Assignment 4, Specification

SFWR ENG 2AA4

April 13, 2019

Joshua Guinness, guinnesj, 400134735

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

In applying the specification, there will be cases that involve undefinedness. We will interpret undefinedness following [?]:

If $p : \alpha_1 \times .... \times \alpha_n \rightarrow \mathbb{B}$ and any of $a_1, ..., a_n$ is undefined, then $p(a_1, ..., a_n)$ is False. For instance, if $p(x) = 1/x < 1$, then $p(0) =$ False. In the language of our specification, if evaluating an expression generates an exception, then the value of the expression is undefined.

Cell ADT Module

Module

Cell

Uses

N/A

Syntax

Exported Constants

None

Exported Types

Cell = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new Cell		Cell	none
new Cell	boolean	Cell	none
get_life		boolean	none
get_neighbours		int	none
set_life	boolean		none
set_neighbours	int		out_of_range

Semantics

State Variables

S: boolean # *Alive or Dead*
N: int # *Number of neighbors*

State Invariant

$n \leq 8$

Assumptions and Design Decisions

- The Cell(S) or 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.

Access Routine Semantics

new Cell():

- transition: $S, N := false, 0$
- output: $out := self$
- exception: none

new Cell(s):

- transition: $S, N := s, 0$
- output: $out := self$
- exception: none

get_life():

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

get_neighbours():

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

set_life(s):

- transition: $S := s$
- output: none
- exception: none

set_neighbours(n):

- transition: $N := n$
- output: none
- exception: $exc := (n > 8 \Rightarrow out\_of\_range)$ *# Can't have more than 8 neighbours*

Game Board ADT Module

Template Module

BoardT

Uses

Cell
View

Syntax

Exported Constants

None

Exported Types

BoardT

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new BoardT	seq of (seq of Cell)	BoardT	invalid_argument
next			
view			

Semantics

State Variables

C: seq of (seq of Cell) *#2D Array of Cells*

State Invariant

$|seq\ of\ Cell| = |seq\ of\ (seq\ of\ Cell)|$ *#2D array is a perfect square*

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 seq of (seq of C) that is passed to the constructor is a perfect square. This means that both sequences are of the same length.
- 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 view() function calls the view module which displays the current state of the game.

Access Routine Semantics

new BoardT(c):

- transition: $C := c$
- output: $out := self$
- exception: invalid_argument

next():

- transition: $C := update\_neighbours\_middle(), update\_neighbours\_leftside(),$ $update\_neighbours\_rightside(), update\_neighbours\_top(),$ $update\_neighbours\_bottom(), update\_corners(), update\_cells()$

  # Gets the next iteration of the game by first updating the neighbour count for each Cell in the seq of (seq of Cell), then updates all Cells to be either alive (true) or dead (false) based on neighbour count
- output: none
- exception: none

view():

- transition: none
- output: none
- exception: none

Local Types

None

Local Functions

update_neighbours_middle: *# Updates all the neighbour counts for Cells that are not on the edge*

update_neighbours_middle() $\equiv \forall i : \mathbb{N} \mid i \in [1..|seq\ of\ C|]\ :\ (\forall j : \mathbb{N} \mid j \in [1..|seq\ of\ C|]\ :\ C[i][j].set\_neighbours(C[i-1][j-1] + C[i-1][j] + C[i-1][j+1] + C[i][j-1] + C[i][j+1] + C[i+1][j-1] + C[i+1][j] + C[i+1][j+1]))$

update_neighbours_leftside: *# Updates all the neighbour counts for Cells on the left side*

update_neighbours_leftside() $\equiv \forall i : \mathbb{N}|i \in [1..|seq\ of\ C|-2] : C[i][0].set\_neighbours(C[i-1][0] + C[i-1][1] + C[i][1] + C[i+1][1] + C[i+1][0])$

update_neighbours_rightside: *# Updates all the neighbour counts for Cells on the right side*

update_neighbours_rightside() $\equiv \forall i : \mathbb{N}|i \in [1..|seq\ of\ C|-2] : C[i][|seq\ of\ C|-2].set\_neighbours(C[i-1][|seq\ of\ C|-2] + C[i-1][|seq\ of\ C|-3] + C[i][|seq\ of\ C|-3] + C[i+1][|seq\ of\ C|-3] + C[i+1][|seq\ of\ C|-2])$

