Assignment 4 Specification

SFWR ENG 2AA4

April 13, 2019

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

Cell Type Module

Module

CellType

Uses

N/A

Syntax

Exported Constants

BOARD_SIZE_X = 30
BOARD_SIZE_Y = 30

Exported Types

CellT = tuple of (live_state: $\mathbb{B}$, live_neighbours: $\mathbb{N}$)

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Game Board ADT Module

Template Module

Board

Uses

CellType
Read

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
new Board	filename: string	Board	
get_board		seq of (seq of CellT)	
next_gen			

Semantics

State Variables

board: seq of (seq of CellT) # *Grid Representing Gameboard*

State Invariant

$|board| = BOARD\_SIZE\_Y$

Assumptions & Design Decisions

- The gameboard is a grid of size 30 x 30, where each element is of the type CellT.
- The gameboard has a hard border, where cells can reach the border but once they grow out of the border, all cells outside of the board are considered dead. This means once cells reach the edge of the gameboard, the behaviour would differ from the game because any cells that move outside of the board are dead.
- Routine next_gen iterates the board to the next generation, using several local functions as helper functions.

- Because the gameboard constructor relies on the read module, it will throw the same exception if given an invalid file input when taking in a string representing the filename.
- This module only handles the state of the gameboard, the View module handles the graphical terminal output.

Access Routine Semantics

Board($filename$):

- transition: $board :=$ read_state($filename$)
- exception: None

get_board():

- output: *out* := *board*
- exception: None

next_gen():

- transition: *# procedural specification*

```
set_neighbour()
for all $i$ in $board$
    for all $j$ in $i$
        if board[i][j].live_state == True
            if board[i][j].live_neighbours < 2
                board[i][j].live_state = False
            else if board[i][j].live_neighbours == 2 | board[i][j].live_neighbours == 3
                board[i][j].live_state = True
            else if board[i][j].live_neighbours > 3
                board[i][j].live_state = False
        else if board[i][j].live_state != True
            if board[i][j].live_neighbours == 2
                board[i][j].live_state = True
```

- exception: none

Local Functions

clear_neighbour :
($\forall i \in$ [0..BOARD_SIZE_X] $\wedge j \in$ [0..BOARD_SIZE_Y] : board[i][j].live_neighbours = 0)

set_neighbour():

- transition: # *procedural specification*

```
set_neighbour()
for all $i$ in $board$
    for all $j$ in $i$
        if i == 0 and j == 0
            if board[0][1].live_state == True
                board[i][j].live_neighbours++
            if board[1][0].live_state == True
                board[i][j].live_neighbours++
            if board[1][1].live_state == True
                board[i][j].live_neighbours++
        else if i == 0 and j == 29
            if board[0][28].live_state == True
                board[i][j].live_neighbours++
            if board[1][29].live_state == True
                board[i][j].live_neighbours++
            if board[1][28].live_state == True
                board[i][j].live_neighbours++
        else if i == 29 and j == 0
            if board[28][0].live_state == True
                board[i][j].live_neighbours++
            if board[29][1].live_state == True
                board[i][j].live_neighbours++
            if board[28][1].live_state == True
                board[i][j].live_neighbours++
        else if i == 29 and j == 29
            if board[29][28].live_state == True
                board[i][j].live_neighbours++
            if board[28][29].live_state == True
                board[i][j].live_neighbours++
            if board[28][28].live_state == True
                board[i][j].live_neighbours++
        else if i == 29 and j ∈ [0..29]
            $(\forall new\_i \in [0..1] \wedge new\_j \in [j-1..j+1] \wedge board[new_i][new_j].live\_state : board[i][j].live\_neighbours++)$
        else if j == 0
            $(\forall new\_i \in [i-1..i+1] \wedge new\_j \in [j..j+1] \wedge board[new_i][new_j].live\_state : board[i][j].live\_neighbours++)$
```

```
else if j == 29
    $(\forall\, new\_i \in [i-1..i+1] \land new\_j \in [j-1..j] \land board[new_i][new_j].live\_state : board[i][j].live\_neighbours++)$
else if i == 29
    $(\forall\, new\_i \in [i-1..i] \land new\_j \in [j-1..j+1] \land board[new_i][new_j].live\_state : board[i][j].live\_neighbours++)$
else
    $(\forall\, new\_i \in [i-1..i+1] \land new\_j \in [j-1..j+1] \land board[new_i][new_j].live\_state : board[i][j].live\_neighbours++)$
if board[i][j].live_state == True
    board[i][j].live_neighbours–
```

Read Module

Module

Read

Uses

CellType

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
read_state	$filename$: string		

Semantics

Environment Variables

input: File for the input to the Read module, containing the specified format for the gameboard

State Variables

None

State Invariant

None

Assumptions

The input file will match the given specification.

Access Routine Semantics

read_state($filename$)

- transition: read data from the file input associated with the string filename. Use this data to initialize the values for Board in the Board constructor. Returns a 2D vector where every element is a CellT, with initial values of 0 for live_neighbours and True or False for live_state, True if the text file value is 1 for that index and False otherwise.

