COMPSCI 2ME4 A4 Specification

Song Tao Wu, wus92

April 13, 2019

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

The Rules for Game of Life:

For a space that is 'populated':

Each cell with one or no neighbors dies, as if by solitude.

Each cell with four or more neighbors dies, as if by overpopulation.

Each cell with two or three neighbors survives.

For a space that is 'empty' or 'unpopulated':

Each cell with three neighbors becomes populated.

Grid Types Module

Module

GridTypes

Uses

N/A

Syntax

Exported Constants

row_limit = 15
column_limit = 15

Exported Types

GridT = sequence of [row_limit] [column_limit] of $\mathbb{B}$.

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

File Utility Module

Library

FileUtil

Uses

Grid Types
Game Board ADT

Syntax

Exported Constants

N/A

Exported Types

N/A

Exported Access Programs

Routine name	In	Out	Exceptions
readFile	String	GridT	file_not_find
writeToFile	GameBoard, String		

Semantics

State Variables

None

State Invariant

None

Access Routine Semantics

readFile(fileName):

- transition: read data from the file associated with the string fileName. Use this data to construct a new gridT, which will be used to construct the new gameboard.

  The text file has the following format, where C_{xy} stand for strings that represent the state of each cell. The state is represented by either the string "0" or "*" which represents true(occupied) and false(unoccupied) respectively. All data values in a row are separated by spaces. Rows are separated by a new line.

$$\begin{array}{ccccc} c_{00}, & c_{01}, & c_{02}, & \ldots, & c_{0column\_limit} \\ c_{10}, & c_{11}, & c_{12}, & \ldots, & c_{1column\_limit} \\ \ldots, & \ldots, & \ldots, & \ldots, & \ldots \\ c_{row\_limit0}, & c_{row\_limit1}, & c_{row\_limit2}, & \ldots, & c_{row\_limit\ column\_limit} \end{array} \tag{1}$$

- out := G where $\forall\, x, y : \mathbb{N} | x \in [0..\text{row\_limit}] \wedge y \in [0..\text{column\_limit}] : G[x][y] := c_{xy}$)

- exception: exc := (fileName does not exist $\implies$ file_not_find)

writeToFile(board, fileName):

- transition: write the grid data from board to the file associated with the string fileName. If the file does not exist, create a new file.

  The text file has the following format, where ($\forall\, x, y : \mathbb{N} | x \in [0..\text{row\_limit}] \wedge y \in [0..\text{column\_limit}] : C_{xy} :=$ board.getGrid(x, y)) C_{xy} stands for the character that represents the state of each cell. The state is represented by either the string "0" or "*" which represents true(occupied) and false(unoccupied) respectively. All data values in a row are separated by spaces. Rows are separated by a new line.

$$\begin{array}{ccccc} c_{00}, & c_{01}, & c_{02}, & \ldots, & c_{0column\_limit} \\ c_{10}, & c_{11}, & c_{12}, & \ldots, & c_{1column\_limit} \\ \ldots, & \ldots, & \ldots, & \ldots, & \ldots \\ c_{row\_limit0}, & c_{row\_limit1}, & c_{row\_limit2}, & \ldots, & c_{row\_limit\ column\_limit} \end{array} \tag{2}$$

- exception: none

View Module

Library

View

Uses

Grid Types
Game Board ADT

Syntax

Exported Constants

N/A

Exported Types

N/A

Exported Access Programs

Routine name	In	Out	Exceptions
display	GameBoard		

Semantics

State Variables

None

State Invariant

None

Access Routine Semantics

display(board):

- transition: output the grid data from board to the standard output.
  The output has the following format, where $(\forall x, y : \mathbb{N} | x \in [0..\text{row\_limit}] \wedge y \in [0..\text{column\_limit}] : C_{xy} := \text{board.getGrid(x, y)})$ C_{xy} stands for the character that represents the state of each cell. The state is represented by either the string "0" or "*" which represents true(occupied) and false(unoccupied) respectively. All data values in a row are separated by spaces. Rows are separated by a new line.

