Assignment 4, Specification

SFWR ENG 2AA4

April 10, 2019

This Module Interface Specification (MIS) document contains modules, types and methods for implementing the model and view of Conway's Game of Life.

Cell Type Module

Module

Cell Type

Uses

N/A

Syntax

Exported Constants

Exported Types

CellT = tuple of (x:$\mathbb{N}$, y:$\mathbb{N}$, status:$\mathbb{B}$)

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Read Module

Module

Read

Uses

Cell Types Board

Syntax

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
ReadFile	s : string	*state* : seq of(seq of CellT)	runtime_error

Semantics

Environment Variables

boardLoad: File containing initial board pattern

State Variables

None

State Invariant

None

Assumptions

The input file matches the specification.

Access Routine Semantics

ReadFile(s)

- transition: read data from the file boardLoad associated with the string s. Use this data to update the state of the Board module. The sequence of sequence of $CellT$ produced by ReadFile will be passed to Board to initialize it.

  The text file has the following format, where $cstat_m n$, stands for a boolean that represent the status of the cell, essentially whether that cell is dead or alive. With 0 being dead and 1 being alive. The values in a row are separated by spaces. The configuration has dimensions n for columns and m for rows, where n and m are $\mathbb{N}$. The data shown below is the format of a file.

$$\begin{array}{ccccc} cstat_{00} & cstat_{01} & cstat_{02} & cstat_{03}... & cstat_{0n-1} \\ cstat_{10} & cstat_{11} & cstat_{12} & cstat_{13}... & cstat_{1n} \\ cstat_{20} & cstat_{21} & cstat_{22} & cstat_{23}... & cstat_{2n} \\ ... & ... & ... & ... & ... \\ cstat_{m-10} & cstat_{m-11} & cstat_{m-12} & cstat_{m-13}... & cstat_{m-1n-1} \end{array} \quad (1)$$

- exception: runtime_error

View Module

Module

View

Uses

Cell Types
Board

Syntax

Exported Types

None

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
viewScreen	*state* : seq of(seq of CellT)	Console *see environment variables*	none
WriteFile	*state* : seq of(seq of CellT)	output *see environment variables*	none

Semantics

Environment Variables

Console: output to the screen
output: File that board state is written to

State Variables

None

State Invariant

None

Assumptions & Design Decisions

- The View module is called by the Board to show the changes to the board once the rules of the game of life have been applied to the cells.

Access Routine Semantics

viewScreen(*state*)

- transition: read data from the passed in sequence of sequence of $CellT$ and output it to the screen.

  The output to the screen has the following format, where $cstat_m n$, stands for a boolean that represent the status of the cell, essentially whether that cell is dead or alive. With 0 being dead and 1 being alive. The values in a row are separated by spaces. The configuration has dimensions n for columns and m for rows, where n and m are $\mathbb{N}$. The data shown below is the format of a file.

$$\begin{array}{ccccc} cstat_{00} & cstat_{01} & cstat_{02} & cstat_{03}... & cstat_{0n-1} \\ cstat_{10} & cstat_{11} & cstat_{12} & cstat_{13}... & cstat_{1n} \\ cstat_{20} & cstat_{21} & cstat_{22} & cstat_{23}... & cstat_{2n} \\ ... & ... & ... & ... & ... \\ cstat_{m-10} & cstat_{m-11} & cstat_{m-12} & cstat_{m-13}... & cstat_{m-1n-1} \end{array} \tag{2}$$

- exception: none

WriteFile (*state*)

- transition: Take the data given from the passed in sequence of sequence of $CellT$ and write the data into a file. The file has this format, here $cstat_m n$, stands for a boolean that represent the status of the cell, essentially whether that cell is dead or alive. With 0 being dead and 1 being alive. The values in a row are separated by spaces. The configuration has dimensions n for columns and m for rows, where n and m are $\mathbb{N}$. The data shown below is the format of a file.

$$\begin{array}{ccccc} cstat_{00} & cstat_{01} & cstat_{02} & cstat_{03}... & cstat_{0n-1} \\ cstat_{10} & cstat_{11} & cstat_{12} & cstat_{13}... & cstat_{1n} \\ cstat_{20} & cstat_{21} & cstat_{22} & cstat_{23}... & cstat_{2n} \\ ... & ... & ... & ... & ... \\ cstat_{m-10} & cstat_{m-11} & cstat_{m-12} & cstat_{m-13}... & cstat_{m-1n-1} \end{array} \tag{3}$$

- exception: none

Game Board Module

Module

Board

Uses

CellTypes

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
new Board	seq of (seq of CellT)	Board	none
update			none
getNewState		seq of (seq of CellT)	none

Semantics

State Variables

board_state: seq of (seq of CellT)
new_state: seq of (seq of CellT)

State Invariant

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 Board is initialized with a sequence of sequence of CellT outputted from the Read module. Therefore the Read module must be called before the Board constructor.
- Each time the board is modified the new state is the state of the board that is passed to the view module.
- Once a cell is out of bounds of the board dimensions it is no longer considered when counting the neighbours to follow the rules of the game of life.

