Introduction to Cognitive Robotics

Module 10: Using Turtlesim with CRAM
Lecture 9: Writing high-level plans for TurtleSim

www.cognitiverobotics.net

The CRAM Beginner Tutorials

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

Writing High-level Plans for TurtleSim

Based on Writing plans for the TurtleSim
http://cram-system.org/tutorials/beginner/high_level_plans

Writing High-level Plans for TurtleSim

Designators: Actions vs Motions

- Motion designators are used to represent motions of the robot, e.g. moving an arm or driving
- Action designators are used to describe abstract, i.e. high-level, plans, e.g. tidying up a room

Writing High-level Plans for TurtleSim

Designators: Actions vs Motions

The `perform` function handles action designators and motion designators differently

- Both are first referenced
- Motion designators are executed by passing them to a matching process module
- Action designators are executed by
  - Looking up a function **with the same name as the resolved designator**
  - The rest of the designator is passed as arguments to the function

Writing High-level Plans for TurtleSim

Designators: Actions vs Motions

- Let's take an example `designator`
  `(desig:an action (type tidying-up) (what the-room))`

- This could be resolved to something like `(tidy-up the-room)`

- So there has to be a function `tidy-up` which takes one parameter `the-room` for `perform` to be able to execute the designator

- Action designators can optionally contain a goal-key which would be checked to see if the goal has already been achieved in which case the action doesn't have to be performed

We won't address this here

Writing High-level Plans for TurtleSim

A plan to draw a house

- Let's implement a plan to draw a house with the turtle's pen
- We already have all the low-level functions necessary to achieve this
- Now we need to define action designators for this plan
  - A house consists of some rectangles and a triangle for the roof
  - We will define a plan to draw the simple shapes by tracing out a set of vertices
  - We will define a higher level plan to draw the house by using the plan to draw simple shapes

Writing High-level Plans for TurtleSim

A plan to draw a house

- We will put the shape drawing plan in action-designators.lisp
- We will put the house-drawing plan in high-level-plans.lisp

Writing High-level Plans for TurtleSim

Defining the action designators

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 separate files
- (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 action-designators.lisp
- Add the new code to high-level-plans.lisp
- Test the code

We will place the new code is separate Lisp files

Writing High-level Plans for TurtleSim

Defining the action designators

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$
```

Writing High-level Plans for TurtleSim

Defining the action designators

Update the ASDF dependencies

Edit cram-my-beginner-tutorial.asd

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

Writing High-level Plans for 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 "action-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"))
              (:file "high-level-plans" :depends-on ("package"
                                                     "motion-designators"
                                                     "location-designators"
                                                     "action-designators"
                                                     "process-modules")))))
```

Add these lines

Writing High-level Plans for TurtleSim

Defining inference rules for the action designators

Create a new Lisp file for the action 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$
```

Writing High-level Plans for TurtleSim

Defining inference rules for the action designators

Create a new Lisp file for the action designators code:

Edit action-designators.lisp

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

Writing High-level Plans for TurtleSim

Defining inference rules for the action designators

Create a new Lisp file for the action designators code:

Edit action-designators.lisp

Copy and paste the code from the following slide

```lisp
(in-package :tut)

(defun get-shape-vertices (type &rest parameters)
  (with-fields (x y)
      (value *turtle-pose*)
    (ecase type
      (:triangle
       (let ((base-width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x base-width) y 0)
          (list (+ x (/ (float base-width) 2)) (+ y height) 0)
          (list x y 0))))
      (:rectangle
       (let ((width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x width) y 0)
          (list (+ x width) (+ y height) 0)
          (list x (+ y height) 0)
          (list x y 0)))))))

(def-fact-group turtle-action-designators (action-grounding)
  (<- (desig:action-grounding ?desig (draw-house))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :house)))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :rectangle))
    (desig-prop ?desig (:width ?width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :rectangle ?width ?height ?vertices))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :triangle))
    (desig-prop ?desig (:base-width ?base-width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :triangle ?base-width ?height ?vertices))

  (<- (desig:action-grounding ?desig (navigate ?target))
    (desig-prop ?desig (:type :navigating))
    (desig-prop ?desig (:target ?target))))
```

```
(in-package :tut)

