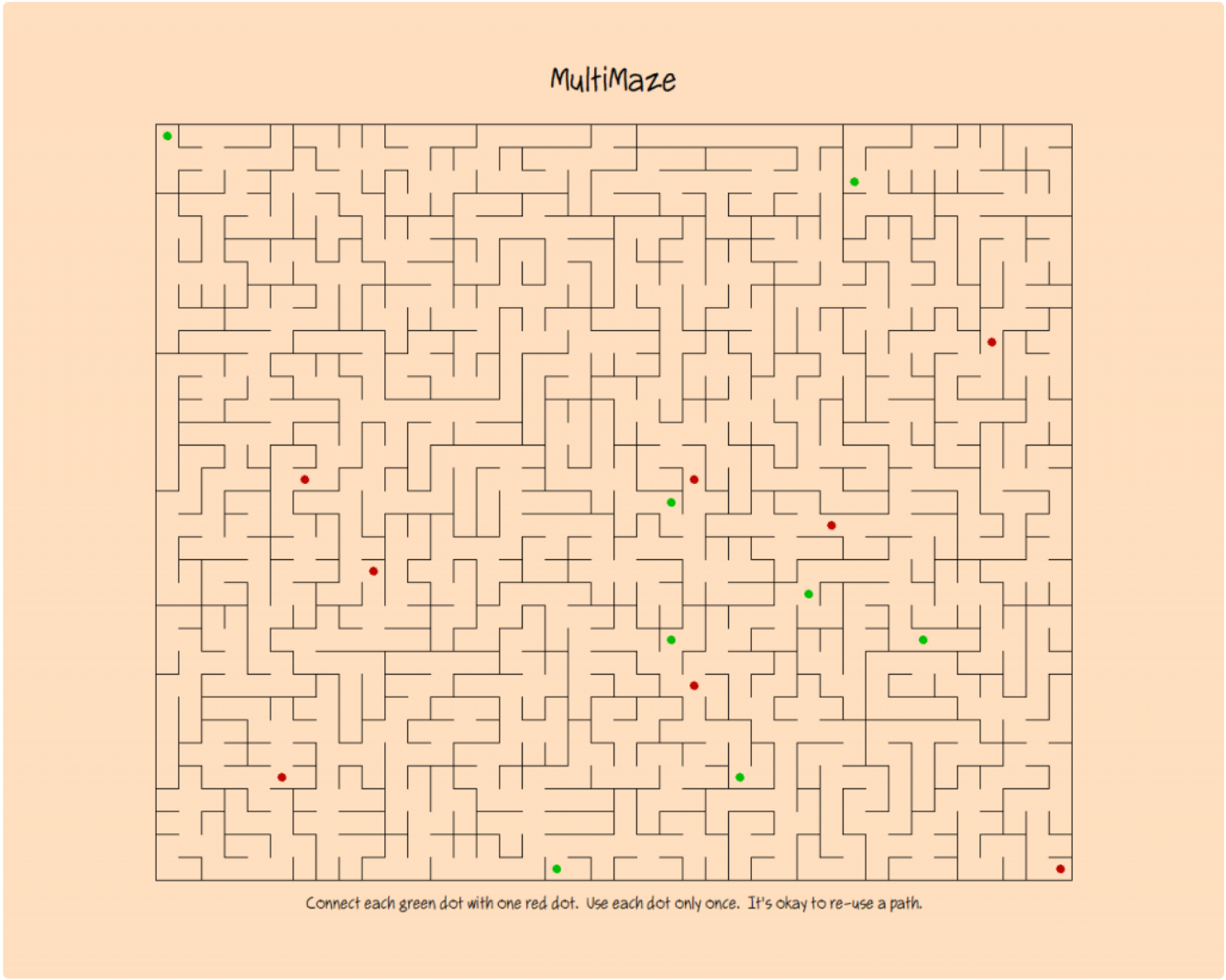
The Amazing MultiMaze

A MultiMaze is a maze with multiple, player-selected beginnings and endings. Here is a sample:



The object is to connect each green dot with one, and only one, red dot (with or without overlapping paths).

