University of Maine
School of Computing and Information Science

COS 470/570: Introduction to AI

Agents & Search Programming Assignment

Fall 2021

Assigned: 9/13/2021 **Due: 10/6/2021**

Contents

1 Introduction **1**

2 The Robot World **2**

2.1 Agents & agent programs in the robot world 2

2.2 Percepts 3

2.3 Actions 4

2.4 Simulation runs 5

2.4.1 Reflex agents 5

2.4.2 Hill-climbing agent, search agents 5

2.5 Performance measure 5

3 Search in another domain **6**

4 The `run-assignment.lisp` file **7**

5 Discussion/Write-up **7**

6 What to turn in **9**

7 Grading **9**

8 Simulator **10**

8.1 Downloading the simulator files 10

8.2 Loading the simulator 10
8.3 Running the simulator 12
8.4 Adding new robot types for this assignment 13

COS 470/570: Agents & Search Programming Assignment

Assigned: 9/13/2021 Due: 10/6/2021

It has been my experience that this program is harder for students than it seems. **START EARLY** so you can get help as you need it! Suggested milestones: 9/20: Uninformed search done; 9/30: Heuristic search done.

1 Introduction

In this assignment, you will create several types of agent programs and test them in a "robot world" simulator that I will provide[1] (described in section 8) and in another domain of your choice.

The agent programs you will write are:

1. a simple reflex agent;
2. a reflex agent that can keep track of the world and it's own actions (i.e, a model-based agent);
3. a hill-climbing agent;
4. a search agent that uses uniform-cost search; and
5. *two* search agents that uses A*, each with a different heuristic function.

For 1 and 2, in the robot world the goal will be for the agent to find a corner and remain there, no matter where the agent is placed in the world. Note that neither of these know anything about the world other than what their sensors (percepts) tell them. In particular, they have no way of knowing where in the world they or their goal is at any time, nor can they recognize even that they *have* a goal: they just behave according to their reflexes. For example, in the robot world (below), if the agent attempts to move into a space where there is an obstacle, it will feel the obstacle via a bump sensor (discussed below), but will not be able to move from its current position.

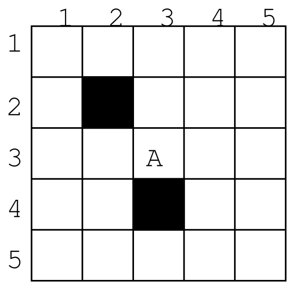
For 3, the agent *will* need to be told where its goal is, and it will need to keep track of where it is itself. The choice of the agent's heuristic is up to you. The agent, however, will *not* know in the robot world where obstacles are ahead of time—that is, it won't have a map of the world.

For 4 and 5, an agent will also need to know were its goal is and where it is to begin with, *and* it will need to be given a map of the world; for example in the robot world, this would list the locations of obstacles and boundaries. Note that search will happen "in the agent's head" for these agents, as per our discussion of a general search agent in class, whereas the others perform their search in the world without planning ahead.

[1] Unlike in prior semesters, where students were sweating bullets writing it in Lisp. (Of course, that just means I'm making you do more assignments. . .)

2 The Robot World

The “Robot World” was mentioned in class. It consists of a rectangular grid of cells, each of which can contain an obstacle or a robot, and the world is surrounded by a wall Although you will want to be able to change the world between trials of your agent program, you should not worry about the world changing during a particular trial other than the robot moving.



In this world, a robot can move one square forward, backward, left or right; and it can turn 90 degrees left or right. It is equipped with a forward-looking sensor that can detect an obstacle directly in front of it, as well as bump (collision) sensors on its front, left, right, and rear sides.

2.1 Agents & agent programs in the robot world

In this world, the simulated agents are simple robots that accept percepts, make decisions, and take action. The simulator (see section 8) includes code to simulate the agent’s body and its interaction with the environment. All you have to do is write the *agent programs* that will solve the problems in this assignment. Your agent program will need to take a *percept string* (also just called a *percept*) consisting of the information from the robot’s sensors and return a next action for the robot to take.

The formats of the percepts and actions are described both in the simulator documentation and below.

As described in section 8 and the simulator documentation, you will create a subclass of the `robot` CLOS class defined by the simulator for each part of the assignment; for example, you might define your reflex agent as:

```
(defclass reflex-agent (robot)
   ...)
```

The simulator calls a method named `agent-program` of whatever robot or robot subclass instance you have added to the simulator instance. You should define versions of `agent-program` for your robot subclasses carry out the parts of the assignment, for example, you might define:

```
(defmethod agent-program ((self reflex-agent) percept)
   ...reflex agent code...
    action)
```

where `action` is the action you want the agent to do next.[2]

[2]Remember, in Lisp you don’t call `return` to return a value: the value of the last form encountered is returned. In

Your code can be completely object-oriented, that is, with all of it being methods of the classes you define. Or you can just have the `agent-program` call a *function* that is the true agent program. It doesn't really matter for the robot world, but if you go the route of defining all your code as methods of your robot subclass, you will have to change everything for your second domain, since you won't be using the robot subclass then.