(defun get-shape-vertices (type &rest parameters)
  (with-fields (x y)
      (value *turtle-pose*)
    (ecase type
      (:triangle
       (let ((base-width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x base-width) y 0)
          (list (+ x (/ (float base-width) 2)) (+ y height) 0)
          (list x y 0))))
      (:rectangle
       (let ((width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x width) y 0)
          (list (+ x width) (+ y height) 0)
          (list x (+ y height) 0)
          (list x y 0)))))))

(def-fact-group turtle-action-designators (action-grounding)
  (<- (desig:action-grounding ?desig (draw-house))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :house)))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :rectangle))
    (desig-prop ?desig (:width ?width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :rectangle ?width ?height ?vertices))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :triangle))
    (desig-prop ?desig (:base-width ?base-width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :triangle ?base-width ?height ?vertices))

  (<- (desig:action-grounding ?desig (navigate ?target))
    (desig-prop ?desig (:type :navigating))
    (desig-prop ?desig (:target ?target))))
```

A function to return a list of vertices for the `draw-simple-shape` plan to trace

The `&rest parameters` means that when the function is called the `parameters` parameter is set to a list of all the remaining arguments, in this case all the arguments after the first one for the parameter `type` (see Graham 1996, p. 102)

The function uses the current position of the turtle to calculate the coordinates of these vertices

Select the `draw-simple-shape` process module when the action designator has type `:drawing` , shape `:triangle`. base-width is bound to `?base-width` and height is bound to `:height`, which are then used , along with `:triangle` as arguments in the call to `get-shape-vertices` (which, in turn, returns the list of vertex coordinates)

```
(in-package :tut)

(defun get-shape-vertices (type &rest parameters)
  (with-fields (x y)
      (value *turtle-pose*)
    (ecase type
      (:triangle
       (let ((base-width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x base-width) y 0)
          (list (+ x (/ (float base-width) 2)) (+ y height) 0)
          (list x y 0))))
      (:rectangle
       (let ((width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x width) y 0)
          (list (+ x width) (+ y height) 0)
          (list x (+ y height) 0)
          (list x y 0)))))))

(def-fact-group turtle-action-designators (action-grounding)
  (<- (desig:action-grounding ?desig (draw-house))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :house)))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :rectangle))
    (desig-prop ?desig (:width ?width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :rectangle ?width ?height ?vertices))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :triangle))
    (desig-prop ?desig (:base-width ?base-width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :triangle ?base-width ?height ?vertices))

  (<- (desig:action-grounding ?desig (navigate ?target))
    (desig-prop ?desig (:type :navigating))
    (desig-prop ?desig (:target ?target))))
```

$\leftarrow$ Coordinates for the triangle

$\leftarrow$ Coordinates for the rectangle

```
(in-package :tut)

(defun get-shape-vertices (type &rest parameters)
  (with-fields (x y)
      (value *turtle-pose*)
    (ecase type
      (:triangle
       (let ((base-width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x base-width) y 0)
          (list (+ x (/ (float base-width) 2)) (+ y height) 0)
          (list x y 0))))
      (:rectangle
       (let ((width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x width) y 0)
          (list (+ x width) (+ y height) 0)
          (list x (+ y height) 0)
          (list x y 0)))))))

(def-fact-group turtle-action-designators (action-grounding)
  (<- (desig:action-grounding ?desig (draw-house))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :house)))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :rectangle))
    (desig-prop ?desig (:width ?width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :rectangle ?width ?height ?vertices))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :triangle))
    (desig-prop ?desig (:base-width ?base-width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :triangle ?base-width ?height ?vertices))

  (<- (desig:action-grounding ?desig (navigate ?target))
    (desig-prop ?desig (:type :navigating))
    (desig-prop ?desig (:target ?target))))
```

Select the `draw-house` function when the action designator has type :`drawing` and shape :house

```
(in-package :tut)

(defun get-shape-vertices (type &rest parameters)
  (with-fields (x y)
      (value *turtle-pose*)
    (ecase type
      (:triangle
       (let ((base-width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x base-width) y 0)
          (list (+ x (/ (float base-width) 2)) (+ y height) 0)
          (list x y 0))))
      (:rectangle
       (let ((width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x width) y 0)
          (list (+ x width) (+ y height) 0)
          (list x (+ y height) 0)
          (list x y 0)))))))

(def-fact-group turtle-action-designators (action-grounding)
  (<- (desig:action-grounding ?desig (draw-house))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :house)))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :rectangle))
    (desig-prop ?desig (:width ?width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :rectangle ?width ?height ?vertices))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :triangle))
    (desig-prop ?desig (:base-width ?base-width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :triangle ?base-width ?height ?vertices))

