ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 458E
Functional Programming

Term Project

Ali Emre Kaya
150210097
kayaemr21@itu.edu.tr

SPRING 2025

Contents

1 Game of Life

Game of Life is a game designed by mathematician John Horton Conway in 1970. It is Turing complete, and can be simulated with any Turing machine. The traditional Game of Life is a game in which units die and revive in a 2-dimensional infinite plane, with each unit having only 2 states (dead and alive) according to certain rules [1].

Rules:

- A dead cell with exactly three living neighbors becomes alive.
- A living cell with two or three living neighbors remains alive; otherwise, it dies.

1.1 Oscillators

Oscillators are models that return to the states they started with after a while. In this assignment, I ran 40 steps for 5 different oscillators and produced results.

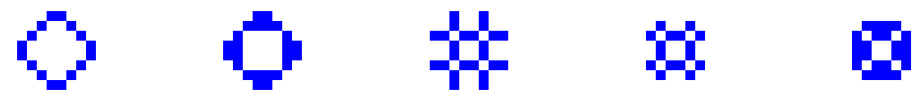
Pinwheel returns to its old state after 4 steps.

Frame 0 Frame 1 Frame 2 Frame 3 Frame 4

Blinker returns to its old state after 2 steps.

Frame 0 Frame 1 Frame 2 Frame 3 Frame 4

Octagon 2 returns to its old state after 5 steps.

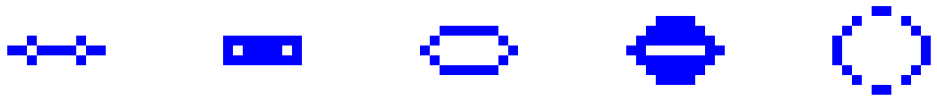


Frame 0 Frame 1 Frame 2 Frame 3 Frame 4

Glider returns to its old state after 4 steps.

Frame 0 Frame 1 Frame 2 Frame 3 Frame 4

Pentadecathon returns to its old state after 15 steps.



Frame 0 Frame 1 Frame 2 Frame 3 Frame 4

2 Code

In general, I had to write a program that would apply rules on the grid and evaluate it as needed. In Assignment 2 [2], I had actually worked with neighbors on the grid, so I proceeded with this assignment by playing around with the neighbor evaluation and cell editing codes from the previous assignment.

2.1 Rules

```
154 -- Next version of Grid according to Rules
155 nextGen :: Grid -> Grid
156 nextGen grid =
157   [[cellNextState (x,y) | y <- [0..gridSize-1]] | x <-
      [0..gridSize-1]]
158   where
159     cellNextState pos =
160       let neighbors = getNeighbors grid pos
161           aliveCount = countAlive neighbors
162           currentCell = getCell grid pos
163       in case currentCell of
164           1 -> if (aliveCount == 2 || aliveCount == 3)
165                 then 1
166                 else 0
167           0 -> if (aliveCount == 3)
168                 then 1
169                 else 0
170           _ -> 0
```

The code above checks the two basic rules in the Game of Life and gives the cell the value it should have. It works separately for each cell, checks the status of its neighbors and ensures the continuity or change of its status within the framework of the specified rules.

I used the same functions that I used in the Schelling assignment to check the neighborhood statuses, and I also wrote the `countAlive` function to keep the live cell count.

2.2 Steps

```
200 let loop :: Grid -> Int -> IO ()
201     loop _ j | j == loopCount + 1 = return ()
202     loop matrix j = do
203       let image = matrixToImage matrix
204           fileName = dirName ++  "/frame" ++ show j ++ ".
      png"
205       savePngImage fileName (ImageRGB8 image)
206       putStrLn ("Saved: " ++ fileName)
207       loop (nextGen matrix) (j + 1)
208 loop grid 0
```

The code above loops as many times as `loopCount`, which is defined somewhere in the first lines of the main code. While performing the loop operation, it first saves the current matrix as png, then runs the `nextGen` function on this matrix and continues its operation recursively with the new matrix.

2.3 Grid and Image Operations

```
8  type Grid = [[Int]]
9  type Position = (Int, Int)
10
11 gridSize :: Int
12 gridSize = 20
13
14 cellSize :: Int
15 cellSize = 10
16
17 imageSize :: Int
18 imageSize = gridSize * cellSize
19
20 colorFor :: Int -> PixelRGB8
21 colorFor 0 = PixelRGB8 255 255 255   -- white
22 colorFor 1 = PixelRGB8 0 0 255       -- blue
23 colorFor _ = PixelRGB8 0 0 0         -- error!
```

I defined the types to make it clearer and easier to define the functions (similar to assignment 2). The rest was already given in the skeleton code, I set the grid size equal to 20 for a better look.

I defined the Oscillators with Grids as below. I also created a function `resizer` below where I can open extra space so that they can move and create their states in the next steps. The first parameter that the `resizer` function takes is the grid itself, the second is the size of the grid that will be created, and the last one specifies where the existing grid will be created in the larger grid.

```haskell
-- Expand with 0s
resizer :: Grid -> Position -> Position -> Grid
resizer source (targetH, targetW) (startY, startX) =
  [ [ getValue y x | x <- [0..targetW - 1] ] | y <- [0..
   targetH - 1] ]
  where
    sourceH = length source
    sourceW = length (head source)

    getValue y x
      | y >= startY && y < startY + sourceH &&
        x >= startX && x < startX + sourceW =
          source !! (y - startY) !! (x - startX)
      | otherwise = 0

-- (p3) Glider
pattern4_ :: Grid
pattern4_ = [
  [0,1,0],
  [0,0,1],
  [1,1,1]
  ]

pattern4 = resizer pattern4_ (20,20) (5,5)
```

Finally, I convert the grid to PNG image with using the `savePngImage` function from the `JuicyPixels` library in Haskell.

REFERENCES

[1] Wikipedia contributors. Conway's game of life — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Conway%27s_Game_of_Life&oldid=1296879698, 2025. [Online; accessed 24-June-2025].

[2] Ali Emre Kaya. Assignment 2 - functional programming (blg458). https://github.com/aliemre2023/ITU-CMPE/tree/main/BLG458%20-%20Functional%20Programming/assignment2, 2024. Accessed: 2025-06-24.