exploring Mazes

Use the `world.ss` teachpack to build a program that lets others interactively explore mazes. Your program should

- react to the player pressing arrow keys by moving an indicator of the current position around in the maze (unless it would be blocked by a wall of course),
- record the path taken by the player (showing that on the maze), and
- in the case the player retraces their steps (ie, moves back to a square it has just come from in the previous move), erase that portion of the trail.

To do this, you will need to design a data definition for the world (from `world.ss`) to cover all of the information you need to provide the functionality above.

3 Helpful Exploration

Extend your maze playing system so that, when the player presses the "h" key (for "help") your program finds the path from the player's current position in the maze to the exit, and then takes just a single step along that path. Thus, if someone were to just repeatedly type "h" your program should move them to the exit.

A straightforward way to do this is to find a path from the current location to the exit and then picks the first step along this path, each time the player presses the "h" key. While this works, it is inefficient, and probably will be noticeable when playing the game.

Instead, when making a maze, build a table that indicates which direction is the best direction to go from any given spot in the maze. Because mazes generated from the above algorithm have the property that there is only a single path from any given spot in the maze to the exit (assuming that you do not retrace your steps) you can adapt the depth-first searching algorithm from class to build this table.

Starting from the exit, at each step in the traversal of the maze, if you encounter a new neighbor of the current node (ie, a space in the maze where you have not been before), you know that the traveling through current node is the best way to get there. So, to build a table that records which neighbor of each node is on the shortest path to the exit, just traverse the graph, adding entries to the table for each new node encountered. Then, when the player types "h", simply look in the table.

4 Requirements

Be sure to follow these requirements when implementing your project:

- You must not write any functions whose recursive calls are processed by other recursive functions, even built-in recursive functions like `map`, `filter`, etc. (Note that `append` processes all but the last of its arguments in this manner.) Instead, use an accumulator. You may wish to write your functions directly first and then transform them to accumulator style. This two-step approach is usually easier and gives you a good way to test the accumulator-style function.
- You must put the definitions in your program into a sensible order. This means your program should be organized in meaningful sections, and each section should be labeled and presented top-down (ie, main functions come before helper functions).
- Similarly, organize your test cases well. You must also use the `testing.ss` teachpack for all of your tests, except for those functions that use random.