  (<- (desig:action-grounding ?desig (navigate ?target))
    (desig-prop ?desig (:type :navigating))
    (desig-prop ?desig (:target ?target))))
```

Select the `draw-simple-shape` function when the action designator has type `:drawing` and shape is `:rectangle`. width is bound to `?width` and height is bound to ?height, which are then used, along with `:rectangle`, as arguments in the call to `get-shape-vertices` (which, in turn, returns the list of vertex coordinates)

```
(in-package :tut)

(defun get-shape-vertices (type &rest parameters)
  (with-fields (x y)
      (value *turtle-pose*)
    (ecase type
      (:triangle
       (let ((base-width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x base-width) y 0)
          (list (+ x (/ (float base-width) 2)) (+ y height) 0)
          (list x y 0))))
      (:rectangle
       (let ((width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x width) y 0)
          (list (+ x width) (+ y height) 0)
          (list x (+ y height) 0)
          (list x y 0)))))))

(def-fact-group turtle-action-designators (action-grounding)
  (<- (desig:action-grounding ?desig (draw-house))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :house)))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :rectangle))
    (desig-prop ?desig (:width ?width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :rectangle ?width ?height ?vertices))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :triangle))
    (desig-prop ?desig (:base-width ?base-width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :triangle ?base-width ?height ?vertices))

  (<- (desig:action-grounding ?desig (navigate ?target))
    (desig-prop ?desig (:type :navigating))
    (desig-prop ?desig (:target ?target))))
```

Select the `draw-simple-shape` function when the action designator has type :`drawing` and shape is `:triangle`. base-width is bound to `?base-width` and height is bound to `?height`, which are then used , along with `:triangle`, as arguments in the call to `get-shape-vertices` (which, in turn, returns the list of vertex coordinates)

```
(in-package :tut)

(defun get-shape-vertices (type &rest parameters)
  (with-fields (x y)
      (value *turtle-pose*)
    (ecase type
      (:triangle
       (let ((base-width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x base-width) y 0)
          (list (+ x (/ (float base-width) 2)) (+ y height) 0)
          (list x y 0))))
      (:rectangle
       (let ((width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x width) y 0)
          (list (+ x width) (+ y height) 0)
          (list x (+ y height) 0)
          (list x y 0)))))))

(def-fact-group turtle-action-designators (action-grounding)
  (<- (desig:action-grounding ?desig (draw-house))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :house)))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :rectangle))
    (desig-prop ?desig (:width ?width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :rectangle ?width ?height ?vertices))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :triangle))
    (desig-prop ?desig (:base-width ?base-width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :triangle ?base-width ?height ?vertices))

  (<- (desig:action-grounding ?desig (navigate ?target))
    (desig-prop ?desig (:type :navigating))
    (desig-prop ?desig (:target ?target))))
```

$\leftarrow$ Select the `navigate function` when the action designator has type `:navigating` and target is bound to `?target` so that the designator simply resolves to a call to navigate using the argument

```
(in-package :tut)

(defun get-shape-vertices (type &rest parameters)
  (with-fields (x y)
      (value *turtle-pose*)
    (ecase type
      (:triangle
       (let ((base-width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x base-width) y 0)
          (list (+ x (/ (float base-width) 2)) (+ y height) 0)
          (list x y 0))))
      (:rectangle
       (let ((width (first parameters))
             (height (second parameters)))
         (list
          (list (+ x width) y 0)
          (list (+ x width) (+ y height) 0)
          (list x (+ y height) 0)
          (list x y 0)))))))

(def-fact-group turtle-action-designators (action-grounding)
  (<- (desig:action-grounding ?desig (draw-house))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :house)))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :rectangle))
    (desig-prop ?desig (:width ?width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :rectangle ?width ?height ?vertices))

  (<- (desig:action-grounding ?desig (draw-simple-shape ?vertices))
    (desig-prop ?desig (:type :drawing))
    (desig-prop ?desig (:shape :triangle))
    (desig-prop ?desig (:base-width ?base-width))
    (desig-prop ?desig (:height ?height))
    (lisp-fun get-shape-vertices :triangle ?base-width ?height ?vertices))

  (<- (desig:action-grounding ?desig (navigate ?target))
    (desig-prop ?desig (:type :navigating))
    (desig-prop ?desig (:target ?target))))
```

We have yet to define these three functions

