Introduction to Cognitive Robotics

David Vernon

www.vernon.eu/cognitive_robotics

Lecture 28

www.vernon.eu/cognitive_robotics/CR_40.pdf

The CRAM Cognitive Architecture: Cognitive Robot Abstract Machine

1. Overview of CRAM
2. The main tools: Lisp, Emacs, CRAM Language, ROS
3. CRAM Beginner Tutorials with Turtlesim
4. A pick-and-place CRAM plan with a simulation of the PR2 robot

Lecture 28

www.vernon.eu/cognitive_robotics/CR_40.pdf

The CRAM Cognitive Architecture: Cognitive Robot Abstract Machine

1. Overview of CRAM
2. The main tools: Lisp, Emacs, CRAM Language, ROS
3. CRAM Beginner Tutorials with Turtlesim
   - Creating a CRAM package
   - Controlling Turtlesim from CRAM
   - Implementing simple plans to move a turtle
   - Using Prolog for reasoning
   - Creating motion designators for TurtleSim
   - Creating process modules
   - Automatically choosing a process module for a motion
   - Using location designators with TurtleSim
   - Writing high-level plans for TurtleSim
   - Implementing failure handling for TurtleSim

The CRAM Beginner Tutorials

Based on CRAM tutorials
http://cram-system.org/tutorials

Using Location Designators with TurtleSim

Based on Using location designators with the TurtleSim
http://cram-system.org/tutorials/beginner/location_designators_2

Using Location Designators with TurtleSim

Location Designators: An Overview

- Location designators provide a way to describe a location in symbolic terms …
- … and have the actual coordinates determined later, when needed at run time
- Location designators are resolved in a different manner compared to motion designators
  - Motion designators are resolved by inference (reasoning) with Prolog using a set of facts and rules
  - Location designators make use of a pair of different types of functions

Using Location Designators with TurtleSim

Location Designators: An Overview

These functions are

1. location-generator

   This creates a lazy list of candidate poses

2. location-validator

   This checks that a candidate pose is actually valid
   i.e. that it is feasible for the robot, e.g. not in collision with objects in the environment

Using Location Designators with TurtleSim

Location Designators: An Overview

- Both generator and validator functions need to be written by the programmer for their specific application
- They must be registered as generators or validators (as appropriate for given designator constraints
- It is also possible to define and register several generators and validators for the same designator
- Having more generators means you have to check more candidates
- Different validators can collaborate to check a candidate

Using Location Designators with TurtleSim

Location Designators: An Overview

Several return values are possible for validators:

- :accept means the candidate is accepted by this validator
- :unknown means this validator cannot decide and leaves the decision to the others
- :maybe-reject means the candidate will be rejected unless some other validator explicitly accepts it
- :reject immediately invalidates the candidate, no further processing needed

Using Location Designators with TurtleSim

Location Designators: An Overview

Therefore, a candidate will be accepted if

No validator rejected it
AND
((at least one validator returned :accept)
     OR
     (all validators returned :unknown)   )

Using Location Designators with TurtleSim

Writing a location designator generator function

We will the code in `location-designators.lisp`

Using Location Designators with TurtleSim

Writing a location designator generator function

As before, when developing new code, we need to

- (Update the dependencies in package.xml) ← We don't need to do this as there are no new packages being used
- Update the dependencies in cram-my-beginner-tutorial.asd ← We need to do this because we are going to put the new code in a separate file
- (Update the dependencies in package.lisp) ← We don't need to do this as there are no new packages being used
- Add the new code to location-designators.lisp ← We will place the new code is a separate Lisp file
- Test the code
  - Run the ROS master
  - Run the Lisp REPL, loading the new program, creating a ROS node
  - Run turtlesim
  - Run turtlesim_teleop
  - Call the new functions

Using Location Designators with TurtleSim

Writing a location designator generator function

Update the ASDF dependencies

Make sure you are in the cram_my_beginner_tutorial sub-directory

```
~$ cd ~/workspace/ros/src/cram_my_beginner_tutorial
~/workspace/ros/src/cram_my_beginner_tutorial$
```

Using Location Designators with TurtleSim

Writing a location designator generator function

Update the ASDF dependencies

Edit cram-my-beginner-tutorial.asd

```
~/workspace/ros/src/cram_my_beginner_tutorial$ emacs cram-my-beginner-tutorial.asd
```

