Assignment 3, Part 1, Specification

SFWR ENG 2AA4

March 12, 2018

The purpose of this software design exercise is to design and implement a portion of the specification for a Geographic Information System (GIS). This document shows the complete specification, which will be the basis for your implementation and testing. In this specification natural numbers ($\mathbb{N}$) include zero (0).

[The parts that you need to fill in are marked by comments, like this one. In several of the modules local functions are specified. You can use these local functions to complete the missing specifications. —SS]

Map Types Module

Module

MapTypes

Uses

N/A

Syntax

Exported Constants

None

Exported Types

CompassT = {N, S, E, W}
LanduseT = {Recreational, Transport, Agricultural, Residential, Commercial}
RotateT = {CW, CCW}

Exported Access Programs

None

Semantics

State Variables

None

State Invariant

None

Point ADT Module

Template Module

PointT

Uses

N/A

Syntax

Exported Types

PointT = ? [SS —SS]

Exported Access Programs

Routine name	In	Out	Exceptions
PointT	$\mathbb{Z}$, $\mathbb{Z}$	PointT	
x		$\mathbb{Z}$	
y		$\mathbb{Z}$	
translate	$\mathbb{Z}$, $\mathbb{Z}$	PointT	

Semantics

State Variables

[SS —SS]
xc: $\mathbb{Z}$
yc: $\mathbb{Z}$

State Invariant

None

Assumptions

The constructor PointT 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(x, y):

- transition: xc, yc := x, y [SS —SS]
- output: *out* := *self* [SS —SS]
- exception: None

x():

- output: *out* := xc
- exception: None

y():

- output: *out* := yc [SS —SS]
- exception: None

translate(Δx, Δy):

- output: PointT(xc+Δx, yc+ Δy) [SS —SS]
- exception: None

Line ADT Module

Template Module

LineT

Uses

PointT, MapTypes [SS —SS]

Syntax

Exported Types

LineT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
LineT	PointT, CompassT, $\mathbb{N}$	LineT	invalid_argument
strt		PointT	
end		PointT	
orient		CompassT	
len		$\mathbb{Z}$	
flip		LineT	
rotate	RotateT	LineT	
translate	$\mathbb{Z}$, $\mathbb{Z}$	LineT	

Semantics

State Variables

s: PointT
o: CompassT
L: $\mathbb{N}$

State Invariant

None

Assumptions

The constructor LineT 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

LineT($st, ornt, l$):

- transition: $s, o, L := st, ornt, l$
- output: $out := self$
- exception: $exc := ((l = 0) \Rightarrow \text{Invalid\_argument})$

strt():

- output: $out :=$ PointT($s.x, s.y$)
- exception: None

end():

- output: PointT(s.x+Lcos(o), s.y+Lsin(o)) [SS —SS]
- exception: None
  I could have made a table but I just wanted to make it as simple as possible, so the starting point of the axiom is 0 so o = 0 and then N means o = 90 and keeps going.

orient():

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

len():

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

flip():

- output: out:= LineT($s, o + 180, L$) [SS —SS]
- exception: None

rotate(r):

- output:

		out :=
$r =$ CW	$o =$ N	270 [? —SS]
	$o =$ S	90 [? —SS]
	$o =$ W	180 [? —SS]
	$o =$ E	0 [? —SS]
$r =$ CCW	$o =$ N	90 [? —SS]
	$o =$ S	270 [? —SS]
	$o =$ W	180 [? —SS]
	$o =$ E	0 [? —SS]

- exception: None

translate(Δx, Δy):

- output: LineT(s.translate(Δx, Δy),o, L) [? —SS]
- exception: None

Path ADT Module

Template Module

PathT

Uses

PointT, LineT, MapTypes

Syntax

Exported Types

PathT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
PathT	PointT, CompassT, $\mathbb{N}$	PathT	
append	CompassT, $\mathbb{N}$		invalid_argument
strt		PointT	
end		PointT	
line	$\mathbb{N}$	LineT	outside_bounds
size		$\mathbb{N}$	
len		$\mathbb{N}$	
translate	$\mathbb{Z}$, $\mathbb{Z}$	LineT	

Semantics

State Variables

s: sequence of LineT

State Invariant

None

Assumptions

- The constructor PathT 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

PathT($st, ornt, l$):

- transition: $s :=< LineT(st, ornt, l) >$ [What is the spec to add the first element to the sequence of LineT? —SS]
- output: $out := self$
- exception: None

append($ornt, l$):

- transition: $s := s|| < LineT(adjPt(ornt), ornt, l) >$
  [What is the missing specification? The appended line starts at a point adjacent to the end point of the previous line in the direction *ornt*. The lines are not allowed to overlap. —SS]
- exception: $exc := (\forall i : PointT | i \in pointsInLine(LineT(adjPt(ornt, l))) \bullet \forall j : LineT | j \in s \bullet \forall x : PointT | x \in pointsInLine(LineT(j)) \bullet i = x \Rightarrow$ Invalid_argument)
  [What is the specification for the exception? An exception should be generated if the introduced line overlaps with any of the previous points in the existing path. —SS]

strt():