So I would suggest either implementing all your code as functions or, if you prefer OOP (which I do), then define *another* class (e.g., `reflex-agent`) to organize your code that can be used in either domain. (In this case, you would also want to create in the robot world a class based on robot that would have an instance variable to hold an instance of your agent program class, and the `agent-program` method would then call one of that object's methods.)

In any case, the only instance variable you should access of the base robot class is `name`! (And even there, you're better off using its `name` method.) Your agent program **must not** access the internal state of the robot, the world, or the simulator. Even for a search agent that needs to know where the obstacles are and the size of the world (e.g., an A^* agent), you need to provide that information to the agent yourself, e.g., via instantiating your robot subclass with a "map" in an instance variable. This preserves the separation between what is actually the case in the world (as known by the simulator) and what your agent *thinks* is the case (based on its sensors and its own knowledge, e.g., a map).

2.2 Percepts

The format of the percept given to your agent program is an *association list*—a list of lists—one for each sensor. Each list is composed of the sensor name (a symbol) followed by the current value. The sensors are named `:front-sensor`, `:front-bump`, `:right-bump`, `:left-bump`, and `:rear-bump`, each of which will have a value of `t` or `nil` in each percept.

Here's an example percept:

```
((:forward-sensor t)
 (:front-bump nil)
 (:right-bump t)
 (:rear-bump nil)
 (:left-bump nil))
```

This would correspond to a situation in which there is something directly in front of the robot, and the last action caused it to bump into something on its right side.

Association lists like this are very common in Lisp, especially when you want to have key/value pairs, but don't want a hash table.[3] There is a special Lisp function, `assoc`, that is made for interacting with association lists; for example, if `percept` holds the percept above, then this:

```
(assoc :forward-sensor percept)
```

will return:

fact, if you *do* call `return`, you'll get a runtime error. You *could* call `(return-from foo value)` to return the value `value` from the function `foo`, but why would you?

[3]Okay, technically an association list has elements that are *cons* cells, not lists; the *car* of the element is the key, and the *cdr* is the value. But for some reason, I prefer to put the value in the car of the cdr. So that's what you get.

```
(:forward-sensor t)
```

A common idiom, since we just want the value, not the key/value pair, is:

```
(cadr (assoc :forward-sensor percept))
```

or

```
(first (assoc :forward-sensor percept))
```

You can set a value in an association list using `setf`, e.g.,

```
(setf (assoc :forward-sensor percept) nili)
```

would result in `percept` having the value:

```
((:forward-sensor nil)
 (:front-bump nil)
 (:right-bump t)
 (:rear-bump nil)
 (:left-bump nil))
```

You may be wondering what is going on with those colons, and why something like

```
(assoc :forward-sensor percept)
```

doesn't give an unbound variable error, since `:forward-sensor` isn't quoted. Recall that all symbols are contained in *packages*, such as `cl-user`, `sim`, etc. There is a special package, `keyword`, that has no displayed name, and so if you see a symbol like `:forward-sensor` with a colon but no name before it, it is a keyword. Symbols in the `keyword` package have the very useful property that they all evaluate to themselves. So you can get something like this:

```
CL-USER> :this-is-a-keyword
:THIS-IS-A-KEYWORD
CL-USER>
```

whereas if you had done that with a symbol of any other package, you would have gotten an error.

2.3 Actions

Your agent program will need to return the next action the robot should take. An action is one of the following keywords:

`:nop`: do nothing

`:forward`: move one square forward

`:backward`: move one square backward

`:left:` move one square to the left, keeping the same heading (i.e., facing the same direction)

`:right:` move one square to the left, keeping the same heading (i.e., facing the same direction)

`:turn-right:` turn right 90 degrees, staying in the same square

`:turn-left:` turn left 90 degrees, staying in the same square

If there are other actions you want to try out, the simulator documentation describes how, but for this assignment, that's it.

2.4 Simulation runs

2.4.1 Reflex agents

Run each agent on three different kinds of 25 × 25 grids, with the goal being to end up in a corner and stay there:

1. worlds without obstacles;
2. worlds with obstacles, but in which there are no "fake corners", neither formed by obstacles nor by an obstacle and a wall; and
3. worlds where fake obstacles occur.

Run the agents multiple times for each, varying where the agent begins, the number and location of obstacles (for the last two kinds of worlds), etc.

Note: **The agent must remain in a corner** in order for a run to count as a success—the simulator **cannot** just stop the agent when it notes that it has entered a corner. The agent has neither a stop action nor a corner sensor, so you'll need to figure out some way to make it stay in a corner once it finds one. Your simulator *may* stop the simulation after it detects that the agent has remained stationary for some number of time units.

2.4.2 Hill-climbing agent, search agents

Run the agents on the same kinds of worlds as above, varying where the agent and the goal are located each time. As discussed below, for the search agents, you will need to run each agent on each of 20 simulated worlds.

2.5 Performance measure

You will need to implement some performance measure(s) for each task and agent. The simplest performance measure for a reflex agent, for example, is whether or not the agent found and remained in a corner. A better measure would reward agents based on shorter paths they took. An even better measure would take into account how the agent's path compares to the optimal path. (Note that you can easily find the optimal path length using your A^* program.)

