Introduction to Cognitive Robotics

David Vernon

www.cognitiverobotics.net

Lecture 37

www.cognitiverobotics.net/CR37.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 37

www.cognitiverobotics.net/CR37.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 the TurtleSim
   - Creating process modules
   - Automatically choosing a process module for a motion
   - Using location designators with the TurtleSim
   - Writing high-level plans for the TurtleSim
   - Implementing failure handling for the TurtleSim
   - Writing tests

The CRAM Beginner Tutorials

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

Creating Motion Designators for the TurtleSim

Based on Creating motion designators for the TurtleSim
http://cram-system.org/tutorials/beginner/motion_designators

Creating Motion Designators for the TurtleSim

A designator is a Common Lisp object that

- Contains a sequence of key-value pairs
- Represents a high-level, symbolic description of some aspect of robot's activity
- Is used to infer concrete parameters
  - when needed
  - based on user-specified rules
  - from the context in which the robot operates, and
  - the symbolic description in the designator

Creating Motion Designators for the TurtleSim

Designators are effectively placeholders

- They require runtime resolution
- Based on the current context of the task action

Creating Motion Designators for the TurtleSim

Resolving or [de]referencing a designator

- Returns an object containing the newly resolved values
- That can then be used by the robot to specify some task

Creating Motion Designators for the TurtleSim

Designator resolution is accomplished by

- Querying a priori knowledge embedded in the plan, or
- Querying knowledge in the KnowRob2 knowledge base, or
- By accessing sensorimotor data through the Perception Executive

Creating Motion Designators for the TurtleSim

There are four types of designator

1. Motion designators (e.g. motor command)
2. Location designators (e.g. 3D pose)
3. Object designators (e.g. grasp configuration)
4. Action designators (e.g. goal)

Creating Motion Designators for the TurtleSim

1. Motion designators

- Describe a low-level motion that a robot should take
- These serve as input to process modules

Creating Motion Designators for the TurtleSim

2. Location designators

- Describe locations taking various constraints into account
- for example, reachability, visibility, ...

Creating Motion Designators for the TurtleSim

3. Object designators

- Describe objects on a symbolic level
- What they are, what they can be used for, …

Creating Motion Designators for the TurtleSim

4. Action designators

- Describe a high-level action
- Which can't be accomplished with a simple (single) motion

Creating Motion Designators for the TurtleSim

Preparing to use motion designators

We will need to add a few more dependencies to the tutorial package files

To do this:

- Update the dependencies in package.xml
- Update the dependencies in cram-my-beginner-tutorial.asd
- Update the dependencies in package.lisp

Creating Motion Designators for the TurtleSim

Update the ROS 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$
```

Creating Motion Designators for the TurtleSim

Update the ROS dependencies

Edit package.xml

```
~/workspace/ros/src/cram_my_beginner_tutorial$ emacs package.xml
```

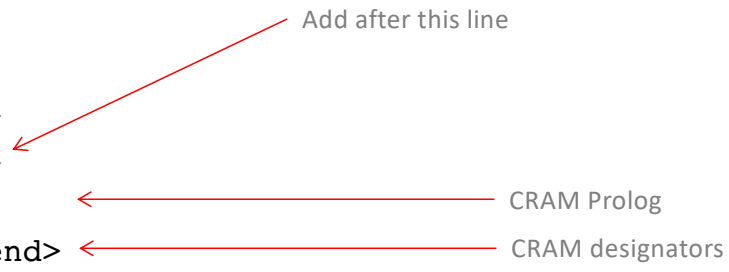
Creating Motion Designators for the TurtleSim

Update the ROS dependencies

Edit package.xml

Add the following lines