update_neighbours_top: *# Updates all the neighbour counts for Cells on the top*

update_neighbours_top() $\equiv \forall i : \mathbb{N}|j \in [1..|seq\ of\ C|-2] : C[0][j].set\_neighbours(C[0][j-1] + C[1][j-1] + C[1][j] + C[1][j+1] + C[0][j+1|])$

update_neighbours_bottom: *# Updates all the neighbour counts for Cells on the bottom*

update_neighbours_bottom() $\equiv \forall i : \mathbb{N}|j \in [1..|seq\ of\ C|-2] : C[|seq\ of\ C|-2][j].set\_neighbours(C[|seq\ of\ C|-2][j-1] + C[|seq\ of\ C|-3][j-1] + C[|seq\ of\ C|-3][j] + C[|seq\ of\ C|-3][j+1] + C[|seq\ of\ C|-2][j+1|])$

update_neighbours_corners: *# Updates the neighbour count for the four corner cells*

update_neighbours_corners() $\equiv$
$C[0][0].set\_neighbours(C[0][1] + C[1][1] + C[1][0])$

$C[0][|seq\ of\ C|-1].set\_neighbours(C[0][|seq\ of\ C|-2]+C[1][|seq\ of\ C|-2]+C[1][|seq\ of\ C|-1])$
$C[|seq\ of\ C|-1][0].set\_neighbours(C[|seq\ of\ C|-2][0]+C[|seq\ of\ C|-2][1]+C[|seq\ of\ C|-1][1])$
$C[|seq\ of\ C|-1][|seq\ of\ C|-1].set\_neighbours(C[|seq\ of\ C|-1][|seq\ of\ C|-2]+C[|seq\ of\ C|-2][|seq\ of\ C|-2]+C[|seq\ of\ C|-2][|seq\ of\ C|-1])$

update_cells: *# Iterates through all cells, setting them to alive (true) or dead (false) based on the rules of the game and the number of neighbours it has*

update_cells() $\equiv$
$\forall x : Cell\ .\ x \in C \wedge x.get\_life = true \mid ((x.get\_neighbours \leq 1 \Rightarrow x.set\_life := false) \vee (x.get\_neighbours \geq 4 \Rightarrow x.set\_life := false) \vee (x.get\_neighbours > 1 \wedge x.get\_neighbours < 4 \Rightarrow x.set\_life := true))$

$\forall x : Cell\ .\ x \in C \wedge x.get\_life = False \mid ((x.get\_neighbours = 3 \Rightarrow x.set\_life := alive) \vee (x.get\_neighbours() > 3 \vee x.get\_neighbours < 3 \Rightarrow x.set\_life() := false))$

Read Module

Module

Read

Uses

Cell

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
read_state	s : String	seq of (seq of Cell)	

Semantics

Environment Variables

initial_state.txt: File which will be the initial state of game

State Variables

None

State Invariant

None

Assumptions and Design Decisions

- The contents of the file are in the right format and will match the given specification.

Access Routine Semantics

read_state(s)

- transition: Read data from the file initial_state associated with the string s. Use this data to create a seq of (seq of Cell). It will then return the seq of (seq of Cell). The text file will be in the form:

  symbol, symbol, symbol, symbol. symbol
  symbol, symbol, symbol, symbol. symbol
  symbol, symbol, symbol, symbol. symbol
  symbol, symbol, symbol, symbol. symbol
  symbol, symbol, symbol, symbol. symbol

  Symbol corresponds to either a -, or a +. - in this case refers to a cell that is "dead" or "unpopulated" while + refers to a cell that is "alive" or "populated." The symbols will be separated by comma's and there will be a new line character at the end of each row. Each row will be of length m, and each column of lenth m, that is they are the same size making a perfect square.