```
draw-house
draw-simple-shape
navigate
```

Writing High-level Plans for TurtleSim

Defining the action designators

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
```

Writing High-level Plans for TurtleSim

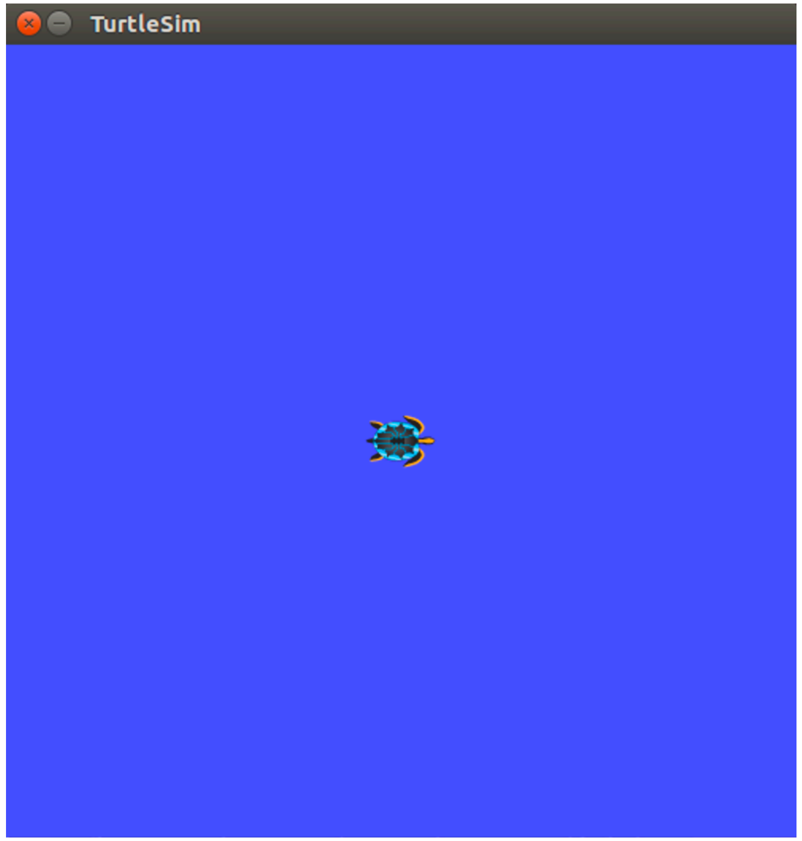
Defining the action designators

Now, start turtlesim

Open a new terminal and enter

```
~$ rosrun turtlesim turtlesim_node
```

This is what you should see



Writing High-level Plans for TurtleSim

Launch the Lisp REPL

Open a new 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>
```

Writing High-level Plans for TurtleSim

Start a ROS node

The name doesn't matter

```
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
```

Writing High-level Plans for TurtleSim

Call the function to perform the initialization

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

Use turtle1 ... remember, this forms the prefix on the topic names
This is the name of the first turtle that turtlesim spawns

Writing High-level Plans for TurtleSim

Defining inference rules for the action designators

Now, let try to create and resolve some example designators

```
TUT> (reference (desig:an action (type drawing) (shape rectangle) (width 5) (height 4)))
(DRAW-SIMPLE-SHAPE
 ((10.544444561004639d0 5.544444561004639d0 0)
  (10.544444561004639d0 9.544444561004639d0 0)
  (5.544444561004639d0 9.544444561004639d0 0)
  (5.544444561004639d0 5.544444561004639d0 0)))
TUT> (reference (desig:an action (type drawing) (shape house)))
(DRAW-HOUSE)
```

Writing High-level Plans for TurtleSim

Writing the plans

Create a new Lisp file for the action 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$
```

Writing High-level Plans for TurtleSim

Writing the plans

Create a new Lisp file for the action designators code:

Edit high-level-plans.lisp

```
~/workspace/ros/src/cram_my_beginner_tutorial/src$ emacs high-level-plans.lisp
```

Writing High-level Plans for TurtleSim

Writing the plans

Create a new Lisp file for the action designators code:

Edit high-level-plans.lisp

- Copy and paste the code from the following slide
  - This code contains the code for the three functions that match the resolved action designator

```
draw-house
draw-simple-shape
navigate
```