The Plain English program that generates MultiMazes is described below. We use the Aldous-Broder algorithm because it always generates a *perfect* maze, ie, a maze with without loops or closed circuits, and without any inaccessible areas. It is also a *uniform* maze generator, which means it generates all possible mazes of a given size with equal probability. And it's a very simple algorithm that requires no extra storage or stack to implement. It's not super fast, but it is *reasonably* fast, typically generating a full-page maze in less than 3 seconds.

The data structures are almost trivial. We define a maze in our program like this:

```
A maze is a thing with a width, a height, a cell size and some cells.
```

And a cell is defined this way:

```
A cell is a thing with a box,
a left flag, a top flag, a right flag, a bottom flag,
a visited flag, a start flag and an end flag.

A neighbor is a cell.
```

Each cell's box describes its position in the maze's "array". The left, top, right, and bottom flags indicate whether or not the corresponding side of the cell is open or closed; the visited flag is required by the Aldous-Broder algorithm, and the start and end flags indicate which cells will be marked with a green or red dot. (By the way, that first definition fit on one line on the screen when I wrote it because we use a tight, proportionally-spaced font and our uncluttered editor always spans the entire width of the screen. Turns out the things that make Plain English sentences easy-to-read — like one thought per line — are not the same things that make artificial, mathematically-influenced programming languages easy-to-read.)

The main routine in our program reads like this:

```
To run:
Start up.
Clear the screen with the tan color.
Create a maze 10 inches by 8 inches using 1/4 inch for the cells.
Display the maze with title and instructions.
Destroy the maze.
Wait for the escape key.
Shut down.
```

Most of that is self-explanatory, so I'll jump to the interesting stuff. This is the top-level create routine:

```
To create a maze a width by a height using a size for the cells:
Allocate memory for the maze.
Put the width into the maze's width.
Put the height into the maze's height.
Put the size into the maze's cell size.
Create some cells for the maze.
Apply the Aldous-Broder algorithm to the maze.
Select starting and ending cells in the maze.
```

The latter three lines do most of the work. The first calls this routine to create the maze's cells:

```
To create some cells for a maze:
If a spot's left is greater than the maze's width, break.
If the spot's top is greater than the maze's height,
add the maze's cell size to the spot's left; repeat.
Create a cell of the maze's cell size at the spot.
Append the cell to the maze's cells.
Add the maze's cell size to the spot's left.
If the spot's left is less than the maze's width, repeat.
```

```
Put 0 into the spot's left.
Add the maze's cell size to the spot's top.
Repeat.
```

Note that the cells are stored in a doubly-linked list, not in an array (as you might expect), because Plain English does not have a native array type (arrays are not native to English grammar).

This is the routine that creates an individual cell:

```
To create a cell of a size at a spot:
Allocate memory for the cell.
Put the spot and the spot plus the size into the cell's box.
Set the cell's left flag.
Set the cell's top flag.
Set the cell's right flag.
Set the cell's bottom flag.
Clear the cell's visited flag.
```

Once we've got our cells, we connect them in the right spots using the Aldous-Broder algorithm I mentioned earlier. This is the Plain English implementation:

```
To apply the Aldous-Broder algorithm to a maze:
Pick a cell in the maze. Set the cell's visited flag.
Loop.
If all of the maze's cells have been visited, break.
Pick a neighbor of the cell in the maze.
If the neighbor's visited flag is not set,
connect the cell with the neighbor.
Set the neighbor's visited flag.
Put the neighbor into the cell.
Repeat.
```

I won't explain what that does because I'd simply be repeating the "code" in this paragraph. Instead, I'll show you the support routines that make that work. Like this one:

```
To pick a neighbor of a cell in a maze:
Void the neighbor.
Loop.
Pick a number between 1 and 4.
If the number is 1,
find the neighbor above the cell in the maze.
If the number is 2,
find the neighbor to the right of the cell in the maze.
If the number is 3,
find the neighbor below the cell in the maze.
If the number is 4,
find the neighbor to the left of the cell in the maze.
If the neighbor is nil, repeat.
```

A neighbor might be above, below, to the right, or to the left of the current cell. We want to pick one of those, if it exists, randomly. So we pick a number between 1 and 4 (corresponding to above, right, below, and left) and see what we can find there. If we fail to find a neighbor in that relative location, we try again and again until we succeed. After all, every cell has at least 2 neighbors.