Access Routine Semantics

Board(seq of (seq of CellT)):

- transition: board_state := seq of (seq of CellT)
  new_state := seq of (seq of CellT)

- exception: None

update():

- transition: board_state := new_state

  $(\forall r : \mathbb{N}, c : \mathbb{N} | r < |\text{ board\_state }| \wedge c < |\text{ board\_state}| : \text{checkNeighbour}(board\_state[r][c]))$

		new_state :=
checkNeighbour($board\_state[r][c]$) < 2	board_state[r][c].status $= 1$	new_state[r][c].status $= 0$
checkNeighbour($board\_state[r][c]$) > 2	board_state[r][c].status $= 1$	new_state[r][c].status $= 0$
checkNeighbour($board\_state[r][c]$) $= 2$	board_state[r][c].status $= 1$	new_state[r][c].status $= 1$
checkNeighbour($board\_state[r][c]$) $= 3$	board_state[r][c].status $= 1$	new_state[r][c].status $= 1$
checkNeighbour($board\_state[r][c]$) $= 3$	board_state[r][c].status $= 0$	new_state[r][c].status $= 1$

- output: None

- exception: None

getNewState():

- output: *out* := new_state

- exception: None

Local Functions

checkNeighbour : CellT $\rightarrow \mathbb{N}$
checkNeigbour(CellT n) $\equiv$

		$out := neigh$
$n.r - 1 \geq 0$	$n.c - 1 \geq 0$	$(+neigh : \mathbb{N} \mid \text{board\_state}[r-1][c-1].status = 1 : 1)$
$n.r - 1 \geq 0$		$(+neigh : \mathbb{N} \mid \text{board\_state}[r-1][c].status = 1 : 1)$
$n.r - 1 \geq 0$	$n.c + 1 \leq \lvert\text{board\_state}\rvert$	$(+neigh : \mathbb{N} \mid \text{board\_state}[r-1][c+1].status = 1 : 1)$
	$n.c - 1 \geq 0$	$(+neigh : \mathbb{N} \mid \text{board\_state}[r][c-1].status = 1 : 1)$
	$n.c + 1 \leq \lvert\text{board\_state}\rvert$	$(+neigh : \mathbb{N} \mid \text{board\_state}[r][c+1].status = 1 : 1)$
$n.r + 1 \leq 0 \lvert\text{board\_state}\rvert$	$n.c - 1 \geq 0$	$(+neigh : \mathbb{N} \mid \text{board\_state}[r+1][c-1].status = 1 : 1)$
$n.r + 1 \leq 0 \lvert\text{board\_state}\rvert$		$(+neigh : \mathbb{N} \mid \text{board\_state}[r+1][c].status = 1 : 1)$
$n.r + 1 \leq 0 \lvert\text{board\_state}\rvert$	$n.c + 1 \leq \lvert\text{board\_state}\rvert$	$(+neigh : \mathbb{N} \mid \text{board\_state}[r+1][c+1].status = 1 : 1)$

Critique of Design

For assignment four we as students were asked to create a specification and implementation of Conway's Game of Life using the software engineering principles we learned this term in 2AA4.

0.1 Informataion Hiding

For my implementation I had three main modules and one type. Each of these modules follows the principle of information hiding and holds a certain "secret". The CellType module's secret is the data structure of cells that make up the board. The Board module's secret is the state of the game board after applying the rules of the game of life. The Read module holds the secret of how the game interacts with the environment variable of text files. The View module's secret is how the user sees the game's state, whether in console or text file. I decided on three modules instead of simply two for this assignment because I felt that adding file reading to the board module would not be a great secret since the module would be interacting with environment variable and changing the state of the board.

0.2 Consistency and Cohesion

For my modules I tried to have cohesion between the methods included in the module and use consistent conventions. There is cohesion in the modules because they have been decomposed in such a way to allow all the related methods to be grouped together. For example the Board module includes the object constructor, getters and methods to update the board's state. This is an example of cohension since these methods all relate to one another. I kept consistency between modules by using similar naming coventions so that across all modules the specification would make sense. For example each cell has two $\mathbb{N}$ that represent its row and column. So in every loop iterating through cells I use r and c to represent the row and column.

0.3 Generality

For generality, I made my version of game of life allow the user to enter any board size, without restrictions. As well as having a different number of rows and columns so that the board is not completely square. This allows this version of the game of life to be used in a variety of cases that may be restricted by other versions.

0.4 Essentiality

I tried to hold to the principal of essentiality when designing my assignment by ensuring I only specified and coded what was needed by the assignment requirements and for the game to function. Each module is fairly small, consisting of only a few methods, each is distinct and performs a function that another method cannot replicate. For example, I could have easily added more methods to the Board module, such as a size getter, but instead I just used built in functions so that it was not being redundant by coding something that does not need to be.

0.5 Minimality

For the principal of minimality I avoided combining independant services together. For example, printing to the screen and writing to a file have a very similar structure to one another in code. Even after noticing this I kept them as separate methods in the view module because if they were combined it would be offer two different services, console output and file writing. Therefore I made the design decision to keep them separate to avoid violating minimality.