BLG 458E - Functional Programming

Final Assignment

Feyza Nur Keleş
820220332

June 26, 2025

Contents

1 Problem Definition

1.1 Conway's Game of Life

Conway's Game of Life is a cellular automaton on a 2D grid, where each cell is either alive or dead. The next state of each cell is determined by its 8 neighbours.

Oscillators: An oscillator is a pattern that returns to its original state after a fixed number of generations.

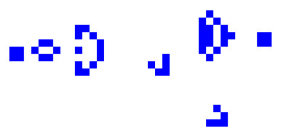
1.2 Project Goal

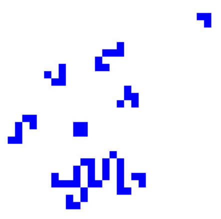
This project aims generating and visualizing five oscillator patterns from Brown et al. using Haskell. Each pattern must be evolved across generations and saved as images in each T.

2 Implementation Motivation

Initially, I planned to simulate more complex and interesting patterns such as the **Gosper Glider Gun**, a **Glider Loop** (inspired by Queen Bee Shuttle architecture), and the **Snark** reflector. These patterns are significant in Game of Life history because they emit gliders, redirect signals, and form components for logic gates or memory systems.

However, after reviewing the assignment requirements more carefully, I realized that the focus was specifically on **oscillator** patterns. Most of the patterns I initially chose are not exactly oscillators: they either produce gliders indefinitely (like the glider gun), or redirect them (like the Snark), but do not return to a stable configuration due to the gliders constantly moving across the grid. However, here is the resultant GIF I ended up with these patterns;


(a) Gosper Glider Gun


(b) Snark Reflector

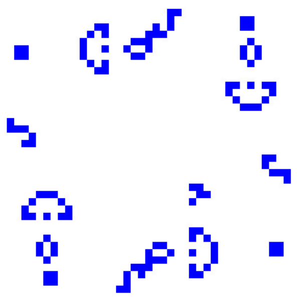

(c) Glider Loop

Figure 1: My Three Initial Implementations - Not Used in Final Submission (Clickable)

As a result, I replaced my pattern set with five historically documented oscillators that meet the criteria. This change made sure that all patterns analyzed in the Results section actually demonstrate oscillatory behavior with defined periods.

3 Implementation Explanation

3.1 Game of Life Implementation Logic

I implemented the rules of Conway's Game of Life using pure functions. The rule logic is defined in `shouldLive`, which evaluates whether a cell will be alive in the next generation. Neighbor counts are calculated using `countNeighbors`, which examines the eight adjacent cells within grid bounds. The next state of the entire grid is computed with `nextGeneration` by applying these rules cell by cell.

```
shouldLive :: Int -> Int -> Bool
shouldLive currentState neighborCount
  | currentState == 1 && (neighborCount == 2 || neighborCount == 3) = True
  | currentState == 0 && neighborCount == 3 = True
  | otherwise = False

countNeighbors :: [[Int]] -> Int -> Int -> Int
countNeighbors grid x y =
  let height = length grid
      width = length (head grid)
      neighbors = [ (x + dx, y + dy) | dx <- [-1,0,1], dy <- [-1,0,1], not
      validNeighbors = filter (\(nx, ny) -> nx >= 0 && ny >= 0 && nx < widt
  in sum [ grid !! ny !! nx | (nx, ny) <- validNeighbors ]

nextGeneration :: [[Int]] -> [[Int]]
nextGeneration grid =
  let height = length grid
      width = length (head grid)
  in [ [ if shouldLive (grid !! y !! x) (countNeighbors grid x y)
         then 1
         else 0
       | x <- [0..width-1]
       ]
     | y <- [0..height-1]
     ]
```