The routine below is typical of the four “find” routines called by the above routine:

```
To find a neighbor above a cell in a maze:
Void the neighbor.
Loop.
Get the neighbor from the maze’s cells.
If the neighbor is the cell, repeat.
If the neighbor is nil, break.
If the neighbor’s bottom line is the cell’s top line, break.
Repeat.
```

Now this is how we connect one cell with another. Cells start with all of their wall flags set, so we simply have to remove the appropriate walls (in both cells) to make a path:

```
To connect a cell with a neighbor:
If the cell’s top line is the neighbor’s bottom line,
clear the cell’s top flag;
clear the neighbor’s bottom flag; exit.
If the cell’s right line is the neighbor’s left line,
clear the cell’s right flag;
clear the neighbor’s left flag; exit.
If the cell’s bottom line is the neighbor’s top line,
clear the cell’s bottom flag;
clear the neighbor’s top flag; exit.
If the cell’s left line is the neighbor’s right line,
clear the cell’s left flag;
clear the neighbor’s right flag; exit.
```

We know the passed cells are neighbors, but we didn’t keep track of their relative locations, so we check all four. (And again, all those IFs fit on one line each on the screen in our editor.)

And that’s it for the Aldous-Broder algorithm, except to see if we’re done, which is this routine’s job:

```
To decide if all of some cells have been visited:
Get a cell from the cells.
If the cell is nil, say yes.
If the cell’s visited flag is not set, say no.
Repeat.
```

We now have a *perfect*, *uniform* maze, and we just have to select some starting and ending points. We pick eight of each, almost at random. I say *almost* because we always select the top left cell as a starting point, and the bottom right cell as an ending point, to make sure that there is a least one long path available for the player.

```
To select starting and ending cells in a maze:
Set the maze’s cells’ first’s start flag.
Set the maze’s cells’ last’s end flag.
Loop.
Pick a cell in the maze.
If the cell’s start flag is set, repeat.
If the cell’s end flag is set, repeat.
```

```
Add 1 to a count. If the count is greater than 14, break.
If the count is odd, set the cell's start flag; repeat.
If the count is even, set the cell's end flag; repeat.
```

And that's the gist of the whole program. A couple of side notes on Plain English looping, however, might be helpful here.

1. When there is no explicit LOOP label, REPEAT means "repeat from the top of the routine".

2. BREAK always passes control to the first statement following the *last* REPEAT.

3. Looping through lists almost always happens like this:

```
To draw some cells:
Get a cell from the cells.
If the cell is nil, break.
Draw the cell with the black pen.
Repeat.
Refresh the screen.
```

If a nil pointer is passed to the "Get" routine, the first item on the list, if any, is returned. "A cell" in this case, indicates a new local variable initialized to zero, so that'll work. If a non-nil pointer is passed to the "Get" routine, the next item on the list, if any, is returned. A nil pointer is returned when the whole list has been processed. All of that is done, not by the compiler, but by a generic routine in our standard "noodle" library:

```
To get a thing from some things:
If the things are empty, void the thing; exit.
If the thing is nil, put the things' first into the thing; exit.
Put the thing's next into the thing.
```

But that's all stuff for other articles. Back to the maze. This is how we actually draw those cells, taking all of those little flags into account:

```
To draw a cell with a pen:
If the cell's left flag is set,
draw the cell's left line with the pen.
If the cell's top flag is set,
draw the cell's top line with the pen.
If the cell's right flag is set,
draw the cell's right line with the pen.
If the cell's bottom flag is set,
draw the cell's bottom line with the pen.
Put the cell's width divided by 3 into a width.
If the cell's start flag is set,
draw a dot the width wide at the cell's box's center with the dark green pen.
If the cell's end flag is set,
draw another dot the width wide at the cell's box's center with the dark red pen.
```

Almost forgot! This is how we pick a random cell anywhere in the maze:

```
To pick a cell in a maze:
Pick a number between 1 and the maze's cells' count.
Void the cell.
```

```
Loop.
Get the cell from the maze's cells.
If the cell is nil, break.
Add 1 to a count. If the count is the number, break.
Repeat.
```

And that's all I have to say about that. I'm speechless. Amazing.