```
<exec_depend>cram_language</exec_depend>

<depend>turtlesim</depend>
<depend>roslisp</depend>
<depend>cl_transforms</depend>
<depend>geometry_msgs</depend>
<depend>cram_prolog</depend>
<depend>cram_designators</depend>
```

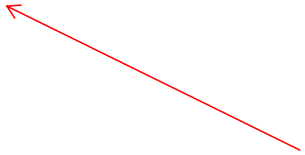


Creating Motion Designators for the TurtleSim

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$
```



You should be there already from the previous step

Creating Motion Designators for the TurtleSim

Update the ASDF dependencies

Edit cram-my-beginner-tutorial.asd

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

Creating Motion Designators for the TurtleSim

Update the ASDF dependencies

Add `cram-designators cram-prolog` in `:depends(...)`

and `(:file "motion-designators" ...)` to `:module(...)`:

```
(defsystem cram-my-beginner-tutorial
  :depends-on (roslisp cram-language
               turtlesim-msg turtlesim-srv
               cl-transforms geometry_msgs-msg
               cram-designators cram-prolog)
  :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"))))))
```

Add this line

Be careful to ensure the open and closing brackets match

The file should now look like this

Creating Motion Designators for the TurtleSim

Update the Lisp package to add `cram-designators` and `cram-prolog` to the namespace

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$
```

Creating Motion Designators for the TurtleSim

Update the Lisp package to add `cram-designators` and `cram-prolog` to the namespace

Edit package.lisp

~/workspace/ros/src/cram_my_beginner_tutorial/src$ emacs package.lisp

Creating Motion Designators for the TurtleSim

Update the Lisp package to add `cram-designators` and `cram-prolog` to the namespace

Edit package.lisp

Add `:cram-designators` to the `(:use :cpl ...)` line and add the `(:import-from ...)` line

```
(defpackage :cram-my-beginner-tutorial
  (:nicknames :tut)
  (:use :cpl :roslisp :cl-transforms :cram-designators)
  (:import-from :cram-prolog :def-fact-group :<- :lisp-fun))
```

Add `:cram-designators` to the namespace so that we don't have to include the package name when we use the functions in the our program

Be careful with the closing brackets

We don't `:use cram-prolog` because it conflicts with some of the similarly-named symbols in the `cram-language` package

Creating Motion Designators for the TurtleSim

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

Creating Motion Designators for the TurtleSim

Create a new Lisp file for the plan code

Edit motion-designators.lisp

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

Creating Motion Designators for the TurtleSim

Update the Lisp package to include the code for the motion designator

Edit motion-designators.lisp

We'll add the code in just a moment …

First, let's do it interactively using the REPL command line

Creating Motion Designators for the TurtleSim

Reload the tutorial in roslisp_repl

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

Creating Motion Designators for the TurtleSim

Create a motion designator in the REPL command line:

```
TUT> (defparameter *my-desig* (desig:a motion (type driving) (speed 1.5)))
*MY-DESIG*
TUT> (desig-prop-value *my-desig* :speed)
1.5
```

Creating Motion Designators for the TurtleSim

Create a motion designator in the REPL command line:

We follow to the normal convention of using the double asterisk for naming global variables in Common Lisp

We use the `a` macro to create designators (internally it uses the `make-designator function`

```
TUT> (defparameter *my-desig* (desig:a motion (type driving) (speed 1.5)))
*MY-DESIG*
TUT> (desig-prop-value *my-desig* :speed)
1.5
```

The designator properties and values

The specified designator type is `motion`

Use `desig-prop-value` function to read the value of a specified property

Creating Motion Designators for the TurtleSim

Let's try to resolve the designator in the REPL command line:

```
TUT> (reference *my-desig*)
Cannot resolve motion designator #<MOTION-DESIGNATOR ((TYPE
                                                       DRIVING)
                                                      (SPEED
                                                       1.5)) {100888CED3}>.
      [Condition of type DESIGNATOR-ERROR]
```

This tells us that there are no rules in place to resolve the designator
... so we need to provide some

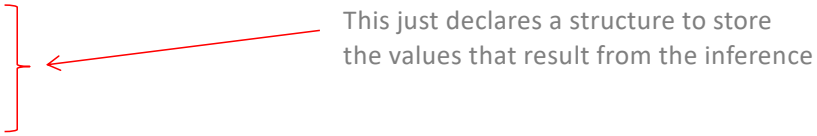
Creating Motion Designators for the TurtleSim

Define inference rules for designators:

- To do this, we add a fact-group `turtle-motion-designators` for referencing motion designators of type `driving`
- The function `reference` uses the CRAM Prolog engine to ground (or resolve) a motion designator into specific motion parameters
- To reference a motion designator, we use the `motion-grounding` Prolog rule:
  - This is called for the given designator
  - The rule binds a tuple of command and specified motion parameters to its second argument
- To see how, add the following code to your motion-designators.lisp file

```
(in-package :tut)

(defstruct turtle-motion
  "Represents a motion."
  speed
  angle)

(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion)))
```

```
(in-package :tut)

(defstruct turtle-motion
  "Represents a motion."
  speed
  angle)

(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion)))
```



```
(in-package :tut)

(defstruct turtle-motion
  "Represents a motion."
  speed
  angle)

(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion)))
```

Define a fact group named `turtle-motion-designators`

`motion-grounding` a public predicate

```
(in-package :tut)

(defstruct turtle-motion
  "Represents a motion."
  speed
  angle)

(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion)))
```

This is the "rule head", i.e. the then-part of the the rule

This is the "rule body", i.e. the if-part of the the rule

Prolog's inference semantics:

'IF there is some assignment to variables such that all elements of the body are true,
THEN use that assignment of variables to evaluate the head'.

Note that in CRAM Prolog for a symbol to be considered a variable, its name must begin with a ?.

```
(in-package :tut)

(defstruct turtle-motion
  "Represents a motion."
  speed
  angle)

(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion)))
```

If the designator contains a key-value pair `(:type :driving)` … and

```
(in-package :tut)

(defstruct turtle-motion
  "Represents a motion."
  speed
  angle)

(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion)))
```

Note: this extracts the value of the argument used when creating the designator

If the designator contain a key-value pair (`:speed` <some variable value>) ... and

```
(in-package :tut)

(defstruct turtle-motion
  "Represents a motion."
  speed
  angle)

(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion)))
```

If the designator contain a key-value pair (`:angle` <some variable value>) ... and

```
(in-package :tut)

(defstruct turtle-motion
  "Represents a motion."
  speed
  angle)

(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion)))
```

the newly created structure

Recall, defining a structure also implicitly defines additional support function

If calling the default function `make-turtle-motion` to create a structure, in this case a `turtle-motion` structure named `?motion`, and assigning the values to the slots, is true

Then ...

```
(in-package :tut)

(defstruct turtle-motion
  "Represents a motion."
  speed
  angle)

(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion)))
```

Then ... associate the drive command (it's just a label) and newly created `?motion` structure (and the extracted values stored in its slots) with the designator `?desig`

More formally, define the value of the `motion-grounding` predicate when applied to `?desig` ...

This value is a list containing the command `drive` and the value of `?motion` (`drive` will be used to identify required operations in the process modules and `?motion` will provide the arguments)

```
(in-package :tut)

(defstruct turtle-motion
  "Represents a motion."
  speed
  angle)

(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion)))
```

Then ... associate the drive command (it's just a label) and newly created `?motion` structure (and the extracted values stored in its slots) with the designator `?desig`

More formally, define the value of the `motion-grounding` predicate when applied to `?desig` ...

This value is a list containing the command `drive` and the value of `?motion` (`drive` will be used to identify required operations in the process modules and `?motion` will provide the arguments)

Here, the inference is very simple:
identify the required command and extract the parameters from the designators

Creating Motion Designators for the TurtleSim

Reload the tutorial in roslisp_repl

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

Creating Motion Designators for the TurtleSim

Create a motion designator in the REPL command line and then reference it:

```
TUT> (defparameter *my-desig2* (desig:a motion (type driving) (speed 1.5) (angle 2)))
*MY-DESIG2*
TUT> (reference *my-desig2*)
(DRIVE #S(TURTLE-MOTION :SPEED 1.5 :ANGLE 2))
```

This time, the designator resolved successfully

We could now use this resolved designator to move the turtle and that's what we'll do in a moment

First, let's add another rule to our turtle-motion-designators fact group and create a fact group for controlling the pen

Creating Motion Designators for the TurtleSim

Add a motion designator for the move plan we developed previously

- Here, we need to designator to resolve to a 3D vector
  (recall that's the argument we provide to `move-to`)

```
(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion))

  ;; move
  (<- (desig:motion-grounding ?desig (move ?motion))
    (desig-prop ?desig (:type :moving))
    (desig-prop ?desig (:goal ?goal))
    (lisp-fun apply make-3d-vector ?goal ?motion)))
```

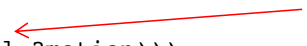
```
(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion))

  ;; move
  (<- (desig:motion-grounding ?desig (move ?motion))
    (desig-prop ?desig (:type :moving))
    (desig-prop ?desig (:goal ?goal))
    (lisp-fun apply make-3d-vector ?goal ?motion)))
```

If the designator contains a key-value pair (`:type :moving`) ... and

```
(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion))

  ;; move
  (<- (desig:motion-grounding ?desig (move ?motion))
    (desig-prop ?desig (:type :moving))
    (desig-prop ?desig (:goal ?goal))
    (lisp-fun apply make-3d-vector ?goal ?motion)))
```

If the designator contain a key-value pair (`:goal` <some variable value>) ... and

```
(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion))

  ;; move
  (<- (desig:motion-grounding ?desig (move ?motion))
    (desig-prop ?desig (:type :moving))
    (desig-prop ?desig (:goal ?goal))
    (lisp-fun apply make-3d-vector ?goal ?motion)))
```



If calling the function `make-3d-vector` to create a structure, in this case a `3d-vector` structure named `?motion`, and assigning the value of `?goal`, is true

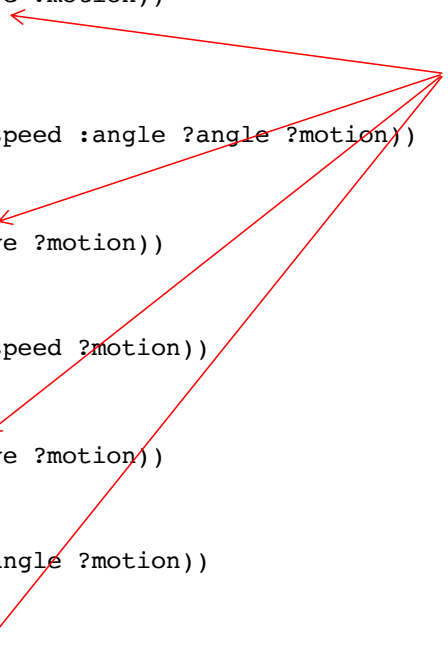
Then ...

```
(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion))

  ;; move
  (<- (desig:motion-grounding ?desig (move ?motion))
    (desig-prop ?desig (:type :moving))
    (desig-prop ?desig (:goal ?goal))
    (lisp-fun apply make-3d-vector ?goal ?motion)))
```

Then ... associate the move command and the newly created `?motion` structure (and the goal value) with the designator `?desig`

More formally, define the value of the `motion-grounding` predicate when applied to `?desig` ... this value is a list containing the command `move` and the value of `?motion`

```
(def-fact-group turtle-motion-designators (motion-grounding)
  ;; for each kind of motion, check for and extract the necessary info

  ;; drive and turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :speed ?speed :angle ?angle ?motion))