The `draw-house` function performs four actions, i.e. executes four process modules by resolving four action designators, one for the house walls, one for the door, one for the window, and one for the roof

```
(in-package :tut)

(defun draw-house ()
  (with-fields (x y)
      (value *turtle-pose*)
    (exe:perform (an action (type drawing) (shape rectangle) (width 5) (height 4.5)))
    (navigate-without-pen (list (+ x 3) y 0))
    (exe:perform (an action (type drawing) (shape rectangle) (width 1) (height 2.5)))
    (navigate-without-pen (list (+ x 0.5) (+ y 2) 0))
    (exe:perform (an action (type drawing) (shape rectangle) (width 1) (height 1)))
    (navigate-without-pen (list x (+ y 4.5) 0))
    (exe:perform (an action (type drawing) (shape triangle) (base-width 5) (height 4)))))

(defun draw-simple-shape (vertices)
  (mapcar
   (lambda (?v)
     (exe:perform (an action (type navigating) (target ?v))))
   vertices))

(defun navigate-without-pen (?target)
  (exe:perform (a motion (type setting-pen) (off 1)))
  (exe:perform (an action (type navigating) (target ?target)))
  (exe:perform (a motion (type setting-pen) (off 0))))

(defun navigate (?v)
  (exe:perform (a motion (type moving) (goal ?v))))
```

Draw the walls (by resolving an action designator and calling the `draw-simple-shape` function)

Draw the door

Draw the window

Draw the roof

```
(in-package :tut)

(defun draw-house ()
  (with-fields (x y)
      (value *turtle-pose*)
    (exe:perform (an action (type drawing) (shape rectangle) (width 5) (height 4.5)))
    (navigate-without-pen (list (+ x 3) y 0))
    (exe:perform (an action (type drawing) (shape rectangle) (width 1) (height 2.5)))
    (navigate-without-pen (list (+ x 0.5) (+ y 2) 0))
    (exe:perform (an action (type drawing) (shape rectangle) (width 1) (height 1)))
    (navigate-without-pen (list x (+ y 4.5) 0))
    (exe:perform (an action (type drawing) (shape triangle) (base-width 5) (height 4)))))

(defun draw-simple-shape (vertices)
  (mapcar
   (lambda (?v)
     (exe:perform (an action (type navigating) (target ?v))))
   vertices))

(defun navigate-without-pen (?target)
  (exe:perform (a motion (type setting-pen) (off 1)))
  (exe:perform (an action (type navigating) (target ?target)))
  (exe:perform (a motion (type setting-pen) (off 0))))

(defun navigate (?v)
  (exe:perform (a motion (type moving) (goal ?v))))
```

The `navigate-without-pen` function is a "helper" function.
It moves the turtle to the start position for the next part of the drawing..
It uses the navigate plan (by resolving an action designator of type `navigating`) to do this

```
(in-package :tut)

(defun draw-house ()
  (with-fields (x y)
      (value *turtle-pose*)
    (exe:perform (an action (type drawing) (shape rectangle) (width 5) (height 4.5)))
    (navigate-without-pen (list (+ x 3) y 0))
    (exe:perform (an action (type drawing) (shape rectangle) (width 1) (height 2.5)))
    (navigate-without-pen (list (+ x 0.5) (+ y 2) 0))
    (exe:perform (an action (type drawing) (shape rectangle) (width 1) (height 1)))
    (navigate-without-pen (list x (+ y 4.5) 0))
    (exe:perform (an action (type drawing) (shape triangle) (base-width 5) (height 4)))))
```

$\leftarrow$ Draw the walls

$\leftarrow$ Draw the door

$\leftarrow$ Draw the window

$\leftarrow$ Draw the roof

```
(defun draw-simple-shape (vertices)
  (mapcar
   (lambda (?v)
     (exe:perform (an action (type navigating) (target ?v))))
   vertices))

(defun navigate-without-pen (?target)
  (exe:perform (a motion (type setting-pen) (off 1)))
  (exe:perform (an action (type navigating) (target ?target)))
  (exe:perform (a motion (type setting-pen) (off 0))))

(defun navigate (?v)
  (exe:perform (a motion (type moving) (goal ?v))))
```

$\leftarrow$ The `draw-simple-shape` uses the `mapcar` function to call a function for each element of a list, i.e., each vertex in the `vertices` list passed as a parameter.

It does this by evaluating the lambda expression (think of it as an unnamed function)

The lambda expression has a parameter `?v` (the goal position) and a body comprising one expression to perform some process module that is identified by resolving the action designator.