- output: $out := s[0].strt$ [What is the missing spec? —SS]
- exception: None

end():

- output: out:= $out := s[|s| - 1].end$ [What is the missing spec? —SS]
- exception: None

line(i):

- output: $out := s[i]$
  [Returns the ith line in the sequence. What is the missing spec? —SS]
- exception:$(\neg(0 \leq i < |s|) \Rightarrow \text{InvalidIndex})$
  [Generate the exception if the index is not in the sequence. —SS]

size:

- output: $out := |s|$
  [Output the number of lines in the path. —SS]
- exception: None

len:

- output: $+(\forall i \in PathT : |i.pointsInLine|)$[Output the number of points on the line. —SS]
- exception: None

translate(Δx, Δy):

- output: Create a new PathT object with state variable s' such that:

$$\forall(i : \mathbb{N} | i \in [0..|s| - 1] : s'[i] = s[i].\text{translate}(\Delta x, \Delta y))$$

- exception: None

Local Functions

pointsInLine: LineT $\rightarrow$ (set of PointT)

pointsInLine (l)

$$\equiv \{i : \mathbb{N} | i \in [0..(l.\text{len} - 1)] : l.\text{strt} := l.\text{strt.translate.}(i * cos(l.o), i * sin(l.o))\}$$

([Complete the spec. —SS]
adjPt: CompassT $\rightarrow$ PointT
adjPt($ornt$) $\equiv$

$ornt = \text{N}$	$s[\|s\| - 1].\text{end.translate.}(0, 1)$[? − − − SS]
$ornt = \text{S}$	$s[\|s\| - 1].\text{end.translate.}(0, -1)$[? − − − SS]
$ornt = \text{W}$	$s[\|s\| - 1].\text{end.translate.}(-1, 0)$[? − − − SS]
$ornt = \text{E}$	$s[\|s\| - 1].\text{end.translate.}(1, 0)$[? − − − SS]

Generic Seq2D Module

Generic Template Module

Seq2D(T)

Uses

N/A

Syntax

Exported Types

Seq2D(T) = ?

Exported Constants

None

Exported Access Programs

Routine name	In	Out	Exceptions
Seq2D	seq of (seq of T), $\mathbb{R}$	Seq2D	invalid_argument
set	PointT, T		outside_bounds
get	PointT	T	outside_bounds
getNumRow		$\mathbb{N}$	
getNumCol		$\mathbb{N}$	
getScale		$\mathbb{R}$	
count	T	$\mathbb{N}$	
count	LineT, T	$\mathbb{N}$	invalid_argument
count	PathT, T	$\mathbb{N}$	invalid_argument
length	PathT	$\mathbb{R}$	invalid_argument
connected	PointT, PointT	$\mathbb{B}$	invalid_argument

Semantics

State Variables

s: seq of (seq of T)
scale: $\mathbb{R}$

nRow: $\mathbb{N}$
nCol: $\mathbb{N}$

State Invariant

None

Assumptions

- The Seq2D(T) constructor is called for each object instance before any other access routine is called for that object. The constructor can only be called once.
- Assume that the input to the constructor is a sequence of rows, where each row is a sequence of elements of type T. The number of columns (number of elements) in each row is assumed to be equal. That is each row of the grid has the same number of entries. $s[i][j]$ means the ith row and the jth column. The 0th row is at the bottom of the map and the 0th column is at the leftmost side of the map.

Access Routine Semantics

Seq2D(S, scl):

- transition: $s, scale, nRow, nCol := S, scl, |S|, |S[0]|$
  [Fill in the transition. —SS]
- output: $out := self$
- exception: $exc := ((scl < 0) \Rightarrow invalid\_argument)|(|S| = 0 \Rightarrow invalid\_argument)|((|S[0]| = 0) \Rightarrow invalid\_argument)|([i : \mathbb{N}|i \in [0..(|s|-2)]](|S[i]| \neq |S[i+1]|) \Rightarrow invalid\_argument)$
  [Fill in the exception. One should be generated if the scale is less than zero, or the input sequence is empty, or the number of columns is zero in the first row, or the number of columns in any row is different from the number of columns in the first row. —SS]

set(p, v):

- transition: $s[p.y][p.x] := v$
  [? —SS]
- exception: $exc := (\neg(validPoint(p)) \Rightarrow outside\_bounds)$
  [Generate an exception if the point lies outside of the map. —SS]

get(p):

- output: $out := s[p.x][p.y]$
  [? —SS]
- exception: $exc := (\neg(validPoint(p)) \Rightarrow outside\_bounds)$
  [Generate an exception if the point lies outside of the map. —SS]

getNumRow():

