Assignment 4, Part 1, 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 of Conway's Game of Life as well as viewing the state. Game of Life is a grid of square cells where each is in one of two possible states, alive or dead. Each cell interacts with its 8 surrounding neighbours to determine the next state. The Game's rules are as follows:

1. Any live cell with fewer than two live neighbours dies, as if by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

In applying the specification, there will be cases that involve undefinedness. We will interpret undefinedness following [**?**]:

If $p : \alpha_1 \times .... \times \alpha_n \rightarrow \mathbb{B}$ and any of $a_1, ..., a_n$ is undefined, then $p(a_1, ..., a_n)$ is False. For instance, if $p(x) = 1/x < 1$, then $p(0) =$ False. In the language of our specification, if evaluating an expression generates an exception, then the value of the expression is undefined.

Cell Module

Module

Cell

Uses

N/A

Syntax

Exported Constants

None

Exported Types

CellT = $\{$Alive, Dead$\}$

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Game State ADT Module

Template Module

StateT

Uses

N/A

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
new StateT	M	StateT	invalid_argument
updateState			none
getCell	$\mathbb{N}$,$\mathbb{N}$	CellT	invalid_argument
size		$\mathbb{N}$	

Semantics

State Variables

S: Matrix #*BoardGrid*

State Invariant

$\forall e \in M : (|M| = |e|)$ #size of all elements in M are equal to size of M ensuring square matrix

Assumptions & Design Decisions

- The StateT constructor is called before any other access routine is called on that instance. Once a StateT has been created, the constructor will not be called on it again.
- The constructor can be used with an empty matrix, however the user should understand no data can be stored or received using this.
- Each element must either be Dead or Alive. (The initializer automatically sets all elements to dead to begin with)

- 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.
- Any area outside the gameboard will be considered Dead Cells and will not be affected by the game in any way. For example if the board is 20x20 then cell 19,20 will be considered dead and wont be affected by cell 19,19.
- Getter functions are provided, though violating the property of being essential, to give a would-be view function easy access to the state of the game, as well as to make testing easier.

Access Routine Semantics

new StateT(M):

- transition: $S := M$
- output: $out := self$
- exception: $exc := (!(\forall e \in M : |M| = |e|) \Rightarrow \text{invalid\_argument})$ #ensures square matrix

updateState():

- transition: $S :=$ M such that $(\forall i, j | i, j \in [0..|S| - 1] : M[i][j] = \text{updateCell}(S, i, j))$
- exception: none

getCell(i,j):

- output: $out := S[i][j]$
- exception: $exc := (!(\text{validPoint}(i, j)) \Rightarrow \text{invalid\_argument})$

size():

- output: $out := |S|$
- exception: None

Local Types

Matrix = seq of (seq of CellT)

Local Functions

updateCell : Matrix x $\mathbb{N}$ x $\mathbb{N} \rightarrow \mathbb{B}$
updateCell(m, i, j) $\equiv$

$S[i][j] = Alive$	countLiveCells(i,j) $= 2$	$m[i][j] := Alive$
	countLiveCells(i,j) $= 3$	$m[i][j] := Alive$
$S[i][j] = Dead$	countLiveCells(i,j) $= 3$	$m[i][j] := Alive$

countLiveCells: $\mathbb{N}$ x $\mathbb{N} \rightarrow \mathbb{N}$
countLiveCells(i,j)$\equiv +(x, y : \mathbb{Z} | x \in \{i-1, i+1, i\} \wedge y \in \{j-1, j+1, j\} \wedge$validPoint$(i, j) \wedge !(i = x \wedge j = y) : 1)$

validPoint: $\mathbb{Z}$ x $\mathbb{Z} \rightarrow \mathbb{B}$
validPoint(i,j) $i \leq 0 \wedge j \leq 0 \wedge \equiv i < |S| \wedge j < |S|$ #Used integer rather than nat for params because i-1 can be passed in when running countLiveCells

View Module

View Module

View

Uses

GameState

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
initializeBoard	s: string	StateT	runtime_error
writeBoard	StateT, s: string		
printState	StateT		

Semantics

Environmental Variables

input: File representing initial board state output: File representing output board state

State Variables

None

State Invariant

None

Assumptions & Design Decisions

- Users will follow the input file format correctly

Access Routine Semantics

initialize(s)

- transition: read data from the file input.txt associated with the string s. Use this data to initialize an instance of the GameState module. This function will first use the input file to create a matrix with the appropriate cells. It will use this matrix to initialize a GameState.

  The text file has the following format, where all data values ina row are two numbers seperated by only a sinle comma, except for the first row which is a single number representing the size of the matrix. For example, if the first line is 20, the matrix will be 20 by 20. This is assumed to be an appropriate number. The following lines represent the locations where a cell is to be Alive. row_n and col_n are numbers representing the co-ordinates of an Alive cells where $x \geq 0$. Cells are automatically instantiated to be Dead otherwise. For example, row_0, col_0 means that Matrix[row_0,col_0] is Alive.

$$\begin{array}{cc} Number & \\ row_1, & col_1 \\ row_2, & col_2 \\ row_3, & col_3 \\ ..., & ... \\ row_n, & col_n \end{array} \tag{1}$$

- exception: if the file s is not found, throw invalid_argument. If any integer value row_n or col_n (where $n \geq 0$) is greater than or equal to $Number$ (first row), throw invalid_argument.

writeBoard (b,s)

- transition: First, b.size() is written to a file (this being the size of the board). Then, for all integer i values less than b.size(), and for all integer j values less than b.size(), if b.getCell(i, j) = Alive, write to the file a new line character followed by the integer value i followed by a comma followed by integer value j. The output file should have the format seen below, where Number represents a single integer number (size of board), and the following lines represent the row number followed by a comma followed by the column number where a cell is Alive in b.

$$\begin{array}{cc} Number & \\ row_1, & col_1 \\ row_2, & col_2 \\ row_3, & col_3 \\ ..., & ... \\ row_n, & col_n \end{array} \tag{2}$$

- exception: none

printState(b):

- transition: this function initializes a string variable. for all integer i values less than b.size() (for all integer j values less than b.size(), if b.getCell(i, j) = Dead, it appends to the string [ ], otherwise it appends [o]) it appends a new line character at the end of each row (when i is about to change). It then prints this string. For example, if b = [[Dead, Dead, Alive], [Dead, Alive, Dead], [Dead, Dead, Alive]] the string will be "[ ][ ][o]n[ ][o][ ]n[ ][ ][o]" where n represents a new line character.
- exception: None

Critique of Design

First and foremost, I believe my design could focus a little more on modularity. Because my modules do not have high cohesion and low coupling because my printString method in my View module highly depends on my GameState method. I believe it would make more sense to add a toString() method in gameState that returns the gameState as a string. This would result in higher cohesion and lower coupling.

My design also violates essentiality due to the additional getter methods for my GameState function that I used to help with unit testing. This can be solved by simply removing those functions.

My design could furthur improve generality and separation of concerns by including a generic matrix class.

In terms of information hiding my program does this well by using state variables and private functions, however I do have a getCell getter function to help for testing and writing. I decided to use a getCell rather than a getState (which would return the matrix) to further include information hiding. A matrix would output all information rather than one cell and therefore a getCell function further emphasizes information hiding.

In terms of minimality, I believe my GameState functions highly incorporate this however while I believe my read and write functions incorporate this it might not be to as high an extent because reading to a file and writing to a file can be more complicated than other simple tasks.

Lastly, in terms of consistency I believe my design performs well with respect to this quality because input parameters, function names have consistent writing conventions, and the logic throughout my code is also very consistent.