Assignment 4 Specification

SFWR ENG 2AA4

April 12, 2019

This Module Interface Specification (MIS) document contains modules, types and methods for implementing the game state of the Conway's Game of Life which is a cellular automaton devised by the British mathematician John Horton Conway in 1970. The game is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how the game moves from one state to another. There are a set of rules that can take the game from one state to another, by determining the future state of a single cell based on its current state of the neighbours. Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

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

In this document we will use True or False to represent the state of each cell based on being 'Alive' or 'Dead'.

Game of Life Module

Template Module

GameState

Uses

None

Syntax

Exported Constants

BoardEdgeSize = 12 *#size of the board in each direction including the Edge*
BoardSize = 10 *#size of the board in each direction excluding the Edge*

Exported Types

CellT = {Alive, Dead}

Exported Access Programs

Routine name	In	Out	Exceptions
new StateT			
getStatus	$\mathbb{N}$, $\mathbb{N}$	$\mathbb{B}$	out_of_range
changeStatus	CellT, $\mathbb{N}$, $\mathbb{N}$	$\mathbb{B}$	out_of_range
count	CellT	$\mathbb{N}$	
updateGame			invalid_argument
is_there_change		$\mathbb{B}$	
gameReset			

Semantics

State Variables

S: GameStateT

State Invariant

Alive = True
Dead = False
$count(Dead) + count(Alive) = BoardSize \times BoardSize$

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.
- Indexing in the game starts from 1, eg. to retrieve the status of the first Cell in the game, we would need to call getStatus(1,1).
- For bettter portablility, instead of a single constructor that reads the initial congifuration from a text file, the current desgin takes advantage of a defualt constructor. Another module, has been desgined to read the initial configuration from a file. This further improves the property of separation of concerns as the client does not have to worry about changing the initial text file to change the start state.
- 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.
- At the start of the game all the cells are set to be Dead. In addition, the is_there_change() function looks if two consecutive GameStates are the same. However, the function does not permanently update the game, only temproray to check if each state would remain the same.
- The GameState edges are all set to Dead as well, this is achieved through creating a GameState that is one Cell bigger than the GameState showed in the view function in every direction. Eg. Model GameState will have size of 7 while View GameState will have size of 5. To improve the information hiding in the module, the Edge Cells can never be accessed or modified and always remain Dead.
- The client is not allowed to directly access the game, only through the getStatus accessor as it breaks the essentiality quality of the program. Although outside of the scope of this assignment, but a additional module can be desgined which given a StateT can return the GameState as a two-dimensional sequence holder.

Access Routine Semantics

StateT():

- transition: $S :=$ init_2d(BoardEdgeSize, $Dead$)
- output : $out := self$
- exception: None

getStatus(i, j):

- output: $out := S[i][j]$
- exception: $exc :=$ (invalidPos$(i, j) \Rightarrow$ out_of_range)

changeStatus(c, i, j):

- transition: $S[i][j] := c$
- exception: $exc :=$ (invalidPos$(i, j) \Rightarrow$ out_of_range)

count(c):

- output: $out := +(i, j : \mathbb{N} | \neg$invalidPos$(i, j) \wedge S[i][j] = c : 1)$
- exception: None

updateGame():

- transition: $S := S'$ such that $\forall(i, j : \mathbb{N} | \neg$invalidPos$(i, j) : S'[i][j] =$ updateCell$(i, j, S))$
- exception: $exc :=$ ($\neg$init_check() $\Rightarrow$ invalid_argument)

is_there_change():

- output: $out := \forall(i, j : \mathbb{N} | \neg$invalidPos$(i, j) : S.$updateGame()$[i][j] = S[i][j])$
- exception: None

gameReset():

- transition: $S :=$ init_2d(BoardEdgeSize, $Dead$)
- exception: None

Local Types

GameStateT = sequence [BoardEdgeSize, BoardEdgeSize] of cellT

Local Functions

countAliveNeighbour: $\mathbb{N} \times \mathbb{N} \times$ GameStateT $\rightarrow \mathbb{N}$
countAliveNeighbour (tempGame, i, j) $\equiv$
$+(i, j : \mathbb{N} | \neg \text{invalidPos}(i, j) \wedge (tempGame[i][j+1] = Alive \vee tempGame[i][j-1] = Alive \vee tempGame[i+1][j+1] = Alive \vee tempGame[i+1][j] = Alive \vee tempGame[i+1][j-1] = Alive \vee tempGame[i-1][j+1] = Alive \vee tempGame[i-1][j] = Alive \vee tempGame[i-1][j-1] = Alive) : 1)$

