Assignment 4 Specification

Leon So

April 13, 2019

This Module Interface Specification (MIS) document contains modules, types and methods for implementing, modelling, and viewing the state of a game of Conway's game of life.

Cell Types Module

Module

CellTypes

Uses

N/A

Syntax

Exported Constants

None

Exported Types

CellState = {Dead, Alive}

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Cell Module

Template Module

Cell

Uses

CellTypes

Syntax

Exported Types

Cell = ?

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
new Cell	$\mathbb{N}$, $\mathbb{N}$	Cell	invalid_argument
getX		$\mathbb{N}$	none
getY		$\mathbb{N}$	none
getState		CellState	none
setState	CellState		none
getAdj		$\mathbb{N}$	none
setAdj	$\mathbb{N}$		none

Semantics

State Variables

X: $\mathbb{N}$
Y: $\mathbb{N}$
$STATE$: CellState
ADJ: $\mathbb{N}$

State Invariant

X < 25
Y < 25

Assumptions & Design Decisions

- The Cell constructor is called for each object instance before any other access routine is called for that object. The constructor can only be called once.
- X and Y, the coordinates of the Cell, will always be less than 25.
- Every Cell will be initialized as Dead, and can be set to Alive.

Access Routine Semantics

new Cell(n_0, n_1):

- transition: $X, Y, STATE, ADJ := n_0, n_1, Dead, 0$
- output: $out := self$
- exception: $exc := ((n_0 \geq 25 \vee n_1 \geq 25) \Rightarrow invalid\_argument)$

getX():

- output: $out := X$
- exception: none

getY():

- output: $out := Y$
- exception: none

setState(s):

- transition: $STATE := s$
- exception: $exc := none$

getAdj():

- output: $out := ADJ$

- exception: none

setAdj(n):

- transition: $ADJ := n$
- exception: $exc := none$

CellGrid Module

Template Module

CellGrid is Seq of (Seq of Cell)

Game Board ADT Module

Template Module

Board

Uses

CellTypes
CellGrid
Cell
View

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
new Board	string	Board	invalid_argument
getTab	$\mathbb{N}, \mathbb{N}$	Cell	out_of_range
nextState			
outputAndView			

Semantics

State Variables

grid: CellGrid *#Grid of Cells*

State Invariant

$|grid| = 25 \times 25$ *#Size of grid fixed to 25 × 25*

Assumptions & Design Decisions

- The Board constructor is called before any other access routine is called on that instance. Once a Board has been created, the constructor will not be called on it again.
- The size of the grid will be fixed, to a 25×25 matrix.

- Any Cell outside of the 25 × 25 matrix will be considered Dead, therefore, when counting neighbours using countNeighbours(), there is no such Alive Cell outside of the 25 × 25 matrix.

Access Routine Semantics

Board($inFile$):

- transition: $grid := init\_board()$
  Read data from inFile. Use this data to update the state of the Board.
  This file contains coordinate of any Cell that is initially alive. Any Cell will have it's state set to Alive using the local function setAlive().

  The text file has the following format. Each cell's coordinate is identified by a string with the x-coordinate and y-coordinate separated by a comma. All cell coordinates are separated by a new line.

- exception: if inFile does not exist, raise an invalid_argument exception.

getCell(x, y):

- output: grid[x][y]

- exception: $exc := ((x \geq 25 \vee y \geq 25) \Rightarrow out\_of\_range)$

nextState():

- transition: calls local function countNeighbours() to count the number of neighbours/adjacents for each Cell in grid.
  $(\forall x : \mathbb{N} | x \in [0..24] : (\forall y : \mathbb{N} | y \in [0..24] :$
  $(grid[x][y].getAdj() = 3 \Rightarrow grid[x][y] := grid[x][y].setState(Alive) |$
  $grid[x][y].getAdj() \leq 1 \Rightarrow grid[x][y] := grid[x][y].setState(Dead) |$
  $grid[x][y].getAdj() \geq 4 \Rightarrow grid[x][y] := grid[x][y].setState(Dead))))$

- exception: None

outputAndView():

- output: calls the view(CellGrid grid) and writeState(CellGrid grid) functions from the View class, passing grid into both functions.

- exception: None

Local Types

None

Local Functions

init_board :
$(\forall x : \mathbb{N} | x \in [0..24] : (\forall y : \mathbb{N} | y \in [0..24] : grid[x][y] := Cell(x, y))$
setAlive : $\mathbb{N} \times \mathbb{N}$
grid[x][y].setState(Alive)

setDead : $\mathbb{N} \times \mathbb{N}$
grid[x][y].setState(Dead)

countNeighbours :
$(\forall x : \mathbb{N} | x \in [0..24] : (\forall y : \mathbb{N} | y \in [0..24] :$
$((x - 1 \geq 0 \wedge y - 1 \geq 0) \Rightarrow (grid[x - 1][y - 1].getState() = Alive \Rightarrow$
$grid[x][y] := grid[x][y].setAdj(grid[x][y].getAdj() + 1)))))$

