Assignment 4, Part 1, Specification

SFWR ENG 2AA4

April 12, 2019

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

Board Types Module

Module

BoardTypes

Uses

N/A

Syntax

Exported Constants

MAX_DIMENSIONS = 20

Exported Types

Status = {Alive, Dead}
CardT = tuple of (s: Status)

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Console Output Module

Module

Output

Uses

BoardTypes

Syntax

Exported Types

None

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
print	seq of seq of Cell, $\mathbb{N}$		none

Semantics

State Variables

None

State Invariant

None

Assumptions & Design Decisions

- The decision to add a natural number parameter to the print function was made for the possible scenario of indicating on the console what generation is being printed.

- The design decision of having console output be a seperate module than the module responsible for reading and writing to and from a file was made. This decision was made to enforce the idea of seperation of concerns. The output to the screen may be different than that of the output file.

Access Routine Semantics

print(*universe*, n):

- output: The current generation given by n is outputted to the console followed by seq of seq of Cell.
- exception: none

File Read And Write Module

Module

ReadAndWrite

Uses

BoardTypes

Syntax

Exported Types

None

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
read	string	seq of seq of Cell	runtime_error, invalid_argument
write	seq of seq of Cell, string		none

Semantics

State Variables

None

State Invariant

None

Assumptions & Design Decisions

- The decision to add a string parameter to the read and write functions was made to allow for the user to have potentially multiple files that they'd want to start multiple games from and write them all to individual files without the risk of overwritting previously written to files.

Access Routine Semantics

read(file):

- output: file gets read. Data from file is in the following format, $C_0, C_1, C_2, ..., C_M$ where C is an ASCII character either "#" or "_", where "#" represents a live cell and "_" represents a dead cell. There must be $MAX\_DIMENSIONS * MAX\_DIMENSIONS$ number of cells. The input file must have MAX_DIMENSIONS number of rows with MAX_DIMENSIONS number of columns.

  While parsing through the data a seq of seq of Cell is built and returned.

- exception: runtime_error if file does not exist
  invalid_argument if number of rows or number of columns does not equal MAX_DIMENSIONS

write(*universe*, file):

- output: A new file is created with the name specified by the file parameter. The format of this file is the same as the input file used in the read function. *universe* is parsed through, a cell that is alive is associated with "#" in the output file and a cell that is dead is associated with "_".

- exception: None

GameBoard ADT Module

Template Module

GameBoard

Uses

BoardTypes

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
GameBoard		GameBoard	
GameBoard	Universe	GameBoard	
hasNeighbours	$\mathbb{N}$, $\mathbb{N}$	$\mathbb{B}$	invalid_argument
numOfNeighbours	$\mathbb{N}$, $\mathbb{N}$	$\mathbb{N}$	invalid_argument
getCell	$\mathbb{N}$, $\mathbb{N}$	Cell	invalid_argument
getUniverse		Universe	
nextState			
gameOver		$\mathbb{B}$	

Semantics

State Variables

U: Universe

State Invariant

None

Assumptions & Design Decisions

- The GameBoard constructor is called before any other access routine is called on that instance. Once a GameBoard has been created, the constructor will not be called on it again.

- Cells at edges of the GameBoard only have neighbouring cells in the Gameboard that is visible. That is, there aren't any cells outside of the visible Gameboard. This is not an infinitely sized board.
- 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, though violating the property of being essential, to give a view function easy access to the state of the game.
- The hasNeighbours function violates minimality as I could use numOfNeighbours and check if it returns zero but having this function would prove useful in reducing computation time for larger gameboards. This function could act as a short-circuit of sorts when computing the nextState.

Access Routine Semantics

GameBoard():

- transition: $U :=$ init_uni()
- exception: None

GameBoard(universe):

- transition: $U :=$ universe
- exception: None

hasNeighbours(n, m):

- output: $out := (\exists x, y : \mathbb{Z} | -1 \leq x \leq 1 \wedge -1 \leq y \leq 1 : U[n+x][m+y].s = Alive)$
- exception: $exc :=$ ($n \geq$ HEIGHT $\vee$ $m \geq$ WIDTH $\Rightarrow$ invalid_argument)

numOfNeighbours(n, m):

- output: $out := (+x, y : \mathbb{Z} | -1 \leq x \leq 1 \wedge -1 \leq y \leq 1 \wedge \text{U}[n+x][m+y].s = Alive : 1)$
- exception: $exc :=$ ($n \geq$ HEIGHT $\vee$ $m \geq$ WIDTH $\Rightarrow$ invalid_argument)

getCell(n, m):

- output: $out := U[n][m]$

- exception: $exc := (n \geq \text{HEIGHT} \vee m \geq \text{WIDTH} \Rightarrow \text{invalid\_argument})$

getUniverse():

- output: $out := U$
- exception: None

nextState():

- transition: $U := N$: Universe such that
  $(\forall x, y : \mathbb{N}|$
  $x, y < \text{MAX\_DIMENSIONS} \wedge (U[x][y].s = Alive \wedge ((\text{numOfNeighbours}(x, y) = 3 \vee$
  $\text{numOfNeighbours}(x, y) = 2)) \vee (U[x][y].s = Dead \wedge \text{numOfNeighbours}(x, y) = 3)) \Rightarrow$
  $N[x][y].s = Alive$
  $|True \Rightarrow N[x][y].s = Dead)$
- exception: None

gameover():

- output: $out := \neg(\exists n, m : \mathbb{N}|n < \text{MAX\_DIMENSIONS} \wedge m < \text{MAX\_DIMENSIONS} :$
  $U[n][m].s = Alive)$
- exception: None

Local Types

Universe = seq of seq of Cell
WIDTH = MAX_DIMENSIONS
HEIGHT = MAX_DIMENSIONS

Local Functions

init_uni :→ Universe
init_seq() ≡ u such that ($|u| = \text{MAX\_DIMENSIONS} \wedge (\forall\, i \in [0..\text{MAX\_DIMENSIONS} -$
$1] : u[i] = v$ such that ($|v| = \text{MAX\_DIMENSIONS} \wedge (\forall\, j \in [0..\text{MAX\_DIMENSIONS} - 1] :$
$v[j].s = Dead$

Critique of Design

The design takes a functional approach to the task of creating the Model and View modules for playing Conway's Game of Life. The design is consistent, that is given an input no matter how many times the game is run with that input, the output will always be the same. For essentiality I have violated it in some locations of this MIS. I have violated it for GameBoard's getCell and getUniverse function as these are needed for testing purposes and for the view of the game's state. Minimality was violated for having a seperate module for printing to the console, I could have done this in the ReadAndWrite module but chose to keep seperation of concerns in mind and also information hiding in mind. I wanted to hide the "secrets" of the way i printed to the console and seperate that from the way I outputted to a txt file. For generality, my MIS fails. I chose to write an MIS dedicated to the task of having a Model and View of Conway's Game of Life. I could have made my MIS more general by having a generic output and a generic read and write. I'm not sure how I would make the GameBoard module more generic. For minimality as a whole I attempted to keep the amount of unneccessary functions to a minimum. Cohesion is decent in the MIS, every module depends on BoardTypes, besides BoardTypes of course, but other than being dependant on BoardTypes the other modules are independed from one another.