3.2 Pattern Centering and Evolution Helpers

Each pattern is centered on a fixed-size grid using the `centerPattern` function, which pads the surrounding space with zeros. This ensures consistent visualization. The evolution is handled by `evolvePattern`, which repeatedly applies `nextGeneration` to simulate multiple steps.

```
centerPattern :: [[Int]] -> [[Int]]
centerPattern pattern =
  let patternHeight = length pattern
      patternWidth = length (head pattern)
      offsetY = (gridSize - patternHeight) `div` 2
      offsetX = (gridSize - patternWidth) `div` 2
  in [ [ if y >= offsetY && y < offsetY + patternHeight &&
            x >= offsetX && x < offsetX + patternWidth
         then (pattern !! (y - offsetY)) !! (x - offsetX)
         else 0
       | x <- [0..gridSize-1]
       ]
     | y <- [0..gridSize-1]
     ]

evolvePattern :: [[Int]] -> Int -> [[Int]]
evolvePattern initial 0 = centerPattern initial
evolvePattern initial n = evolvePattern
(nextGeneration initial) (n - 1)
```

3.3 Pattern Initialization

The oscillator patterns used in this project were given in RLE format. To convert them into usable Haskell `[[Int]]` matrices, I wrote a separate parser that reads the RLE strings and generates binary matrices. A placeholder for the RLE parsing code is included below:

1. Clean and parse the RLE:

```
parseLifeRLE :: String -> [[Bool]]
parseLifeRLE rle = buildRows (tokenize clean)
  where clean = takeWhile (/= '!') $ dropWhile (/= '\n') rle
```

2. Tokenize RLE into (count, symbol):

```
tokenize :: String -> [(Int, Char)]
tokenize [] = []
tokenize str =
  let (num, rest) = span isDigit str
      n = if null num then 1 else read num
  in (n, head rest) : tokenize (tail rest)
```

3. Build the grid from tokens:

```
buildRows :: [(Int, Char)] -> [[Bool]]
buildRows = reverse . build [] []
  where
    build row acc [] = reverse row : acc
    build row acc ((n,'o'):xs) = build
    (replicate n True ++ row) acc xs
    build row acc ((n,'b'):xs) = build
    (replicate n False ++ row) acc xs
    build row acc ((n,'$'):xs) = build []
    (replicate (n-1) [] ++ reverse row : acc) xs
    build row acc (_:xs) = build row acc xs
```

4. Convert to padded `[[Int]]` matrix:

```
gridToMatrix :: [[Bool]] -> [[Int]]
gridToMatrix g = map (map fromEnum . pad) g
  where maxW = maximum (map length g)
        pad row = row ++ replicate (maxW - length row) False
```

Each pattern function (e.g., `pattern1`, `pattern2`, etc.) simply returns the evolved state of a fixed matrix after a given number of generations. The only difference between these functions is the specific matrix used, representing the initial state of the corresponding oscillator.

```
pattern1 :: Int -> [[Int]]
pattern1 generations = evolvePattern pinwheelInitial generations
  where
    pinwheelInitial =
      [ [0,0,0,0,0,0,1,1,0,0,0,0]
      , [0,0,0,0,0,0,1,1,0,0,0,0]
      , [0,0,0,0,0,0,0,0,0,0,0,0]
      , [0,0,0,0,1,1,1,1,0,0,0,0]
      , [1,1,0,1,0,0,1,0,1,0,0,0]
      , [1,1,0,1,0,1,0,0,1,0,0,0]
      , [0,0,0,1,0,0,0,1,1,0,1,1]
      , [0,0,0,1,0,0,0,0,1,0,1,1]
      , [0,0,0,0,1,1,1,1,0,0,0,0]
      , [0,0,0,0,0,0,0,0,0,0,0,0]
      , [0,0,0,0,1,1,0,0,0,0,0,0]
      , [0,0,0,0,1,1,0,0,0,0,0,0]
      ]
```

4 Results

To verify the correct behavior of each oscillator, we were expected to generate images in every generation with the given initial code. These snapshots help visualize how each pattern evolves and eventually returns to its original state.

For each pattern, I generated 36 frames. This made it possible to observe full cycles, even for the longest oscillators like Toad Hassler and Gourmet. (Clicking on the images opens full GIF animations that show the entire cycle.)

4.1 Pinwheel (Period 4)

The Pinwheel is a period-4 oscillator that shifts and rotates through four distinct phases. After four generations, the pattern returns to its starting configuration and repeats.