This action designator is of type `navigating` with a target `?v` so it resolved by calling `navigate` to navigate the turtle to the goal position specified by `?v`.

Refer back to the inference rules in `action-designators.lisp`

```
(in-package :tut)

(defun draw-house ()
  (with-fields (x y)
      (value *turtle-pose*)
    (exe:perform (an action (type drawing) (shape rectangle) (width 5) (height 4.5)))    Draw the walls
    (navigate-without-pen (list (+ x 3) y 0))
    (exe:perform (an action (type drawing) (shape rectangle) (width 1) (height 2.5)))    Draw the door
    (navigate-without-pen (list (+ x 0.5) (+ y 2) 0))
    (exe:perform (an action (type drawing) (shape rectangle) (width 1) (height 1)))      Draw the window
    (navigate-without-pen (list x (+ y 4.5) 0))
    (exe:perform (an action (type drawing) (shape triangle) (base-width 5) (height 4)))))    Draw the roof

(defun draw-simple-shape (vertices)
  (mapcar
   (lambda (?v)
     (exe:perform (an action (type navigating) (target ?v))))
   vertices))

(defun navigate-without-pen (?target)
  (exe:perform (a motion (type setting-pen) (off 1)))
  (exe:perform (an action (type navigating) (target ?target)))
  (exe:perform (a motion (type setting-pen) (off 0))))

(defun navigate (?v)
  (exe:perform (a motion (type moving) (goal ?v))))
```

There is not much in this function yet, apart from the resolution of a motion designator and the execution of the associated process module

However, we will add to it later in the section on failure handling

```
(in-package :tut)

(defun draw-house ()
  (with-fields (x y)
      (value *turtle-pose*)
    (exe:perform (an action (type drawing) (shape rectangle) (width 5) (height 4.5)))
    (navigate-without-pen (list (+ x 3) y 0))
    (exe:perform (an action (type drawing) (shape rectangle) (width 1) (height 2.5)))
    (navigate-without-pen (list (+ x 0.5) (+ y 2) 0))
    (exe:perform (an action (type drawing) (shape rectangle) (width 1) (height 1)))
    (navigate-without-pen (list x (+ y 4.5) 0))
    (exe:perform (an action (type drawing) (shape triangle) (base-width 5) (height 4)))))

(defun draw-simple-shape (vertices)
  (mapcar
   (lambda (?v)
     (exe:perform (an action (type navigating) (target ?v))))
   vertices))

(defun navigate-without-pen (?target)
  (exe:perform (a motion (type setting-pen) (off 1)))
  (exe:perform (an action (type navigating) (target ?target)))
  (exe:perform (a motion (type setting-pen) (off 0))))

(defun navigate (?v)
  (exe:perform (a motion (type moving) (goal ?v))))
```

Note: we omit the `desig:` package namespace specification before the `a` and `an` macros

Writing High-level Plans for TurtleSim

Test the plan

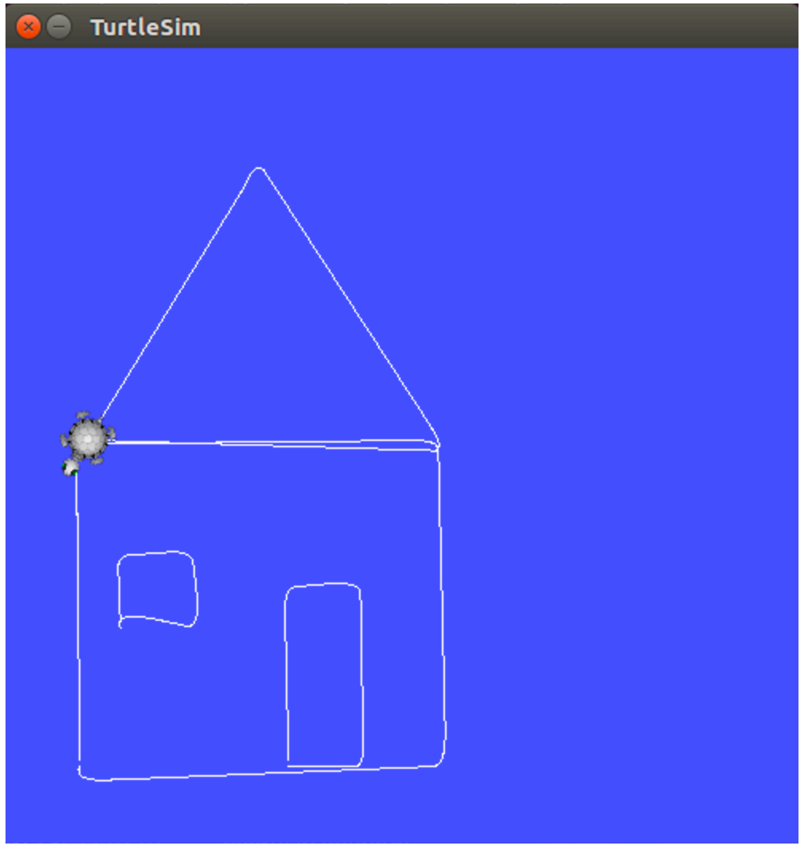
```
TUT> (top-level
       (with-process-modules-running (turtlesim-navigation turtlesim-pen-control)
         (navigate-without-pen '(1 1 0))
         (exe:perform (an action (type drawing) (shape house)))))
```