$(\forall x : \mathbb{N} | x \in [0..24] : (\forall y : \mathbb{N} | y \in [0..24] :$
$((x - 1 \geq 0) \Rightarrow (grid[x - 1][y].getState() = Alive \Rightarrow$
$grid[x][y] := grid[x][y].setAdj(grid[x][y].getAdj() + 1)))))$

$(\forall x : \mathbb{N} | x \in [0..24] : (\forall y : \mathbb{N} | y \in [0..24] :$
$((x - 1 \geq 0 \wedge y + 1 < 25) \Rightarrow (grid[x - 1][y + 1].getState() = Alive \Rightarrow$
$grid[x][y] := grid[x][y].setAdj(grid[x][y].getAdj() + 1)))))$

$(\forall x : \mathbb{N} | x \in [0..24] : (\forall y : \mathbb{N} | y \in [0..24] :$
$((y - 1 \geq 0) \Rightarrow (grid[x][y - 1].getState() = Alive \Rightarrow$
$grid[x][y] := grid[x][y].setAdj(grid[x][y].getAdj() + 1)))))$

$(\forall x : \mathbb{N} | x \in [0..24] : (\forall y : \mathbb{N} | y \in [0..24] :$
$((y + 1 \geq 0) \Rightarrow (grid[x][y + 1].getState() = Alive \Rightarrow$
$grid[x][y] := grid[x][y].setAdj(grid[x][y].getAdj() + 1)))))$

$(\forall x : \mathbb{N} | x \in [0..24] : (\forall y : \mathbb{N} | y \in [0..24] :$
$((x + 1 < 25 \wedge y - 1 \geq 0) \Rightarrow (grid[x + 1][y - 1].getState() = Alive \Rightarrow$
$grid[x][y] := grid[x][y].setAdj(grid[x][y].getAdj() + 1)))))$

$(\forall x : \mathbb{N} | x \in [0..24] : (\forall y : \mathbb{N} | y \in [0..24] :$
$((x + 1 < 25) \Rightarrow (grid[x + 1][y].getState() = Alive \Rightarrow$
$grid[x][y] := grid[x][y].setAdj(grid[x][y].getAdj() + 1)))))$

$(\forall x : \mathbb{N} | x \in [0..24] : (\forall y : \mathbb{N} | y \in [0..24] :$
$((x + 1 < 25 \wedge y + 1 < 25) \Rightarrow (grid[x + 1][y + 1].getState() = Alive \Rightarrow$
$grid[x][y] := grid[x][y].setAdj(grid[x][y].getAdj() + 1)))))$

View Module

Uses

CellTypes
CellGrid
Cell

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
view	CellGrid		none
writeState	CellGrid		none

Semantics

State Variables

None

State Invariant

None

Assumptions & Design Decisions

- The view(CellGrid grid) function will output an ASCII representation of the game state in the console.
- The writeState(CellGrid grid) function will output an output file,"output.txt", which contains the state of the game. Each time this method is called, it will produce a new "output.txt" file if it does not already exist, or override the existing file.

Access Routine Semantics

view(*grid*):

- output: output an ASCII representation of the game state and the grid of cells in console. This function checks the state of each Cell in grid using the getState() method. The ASCII representation is outputted in the following format:
  - Each row contains 25 cells.
  - There are 25 rows.
  - Each row is printed on a new line.
  - Every Dead Cell is represented by '[ ]'.
  - Every Alive Cell is represented by '[#]'.
  - Each Cell in a row is printed side by side with no space in-between.
- exception: none

writeState(*grid*):

- output: outputs a file representing the state of the game and inputted grid, "output.txt", which contains the state of the game. This file contains coordinate of any Cell that is alive. The text file has the following format. Each cells coordinate is identified by a string with the x-coordinate and y-coordinate separated by a comma. All cell coordinates are separated by a new line.
- exception: none

Local Types

None

Local Functions

None

Critique of Design

My design implements information hiding, as all lower level details are hidden through the use of private/local function. There is also high cohesion between my modules, as all the modules are closely related to each other. Also there is relatively low coupling, as the two modules which has the highest fan out are the GameBoard module and View module. They both use Cell, CellTypes, and CellGrid, therefore, with a fan-out of 3, there is relatively low coupling. The majority of this program is essential, except for the CellTypes and CellGrid module. These modules are not needed, however, simplify the process of developing the code through the typedefs and enumerations declared in these modules. All routines in the Cell, GameBoard, and View module are essential. None of the routines in these modules can be replaced by other routines in the module. All public routines and methods are minimal as they each perform an independent service, however my design could be more minimal through decomposing the local functions further such that they each perform a minimum independent service. This program and specification is also consistent, as consistent naming conventions are used throughout the entire program and specification. In addition, there is also consistent ordering of parameters, such as the ordering of x, y coordinates. The GameBoard module is not very general, as it accepts an input file of a specific and assumed formatting. This program also accounts for unexpected inputs such as through raising an out_of_range exception if a user inputs a coordinate out of range for the getCell(nat n_0, nat n_1) function.