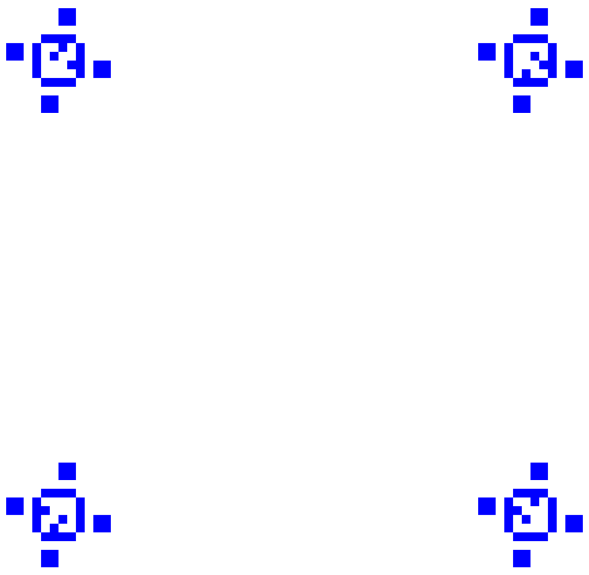


Figure 2: Frames 0 to 3 for Pinwheel (clickable)

4.2 Queen Bee Shuttle (Period 30)

The Queen Bee Shuttle has a period of 30 and features moving gliders that temporarily appear during its evolution. The pattern completes its cycle after 30 generations.

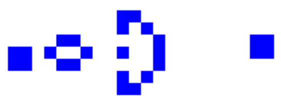


Figure 3: Queen Bee Shuttle (clickable)

4.3 Worker Bee (Period 9)

The Worker Bee is a compact version of the Queen Bee Shuttle. It has a period of 9 and oscillates in a stable, localized region on the grid.

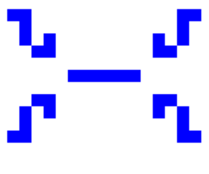


Figure 4: Worker Bee (clickable)

4.4 Toad Hassler (Period 36)

Toad Hassler is made up of interacting toad (larger worker bees) patterns that result in a long period of 36 generations. The animation shows a symmetric and rhythmic evolution.

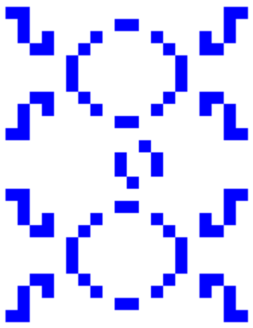


Figure 5: Toad Hassler (clickable)

4.5 Gourmet (Period 32)

The Gourmet oscillator undergoes complex transitions across its 32-generation cycle. Despite its chaotic appearance, it returns precisely to its starting structure.

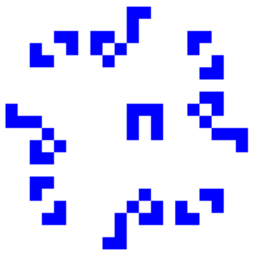


Figure 6: Gourmet (clickable)

5 Comment on the Obtained Results

All oscillators behaved as expected, returning to their original state after the correct number of generations. For example, the Pinwheel repeated after 4 generations, while the Queen Bee Shuttle required 30.

Animations helped confirm these results. Even with small grids, each pattern stayed centered, and no boundary errors occurred, thanks to dynamic grid and cell sizing.

The oscillators were initialized using hardcoded matrices. Their correct behavior verified that the parsing logic and simulation rules were implemented properly.

6 Overall Conclusion

This project implemented known oscillator patterns from Conway's Game of Life in Haskell. All five selected patterns evolved correctly and visually confirmed their expected periods. The helper functions for state evolution, pattern centering, and grid visualization worked consistently across all patterns.

Patterns were represented as hardcoded binary matrices, and a separate RLE parser was written to convert it to binary matrices. I could include the RLE to matrix converter in the code I provided but I did not want to complicate the code asked with something that was not initially expected. Also, the frame amount to be runned for is hard coded too, but it can be detected manually with a more complicated implementation. However, once parsed, every pattern function only varied in its starting matrix and produced the same amount of frames.

Overall, the project met all functional requirements, responded to feedback about hard-coded grids, and produced clean visual results.