Make the turtle go to the bottom left first before executing the plan (to make sure there is room for the drawing)

Writing High-level Plans for TurtleSim

Test the plan

```
TUT> (top-level
       (with-process-modules-running (turtlesim-navigation turtlesim-pen-control)
         (navigate-without-pen '(1 1 0))
         (exe:perform (an action (type drawing) (shape house)))))
```



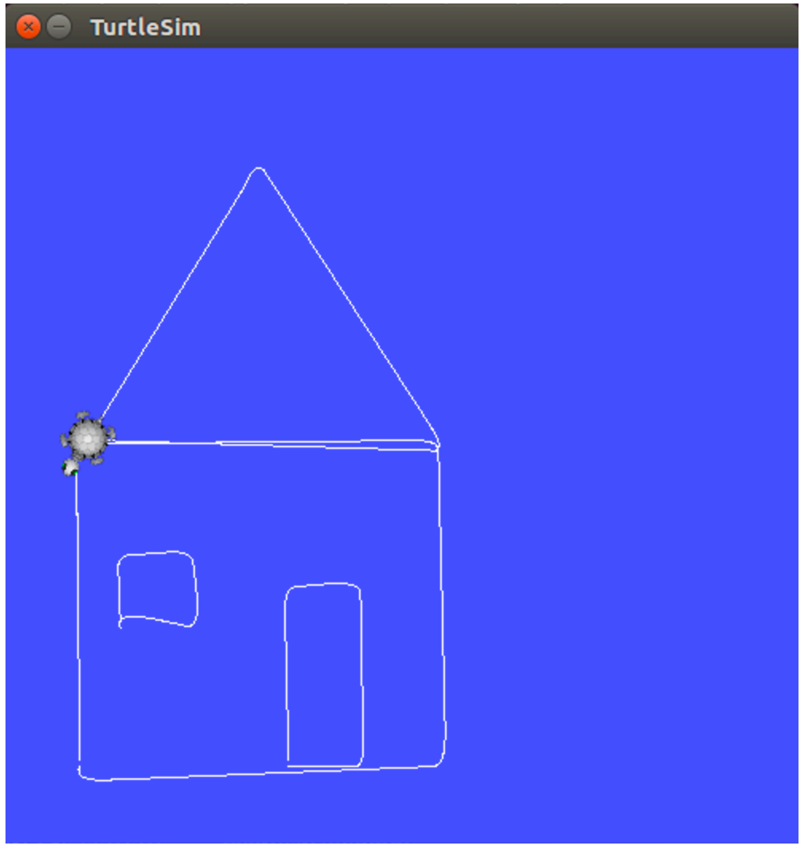
Writing High-level Plans for TurtleSim

Test the plan … we could also do

```
TUT> (top-level
       (with-process-modules-running (turtlesim-navigation turtlesim-pen-control)
         (navigate-without-pen '(1 1 0))
         (draw-house)))
```

Call the function directly rather than resolving an action designator
but only because it is in the body of top-level and with-process-modules-running

But this rather misses the important point that draw-house is a function associated with the resolution of an action designator