Using Location Designators with TurtleSim

```
(defsystem cram-my-beginner-tutorial
  :depends-on (roslisp cram-language
               turtlesim-msg turtlesim-srv
               cl-transforms geometry_msgs-msg
               cram-designators cram-prolog
               cram-process-modules cram-language-designator-support
               cram-executive)
  :components
  ((:module "src"
            :components
            ((:file "package")
             (:file "control-turtlesim" :depends-on ("package"))
             (:file "simple-plans" :depends-on ("package" "control-turtlesim"))
             (:file "motion-designators" :depends-on ("package"))
             (:file "location-designators" :depends-on ("package"))
             (:file "process-modules" :depends-on ("package"
                                                   "control-turtlesim"
                                                   "simple-plans"
                                                   "motion-designators"))
             (:file "selecting-process-modules" :depends-on ("package"
                                                             "motion-designators"
                                                             "process-modules"))))))
```

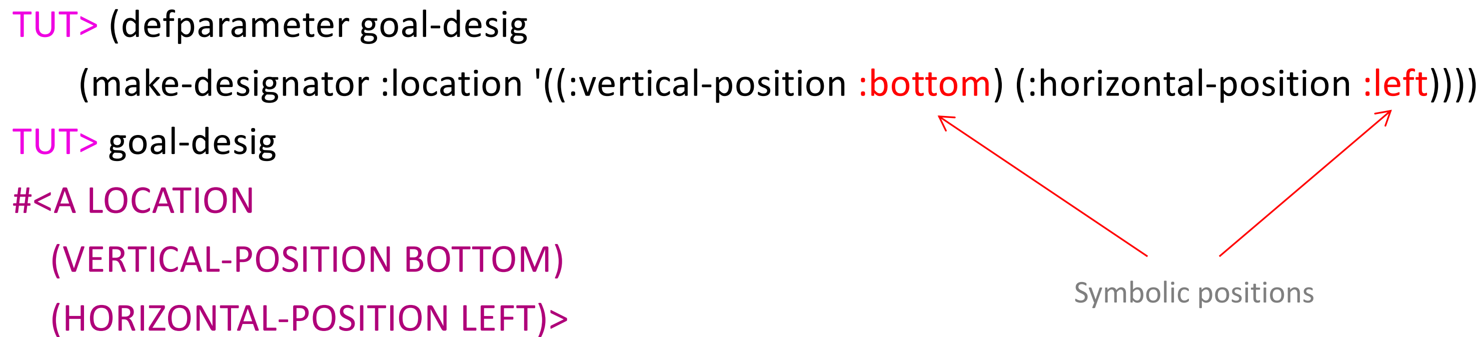
Add this line

Using Location Designators with TurtleSim

Writing a location designator generator function

We would like to specify locations for the turtle to go using spatial relations, e.g.:

```
TUT> (defparameter goal-desig
       (make-designator :location '((:vertical-position :bottom) (:horizontal-position :left))))
TUT> goal-desig
#<A LOCATION
   (VERTICAL-POSITION BOTTOM)
   (HORIZONTAL-POSITION LEFT)>
```



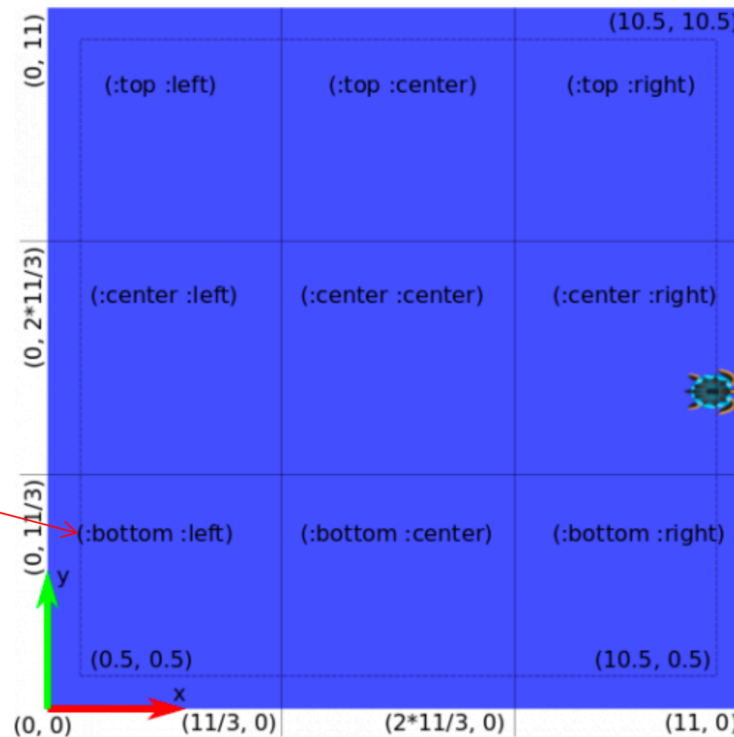
Using Location Designators with TurtleSim

