CONWAY GAME OF LIFE

Course: Concepts of Programming Languages

1.Introduction

Conway's Game of Life is a classic cellular automaton devised by John Conway. It is a zero-player game where the evolution of the system depends entirely on its initial state and a fixed set of rules.

This project implements the same Game of Life logic using two different programming paradigms:

- Imperative Programming
- Functional Programming

The purpose of this project is not only to solve the problem, but to compare how different paradigms approach the same computational task, highlighting differences in:

- State management
- Control flow
- Abstraction
- Readability
- Conceptual thinking

2.Problem Description

The game consists of a 2D grid of cells where:

- Each cell is either **alive (1)** or **dead (0)**.
- At every generation, the state of the grid updates according to these rules:

1. A live cell with **2 or 3 live neighbors** survives.
2. Aa alive cell with **fewer than 2 or more than 3** live neighbors dies.
3. A dead cell with **exactly 3 live neighbors** becomes alive.
4. All other cells remain dead.

The goal is to compute the **next generation** of the grid repeatedly.

3.Overview of Programming Paradigms

3.1 Imperative Programming Paradigm

Imperative programming focuses on **how** a program executes:

- Step-by-step instructions
- Explicit loops
- Mutable state
- Commands that change program state

This paradigm closely resembles how a computer executes instructions.

3.2 Functional Programming Paradigm

Functional programming focuses on **what** the program computes:

- Pure functions
- Immutability
- Higher-order functions
- Recursion instead of loops
- Declarative logic

This paradigm emphasizes **expressiveness, correctness, and predictability**.

4.Functional Programming Implementation

4.1 Design Philosophy

The functional implementation follows these principles:

- **Immutable data structures** (tuples instead of lists)
- **No mutation** of the grid
- **Recursion instead of loops**
- **Higher-order functions**
- **Pattern matching** for rule application

The grid is represented as: (This guarantees immutability.)

```
Grid = Tuple[Tuple[int, ...], ...]
```

4.2 Higher-Order Function: Neighbor Counter

```
def make_neighbor_counter(grid: Grid) -> Callable[[int, int], int]:
```

Concepts Applied:

- Higher-Order Functions (HOF) → Instantiation
  This function returns another function (count).
- Closure
  The inner function captures grid, rows, and cols.
- Functional abstraction

Neighbor counting is expressed using a generator expression, avoiding explicit loops.

4.3 Pattern Matching for Rules

```
match (cell, live_neighbors):
    case (1, n) if n in (2, 3):
```

Concepts Applied:

- Pattern Matching
- Declarative rule expression
- No nested conditionals

This makes the rules easier to read and mathematically expressive.

4.4 Recursive Grid Construction

```
def step_row(next_cell: Callable[[int,int],int], r: int, c: int, cols: int) -> Tuple[int, ...]:
    return (next_cell(r, c),) + step_row(next_cell, r, c+1, cols)


def step_grid(next_cell: Callable[[int,int],int], r: int, rows: int, cols: int) -> Grid:
    return (step_row(next_cell, r, 0, cols),) + step_grid(next_cell, r+1, rows, cols)
```

Concepts Applied:

- Recursion
- Immutability
- No indexing loops

Each generation is built by creating a new grid, not modifying the old one.

4.5 Functional Step Function

```
def step(grid: Grid) -> Grid:
```

- Acts as a pure function
- Same input → same output
- No side effects

This makes the function easier to test, reason about, and reuse.

5.Imperative Programming Implementation

5.1 Design Philosophy

The imperative implementation emphasizes:

- Explicit control flow
- Mutable data structures
- Nested loops
- Direct state updates

The grid is represented as:

```
Grid = List[List[int]]
```

5.2 Neighbor Counting Using Loops

```
def count_live_neighbors(grid: Grid, r: int, c: int) -> int:
    """Count live neighbors around (r, c) using loops."""
    rows = len(grid)
    cols = len(grid[0])
    count = 0

    for dr in (-1, 0, 1):
        for dc in (-1, 0, 1):
            if dr == 0 and dc == 0:
                continue  # skip itself
```

Concepts Applied:

- Nested loops
- Explicit indexing
- Mutable counters
- Step-by-step execution

This style closely matches traditional algorithmic thinking.

5.3 Imperative Step Function

```
def step(grid: Grid) -> Grid: ...
    return new_grid
```

Key Characteristics:

- Uses for loops for traversal
- Builds a mutable new_grid
- Uses conditional statements (if / elif / else)
- Control flow is explicit and sequential

6. Paradigm Comparison

Concept	Imperative	Functional
State	Mutable lists	Immutable tuples
Control Flow	For-loops	Recursion
Rule Expression	Conditionals	Pattern matching
Side Effects	Present	None
Abstraction	Procedures	Functions
Readability	Step-based	Declarative
Testing	Harder	Easier

6. Graphical User Interface

Both implementations are connected to a Tkinter-based graphical user interface. The GUI allows users to start, pause, step, randomize, and clear the simulation. Although the GUIs look identical, their internal state management differs according to the paradigm used.

7. Learning Outcomes

Through this project, the differences between imperative and functional programming became clear. The project demonstrated how paradigms influence program structure, abstraction, and reasoning.

8. Conclusion

This project shows that programming paradigms are not merely stylistic choices. They fundamentally shape how solutions are designed and understood. Implementing the same problem using two paradigms provided valuable insight into both.

9.References

1. Conway, J. H. (1970). *The Game of Life*. Scientific American, 223(4), 120–123.
2. Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and Interpretation of Computer Programs* (2nd ed.). MIT Press.
3. Van Roy, P., & Haridi, S. (2004). *Concepts, Techniques, and Models of Computer Programming*. MIT Press.
4. Python Software Foundation. *The Python Programming Language Documentation*. https://docs.python.org/3/
5. Python Software Foundation. *PEP 622 – Structural Pattern Matching (Specification)*. https://peps.python.org/pep-0622/
6. Python Software Foundation. *Tkinter — Python Interface to Tcl/Tk*. https://docs.python.org/3/library/tkinter.html
7. Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3), 359–411.
8. Scott, M. L. (2015). *Programming Language Pragmatics* (4th ed.). Morgan Kaufmann.