  The text file has the following format, where 0 represents a live_state value of false and 1 represents a live_state value of true. The text file is in a 30 x 30 array of these values, where all elements are separated by a single space, and new rows in the array are separated by a new line. Below is a sample of what a potential input in the file would look like, where ... denotes that the rest of the values go up until there are 30 elements in a row, and likewise for number of rows.

$$\begin{array}{cccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0... & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0... & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0... & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0... & 0 \\
... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0... & 0
\end{array} \tag{1}$$

- exception: none

Write Module

Module

Write

Uses

CellType, GameBoard

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
write_state	Board		

Semantics

Environment Variables

out: File for the output of the Write module, where the module writes the passed in gameboard to the output file in the same format as the input file in the Read module, so it can be passed to the Read.

State Variables

None

State Invariant

None

Assumptions

The input file will match the given specification.

Access Routine Semantics

write_state(*board*)

- transition: write to the output text file from the given gameboard input. Iterates through the gameboard and writes in the same 30 x 30 grid format as the input text file in the Read module. Checks every cell in the gameboard and writes a 0 if the Cell live_state is false, and 1 if it is true. The other state variable for the number of live neighbours is not relevant for the initial reading so is omitted here as well.

  The output text file has the following format, where 0 represents a live_state value of false and 1 represents a live_state value of true. The text file is in a 30 x 30 array of these values, where all elements are separated by a single space, and new rows in the array are separated by a new line. Below is a sample of what a potential input in the file would look like, where ... denotes that the rest of the values go up until there are 30 elements in a row, and likewise for number of rows.

$$\begin{array}{cccccccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0... & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0... & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0... & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0... & 0 \\ ... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0... & 0 \end{array} \tag{2}$$

- exception: none

View Module

Module

View

Uses

GameBoard

Syntax

Exported Constants

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
display_state	Board		

Semantics

Environment Variables

None

State Variables

None

State Invariant

None

Assumptions

The input gameboard is valid and is of the size 30 x 30.

Access Routine Semantics

write_state(*board*)

- transition: Iterates through the gameboard and prints out the gameboard to the terminal to see the state of the game. If the Cell live_state is false then it displays a "." symbol, while if it is true then it displays a "@" symbol. All elements in the gameboard as separated by a single space, and rows are separated by a new line character. This ensures that the live cells are easily seen and stand out from the dead cells. On the side of the gameboard is a column of "—" symbols and row numbers to easily check and compare with the actual Game of Life for verification of correct game behaviour. Below is a sample output of what it would look like on the terminal.

```
 .   .   .   .   .   .   .   @   .   .   .   .   ....  |0
 .   .   .   .   .   .   @   @   .   .   .   .   ....  |1
 .   .   .   .   .   .   .   @   .   .   .   .   ....  |2
 .   .   .   .   .   .   .   .   .   .   .   .   ....  |3
...  ... ... ... ... ... ... ... ... ... ... ... ...   ...
 .   .   .   .   .   .   .   .   .   .   .   .   ....  |29
```
(3)

- exception: none

Critique of Design

0.1 Essentiality

Essentiality is enforced to the extent that the other design principles are not violated. All functions in every module are largely essential, but there are some places where it could've been further made essential but wasn't to avoid violating the property of Minimality. For example, the View and Write modules could have been combined and same with their functions, where when it reads in a given gameboard it both outputs the state to the terminal and writes it to the output file. However, if this was done then this function would have multiple outputs and purposes, making the design less modular. For example, the user may not want to update their output file and only want to view the state of the game on the terminal. Additionally, some of the local functions in GameBoard could've been combined with the next_gen function, but separating these functions allow the design to be more modular, being easier to test, debug and maintain.

0.2 Generality

The design maintains generality by having functions that can take in a wide range of parameters. For example, previously the read function in the Read module didn't take in any input parameters, instead only having a set file name it could read from. It was modified so it could take in a filename parameter that isn't named the environment variable,, making it more applicable to a general range of functions.

0.3 Minimality

All modules and functions were created with the principle of minimality in mind. This is why the design decision was made to separate the Read and Write module outside of the GameBoard module, so the specification was made by designing by secrets, every module has a single purpose of functions. The Write module only handles writing, the View module only handles output to the terminal, and the Read module only handles reading an input text file. Additionally, the functions are all minimal as well. The Read, Write and View modules all only perform a single function, and the 3 public methods for the GameBoard only perform one purpose. The accessor get_board returns the gameboard, the gameboard constructor constructs the board and next_gen iterates the board to the next state.

0.4 Consistency

All syntax, variable names and module names are consistent to the MIS standard. Module names are all capitalized and done with camelcase (ie. GameBoard and CellType), while function names are done with snake case, such as get_Board, next_gen, display_state etc. State variables such as board, or live_neighbours or live_state also follow this convention. Constants such as BOARD_SIZE_X are in all capitals.

0.5 Cohesion

The principle of high cohesion and low coupling are maintained in this design specification. All functions in a module are highly related to each other, for example in GameBoard all functions are only related to handling the state of the game. Low coupling is maintained by the lack of dependence between modules, if the View module fails the Write module still performs etc. Following these design principles makes the design easy to maintain, it will be more immediate obvious which module is causing an error in the program due to high cohesion and low coupling.

0.6 Information Hiding

Information hiding is maintained so that anything that the user doesn't need to see are hidden. Examples of this are mainly seen in the GameBoard module, where the local functions set_neighbour and clear_neighbours hide the implementation of the next_gen from the user. State variables for example the actual 2-D array for the board are kept hidden, and only accessible to the user through the accessor get_board.