  ;; drive
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:speed ?speed))
    (lisp-fun make-turtle-motion :speed ?speed ?motion))

  ;; turn
  (<- (desig:motion-grounding ?desig (drive ?motion))
    (desig-prop ?desig (:type :driving))
    (desig-prop ?desig (:angle ?angle))
    (lisp-fun make-turtle-motion :angle ?angle ?motion))

  ;; move
  (<- (desig:motion-grounding ?desig (move ?motion))
    (desig-prop ?desig (:type :moving))
    (desig-prop ?desig (:goal ?goal))
    (lisp-fun apply make-3d-vector ?goal ?motion)))
```



These commands, i.e. `drive` and move, will be used in the process-modules.lisp file to identify the require operations, i.e. the functions to call to accomplish the motion using the `?motion` argument

Creating Motion Designators for the TurtleSim

Reload the tutorial in roslisp_repl

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

Creating Motion Designators for the TurtleSim

Create a motion designator of type moving in the REPL command:

```
TUT> (desig:a motion (type moving) (goal (1 1 0)))
#<A MOTION
      (TYPE MOVING)
      (GOAL (1 1 0))>
```

We pass as list of number as the goal to the designator (just as we passed a list of numbers to the move-to() function previously.

This make sense because we are going to use this designator with that function later.

Creating Motion Designators for the TurtleSim

Finally, add a motion for controlling the pen:

- To do this, we add another fact-group `turtle-pen-motion-designators` for referencing motion designators of type `setting-pen`

- Add the following code to your motion-designators.lisp file

```
(defstruct pen-motion
  "Represents a pen motion."
  (r 255) (g 255) (b 255)
  (width 1)
  (off 0))

(def-fact-group turtle-pen-motion-designators (motion-grounding)
  ;; for each kind of pen motion, check for and extract the necessary info

  ;; change color, width and on/off status
  (<- (desig:motion-grounding ?desig (set-pen ?motion))
    (desig-prop ?desig (:type :setting-pen))
    (desig-prop ?desig (:r ?r))
    (desig-prop ?desig (:g ?g))
    (desig-prop ?desig (:b ?b))
    (desig-prop ?desig (:width ?width))
    (desig-prop ?desig (:off ?off))
    (lisp-fun make-pen-motion :r ?r :g ?g :b ?b :width ?width :off ?off ?motion))

  ;; change color and width (implicates setting the pen to 'on')
  (<- (desig:motion-grounding ?desig (set-pen ?motion))
    (desig-prop ?desig (:type :setting-pen))
    (desig-prop ?desig (:r ?r))
    (desig-prop ?desig (:g ?g))
    (desig-prop ?desig (:b ?b))
    (desig-prop ?desig (:width ?width))
    (lisp-fun make-pen-motion :r ?r :g ?g :b ?b :width ?width :off 0 ?motion))

  ;; change on/off status (if set to 'on' the pen will have a default color and width)
  (<- (desig:motion-grounding ?desig (set-pen ?motion))
    (desig-prop ?desig (:type :setting-pen))
    (desig-prop ?desig (:off ?off))
    (lisp-fun make-pen-motion :off ?off ?motion)))
```

Creating Motion Designators for the TurtleSim

Reload the tutorial in roslisp_repl

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

Creating Motion Designators for the TurtleSim

Create a motion designator in the REPL command line and then reference it:

```
TUT> (defparameter *my-desig3* (desig:a motion (type setting-pen) (r 100) (g 150) (b 0) (width 5)))
*MY-DESIG3*
TUT> (reference *my-desig3*)
(SET-PEN #S(PEN-MOTION :R 100 :G 150 :B 0 :WIDTH 5 :OFF 0))
```

Creating Motion Designators for the TurtleSim

Now we have designators and know how to define and handle them

Next, we need to use them to control the robot

CRAM process modules allow us to do just that

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

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`