Assignment 4 Specifications

SFWR ENG 2AA4

April 8, 2018

This document shows the complete specification for the modules used to store the state of the game Freecell. In this specification natural numbers ($\mathbb{N}$) include zero (0). In addition, this specification assumes that the first element in a sequence is indexed at 0 (i.e 0 based indexing).

Card Types Module

Module

CardTypes

Uses

N/A

Syntax

Exported Constants

None

Exported Types

Suit = {hearts, diamonds, spades, clubs}
Value = {ace, two, three, four, five, six, seven, eight, nine, ten, jack, queen, king}

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Pile Types Module

Module

PileTypes

Uses

N/A

Syntax

Exported Constants

None

Exported Types

PileType $=\{$foundation, cell, tableau$\}$

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Card ADT Module

Template Module

CardT

Uses

CardTypes

Syntax

Exported Types

CardT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
CardT	Value, Suit	CardT	
S		Suit	
V		Value	
isOppositeColour	CardT	$\mathbb{B}$	
isOneLess	CardT	$\mathbb{B}$	

Semantics

State Variables

s: Suit
v: Value

State Invariant

None

Assumptions

The constructor CardT is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object.

Access Routine Semantics

PointT(val, st):

- transition: $v, s := val, st$
- output: $out := self$
- exception: None

S():

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

V():

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

isOppositeColour($card$):

- output:

		$out :=$
$v =$ hearts	$card.V() =$ hearts	$false$
	$card.V() =$ diamonds	$false$
	$card.V() =$ spades	$true$
	$card.V() =$ clubs	$true$
$v =$ diamonds	$card.V() =$ hearts	$false$
	$card.V() =$ diamonds	$false$
	$card.V() =$ spades	$true$
	$card.V() =$ clubs	$true$
$v =$ spades	$card.V() =$ hearts	$true$
	$card.V() =$ diamonds	$true$
	$card.V() =$ spades	$false$
	$card.V() =$ clubs	$false$
$v =$ clubs	$card.V() =$ hearts	$true$
	$card.V() =$ diamonds	$true$
	$card.V() =$ spades	$false$
	$card.V() =$ clubs	$false$

- exception: None

isOneLess($card$):

- output: $out := ((\mathrm{numVal}(v) - \mathrm{numVal}(card.\mathrm{V}())) = -1)$
- exception: None

Local Functions

numVal: Value $\rightarrow \mathbb{N}$

numVal(v) $\equiv$

$v =$ ace	0
$v =$ two	1
$v =$ three	2
$v =$ four	3
$v =$ five	4
$v =$ six	5
$v =$ seven	6
$v =$ eight	7
$v =$ nine	8
$v =$ ten	9
$v =$ jack	10
$v =$ queen	11
$v =$ king	12

Pile ADT Module

Template Module

PileT

Uses

CardT

Syntax

Exported Types

PileT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
PileT		PileT	
add	CardT		
top		CardT	
rm		CardT	
size		$\mathbb{N}$	

Semantics

State Variables

s: sequence of CardT

State Invariant

None

Assumptions

The constructor PileT is called for each object instance before any other access routine is called for that object. The constructor cannot be called on an existing object. It is also assumed that the top() and rm() routines are not called when the number of cardT's in s is 0. In addition, the output is assumed to occur before the transition in the rm() routine.

Access Routine Semantics

PileT():

- transition: None
- output: $out := self$
- exception: None

add($card$):

- transition: $s := s || < card >$
- exception: None

top():

- output: $out := s[|s| - 1]$
- exception: None

rm():

- output: $out := s[|s| - 1]$
- transition: $s := s \backslash < s[|s|-1] >$ *#That is, the sequence is the same as before except the last element is removed*
- exception: None

size():

- output: $out := |s|$
- exception: None

Freecell Game ADT

Template Module

FreecellGame

Uses

PileTypes, CardTypes, CardT

Syntax

Exported Types

FreecellGame = ?