$$\begin{array}{ccccc} c_{00}, & c_{01}, & c_{02}, & \ldots, & c_{0 column\_limit} \\ c_{10}, & c_{11}, & c_{12}, & \ldots, & c_{1 column\_limit} \\ \ldots, & \ldots, & \ldots, & \ldots, & \ldots \\ c_{row\_limit 0}, & c_{row\_limit 1}, & c_{row\_limit 2}, & \ldots, & c_{row\_limit\ column\_limit} \end{array} \tag{3}$$

- exception: none

Game Board ADT Module

Template Module

GameBoard

Uses

GridTypes

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
new GameBoard	GridT	GameBoard	invalid_argument
new GameBoard		GameBoard	
next_state	$\mathbb{N}$, $\mathbb{N}$	$\mathbb{B}$	
num_of_alive_neighbours	$\mathbb{N}$, $\mathbb{N}$	$\mathbb{B}$	
getGrid		GridT	
update			

Semantics

State Variables

grid: GridT # *The game board grid*

State Invariant

size(grid) = row_limit $\times$ column_limit

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.
- 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

GameBoard(*iniitial_states*):

- transition: $grid := initial\_states$
- exception: $exc := (\neg\ (|\text{grid}|\ = \text{row\_limit}) \implies \text{invalid\_argument})$

GameBoard():

- transition: $grid :=$ G where $(\forall\, x, y : \mathbb{N} | x \in [0..\text{row\_limit}] \wedge y \in [0..\text{column\_limit}] : G[x][y] = false)$
- exception: none

next_state(x, y):

- output: $(\neg grid[x][y] \implies \text{num\_of\_alive\_neighbours}(x, y) = 3) \vee grid[x][y] \implies (\text{num\_of\_alive\_neighbours}(x, y) = 3 \vee \text{num\_of\_alive\_neighbours}(x, y) = 2))$
- exception: none

num_of_alive_neighbours(x, y):

- output : $(+\ a, b : \mathbb{N} \mid a \in [max(x-1, 0)..min(x+1, \text{row\_limit})] \wedge b \in [max(y-1, 0)..min(y+1, \text{column\_limit})] \wedge grid[a][b] : 1)$
- exception: none

getGrid():

- output : grid
- exception: none

update():

- transition: grid := G where $\forall\, x, y : \mathbb{N} | x \in [0..\text{row\_limit}] \wedge y \in [0..\text{column\_limit}] : G[x][y] := next\_state(\text{x, y})$
- exception: none

Critique of Design

GridTypes Module

- The module is just a library exporting constants like row size and column size and the whole gridT type - 2 dimensional vector of boolean value.
- The constants and type will be used in all other files. By defining those value and types here, I modularize the code to avoid the need to redefining those values in every other modules.

FileUtil Module

- The module provides methods to write board to and read data from a file. There're two methods - readFile and writeToFile. Since both methods have to do with file utilities. This modularization shows the programming design principle - separation of concern.
- If we were to add more methods to interact with files, we could add the methods in this files.

GameBoard Module

- GameBoard provides methods to update the whole grid, and move on to the next state of the game.
- There are two constructors, one getter for state variable, and three methods to update the whole grid. the two methods - num_of_alive_neighbours and next_state are used to update the whole grid. The two methods could have been made private to enhance information hiding principle. But for unit testing purpose, I decided to make them public to give me more options and freedom to interact with the grid.
- The default constructor is used to set the whole grid to be unpopulated to avoid errors from not passing any parameters to the GameBoard.

View Module

- The module only provides one method to perform the basic view function. It takes a GameBoard and output it to the standard output.

- the function could have been included in the GameBoard module. But for the sake of MVC - Module View Controller and principle of separation of concern, the display method is included here in the module.