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 cellular automaton of Conway's Game of Life.

Game Types Module

Module

GameTypes

Uses

N/A

Syntax

Exported Constants

BOARDLENGTH = 25

Exported Types

CellState = {Dead, Alive}

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Game Board ADT Module

Template Module

GridT

Uses

BoardTypes

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
new GridT	seq of (seq of CellState)	GridT	t
checkCell	$\mathbb{N}$, $\mathbb{N}$	CellState	
update		GridT	
neighbourCnt	GridT, $\mathbb{N}$, $\mathbb{N}$	$\mathbb{N}$	invalid_argument
stateCount	CellState	$\mathbb{N}$	
init_seq	$\mathbb{N}$	seq of (seq of CellState)	invalid_argument

Semantics

State Variables

Grid: seq of (seq of CellState) # *Grid*

State Invariant

stateCount(Alive) + stateCount(Dead) = BOARDLENGTH x BOARDLENGTH

Assumptions & Design Decisions

- The GridT 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.
- The edge of the board will process non-existent cells on its border as dead cells.

- 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.
- The getter function is provided to obtain the state of a cell and will violate the property of being essential. Regardless, this function is needed to obtain information about the game and ensure the ease of implementation. It is used in the view module as well as part of the Model View Controller design pattern implementation. (`https://blog.codinghorror.com/understanding-model-view-controller/`)
- The length of the grid at 25 which is used to create a set 25x25 grid, similar to how the dimensions of a board game are determined by the designer. The input/output files will follow this constraint.

Access Routine Semantics

Constructor for the grid GridT(grid):

- transition: $Grid := grid$
- output: $out := self$

Updates the grid for the next stage iteration
update():

- output: $out := (\forall x, y : \mathbb{N} | x, y \in [0...24] : updateCell(x, y))$

Returns the state of the cell
checkCell(n_1, n_2):

- output: $out := Grid[n_1][n_2]$

Returns the number of neighbours
neighbourCnt(grid, n_1, n_2):

- output: $out := +(c : CellState, x : \mathbb{N}, y : \mathbb{N} | x, y \in [-1..2] \wedge (x \neq 0 \wedge y \neq 0) \wedge c = Alive \implies Grid[n_1 + x][n_2 + y] = c : 1)$
- exception: ($n_1 < 0 \vee n_1 >$ BOARDLENGTH$\vee n_2 < 0 \vee n_2 >$ BOARDLENGTH $\implies$ invalid_argument)

Returns the number of alive/dead states
stateCount(state):

- output: $out := (+n_1 : \mathbb{N}, n_2 : \mathbb{N} | n_1, n_2 \in [0...24] \wedge checkCell(n_1, n_2) = state : 1)$

Creates an empty grid
init_seq(n):

- output: $out := s$ such that $(|s| = n \wedge (\forall\ c : CellState | i, j \in [0..n-1] \wedge c = Dead : s[i][j] = c)$
- exception: n $\neq$ BOARDLENGTH $\implies$ invalid_argument

Local Types

N/A

Local Functions

Updates a cell according to its neighbours
updateCell(x, y): $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$
updateCell(x, y) $\equiv$

- transition:

cellState(x, y) = Alive	neighbourCnt(x, y) $<$ 2 $\vee$ neighbourCnt(x, y) $>$ 3	Grid[x][y] = Dead
	neighbourCnt(x, y) = 2 $\vee$ neighbourCnt(x, y) = 3	Grid[x][y] = Alive
cellState(x, y) = Dead	neighbourCnt(x, y) = 3	Grid[x][y] = Alive
	neighbourCnt(x, y) $\neq$ 3	Grid[x][y] = Dead

- output: $out := True$

View Module

Template Module

GridT

Uses

BoardTypes, GameBoard

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
Read	String	GridT	
show	GridT		
write	GridT		

Semantics

Environment Variables

readboard: A file containing a gameboard's initial state writeboard: A file showing a current gameboard state.

State Variables

State Invariant

stateCount(Alive) + stateCount(Dead) = BOARDLENGTH x BOARDLENGTH

Assumptions & Design Decisions

- The Read()'s inout file will be formatted correctly.

Access Routine Semantics

Read(file):

- output: Read data from the file and initialize a GridT that will act as a starting gameboard for the module. The file contains 25x25 square brackets with either a blank space or a * in between. The blank represents a dead cell and the * represents a live cell. Takes this data and create a matching GridT ADT.
- exception: None

Show(grid):

- output: Displays the input grid on the console with the same format as the input file. A 25x25 board containing square brackets with a blank or * to represent dead or alive cells, respectively.
- exception: None

Write(grid):

- output: Writes the input grid out to an output file with the same format as the input file. A 25x25 board containing square brackets with a blank or * to represent dead or alive cells, respectively.
- exception: None

Critique of Design

The design of this game was split into two main portions: the GameBoard and the View Module. As well, a BoardTypes module was constructed to outline two major facts about the board: the length and the cell types. The reason these modules were split were to outline the Model and the View portions of the Model-View-Controller implementation required. For the design, some decisions and assumptions were made to implement the design. First, the border cells treated the imaginary adjacent cells off the board as dead cells. This was made do that only live cells on the board would affect the game. Also, the file format was chosen to go hand-in-hand with the ASCII character display for continuity. More so, the instances created all specifically allocated tasks to increase readability and code organization.

The design is consistent across the different modules and the MIS. The MIS's outlines are followed by the code effectively and the requirements are outlined to correctly represent the program. As stated before, the file formats and display were consistent with one another. It is crucial for the formats of the file to be similar as it is not only required, but the code processes them similarly.

The design violated some aspects of essentiality as code organization and ease of use were prioritized. For example, I could have gathered and created an empty sequence without the extra function call of init_seq. Instead, this instance was created for better organization and increased cohesion. Minimality was not violated since every function served only one purpose and could not be split into two tasks. The design also respected information hiding since the state variables could not be accessed externally and instances were privatized, like init_seq. Generality was ignored in a sense through the GameBoard module when only a sequence of a sequence of CellStates was created. The module would have been general to accommodate all types representing Alive or Dead, but the module was limited to Cellstate for consistency. Only this type could be used in the implementation of the game.

Lastly, the code was cohesive and well-structured as the set of instances in both modules are closely related to one another. For example, in the GameBoard module, all instances are accessing the cells and either obtaining similar information or modifying its state. In the View Module, the tree functions are processing the exact same file/display format, with read and write performing identical but opposing functions.