  The function will create a seq of (seq of Cell), and as it reads each line, if the symbol is a -, it will set the life to be false, and if the symbol is +, it will set the life to be +.
- output: seq of (seq of Cell)
- exception: none

View Module

Module

View

Uses

BoardT
Cell

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	In	Out	Exceptions
view_state	seq of (seq of Cell)		

Semantics

State Variables

None

State Invariant

None

Assumptions and Design Decisions

Access Routine Semantics

view_state(s)

- transition: Out data from the seq of (seq of Cell) to text file view_state.txt. The text file will be in the form:

  symbol, symbol, symbol, symbol. symbol
  symbol, symbol, symbol, symbol. symbol
  symbol, symbol, symbol, symbol. symbol
  symbol, symbol, symbol, symbol. symbol
  symbol, symbol, symbol, symbol. symbol

  Symbol corresponds to either a -, or a +. - in this case refers to a cell that is "dead" or "unpopulated" while + refers to a cell that is "alive" or "populated." The symbols will be separated by comma's and there will be a new line character at the end of each row. Each row will be of length m, and each column of length m, that is they are the same size making a perfect square.

  The function will go through each row in the seq of (seq of Cell), and if the get_life() is false, it will output a "-, " and if it is true, it will output a "+, ". If it is the last element in the row the output will be either "-\n" or "+\n". It will do this for every row in the seq of (seq of Cell).

- output: none
- exception: none

Critique of Design

To begin, the spec was designed to uphold and enforce the software engineering principles and best practises that have been learned over the term. Although there is room for improvement within the spec, care was taken when designing it as well as time constraints. Below is a more in depth analysis on six of these components.

Essentiality:
This refers to omitting unnecessary features from the spec and the implementation. As a whole, this was decently done in the spec but there are a few cases where a little more care to uphold this principle could have been taken. In each module, the methods solely exist to implement the secret of that module. However, there are two cases where this could have been improved. For one, the way the spec and implementation is done is to have a seq of (seq of Cell). This game could also have been implemented by having a seq of (seq of bool), and not needing a Cell ADT at all. To me, having a cell ADT made the most sense and I think makes certain things easier and simpler, like transitioning to the next iteration, but the program could definitely have been implemented without it. The second case is the local functions used to help the public method next. There are multiple location functions for updating the number of neighbours each Cell has in the seq of (seq of Cell). This could have been done within the next function, or in a single local function, but I felt splitting it up makes it easier to understand what is happening, and easier to implement.

Generality:
This software engineering principle is about solving a more general problem if possible than a specific one. The spec I created, while still sufficiently general is lacking in one main area. It is general in the fact that any number of iterations can be run. However, the main aspect that is affecting the specs generality is that the input has to be a perfect square, that is same number of rows and columns. This would be very easy to modify in the spec and program however due to time constraints, this was the quickest thing to do.

Minimality:
Minimality is the idea that each method only does one thing and does not do two independent services. In this spec, care was taken to ensure each method completes only its specific task. This was aided in the fact that the game being designed isn't particularly complicated. For example, in the Cell module, each method is either a constructor, getter, or a setter and does only that function. The methods in the Read and View modules exist only to read in data, or output. In the Game Board module, the public three methods also only preform one task which is to create an instance, get the next iteration, or view the

output. The local functions help the public ones, yet also uphold minimality as they are split further into even more specific tasks to make implementation and planning easier, as well as preserve the concept.

Consistency:
Consistency refers to whether the language and terminology throughout the spec is the same. In the spec I created, symbols and terminology used are the exact same throughout the entire spec. For example, Cell is known as the ADT that describes a single cell or pixel. That is used constantly throughout the spec and is not referred to by something else.

Cohesion:
Cohesion refers to the fact that components inside a module are closely related together. The spec and implementation definitely uphold this software principle because all components within a module work to perform one goal. In the case of Cell, each method is needed to be able to work with the ADT. All the methods within this module do nothing else besides that. In the other modules, the various items exclusively implement the secret for the module.

Information Hiding:
This principle for dividing up modules refers to the fact that the implementation is hidden from other modules, information secrets are hidden to clients, and there is one secret per module. This is one of the most important object oriented concepts. The spec written enforces information hiding because each module only contains one secret or does one thing. The Cell module is just an ADT for a pixel or cell, Read and View just read in the input and output it, respectively, and the Game Board controls the state of the game. The way the modules are designed, the implementation of everything is hidden to the client, and the only way the client can use the modules features is through the interface. This is done through public and private methods as outline in the MIS. This allows the implementation to change without affecting the input or output.