# Assignment 4, Part 1, Specification

SFWR ENG 2AA4

April 12, 2019

Justin Prez

This Module Interface Specification (MIS) document contains modules, types and methods for implementing the state of a game of Conway's Game of Life.

# Game Board ADT Module

## Template Module

BoardT

## Uses

N/A

## Syntax

### Exported Access Programs

| Routine name | In | Out | Exceptions |
|---|---|---|---|
| new BoardT | $s$ : string | BoardT | invalid_argument, out_of_range |
| next_generation | | | |
| get_grid | | seq of $\mathbb{N}$ | |
| get_columns | | $\mathbb{N}$ | |
| get_rows | | $\mathbb{N}$ | |

## Semantics

### State Variables

$G$: Seq of $\mathbb{N}$ # *Grid of 0s and 1s*
$C$: $\mathbb{N}$ # *Number of Columns*
$R$: $\mathbb{N}$ # *Number of Rows*

### State Invariant

$|G| = R * C$

### Assumptions & Design Decisions

- The BoardT constructor is called before any other access routine is called on that instance. Once a BoardT has been created, the constructor will not be called on it again.
- Once instantiated the size of the board will not change.

- The input file will match the given specifications below.
- This implementation of the game board assumes infinite "wrap around" property of the board. The cells on the right most edge are the neighbors of the left most edge, and the same goes for the top and bottom most edges.
- For better scalability, this module is specified as an Abstract Data Type (ADT) instead of an Abstract Object. This would allow multiple games to be created and tracked at once by a client.

**Access Routine Semantics**

BoardT(*filename*):

- transition: read data from an input file associated with the input string. Use this data to instantiate the BoardT state variables.

  The text file has the following format where $row_i$, $col_j$, and $c_{ij}$ stand for values that represents the number of rows, number of columns and the life state of each cell, respectively. Note that a cell value can either be 0 representing a dead cell or 1 representing a live cell. All data values in a row are separated by a single white space. Rows are separated by a new line. The total number of $c_i$ values is equal to the value corresponding to $row_i * col_j$.

$$\begin{array}{cccccccc} row_i & col_j & & & & & & \\ c_{00} & c_{01} & c_{02} & c_{03} & c_{04} & \ldots & c_{0j} \\ c_{10} & c_{11} & c_{12} & c_{13} & c_{14} & \ldots & c_{1j} \\ & & & & & & & \\ \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots & \ldots \\ c_{i0} & c_{i1} & c_{i2} & c_{i3} & c_{i4} & \ldots & c_{ij} \end{array} \tag{1}$$

- exception: $exc := ((|G| \neq R * C) \Rightarrow$ out_of_range)
- exception: if the input string does not correspond to an existing file or a valid file $\Rightarrow$ invalid_argument

next_generation():

- transition: $(i : \mathbb{N} | i \in [0..R-1] : (j : \mathbb{N} | j \in [0..C-1] :$

| **current state** | **count_neighbours** | **transition** |
|---|---|---|
| $G[i * C + j] = 0$ | count_neighbors($j$,$i$) $= 3$ | $G[i * C + j] := 1$ |
| $G[i * C + j] = 1$ | count_neighbors($j$,$i$) $< 2 \vee$ count_neighbors($j$,$i$) $> 3$ | $G[i * C + j] := 0$ |

- exception: none

get_grid():

- output: $out := G$
- exception: None

get_columns():

- output: $out := C$
- exception: None

get_rows():

- output: $out := R$
- exception: None

## Local Functions

count_neighbors : $\mathbb{N} \times \mathbb{N} \Rightarrow \mathbb{N}$
count_neighbors$(x, y) \equiv +(i : \mathbb{Z} | i \in [-1..1] : (j : \mathbb{Z} | j \in [-1..1] : G[(x + i + C)\%C + ((y + j + R)\%R) * C)])) - G[x + y * C]$

# Write Module

## Module

Write

## Uses

BoardT

## Syntax

### Exported Constants

None

### Exported Access Programs

### Exported Access Programs

| Routine name | In | Out | Exceptions |
|---|---|---|---|
| write_to_terminal | seq of $\mathbb{N}$, $\mathbb{N}$, $\mathbb{N}$ | | |
| write_to_file | seq of $\mathbb{N}$, $\mathbb{N}$, $\mathbb{N}$, $s$ : string | | invalid_argument |

## Semantics

### State Variables

None

### State Invariant

None

### Assumptions & Design Decisions

- output file could potentially be used again as input again, so the formatting needs to be the exact same as specified above.

**Access Routine Semantics**

write_to_file($G$,$C$,$R$,$s$):

- output: The current state of the Boards grid, as well as the number of rows and columns are passed to the function along with the string $s$ for the output file. The textfile will have the following format: The first line consists of the number of rows and columns separated by a space. There will be $R$ rows after separated by new lines with $C$ columns per row separated by white spaces.

$$\begin{array}{cccccccc} row_i & col_j & & & & & \\ c_{00} & c_{01} & c_{02} & c_{03} & c_{04} & ... & c_{0j} \\ c_{10} & c_{11} & c_{12} & c_{13} & c_{14} & ... & c_{1j} \\ & & & & & & \\ ..., & ..., & ..., & ..., & ..., & ... & ... \\ c_{i0} & c_{i1} & c_{i2} & c_{i3} & c_{i4} & ... & c_{ij} \end{array} \tag{2}$$

- exception: If there is an error opening the file ⇒ invalid_argument

write_to_terminal($G$,$C$,$R$):

- output: The current state of the Boards grid, as well as the number of rows and columns are passed to the function. The grid will be printed to the console in the following format: The first line consists of the number of rows and columns separated by a space. There will be $R$ rows after separated by new lines with $C$ columns per row separated by white spaces.

$$\begin{array}{cccccccc} row_i & col_j & & & & & \\ c_{00} & c_{01} & c_{02} & c_{03} & c_{04} & ... & c_{0j} \\ c_{10} & c_{11} & c_{12} & c_{13} & c_{14} & ... & c_{1j} \\ & & & & & & \\ ..., & ..., & ..., & ..., & ..., & ... & ... \\ c_{i0} & c_{i1} & c_{i2} & c_{i3} & c_{i4} & ... & c_{ij} \end{array} \tag{3}$$

- exception: None

# Critique of Design

I believe that my implementation was consistent as for the most part in my code. I avoided using namespace std to see the derivatives of where the functions were coming from, and defined unsigned integers as "nat" to improve understandability and consistency in the code. Essentiality was violated with the accessor methods but was a necessary violation in order to test the code segments, and view the state of the board (via the write functions). This implementation is not as general as it requires a very specific input and only accepts grids of 0s and 1s. This could be improved by accepting a wider variety of input formats and recogonzining different patterns of pairs besides simply 1s and 0s. This implementation preserves minimality as all access routines provide only 1 independent service that is not provided by any other accessor method routine. My implementation has high cohesion and low coupling amoung the modules as the components of the modules are closely related (i.e. accessor methods of state variables, updating the state variable...) and do not strongly depend on other modules. Information Hiding as the state variables are protected by encapsulation, and methods like count_neighbors(x,y) which changes for every cell on every generation update is only accessed within the BoardT module can not be accessed publically.