Constants Module

Module

Constants

Uses

N/A

Syntax

Exported Constants

MAX_X = 180 *//dimension in the x-direction of the problem area*
MAX_Y = 160 *//dimension in the y-direction of the problem area*
TOLERANCE = 5 *//space allowance around obstacles*
VELOCITY_LINEAR = 15 *//speed of the robot when driving straight*
VELOCITY_ANGULAR = 30 *//speed of the robot when turing rad*

Exported Access Programs

none

Semantics

State Variables

none

State Invariant

none

Point ADT Module

Template Module

PointT

Uses

Constants

Syntax

Exported Types

PointT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
PointT	real, real	PointT	InvalidPointException
xcrd		real	
ycrd		real	
dist	PointT	real	

Semantics

State Variables

xc: real
yc: real

State Invariant

none

Assumptions

The constructor PointT is called for each abstract object 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: $xc, yc := x, y$
- output: $out := self$
- exception $exc := ((\neg(0 \leq x \leq \text{Contants.MAX\_X}) \vee \neg(0 \leq y \leq \text{Constants.MAX\_Y})) \Rightarrow \text{InvalidPointException})$

xcrd():

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

ycrd():

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

dist(p):

- output: $out := \sqrt{(self.xc - p.xc)^2 + (self.yc - p.yc)^2}$
- exception: none

Region Module

Template Module

RegionT

Uses

PointT, Constants

Syntax

Exported Types

RegionT = ?

Exported Access Programs

Routine name	In	Out	Exceptions
RegionT	PointT, real, real	RegionT	InvalidRegionException
pointInRegion	PointT	boolean	

Semantics

State Variables

lower_left: PointT *//coordinates of the lower left corner of the region*
width: real *//width of the rectangular region*
height: real *//height of the rectangular region*

State Invariant

None

Assumptions

The RegionT constructor is called for each abstract object before any other access routine is called for that object. The constructor can only be called once.

Access Routine Semantics

RegionT(p, w, h):

- transition: $lower\_left, width, height := p, w, h$
- output: $out := self$
- exception: $exc := \exists(pt : \text{PointT} | \text{insideRegion}(pt, self.lower\_left, self.width, self.height) : \neg\text{insideRegoin}(pt, \text{PointT}(0,0), \text{Constants.MAX\_X}, \text{Constants.MAX\_Y}) \Rightarrow \text{InvalidRegionException}$

pointInRegion(p):

- output: $out := \exists(pt : \text{PointT} | \text{insideRegoin}(pt, self.lower\_left, self.width, self.height) : pt.\text{dist}(p) \leq \text{Constants.TOLERANCE})$
- exception: none

Local Functions

insideRegion: PointT $\times$ PointT $\times$ real $\times$ real $\rightarrow$ boolean
insideRegion($p, preg, w, h$) $\equiv$ $preg.\text{xcrd}() \leq p.\text{xcrd}() \leq preg.\text{xcrd}() + w \wedge$
$preg.\text{ycrd}() \leq p.\text{ycrd}() \leq preg.\text{ycrd}() + h$

Generic List Module

Generic Template Module

GenericList(T)

Uses

N/A

Syntax

Exported Types

GenericList(T) = ?

Exported Constants

MAX_SIZE = 100

Exported Access Programs

Routine name	In	Out	Exceptions
GenericList		GenericList	
add	integer, T		FullSequenceException, InvalidPositionException
del	integer		InvalidPositionException
setval	integer, T		InvalidPositionException
getval	integer	T	InvalidPositionException
size		integer	

Semantics

State Variables

s: sequence of T

State Invariant

$|s| \leq$ MAX_SIZE

Assumptions

The GenericList() constructor is called for each abstract object before any other access routine is called for that object. The constructor can only be called once.

Access Routine Semantics

GenericList():

- transition: $self.s := <>$
- output: $out := self$
- exception: none

add(i, p):

- transition: $s := s[0..i-1] || < p > || s[i..|s|-1]$
- exception: $exc := (|s| = \text{MAX\_SIZE} \Rightarrow \text{FullSequenceException} \mid i \notin [0..|s|] \Rightarrow \text{InvalidPositionException})$

del(i):

- transition: $s := s[0..i-1] || s[i+1..|s|-1]$
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{InvalidPositionException})$

setval(i, p):

- transition: $s[i] := p$
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{InvalidPositionException})$

getval(i):

- output: $out := s[i]$
- exception: $exc := (i \notin [0..|s|-1] \Rightarrow \text{InvalidPositionException})$

size():

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

Path Module

Template Module

PathT is GenericList(PointT)

Obstacles Module

Template Module

Obstacles is GenericList(RegionT)

Destinations Module

Template Module

Destinations is GenericList(RegionT)

SafeZone Module

Template Module

SafeZone extends GenericList(RegionT)

Exported Constants

MAX_SIZE = 1

Map Module

Module

Map

Uses

Obstacles, Destinations, SafeZone

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
init	Obstacles, Destinations, SafeZone		
get_obstacles		Obstacles	
get_destinations		Destinations	
get_safeZone		SafeZone	

Semantics

State Variables

obstacles : Obstacles
destinations : Destinations
safeZone : SafeZone

State Invariant

none

Assumptions

The access routine init() is called for the abstract object before any other access routine is called. If the map is changed, init() can be called again to change the map.