Writing a location designator generator function

Symbolic positions, e.g. `:bottom :left`

We want the designator to resolve by producing coordinates in whichever of the nine sectors are specified by the nine combinations of

`{:top, :center, :bottom}` × `{:left, :center, :right}`



Using Location Designators with TurtleSim

Writing a location designator generator function

Create a new Lisp file for the location designators code:

Make sure you are in the cram_my_beginner_tutorial/src sub-directory

```
~$ cd ~/workspace/ros/src/cram_my_beginner_tutorial/src
~/workspace/ros/src/cram_my_beginner_tutorial/src$
```

Using Location Designators with TurtleSim

Writing a location designator generator function

Create a new Lisp file for the location designators code:

Edit location-designators.lisp

```
~/workspace/ros/src/cram_my_beginner_tutorial/src$ emacs location-designators.lisp
```

Using Location Designators with TurtleSim

Writing a location designator generator function

Create a new Lisp file for the location designators code:

Edit location-designators.lisp

Copy and paste the code from the following slide

```
(in-package :tut)

(defun navigation-goal-generator (designator)
  (declare (type location-designator designator))
  (with-desig-props (vertical-position horizontal-position) designator
    (let ((x-offset (ecase horizontal-position
                       (:left 0)
                       (:center (/ 11.0 3.0))
                       (:right (* (/ 11.0 3.0) 2))))
          (y-offset (ecase vertical-position
                       (:bottom 0)
                       (:center (/ 11.0 3.0))
                       (:top (* (/ 11.0 3.0) 2)))))
      (loop repeat 5
            collect (cl-transforms:make-3d-vector
                     (+ x-offset (random (/ 11.0 3.0)))
                     (+ y-offset (random (/ 11.0 3.0)))
                     0)))))
```

```
(in-package :tut)

(defun navigation-goal-generator (designator)
  (declare (type location-designator designator))
  (with-desig-props (vertical-position horizontal-position) designator
    (let ((x-offset (ecase horizontal-position
                      (:left 0)
                      (:center (/ 11.0 3.0))
                      (:right (* (/ 11.0 3.0) 2))))
          (y-offset (ecase vertical-position
                      (:bottom 0)
                      (:center (/ 11.0 3.0))
                      (:top (* (/ 11.0 3.0) 2)))))
      (loop repeat 5
            collect (cl-transforms:make-3d-vector
                     (+ x-offset (random (/ 11.0 3.0)))
                     (+ y-offset (random (/ 11.0 3.0)))
                     0)))))

(register-location-generator 5 navigation-goal-generator)
```

the generator has one parameter: a designator ...

Declare the type of the designator to be a `location-designator`

Use `the vertical-position` and `horizontal-position` designator properties

x origin for :left, :center, and :right sectors

y origin for :bottom, :center, and :top sectors

Generate and return 5 random sets of coordinates in the specified sector offset from the x and y origins

the function generator returns a list of 5 3d-vectors

Register the function as a location designator generator

The priority of the generator
Generators are called in priority order: lower values imply higher priority and the earlier they are called

Using Location Designators with TurtleSim

Writing a location designator generator function

Now, let's experiment with this code

First, we need to make sure a ROS master is running

If you have not already done it, open a terminal and enter

```
~$ roscore
```

Using Location Designators with TurtleSim

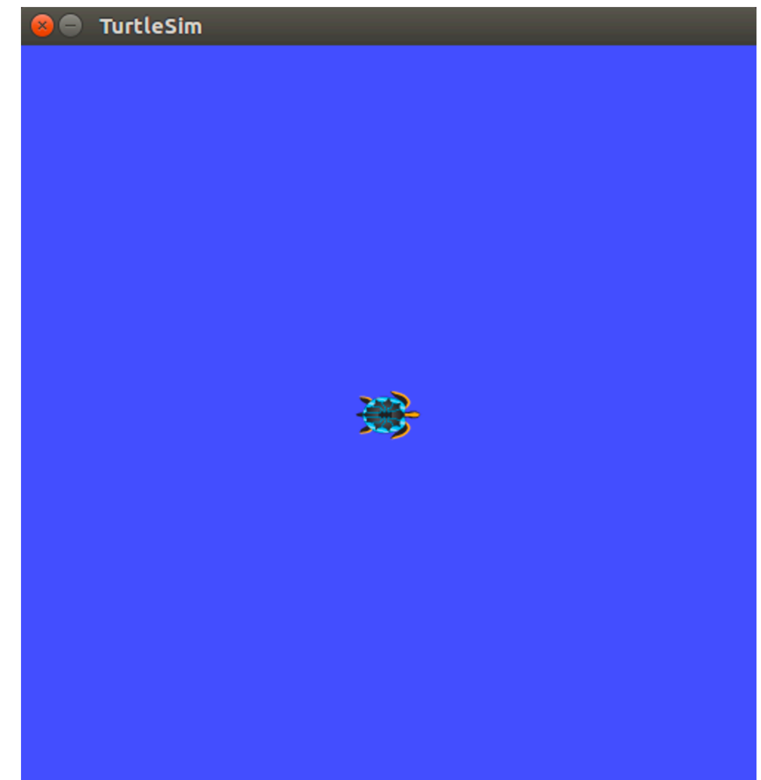
Writing a location designator generator function

