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 line and position in the line. The line number and position of the character in the file must be reflected in B.

  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. If the file is empty, 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. The line number and position of the character in B must be reflected in the file.

- 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: $exc :=$ (¬isValidCell(n_0, n_1)) ⇒ 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) × ℕ × ℕ → $BoardT$
BoardT(s, r, c)
≡ 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: ℕ × ℕ → 𝔹
isValidCell(n_0, n_1) ≡ $(n_0 < 0 \vee n_0 \geq rows \vee n_1 < 0 \vee n_1 \geq columns) \Rightarrow$ false

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

survives: ℕ × ℕ → 𝔹
survives (n_0, n_1) ≡

$B[n_0][n_1] = \text{DEAD}$	$neighbourCount(n_0, n_1) = 3$	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 output 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.

  The view 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 view will either be " " or "o". The number of lines and characters on each line depends on the row and column state variables respectively, from the gameboard. The line number and position of the character in the view must be consistent with the gameboard's B state variable.

$$\begin{array}{ccc} o & o & o \\ o & o & o \\ o & o & o \end{array} \tag{1}$$

- exception: none

Critique of Design

This specification was written with a mix of natural language and formal language. Natural language was used for the view's print method, as there are properties in prints to the console that cannot be described by using formal language. Natural language was also used for the read and write of the gameboard module, as the format of the text file has properties that cannot be described using formal language. By using formal language for the other functions in the module, the specification that was written in formal language is unambiguous by definition. Parnas tables were used accordingly to ensure that the specification is complete, and covers all necessary cases.

There are two modules in this specification: One module for view, called View, and one module for the model, called BoardT. The View module displays the gameboard, and the BoardT module represents the state of the game, and has functions that allow for initializing the state, exporting the state, and calculating the state in the next "turn" of the game. The components in this module are closely related, showing high cohesion, which is a desired property in software design. By having separte modules for separate concerns: displaying and state functionality, this design shows good modularity, and separation of concerns. Although the View module uses some access programs provided by the BoardT module to access information about the gameboard, it does not rely on the functions in BoardT to display, showing a reasonable amount of coupling between the two modules.

The BoardT module allows for Generality, since it allows for gameboards of any length and width as long as the length of each line in the file is consistent. However, many other aspects and functions such as the character to represent a "live cell" have been constrained, in order to reduce possible errors and bugs. For example, if a live cell could be represented by any character in the english alphabet, in the next "turn" of the gameboard, there are no rules implemented in the game to determine what will represent the "live" and "dead" cells. For simplicity, only the "o" character can be used.

The modules in this design were designed to be opaque and essential, as all helper functions were made local functions, and all necessary functions for the module were made exported access programs. Each function in the BoardT module, with the help of several helper functions are minimal, each function performs a clear and specific function allowing the design to be general. For example, getRows exported access program is an accessor, that does not mutate at the same time, while having a descriptive function name. The variable names in the design were consistent. For example, for the 2D sequence, the n_0, and is always first and always represents the "row" index, and the n_0 is always second and always represents the "column" index.