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

getNumCol():

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

getScale():

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

count(t: T):

- output: $out := +((i \in [0..nRow - 1]) \wedge (\forall j \in [0..nCol - 1])|s[i][j] = t : 1)$
  [Count the number of times the value t occurs in the 2D sequence. —SS]
- exception: None

count(l: LineT, t: T):

- output: $out := +((i : PointT)|i \in pointsInLine(l)|s[i.x][i.y] = t : 1)$
  [Count the number of times the value t occurs in the line l. —SS]
- exception: $exc := ((\neg(validLine(l)) \Rightarrow invalid\_argument)$
  [Exception if any point on the line lies off of the 2D sequence (map) —SS]

count(pth: PathT, t: T):

- output: $out := +((i : LineT)|i \in pth|s.count(i) = t \Rightarrow: 1)$
  [Count the number of times the value t occurs in the path pth. —SS]
- exception: $exc := (\neg(validPath(pth)) \Rightarrow invalid\_argument)$
  [Exception if any point on the path lies off of the 2D sequence (map) —SS]

length(pth: PathT):

- output: $out := scale * pth.len$[Use the scale to find the length of the path. —SS]
- exception: $exc := (\neg(validPath(pth)) \Rightarrow invalid\_argument)$
  [Exception if any point on the path lies off of the 2D sequence (map) —SS]

connected(p_1: PointT, p_2: PointT):

- output: $out := (\exists x : PathT, i : \mathbb{N} | validPath(x) \wedge i \in [0..|pointsInPath(x)| - 2] : p_1, p_2 \in pointsInPath(x) \wedge s.get(pointsInPath[i]) = s.get(pointsInPath[i+1])$
  [Return true if a path exists between p_1 and p_2 with all of the points on the path being of the same value. —SS]
- exception: $(\neg validPoint(p_1) \vee \neg validPoint(p_2) \Rightarrow invalid\_argument)$[Return an exception if either of the input points is not valid. —SS]

Local Functions

validRow: $\mathbb{N} \to \mathbb{B}$
validRow(n) $\equiv 0 \leq n < |S|$
[returns true if the given natural number is a valid row number. —SS]

validCol: $\mathbb{N} \to \mathbb{B}$
validCol(a) $\equiv 0 \leq a < |S[0]|$
[returns true if the given natural number is a valid column number. —SS]

validPoint: PointT $\to \mathbb{B}$
validPoint(p)$\equiv (S[0][0] \leq p.x < S[0][|S|] \wedge S[0][0] \leq p.y < S[|S|][0])$
[Returns true if the given point lies within the boundaries of the map. —SS]

validLine: LineT $\to \mathbb{B}$
validLine (l) $\equiv (i : PointT | \forall i \in pointsInLine(l) \bullet validPoint(i))$
[Returns true if all of the points for the given line lie within the boundaries of the map. —SS]

validPath: PathT $\to \mathbb{B}$
validPath(p) $\equiv (i : PointT | \forall i \in pointsInPath(p) \bullet validPoint(i))$
[Returns true if all of the points for the given path lie within the boundaries of the map. —SS]

pointsInLine: LineT $\rightarrow$ (set of PointT)
pointsInLine $(l) \equiv \{i : \mathbb{N} | i \in [0..(l.\text{len} - 1)] : l.\text{strt} := l.\text{strt.translate}.(i * cos(l.o), i * sin(l.o))\}$
[The same local function as given in the Path module. —SS]
pointsInPath: PathT $\rightarrow$ (set of PointT)
pointsInPath$(p) \equiv (l : LineT | \forall l \in [0..|p| - 1] \wedge j \in [0..|p[0]| - 1] \bullet pointsInLine(l)[j])$
[Return the set of points that make up the input path. —SS]

LanduseMap Module

Template Module

LanduseMapT is Seq2D(LanduseT)

DEM Module

Template Module

DEMT is Seq2D($\mathbb{Z}$)

Critique of Design

Write a critique of the interface for the modules in this project. Is there anything missing? Is there anything you would consider changing? Why?
I didn't really find anything missing however, I feel like if we add exception to flip function would be better. What if when we flip the line the new path doesn't have the same type of the previous path. Therefore I think it makes sense if we use an exception call in LineADT to check if that case happens it display to the user flipping function cannot flip the path. The two thing that I would consider changing are compass and local functions. Basically for the map function the way compass has been defined is not in a proper way, we can define the compass like N which is north equal to 90 degree and so on for the rest of the directions. This would make the programmers understand it better and it would help them to get the desire result faster. Besides it would be less confusion. And I believe local functions should have been written in a way that they use only all the inputs and based on the inputs they do their jobs, however based on what can be seen in the local function, it just displays part of the requirement, and the programmer must to make an assumption for instant using S(sequence of sequence of T).