Now, start turtlesim

Open a new terminal and enter

~$ rosrun turtlesim turtlesim_node

This is what you should see



Using Location Designators with TurtleSim

Launch the Lisp REPL

If you have not already done it, open a terminal and enter

```
~/workspace/ros$ roslisp_repl
```

Load the system

```
CL-USER> (ros-load:load-system "cram_my_beginner_tutorial" :cram-my-beginner-tutorial)
```

Switch to the package

```
CL-USER> (in-package :tut)
TUT>
```

Using Location Designators with TurtleSim

Start a ROS node

```
TUT> (start-ros-node "turtle1")
[(ROSLISP TOP) INFO] 1292688669.674: Node name is turtle1
[(ROSLISP TOP) INFO] 1292688669.687: Namespace is /
[(ROSLISP TOP) INFO] 1292688669.688: Params are NIL
[(ROSLISP TOP) INFO] 1292688669.689: Remappings are:
[(ROSLISP TOP) INFO] 1292688669.691: master URI is 127.0.0.1:11311
[(ROSLISP TOP) INFO] 1292688670.875: Node startup complete
```

Using Location Designators with TurtleSim

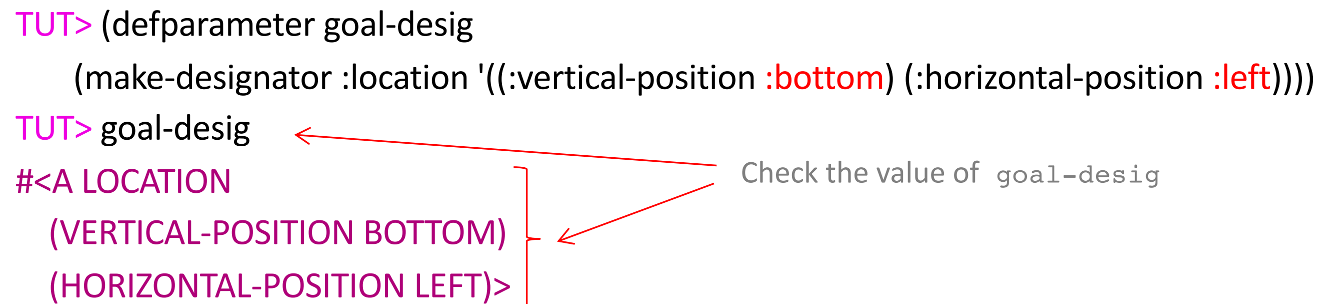
Call the function to perform the initialization

```
TUT> (init-ros-turtle "turtle1")
```

Using Location Designators with TurtleSim

Writing a location designator generator function

Define the example location designator



Using Location Designators with TurtleSim

Writing a location designator generator function

Now, test the example designator resolving function

```
TUT> (navigation-goal-generator goal-desig)
(#<3D-VECTOR (1.0404995679855347d0 0.035078644752502444d0 0.0d0)>
 #<3D-VECTOR (3.0929160118103027d0 0.8247395753860474d0 0.0d0)>
 #<3D-VECTOR (3.048727035522461d0 1.0249313116073608d0 0.0d0)>
 #<3D-VECTOR (2.1428534984588623d0 2.775157928466797d0 0.0d0)>
 #<3D-VECTOR (3.3696107864379883d0 0.026859402656555176d0 0.0d0)>)
```

The generator function takes a location designator as an argument and returns a list of solutions

The 5 generated sets of random coordinates in the specified sector (:bottom :left) i.e. in the range 0 <= x, y <= 3.667 (11//3)

Note: it returns them as a list

Using Location Designators with TurtleSim

Writing a location designator generator function

And now, try resolving (referencing) the example location designator

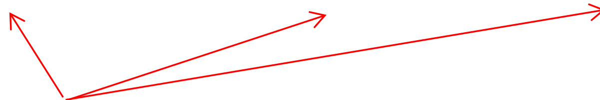
```
TUT> (reference goal-desig)
#<3D-VECTOR (1.1420983076095558d0 2.184809684753418d0 0.0d0)>
```

