Assignment 3, Part 1, Specification

SFWR ENG 2AA4

April 10, 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. The game involves a grid of size N x N, of cells that can be "ALIVE" or "DEAD". The player can set the start state of the game through a txt file that will be read by the program. The program will then construct the game according to the txt file and display the changes from one game state to the next game state.

Board Types Module

Module

BoardTypes

Uses

N/A

Syntax

Exported Constants

SIZE = 4

Exported Types

cellT = {ALIVE, DEAD}

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Game Board ADT Module

Template Module

BoardT

Uses

BoardTypes

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
new BoardT	s : string	BoardT	
read	s : string		invalid_argument
write	s : string		
getRows		$\mathbb{N}$	
getColumns		$\mathbb{N}$	
nextState		BoardT	
getCell	$\mathbb{N}$, $\mathbb{N}$	cellT	invalid_argument

Semantics

State Variables

B: seq of (seq of T) # *Gameboard*
rows: int # *Number of rows in gameboard*
columns: int # *Number of columns in gameboard*

State Invariant

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.
- The input file will match the given specification

- 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

BoardT(s):

- transition: $B :=$ read(s)
- exception: *none*

read(s):

- transition: read data from the file associated with the string s. The data in the text file will be used to initiate the gameboard state variables B, $rows$ and $columns$, where $rows$ is the number of lines in the file, $columns$ is the number of characters on each line in the file, and B holds the CellT at a particular row and column.

  The text file has the following format, where " " stand for a CellT with a value of DEAD and "o" stands for CellT with a value of ALIVE. All characters in the file will either be " " or "o". lines are separated by a carriage return. There can be any finite number of lines in the file with finite number of characters on each line as long as the length of each line in the file is the same for every line. If the length of each line in the file is not the same for the whole file, there is an invalid_argument exception

- exception: invalid_argument

write(s):

- transition: write gameboard state to the file associated with the string s.

  The text file has the following format, where " " will be used for cellT with a value of DEAD and "o" will be used for cellT with a value of ALIVE. lines are separated by a carriage return. Each line will represent a seq in B. Each character in the line will be a representation of the CellT values at that seq and position in the seq

- exception: none

getRows():

- output: $out := rows$

- exception: none

getColumns():

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

getCell(n_0, n_1):

- output: $out := B[n_0][n_1]$
- exception: $exc := (\neg \text{isValidCell}(n_0, n_1)) \Rightarrow \text{invalid\_argument})$

nextState():

- output: $out := newBoardT(s, rows, columns)$ such that $(\forall\, i \in [0..rows-1] : (\forall\, j \in [0..columns-1] : survives(i,j) \Rightarrow s[i][j] = ALIVE \wedge \neg survives(i,j) \Rightarrow s[i][j] = DEAD))$
- exception: none

Local Functions

BoardT: (seq of seq of CellT) $\times\, \mathbb{N} \times \mathbb{N} \rightarrow BoardT$
BoardT(s, r, c)
$\equiv B_2$ such that $(B_2.getRows() = r \wedge B_2.getColumns = c \wedge (\forall\, i \in [0..r-1] : (\forall\, j \in [0..c-1] : B_2[i][j] = s[i][j]))$

isValidCell: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$
isValidCell$(n_0, n_1) \equiv (n_0 < 0 \vee n_0 \geq rows \vee n_1 < 0 \vee n_1 \geq columns) \Rightarrow$ false

neighbourCount: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
neighbourCount$(n_0, n_1) \equiv +(\forall i, j : \mathbb{N} | i \in (n_0 - 1, n_0 + 1) \wedge j \in (n_1 - 1, n_1 + 1) \wedge isValidCell(i,j) \wedge (getCell(i,j)) = ALIVE : 1)$

survives: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$
survives $(n_0, n_1) \equiv$

$B[n_0][n_1] = \text{DEAD}$	$neighbourCount(n_0, n_1) = 3$	True
	$neighbourCount(n_0, n_1) \neq 3$	False
$B[n_0][n_1] = \text{ALIVE}$	$neighbourCount(n_0, n_1) = 3 \vee neighbourCount(n_0, n_1) = 2$	True
	$neighbourCount(n_0, n_1) \neq 3 \wedge neighbourCount(n_0, n_1) \neq 2$	False

View Module

Module

View

Uses

BoardTypes GameBoard

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
new View		View	
print	s : BoardT		

Semantics

Environment Variables

State Variables

None

State Invariant

None

Assumptions

The input file will match the given specification.

Access Routine Semantics

View(s)

- Constructor for view
- exception: none

print(s)

- displays the state of the gameboard using text-based ASCII graphics. Dead cells will be represented using " " and live cells will be represented using the "o" character. The number of rows and columns depends on the row and column state variables from the gameboard

$$\begin{matrix} o & o & o & ... \\ o & o & o & ... \\ o & o & o & ... \\ ... & ... & ... & ... \end{matrix} \tag{1}$$

- exception: none

Critique of Design

The interface for the modules has rigour and formality. It uses language from discrete math, which has predefined symbols formal syntax, and precise semantics, removing all ambiguity. As well, arguments are checked to see if they are valid, and if not, an exception is called. Modules exercise a proper amount of separation of concerns, with high cohesion and low coupling. All modules are components of a game, so they are closely related. However, modules only call upon each other when necessary. For example, the StackT object does not call upon any other modules , but only provides the necessary functions for itself. The StackT module exhibits generality, as the T value can be of any type. As a result, StackT can be used to represent stacks of other types. Each function only performs one task, so it is clear how each function could be, or will be used. For example, there is a function "is_valid_tab_mv," which checks to make sure the cardT values are correct, and then a "is_valid_pos" function which checks to make sure the arguments are valid, and then a "tab_mv" function which actually moves the values.

A possible improvement would be making the "valid_tab_tab" and other valid functions more general, so that the functions could be reused for other types of solitare . Another related improvement would be the sizes for the tableau and foundation could be a exported constant, so that the number of tableaus and foundation could be changed. Possible considerations would be limiting the size, or setting a constant for a maximum size of the stack of tableaus, as when the game is actually implemented, the tableau will show an image of the cards, and the number of cards in a tableau stack shouldn't be able to extend out of the game window. This will also limit ways the user could use the applications, and prevent possible bugs or errors

[Write a critique of the interface for the modules in this project. Is there anything missing? Is there anything you would consider changing? Why? —SS]