Writing High-level Plans for TurtleSim

Test the plan

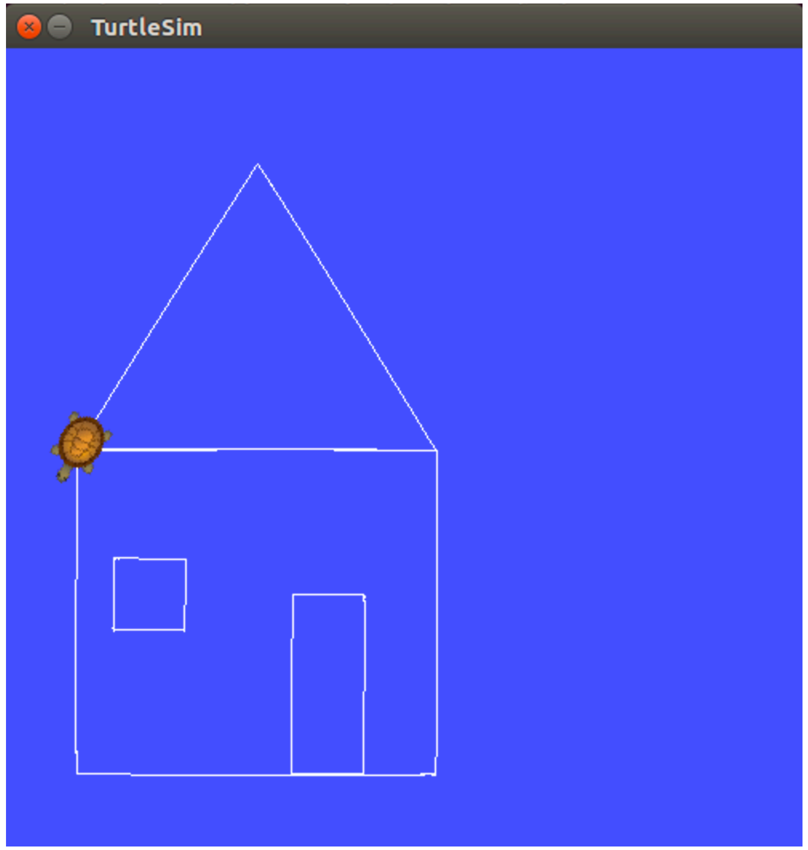
```
TUT> (top-level
               (with-process-modules-running (turtlesim-navigation turtlesim-pen-control)
                 (navigate-without-pen '(1 1 0))
         (exe:perform (an action (type drawing) (shape house)))))
[(TURTLE-PROCESS-MODULES) INFO] 1503577044.541: TurtleSim pen control invoked with motion designator `#<MOTION-DESIGNATOR ((TYPE
                                                  SETTING-PEN)
                                                 (OFF
                                                  1)) {1008B23133}>'.
[(TURTLE-PROCESS-MODULES) INFO] 1503577044.559: TurtleSim navigation invoked with motion designator `#<MOTION-DESIGNATOR ((TYPE
                                                 MOVING)
                                                (GOAL
                                                 (1 1
                                                  0))) {1009A325A3}>'.
[(TURTLE-PROCESS-MODULES) INFO] 1503577047.556: TurtleSim pen control invoked with motion designator `#<MOTION-DESIGNATOR ((TYPE
                                                  SETTING-PEN)
                                                 (OFF
                                                  0)) {10088E73D3}>'.
[(TURTLE-PROCESS-MODULES) INFO] 1503577047.573: TurtleSim navigation invoked with motion designator `#<MOTION-DESIGNATOR ((TYPE
                                                 MOVING)
                                                (GOAL
                                                 (6.001096606254578d0
                                                  1.0863173007965088d0
                                                  0))) {1008E02D63}>'.

[ ... ]

[(TURTLE-PROCESS-MODULES) INFO] 1503577079.916: TurtleSim navigation invoked with motion designator `#<MOTION-DESIGNATOR ((TYPE
                                                 MOVING)
                                                (GOAL
                                                 (1.0146702527999878d0
                                                  5.501473426818848d0
                                                  0))) {1003C1D383}>'.
(T T T)
TUT>
```

Writing High-level Plans for TurtleSim

Test the plan

```
TUT> (top-level
       (with-process-modules-running (turtlesim-navigation turtlesim-pen-control)
         (navigate-without-pen '(1 1 0))
         (exe:perform (an action (type drawing) (shape house)))))
```

This is the result if we replace the move-to function with one based on the divide-and-conquer algorithm we covered earlier in the course, using a threshold of 0.01 on distance



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
Writing plans for the TurtleSim	http://cram-system.org/tutorials/beginner/high_level_plans
Implementing failure handling for the TurtleSim	http://cram-system.org/tutorials/beginner/failure_handling

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`