Using Location Designators with TurtleSim

Writing a location designator generator function

And now, try resolving (referencing) the example location designator

```
TUT> (reference goal-desig)
#<3D-VECTOR (1.142098307609558d0 2.184809684753418d0 0.0d0)>
```



Why are these values not the same as the first element. of the list we got when we performed the evaluation previously?

```
TUT> (navigation-goal-generator goal-desig)
(#<3D-VECTOR (1.0404995679855347d0 0.035078644752502444d0 0.0d0)>
#<3D-VECTOR (3.0929160118103027d0 0.8247395753860474d0 0.0d0)>
 #<3D-VECTOR (3.048727035522461d0 1.0249313116073608d0 0.0d0)>
 #<3D-VECTOR (2.1428534984588623d0 2.775157928466797d0 0.0d0)>
 #<3D-VECTOR (3.3696107864379883d0 0.026859402656555176d0 0.0d0)>)
```

Because when referencing goal-desig, the generator creates a new and different list of candidates

Using Location Designators with TurtleSim

Writing a location designator generator function

And now, try resolving (referencing) the example location designator

```
TUT> (reference goal-desig)
#<3D-VECTOR (1.142098307609558d0 2.184809684753418d0 0.0d0)>
TUT> (reference goal-desig)
#<3D-VECTOR (1.142098307609558d0 2.184809684753418d0 0.0d0)>
```

It resolves to a 3d-vector with appropriate values

On the second resolution, it also resolves to a 3d-vector with appropriate values

... but why does it have the same coordinates? Surely, the random number generator in `navigation-goal-generator` didn't create the exact same numbers twice?

It didn't; it generated 5 solution ... but to get the other elements in the list of solutions we need another function: `next-solution`

Using Location Designators with TurtleSim

Writing a location designator generator function

To get the other elements in the list of solutions we need another function: `next-solution`

Call `next-solution` with the location designator `goal-desig` as an argument

That's odd: it returns a location designator, not a 3d-vector with the coordinates

```
TUT> (next-solution goal-desig)
#<LOCATION-DESIGNATOR ((:VERTICAL-POSITION :BOTTOM)
                       (:HORIZONTAL-POSITION :LEFT)) {10086B5A23}>
```

- Confusingly, `next-solution` doesn't actually return the next solution, but rather a new designator
  - This designator has identical properties to the `goal-desig` location designator
  - If no other solution exists, then `next-solution` returns `nil`
- If we actually want to see this solution, assuming it exists, we need to resolve it, i.e. reference it ...

Using Location Designators with TurtleSim

Writing a location designator generator function

To see this solution, assuming it exists, we need to resolve it, i.e. reference it …

```
TUT>  (reference (next-solution goal-desig))
#<3D-VECTOR (0.261885404586792d0 2.649489164352417d0 0.0d0)>
```

- If we called (reference (next-solution goal-desig)) again, we would get this same solution again
- If goal-desig stays the same, then the next solution designator also stays the same, and so does the solution associated with it

Using Location Designators with TurtleSim

Writing a location designator generator function

One way to iterate over the solutions would be:

```
TUT> (loop for desig = goal-desig then (next-solution desig)
           while desig
           do (print (reference desig)))
#<3D-VECTOR (1.142098307609558d0 2.184809684753418d0 0.0d0)>
#<3D-VECTOR (0.261885404586792d0 2.649489164352417d0 0.0d0)>
#<3D-VECTOR (2.560281276702881d0 1.5541117191314697d0 0.0d0)>
#<3D-VECTOR (3.182616949081421d0 1.329969882965088d0 0.0d0)>
#<3D-VECTOR (1.3107243776321411d0 2.9240825176239014d0 0.0d0)>
NIL
```

Using Location Designators with TurtleSim

Writing a location designator generator function

One way to iterate over the solutions would be:

```
TUT> (loop for desig = goal-desig then (next-solution desig)
           while desig
           do (print (reference desig)))
#<3D-VECTOR (1.142098307609558d0 2.184809684753418d0 0.0d0)>
#<3D-VECTOR (0.261885404586792d0 2.649489164352417d0 0.0d0)>
#<3D-VECTOR (2.560281276702881d0 1.5541117191314697d0 0.0d0)>
#<3D-VECTOR (3.182616949081421d0 1.329969882965088d0 0.0d0)>
#<3D-VECTOR (1.3107243776321411d0 2.9240825176239014d0 0.0d0)>
NIL
```

14.5 The Loop Facility

The `loop` macro was originally designed to help inexperienced Lisp users write iterative code. Instead of writing Lisp code, you express your program in a form meant to resemble English, and this is then translated into Lisp. Unfortunately, `loop` is more like English than its designers ever intended: you can use it in simple cases without quite understanding how it works, but to understand it in the abstract is almost impossible.