updateCell: $\mathbb{N} \times \mathbb{N} \times$ GameStateT $\rightarrow \mathbb{B}$
updateCell (i, j, tempGame) $\equiv$

tempGame[i][j] = Alive	countAliveNeighbour(i, j, tempGame) <2	Dead
	countAliveNeighbour(i, j, tempGame) = 2	Alive
	countAliveNeighbour(i, j, tempGame) = 3	Alive
	countAliveNeighbour(i, j, tempGame) >3	Dead
tempGame[i][j] = Dead	countAliveNeighbour(i, j, tempGame) = 3	Alive

init_check: StateT $\rightarrow \mathbb{B}$
init_check($tempState$) $\equiv$ count($Dead$) + count($Alive$) = BoardSize $\times$ BoardSize

getBoard: void $\rightarrow$ GameStateT
getBoard() $\equiv S$

init_2d: $\mathbb{N} \times$ CellT $\rightarrow$ GameStateT
init_2d$(n, c) \equiv s$ such that $(|s| = n \wedge (\forall\, i \in [0..n-1] : s[i] = (c_{n-1}, c_{n-2}, c_{n-3}, ..., c_{n-n}))$

invalidPos : $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$
invalidPos$(i, j) \equiv \neg(1 \leq i \wedge i \leq$ BoardSize$) \vee \neg(1 \leq j \wedge j \leq$ BoardSize$)$

View Module

Module

View

Uses

GameState

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
initialConfigRead	String, StateT	StateT	
outputGameTerminal	StateT		
outputGameText	String, StateT		

Semantics

Environment Variables

read_input_config: Text file including the input config
write_final_config: Text file used to export the output config

State Variables

None

State Invariant

None

Assumptions

- The input file will match the given specification.
- This module will not be explicitly tested. Nevertheless, each method in the module was tested by simply looking if the outputted game looks how it's supposed to look like. The outputGameText and readInitialCongif methods have also been tested together by using the output of outputGameText as the input of readInitialConfig which worked as it should.

Access Routine Semantics

readInitialConfig($filename, tempGame$)

- transition: read data from the file read_input_config associated with the string filname-name. Use this data to update the state of the GameState module. Read will first initialize an object file of class StateT which has all the data to be initially set to be dead. Then, the Ascii Gui will be read character by character from the text file, and put into the GameStateT of the object file created. The newly initialized object file is then returned, allowing the client to work on it after it has been initialized from the text file.

  The text file has the following format, where 'X' represents an Alive cell and '.' represents a Dead cell. The Ascii gui is then made up of the two symbols (X, .) in the form of a square with each side equal to BoardSize constant, counting the number of symbols. Rows are separated by a new line. The data shown below is for a constant BoardSize of 5 where the cells (3,2), (3,3), (3,4) are Alive.

$$\begin{array}{ccccc} . & . & . & . & . \\ . & . & . & . & . \\ . & X & X & X & . \\ . & . & . & . & . \\ . & . & . & . & . \end{array} \tag{1}$$

- exception: None

outputGameText($tempState$)

- transition: The function is given an object of class StateT which potentiall includes Alive states or could be all Dead cells. The function goes through the sequence of cells making our GameStateT in which prints symbol 'X' as Alive and symbol '.' as Dead to a newly created text file associated with the string s. The output result

will be very similar to the format showd in figure (1) as it might include different Alive and Dead cells.

- exception: None

outputGameTerminal($filename, tempState$)

- transition: The function is given an object of class StateT which potentiall includes Alive states or could be all Dead cells. The function goes through the sequence of cells making our GameStateT in which prints symbol 'X' as Alive and symbol '.' as Dead to the terminal. The output result will be very similar to the format showd in figure (1) as it might include different Alive and Dead cells.

- exception: None

Critique of Design

Using the current design the edges of the viewable GameState are all updated considering the unviewable cells as Dead. This is done through use of a two dimensional vector where all the elements starting from index 1 can be accessed. This means that our inital graph will be made of seven by seven vector while we can only access, change and update the five by five inner GameState in the bigger vector. Then using information hiding we can disable the client to change or access any of the cells outside of the viewable game. Although this design works perfectly when given a pattern, it does not work well when the changing shape partially goes off of the screen since although they might be Alive, since they are off the screen they are counted as Dead. This can be done using a large enough defualt board which can be zoomed in and out, instead of changing the intial size of the board. This would allow for far better user interface while keeping our pragram in tune for portability and scalibilty problems.