Exported Constants

F_C_SIZE = 4
T_SIZE = 8

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
FreecellGame		FreecellGame	
newGame	seq of CardT		invalid_deck
getCard	PileType, $\mathbb{N}$	CardT	invalid_availability, invalid_index
size	PileType, $\mathbb{N}$	$\mathbb{N}$	invalid_index
moveCard	PileType, PileType, $\mathbb{N}$, $\mathbb{N}$		invalid_move
gameWon		$\mathbb{B}$	
noValidMoves		$\mathbb{B}$	

Semantics

State Variables

foundP: set of PileT
cellP: set of PileT
tabP: set of PileT

State Invariant

|foundP| = F_C_SIZE
|cellP| = F_C_SIZE
|tabP| = T_SIZE

Assumptions

- The FreecellGame() constructor and the newGame(deck) routine is called for each object instance before any other access routine is called for that object. The constructor can only be called once but the newGame(deck) routine can be called many times, as it essentially resets the game.
- Assume that the state variables, foundP, cellP, and tabP correspond to a sequence of foundation piles, cell piles, and tableau piles. Foundation piles generally correspond to the piles in the top right of a freecell game, cell piles in the top left, and tableau piles are the center playing piles. Also assume that the deck of cards used with the newGame(deck) routine includes cards that are shuffled.

Access Routine Semantics

FreecellGame():

- transition: None
- output: $out := self$
- exception: None

newGame($deck$):

- transition:

$foundP := \{PileT(), PileT(), PileT(), PileT()\}$
$cellP := \{PileT(), PileT(), PileT(), PileT()\}$
$tabP[0] := \{i : \mathbb{N} \mid i \in [0..6] : deck[i]\}$
$tabP[1] := \{i : \mathbb{N} \mid i \in [7..13] : deck[i]\}$
$tabP[2] := \{i : \mathbb{N} \mid i \in [14..20] : deck[i]\}$
$tabP[3] := \{i : \mathbb{N} \mid i \in [21..27] : deck[i]\}$
$tabP[4] := \{i : \mathbb{N} \mid i \in [28..33] : deck[i]\}$
$tabP[5] := \{i : \mathbb{N} \mid i \in [34..39] : deck[i]\}$
$tabP[6] := \{i : \mathbb{N} \mid i \in [40..45] : deck[i]\}$
$tabP[7] := \{i : \mathbb{N} \mid i \in [46..51] : deck[i]\}$

#All of these transitions occur

- exception: $exc :=$ areDuplicateCards($deck$)$|\neg(|deck| = 52) \Rightarrow$ invalid_deck

getCard($pile, i$):

- output:

	$out :=$
$pile =$ foundation	$foundP[i]$.top()
$pile =$ cell	$cellP[i]$.top()
$pile =$ tableau	$tabP[i]$.top()

- exception:

	$exc :=$
$pile =$ foundation	$\neg(i \in [0..(\text{F\_C\_SIZE} - 1)]) \Rightarrow$ invalid_index | ($foundP[i]$.size() $= 0) \Rightarrow$ invalid_availability
$pile =$ cell	$\neg(i \in [0..(\text{F\_C\_SIZE} - 1)]) \Rightarrow$ invalid_index | ($cellP[i]$.size() $= 0) \Rightarrow$ invalid_availability
$pile =$ tableau	$\neg(i \in [0..(\text{T\_SIZE} - 1)]) \Rightarrow$ invalid_index | ($tabP[i]$.size() $= 0) \Rightarrow$ invalid_availability

size($pile, i$):

- output:

	$out :=$
$pile =$ foundation	$foundP[i]$.size()
$pile =$ cell	$cellP[i]$.size()
$pile =$ tableau	$tabP[i]$.size()

- exception:

	$exc :=$
$pile =$ foundation	$(\neg(i \in [0..(\text{F\_C\_SIZE} - 1)]) \Rightarrow$ invalid_index
$pile =$ cell	$(\neg(i \in [0..(\text{F\_C\_SIZE} - 1)]) \Rightarrow$ invalid_index
$pile =$ tableau	$(\neg(i \in [0..\text{T\_SIZE} - 1)]) \Rightarrow$ invalid_index

moveCard(f, to, i, j):