There are three phases in the evaluation of a `loop` expression, and a given clause can contribute code to more than one phase. The phases are as follows:

1. *Prologue.* Evaluated once as a prelude to iteration. Includes setting variables to their initial values.
2. *Body.* Evaluated on each iteration. Begins with the termination tests, followed by the body proper, then the updating of iteration variables.
3. *Epilogue.* Evaluated once iteration is completed. Concludes with the return of the value(s) of the `loop` expression.

Using Location Designators with TurtleSim

Writing a location designator validation function

It is sometimes necessary to validate the generated solutions

- For example, the points we get from `navigation-goal-generator` range from 0 to 11
- But whenever the turtle has a coordinate above 10.5 or less than 0.5 a part of it disappears from the screen
- Let's improve our location designator resolution by rejecting the points that lie outside of the range 0.5 -10.5

Using Location Designators with TurtleSim

Writing a location designator validation function

Update the location designators code:

Edit location-designators.lisp

Copy and paste the code from the following slide

```
(defun navigation-goal-validator (designator solution)
  (declare (type location-designator designator))
  (when (and (desig-prop-value designator :vertical-position)
             (desig-prop-value designator :horizontal-position))
    (when (typep solution 'cl-transforms:3d-vector)
      (when
          (and
           (>= (cl-transforms:x solution) 0.5)
           (>= (cl-transforms:y solution) 0.5)
           (<= (cl-transforms:x solution) 10.5)
           (<= (cl-transforms:y solution) 10.5))
        :accept))))

(register-location-validation-function
 5 navigation-goal-validator)
```

The validation function has two parameters: a location designator and a solution

```
(defun navigation-goal-validator (designator solution)
  (declare (type location-designator designator))
  (when (and (desig-prop-value designator :vertical-position)
             (desig-prop-value designator :horizontal-position))
    (when (typep solution 'cl-transforms:3d-vector)
      (when
          (and
           (>= (cl-transforms:x solution) 0.5)
           (>= (cl-transforms:y solution) 0.5)
           (<= (cl-transforms:x solution) 10.5)
           (<= (cl-transforms:y solution) 10.5))
        :accept))))

(register-location-validation-function 5 navigation-goal-validator))
```

If the designator property values are what we expect (refer back to the definition of the function generator)

If the type of `solution` is `cl-transforms:3d-vector`

if the values of the x and y coordinates are in the required range the return :accept (i.e. T) and reject (NIL) everything else

Register the function as a location designator validator

The priority of the validator
Just like generators, validators are called in priority order:
lower values imply higher priority and the earlier they are called

Using Location Designators with TurtleSim

Writing a location designator validation function

Re-load the system (or recompile)

```
TUT> (ros-load:load-system "cram_my_beginner_tutorial" :cram-my-beginner-tutorial)
```

Using Location Designators with TurtleSim

Writing a location designator validation function

Check to see if the validation function works

```
TUT> (defparameter another-goal
       (make-designator :location '((:vertical-position :bottom) (:horizontal-position :left))))
ANOTHER-GOAL
TUT> (loop for desig = another-goal then (next-solution desig)
                 while desig
                  do (print (reference desig)))
#<3D-VECTOR (1.5110043287277222d0 2.1800525188446045d0 0.0d0)>
#<3D-VECTOR (2.030768394470215d0 2.731144428253174d0 0.0d0)>
#<3D-VECTOR (0.7122993469238281d0 3.623168706893921d0 0.0d0)>
NIL
```

Define a new location designator

Print all the coordinates after they have been validated

In this case, only three of the five generated solutions have been validated;
the other two must have been outside the allowed range

Using Location Designators with TurtleSim

Using a location designator

Append the following code to `motion-designators.lisp`

Using Location Designators with TurtleSim

```
(def-fact-group goal-motions (motion-grounding)
  (<- (motion-grounding ?desig (go-to ?point))
    (desig-prop ?desig (:type :going-to))
    (desig-prop ?desig (:goal ?point))))
```

This will resolve any motion designator with properties

`( (:type :going-to) (:goal some-location-designator) )`

into

`(go-to some-location-designator) instruction.`

Using Location Designators with TurtleSim

Using a location designator

Append the following code to `process-modules.lisp`

- Specifically,
  add the `go-to` part to the case in the `turtlesim-navigation` process module