Access Routine Semantics

init(o, d, sz):

- transition: $obstacles, destinations, safeZone := o, d, sz$
- exception: none

get_obstacles():

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

get_destinations():

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

get_safeZone():

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

Path Calculation Module

Module

PathCalculation

Uses

Constants, PointT, RegionT, PathT, Obstacles, Destinations, SafeZone, Map

Syntax

Exported Access Programs

Routine name	In	Out	Exceptions
is_validSegment	PointT, PointT	boolean	
is_validPath	PathT	boolean	
is_shortestPath	PathT	boolean	
totalDistance	PathT	real	
totalTurns	PathT	integer	
estimatedTime	PathT	real	

Semantics

is_validSegment(p, pt):

- output:
  $out := \forall(i, t : \mathbb{N} | i \in [0..\text{Map.get\_obstacles.size}() - 1] \land 0 \leq t \leq 1 : \text{Map.get\_obstacles.getval}(i).\text{pointInRegion}(\text{paraLinePt}(p, pt, t))$
- exception: none

is_validPath(p):

- output: $out := \text{isSafe}(p) \land \text{isPass}(p) \land \text{notObst}(p)$
- exception: none

is_shortestPath(p):

- output: $out := \forall(pa : \text{PathT} | \text{is\_validPath}(pa) : p.\text{size}() \leq pa.\text{size}())$

- exception: none

totalDistance(p):

- output: $out := +(i, j : \mathbb{N} | i \in [0..p.\text{size}() - 2] \wedge j \in [1..p.\text{size}() - 1] : p.\text{getval}(i).\text{dist}(p.\text{getval}(j))$
- exception: none

totalTurns(p):

- output: $out := +(i : \mathbb{N} | i \in [0..p.\text{size}()-3] \wedge \text{changeOrien}(p.\text{getval}(i), p.\text{getval}(i+2) : 1)$
- exception: none

estimatedTime(p):

- output: $out :=$ totalDistance(p) $*$ Constants.VELOCITY_LINEAR + totalTurns(p) $*$ Constants.VELOCITY_ANGULAR
- exception: none

Local Functions

paraLinePt: PointT $\times$ PointT $\times$ real $\rightarrow$ PointT
paraLinePt$(p, pt, t) \equiv$ PointT$((1-t)*p.\text{xcrd}()+t*pt.\text{xcrd}(), (1-t)*p.\text{ycrd}()+t*pt.\text{ycrd}())$

isSafe: PathT $\rightarrow$ boolean
isSafe$(p) \equiv$ Map.get_safeZone().getval(0).pointInRegion(p.getval(0)) $\wedge$ Map.get_safeZone().getval(0).pointInRegion(p.getval(p.size()-1))

isPass: PathT $\rightarrow$ boolean
isPass$(p) \equiv \forall(i : \mathbb{N} | i \in [0..\text{Map.get_destinations}().\text{size}() - 1] : \exists(j : \mathbb{N} | j \in [0..p.\text{size}()-1] : \text{Map.get_destinations}().\text{getval}(i).\text{pointInRegion}(p.\text{getval}(j))))$

notObst: PathT $\rightarrow$ boolean
notObst$(p) \equiv \forall(i, j : \mathbb{N} | i \in [0..\text{Map.get_obstacles}().\text{size}() - 1] \wedge j \in [0..p.\text{size}()-1] : \neg \text{Map.get_obstacles}().\text{getval}(i).\text{pointInRegion}(p.\text{getval}(j)))$

changeOrien: PointT $\times$ PointT $\rightarrow$ boolean
changeOrien$(p, pt) \equiv p$.xcrd() $\neq pt$.xcrd() $\wedge$ p.ycrd() $\neq pt$.ycrd()

Crtique of the interface

PathT specification

Interface could use a better representation of the path instead of using a genericlist of points. To avoid ambiguous, the interface could use line segment to define path instead of points. There are two problems when we use a set of points to define a path: 1) the way we connect each points could varies. 2) the order of the point could be considered as an important factor when we are trying to connect a set of points, but this is not stated explicitly in the module.

SafeZone extends GenericList

Another wired thing I found with this modules is SafeZone module. SafeZone module inheritence from GenericList¡RegionT¿ and then define MAX_SIZE to be 1. After I finished the pathCalculation module semantics, I found that the module will be more understandable if we just simply create an instant object using ADT RegionT, and then call the object SafeZone. There is no need to create one more list with only one element.

Path calculation module based on Map

Path Calculation module could be considered as useless if we dont have a valid map obejcts defined. Without a map obejct as backup, this module could not run at all. I think this situation should be stated in the module, by exception or assumptions.

For i $\in [0..|p|-1]$ compare $p[i]$ and $p[i+2]$

Suppose the order of the points in the path set is the order of connecting thoese points, $p[i]$ to $p[i+2]$ connects three points in a line, which is the

least amount to tell that there is a change of orientation. So inside the specification, I compare these two elements and iterate through the end of the list. The comparison method in my specification is just simply compare if these two coordiates have the at least one same x or y coordinates. If they do not have any common x or y coordinats, indicates there is a change of orientation. On the other hand, my specification is based on the fact that each orientation only rotate 90 degrees.

If orientations could rotate any degrees between 0 and 360

In this case, we must come up with an equations. Suppose x,x2 is the point we need to compare. But in this case, the rotation is not 90 degrees. In order to find the degree of rotation, we need to find cos(Θ) which is adjacent over hypotenuse lines. Adjacent line is indicated by point x's x-coordinates. Hypotenuse line could be represented by the distance from origin to x2. In math equation, we could write:

$$\cos(\Theta) = \frac{x.xcrd()}{x2.dist(PointT(0,0))}$$