- transition:

$f =$ foundation	$to =$ foundation	$foundP[i] := foundP[i]$.add($foundP[j]$.rm())
	$to =$ cell	$foundP[i] := foundP[i]$.add($cellP[j]$.rm())
	$to =$ tableau	$foundP[i] := foundP[i]$.add($tabP[j]$.rm())
$f =$ cell	$to =$ foundation	$cellP[i] := cellP[i]$.add($foundP[j]$.rm())
	$to =$ cell	$cellP[i] := cellP[i]$.add($cellP[j]$.rm())
	$to =$ tableau	$cellP[i] := cellP[i]$.add($tabP[j]$.rm())
$f =$ tableau	$to =$ foundation	$tabP[i] := tabP[i]$.add($foundP[j]$.rm())
	$to =$ cell	$tabP[i] := tabP[i]$.add($cellP[j]$.rm())
	$to =$ tableau	$tabP[i] := tabP[i]$.add($tabP[j]$.rm())

f = foundation	to = foundation	$\neg$canMoveFtoF$(i, j) \Rightarrow$ invalid_move
	to = cell	$\neg$canMoveFtoC$(i, j) \Rightarrow$ invalid_move
	to = tableau	$\neg$canMoveFtoT$(i, j) \Rightarrow$ invalid_move
f = cell	to = foundation	$\neg$canMoveCtoF$(i, j) \Rightarrow$ invalid_move
	to = cell	$\neg$canMoveCtoC$(i, j) \Rightarrow$ invalid_move
	to = tableau	$\neg$canMoveCtoT$(i, j) \Rightarrow$ invalid_move
f = tableau	to = foundation	$\neg$canMoveTtoF$(i, j) \Rightarrow$ invalid_move
	to = cell	$\neg$canMoveTtoC$(i, j) \Rightarrow$ invalid_move
	to = tableau	$\neg$canMoveTtoT$(i, j) \Rightarrow$ invalid_move

- exception: (table above)

gameWon():

- output: $out := \forall(i : \mathbb{N} | i \in [0..(\text{F\_C\_SIZE} - 1)] : foundP[i].size() = 13)$
- exception: None

noValidMoves():

- output: $out := \forall(i, j : \mathbb{N} | i \in [0..(\text{F\_C\_SIZE}-1)] | j \in [0..(\text{T\_SIZE})] : \neg\text{canMoveFtoF}(i, i) \wedge \neg\text{canMoveFtoC}(i, i) \wedge \neg\text{canMoveFtoT}(i, j) \wedge \neg\text{canMoveCtoF}(i, i) \wedge \neg\text{canMoveCtoC}(i, i) \wedge \neg\text{canMoveCtoT}(i, j) \wedge \neg\text{canMoveTtoF}(j, i) \wedge \neg\text{canMoveTtoC}(j, i) \wedge \neg\text{canMoveTtoT}(j, j))$
- exception: None

Local Functions

areDuplicateCards: seq of CardT $\rightarrow \mathbb{B}$
areDuplicateCards$(deck) \equiv \neg(\forall(i, j : \mathbb{N} | i \in [0..(|deck| - 2)] | j \in [i + 1..(|deck| - 1)] : (deck[i].S() \neq deck[j].S()) \wedge (deck[i].V() \neq deck[j].V())))$

canMoveFtoF: N $\times$ N $\rightarrow \mathbb{B}$
canMoveFtoF$(i, j) \equiv (i \in [0..(\text{F_C_SIZE}-1)]) \wedge (j \in [0..(\text{F_C_SIZE}-1)]) \wedge (foundP[i].\text{size}() \neq 0) \wedge (foundP[j].\text{size}() = 0) \wedge (foundP[i].\text{top}().\text{V}() = ace)$