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion)))
      (move
        (move-to motion))
      (go-to
       (when (typep motion 'location-designator)
         (let ((target-point (reference motion)))
           (roslisp:ros-info (turtle-process-modules)
                             "Going to point ~a." target-point)
           (move-to target-point))))))))
...
```

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion)))
      (move
        (move-to motion))
      (go-to
       (when (typep motion 'location-designator)
         (let ((target-point (reference motion)))
           (roslisp:ros-info (turtle-process-modules)
                             "Going to point ~a." target-point)
           (move-to target-point))))))))
...
```

We now have a third type of motion designator, i.e. a new command go-to

```
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
       (send-vel-cmd
        (turtle-motion-speed motion)
        (turtle-motion-angle motion)))
      (move
        (move-to motion))
      (go-to
       (when (typep motion 'location-designator)
         (let ((target-point (reference motion)))
           (roslisp:ros-info (turtle-process-modules)
                             "Going to point ~a." target-point)
           (move-to target-point))))))))
...
```

If motion is a location designator ...

resolve the designator and assign the value to target-point

and call move-to to make the turtle go to that location

Using Location Designators with TurtleSim

Using a location designator

And append the following code to `process-modules.lisp`

Specifically, add the `goto-location` function in the `turtlesim-navigation` process module

```
(in-package :tut)

...

(defun drive (?speed ?angle)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let ((trajectory (desig:a motion (type driving) (speed ?speed) (angle ?angle))))
        (pm-execute 'turtlesim-navigation trajectory)))))

(defun goto-location (?horizontal-position ?vertical-position)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let* ((?area (desig:a location
                             (horizontal-position ?horizontal-position)
                             (vertical-position ?vertical-position)))
             (goal (desig:a motion (type going-to) (goal ?area))))
        (cram-executive:perform goal)))))
```

Because we use `perform`, we need to make sure the correct process module for our `going-to` designator can be found

For that we add a rule to the fact-group `available-turtle-process-modules` from the selecting-process-modules.lisp file

Using Location Designators with TurtleSim

Using a location designator

The complete process-modules.lisp module should now look like the following

```lisp
(in-package :tut)

(def-process-module turtlesim-navigation (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim navigation invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (drive
         (send-vel-cmd
          (turtle-motion-speed motion)
          (turtle-motion-angle motion)))
      (move
       (move-to motion))
      (go-to
       (when (typep motion 'location-designator)
         (let ((target-point (reference motion)))
           (roslisp:ros-info (turtle-process-modules)
                             "Going to point ~a." target-point)
           (move-to target-point))))))))

(def-process-module turtlesim-pen-control (motion-designator)
  (roslisp:ros-info (turtle-process-modules)
                    "TurtleSim pen control invoked with motion designator `~a'."
                    motion-designator)
  (destructuring-bind (command motion) (reference motion-designator)
    (ecase command
      (set-pen
       (call-set-pen
        (pen-motion-r motion)
        (pen-motion-g motion)
        (pen-motion-b motion)
        (pen-motion-width motion)
        (pen-motion-off motion))))))

(defun drive (?speed ?angle)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let ((trajectory (desig:a motion (type driving) (speed ?speed) (angle ?angle))))
        (pm-execute 'turtlesim-navigation trajectory)))))

(defun move (?x ?y)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let ((goal (desig:a motion (type moving) (goal (?x ?y 0)))))
        (pm-execute 'turtlesim-navigation goal)))))

(defun goto-location (?horizontal-position ?vertical-position)
  (top-level
    (with-process-modules-running (turtlesim-navigation)
      (let* ((?area (desig:a location
                             (horizontal-position ?horizontal-position)
                             (vertical-position ?vertical-position)))
             (goal (desig:a motion (type going-to) (goal ?area))))
        (cram-executive:perform goal)))))
```

Using Location Designators with TurtleSim

Using a location designator

Because we use `perform` in the `goto-location` function, we need to make sure the correct process module for our going-to designator can be found

For that we add a rule to the fact-group `available-turtle-process-modules` from the selecting-process-modules.lisp file

Using Location Designators with TurtleSim

Using a location designator

Add the following code to selecting-process-modules.lisp

Using Location Designators with TurtleSim

```
(in-package :tut)

(def-fact-group available-turtle-process-modules (available-process-module
                                                  matching-process-module)
  (<- (available-process-module turtlesim-navigation))
  (<- (available-process-module turtlesim-pen-control))

  (<- (matching-process-module ?desig turtlesim-navigation)
    (desig-prop ?desig (:type :driving)))
  (<- (matching-process-module ?desig turtlesim-navigation)
    (desig-prop ?desig (:type :moving)))
  (<- (matching-process-module ?desig turtlesim-navigation)
    (desig-prop ?desig (:type :going-to)))
  (<- (matching-process-module ?desig turtlesim-pen-control)
    (desig-prop ?desig (:type :setting-pen))))

(defun perform-some-motion (motion-desig)
  (top-level
    (with-process-modules-running (turtlesim-navigation turtlesim-pen-control)
      (cram-executive:perform motion-desig))))
```

