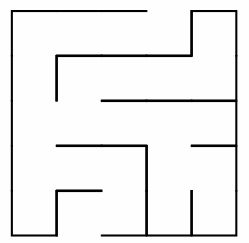
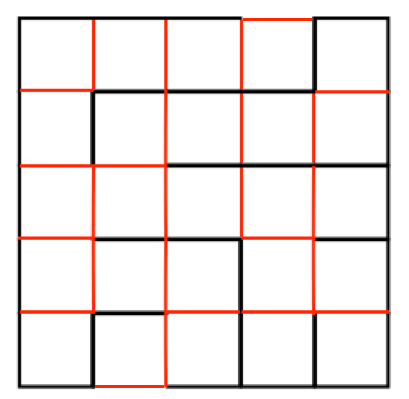
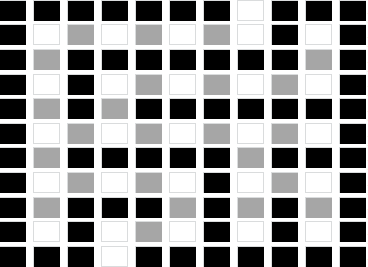
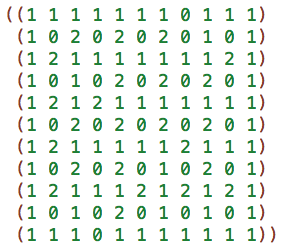
**4- Maze Solver:**

***4.1 High-level description***

Our Maze Solver program allows users to input the dimensions of the maze they would like to create. For example, a user can enter any two positive integers. Our program then uses recursive division to generate a random maze. Before we explain what is recursive division and how it works to design our mazes, let us first talk about how we represent a maze in Scheme. In Figure 1 we illustrate how we converted a typical maze into our Scheme-maze 

Maze: typical maze representation Maze: cellular representation

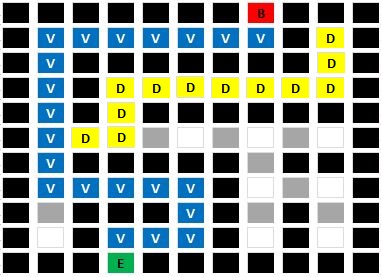
  Maze: Scheme representation Maze: Cellular representation

In order to represent the maze in Figure 1.a in Scheme we have to first think of it as if it as a “cellular grid”. The lines in Figure 1.b should in fact be cells if we want to represent that maze in Scheme. You can see that in Figure 1.c. All black lines represent a “wall” in the maze and all red lines represent an “open gate”. We give all walls a representative value equal to 1 and all gates a value equal to 2. Lastly, the while cells represent a tile in the maze. We give all white cells a representative value equal to 0. As you can see, such representation results in a (2n+1) (2m+1) Scheme-maze.

There are four directions the non-deterministic program (we will call it player from now on) could take, namely forward, backward, right or left. The current position of the player gets passed to the direction’s function, which “moves” the player. The position of the player is represented in a (row\_index, column\_index) pair. A move to the right means that new position is (row\_index, column\_index +1), left is (row\_index, column\_index -1), forward is (row\_index+1, column\_index) and backward is (row\_index-1, column\_index). Those four directions are amb’s finite list of choices as we introduced in Section 2 of this document.

Next we introduce our game rules and restrictions. Each position (row\_index, column\_index) has a value associated with it. As we mentioned earlier, this value could be either 0, 1 or 2. If the player makes a move into a position that has a value 0 or 2 that means it is a valid move and we can update the player’s current position to this new position. Then we store it into a list we call *correct path*. Otherwise, it is an invalid move and the player has to go back and make another choice. Also, if the player’s current position is equal to the starting point of the maze, then the player should not move backward. In other words, a backward move in this case is invalid and would result in failure of the current program. Our last rule ensures that when the player chooses a direction for their next move, the new position is not equal to his previous position. This is to make sure that the player is always advancing.

What if the player made a sequence of choices that led him to a dead-end? Consider this case below.



[Figure Dead-end]

The player can neither move right, left, forward, nor backward. The player cannot move backward is due to our restrictions which state that a player’s new position cannot be the same as their previous one. Of course, the player cannot move in the other directions because of our first rule, which states that a player cannot move into a cell that has a value of 1 (because you cannot move into a wall!). That means the player has exhausted all options and is now stuck. Furthermore, that means that our current *correct path* list does not have the correct path to solving the matrix. Well, that is only half true. The correct path list had the correct path up until a point where the player made a valid, yet wrong decision that led to this dead-end. That is why we came up with a new way of solving this problem. We allow the player to change the value of his current position if he found that the path they chose led to a dead-end. In that case, we pop the player’s current position from the correct path list and assign their previous correct position (*cadr* of *correct path* list) to be their current position and we hope that this would be sufficient to get the player out of that dead-end. If it was not, then we repeat that *backtracking* process until the player is out of that dead-end. With those rules and restrictions, a player can solve any given Scheme-maze.