canMoveFtoC: N $\times$ N $\rightarrow \mathbb{B}$
canMoveFtoC$(i, j) \equiv (i \in [0..(\text{F_C_SIZE}-1)]) \wedge (j \in [0..(\text{F_C_SIZE}-1)]) \wedge (foundP[i].\text{size}() \neq 0) \wedge (cellP[j].\text{size}() = 0)$

canMoveFtoT: N $\times$ N $\rightarrow \mathbb{B}$

canMoveFtoT$(i,j) \equiv (i \in [0..(\text{F_C_SIZE}-1)]) \wedge (j \in [0..(\text{T_SIZE}-1)]) \wedge (foundP[i].\text{size}() \neq 0) \wedge ((tabP[j].\text{size}() = 0) \vee foundP[i].\text{top}().\text{isOppositeColour}(tabP[j].\text{top}()) \wedge (foundP[i].\text{top}().\text{isOneLess}(tabP[j].\text{top}())))$

canMoveCtoF: $\text{N} \times \text{N} \rightarrow \mathbb{B}$
canMoveCtoF$(i,j) \equiv (i \in [0..(\text{F_C_SIZE}-1)]) \wedge (j \in [0..(\text{F_C_SIZE}-1)]) \wedge (cellP[i].\text{size}() \neq 0) \wedge ((foundP[j].\text{size}() = 0) \vee cellP[i].\text{top}().\text{isOppositeColour}(foundP[j].\text{top}()) \wedge (cellP[i].\text{top}().\text{isOneLess}(foundP[j].\text{top}())))$

canMoveCtoC: $\text{N} \times \text{N} \rightarrow \mathbb{B}$
canMoveCtoC$(i,j) \equiv (i \in [0..(\text{F_C_SIZE}-1)]) \wedge (j \in [0..(\text{F_C_SIZE}-1)]) \wedge (cellP[i].\text{size}() \neq 0) \wedge (cellP[j].\text{size}() = 0)$

canMoveCtoT: $\text{N} \times \text{N} \rightarrow \mathbb{B}$
canMoveTtoT$(i,j) \equiv (i \in [0..(\text{F_C_SIZE}-1)]) \wedge (j \in [0..(\text{T_SIZE}-1)]) \wedge (cellP[i].\text{size}() \neq 0) \wedge ((tabP[j].\text{size}() = 0) \vee cellP[i].\text{top}().\text{isOppositeColour}(tabP[j].\text{top}()) \wedge (cellP[i].\text{top}().\text{isOneLess}(tabP[j].\text{top}())))$

canMoveTtoF: $\text{N} \times \text{N} \rightarrow \mathbb{B}$
canMoveTtoF$(i,j) \equiv (i \in [0..(\text{T_SIZE}-1)]) \wedge (j \in [0..(\text{F_C_SIZE}-1)]) \wedge (tabP[i].\text{size}() \neq 0) \wedge ((foundP[j].\text{size}() = 0) \vee \neg(tabP[i].\text{top}().\text{isOppositeColour}(foundP[j].\text{top}())) \wedge (foundP[j].\text{top}().\text{isOneLess}(tabP[i].\text{top}())))$

canMoveTtoC: $\text{N} \times \text{N} \rightarrow \mathbb{B}$
canMoveTtoC$(i,j) \equiv (i \in [0..(\text{T_SIZE}-1)]) \wedge (j \in [0..(\text{F_C_SIZE}-1)]) \wedge (tabP[i].\text{size}() \neq 0) \wedge (cellP[j].\text{size}() = 0)$

canMoveTtoT: $\text{N} \times \text{N} \rightarrow \mathbb{B}$
canMoveTtoT$(i,j) \equiv (i \in [0..(\text{T_SIZE}-1)]) \wedge (j \in [0..(\text{T_SIZE}-1)]) \wedge (tabP[i].\text{size}() \neq 0) \wedge ((tabP[j].\text{size}() = 0) \vee tabP[i].\text{top}().\text{isOppositeColour}(tabP[j].\text{top}()) \wedge (tabP[i].\text{top}().\text{isOneLess}(tabP[j].\text{top}())))$