Add this

Using Location Designators with TurtleSim

Using a location designator

Re-load the system (or recompile)

```
TUT> (ros-load:load-system "cram_my_beginner_tutorial" :cram-my-beginner-tutorial)
```

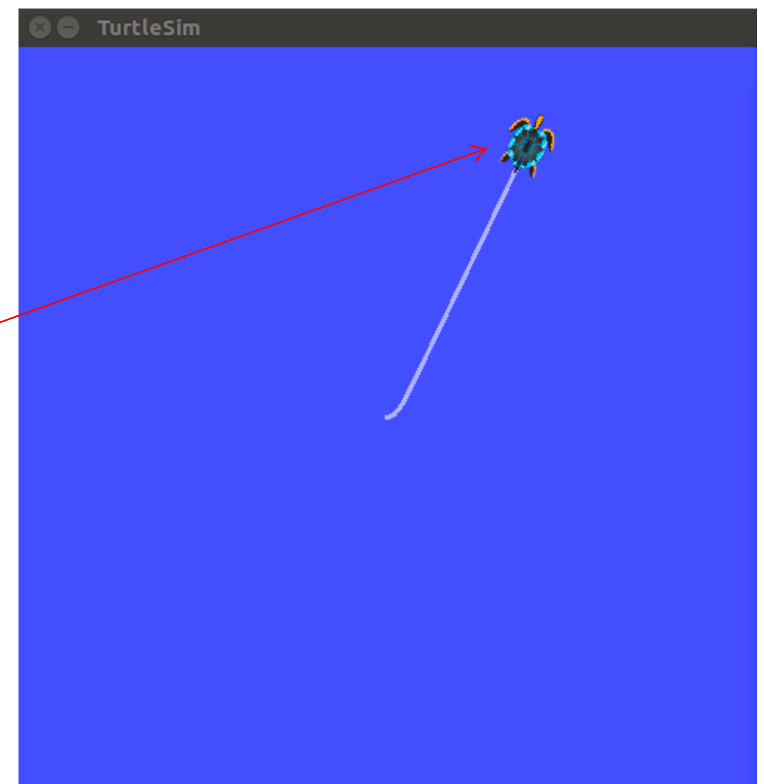
Automatically Choosing a Process Module for a Motion

Using a location designator

Let's try `goto-location`

TUT> (goto-location :right :top)

The turtle drives to the top-right sector



Using Location Designators with TurtleSim

Using a location designator

Let's try `goto-location`

```
TUT> (goto-location :right :top)
[(TURTLE-PROCESS-MODULES) INFO] 1562698457.619: TurtleSim navigation invoked with motion     designator `#<A MOTION
  (TYPE GOING-TO)
  (GOAL #<A LOCATION
   (HORIZONTAL-POSITION RIGHT)
   (VERTICAL-POSITION TOP)>)>'.
[(TURTLE-PROCESS-MODULES) INFO] 1501153970.691: Going to point #<3D-VECTOR (10.131428718566895d0 8.874866485595703d0 0.0d0)>.
T
```

CRAM Beginner Tutorials

Create a CRAM Package	http://cram-system.org/tutorials/beginner/package_for_turtlesim
Controlling turtlesim from CRAM	http://cram-system.org/tutorials/beginner/controlling_turtlesim_2
Implementing simple plans to move a turtle	http://cram-system.org/tutorials/beginner/simple_plans
Using Prolog for reasoning	http://cram-system.org/tutorials/beginner/cram_prolog
Creating motion designators for the TurtleSim	http://cram-system.org/tutorials/beginner/motion_designators
Creating process modules	http://cram-system.org/tutorials/beginner/process_modules_2
Automatically choosing a process module for a motion	http://cram-system.org/tutorials/beginner/assigning_actions_2
Using location designators with the TurtleSim	http://cram-system.org/tutorials/beginner/location_designators_2

Background Reading

G. Kazhoyan, Lecture notes: Robot Programming with Lisp 7. Coordinate Transformations, TF, ActionLib, slides 5-8.
https://ai.uni-bremen.de/_media/teaching/7_more_ros.pdf

http://wiki.ros.org/tf/Overview/Transformations

T. Rittweiler, CRAM – Design and Implementation of a Reactive Plan Language, Bachelor Thesis, Technical University of Munich, 2010.
https://common-lisp.net/~trittweiler/bachelor-thesis.pdf