For the three goal-based search agents (uniform-cost and the two A^* agents), you need to somehow measure the search work they did, since they should both traverse optimal paths in this domain. A good approach here is to note the number of search nodes they expand or create or how many edges of the search space the algorithm traverse. You should also consider keeping track of an estimate of the maximum space used; this can be done by tracking the maximum size of the searches queue or open list, for example.

3 Search in another domain

Choose another domain in which to implement the two goal-based searches (uniform cost and A*) to compare with each other and to your robot world. This should be one of the traditional AI toy search problems:

- Foxes and chickens: You have n foxes and n chickens that you want to get from one side of a river to another, but your canoe will hold only you and two animals. If at any time the number of foxes $>$ number of chickens on one side of the river, the foxes will eat the chickens.
- Towers of Hanoi: You have three pegs with three disks, a small, medium, and large one. Initially, the disks are stacked on the leftmost peg with the large one on the bottom and the small one on top. You want all the disks on the rightmost peg, but at any point, you can move only one disk, and you can never stack a larger disk on a smaller one.
- Cryptarithmetic: You have an arithmetic problem, but instead of numerals, you have letters that form words. For example:

  ```
   SEND
  +MORE
  -----
  MONEY
  ```

  Your task is to find an assignment of numerals to letters so that (1) no two letters correspond to the same numeral and (2) the arithmetic problem is valid.
- Traveling salesman: Given n cities, each of which is connected to all the others (i.e., they form a fully-connected graph) and the distances between each pair (the weights of the edges), find the shortest path that starts at a city, visits each other city exactly once, and ends up back at the start city.
- 8-puzzle: This is the sliding tile puzzle you've seen where you have to arrange the numbered tiles in order (e.g.).
- 8 queens: Given a chessboard and 8 queens, how do you arrange the queens so none of them can attack any of the others?
- Water jug problem: Given two water jugs, one that holds 3 gallons and one that holds 4, how do you get exactly 2 gallons in the 4-gallon jar, with only the operations: fill a jar at the faucet; pour all the contents of a jar onto the ground; and empty one jar into the other?
- Monkeys and bananas: Given a room containing a monkey, a box, and some bananas that the monkey can only grab if it is standing on the box, how do you go from state of the monkey being on the ground with no bananas to the state of the monkey having the bananas? Your only operations are move (the monkey); push the box; climb on the box; and grab the bananas if they are in reach.

- Blocks world problems: For example, in a world containing an infinite table (available on Wayfair?), some blocks labeled A, B, and C, and a robot arm, how do you go from the state of, say A being on B and B and C on the table to the state of all blocks stacked up with A on top and C on the bottom? Your only actions are grasp a block (with the arm), pick up a block from the table, unstack a block from on top of another, stack a block on top of another, and put a block on the table; the arm can only hold one block at a time, it can only pick up a block or unstack a block if the block has nothing on top of it, and it can only stack a block on another if the other one has nothing on top of it.

You should pick a problem that will help you to understand more about uniform cost search and A*. You will be asked to justify your selection in the write-up for this program. You will also need to vary the problem for test runs, so you should choose a domain where the initial and goal states can be easily varied to create different problems. Choose a reasonable heuristic for the A* search; you don't need to choose two different ones for this domain.

4 The `run-assignment.lisp` file

Create a file named `run-assignment.lisp` that:

1. Loads your assignment solution files, including the `simulator.lisp`, `messages.lisp`, and `new-symbol.lisp` files discussed below in section 8.
2. Defines a function, `run-robots`, that demonstrates *all* your robot world agents running – i.e., for each, creates a simulator, adds the agent, and runs the simulator until the agent encounters the stopping criterion, showing each step it takes via `world-sketch`. (If you have a nicer way of demonstrating how the agent runs, include screenshots of that in your write-up.)
3. Defines a function, `run-other`, that demonstrates your searches in the other domain you chose.

As part of grading your assignment, we may load this file and call each of these functions.

5 Discussion/Write-up

For this assignment, you will turn in not only the code, but a write-up that describes and discusses your results. This will need to include:

- thorough discussion of your agent programs;
- discussion of your simulation runs and their results;
- discussion of your second domain, including how you had to change your programs to work in the domain and the results of running your programs;
- comparison of the different agent programs' performance in the robot world; and
- comparison of searching and search results in the second domain and in the robot world.

Your write-up should address at least the following questions:

1. What did you have to change as you changed agent types and tasks? Do these changes tell you anything interesting about the nature of these types of agents?

2. Would you think your agents would "scale-up"?

   (a) Would your agents be able to perform the tasks on a larger grid?

   (b) Would your agents be able to perform the tasks with (a lot) more obstacles?

3. What is your second domain? What operators did you use, what is different in the test runs, what heuristic functions did you use, and why did you choose it?

4. How did the results compare?

   (a) Compare the number of nodes created for uniform cost and for A^* (for each heuristic) by running each 20 randomly-generated environments; i.e., for each of the environments, run both uniform cost search and the two A^* agents. You will want to look at both the average number of nodes created for all 20 runs and the number created for each individual run. You will need to do statistical analysis of the results; I'd suggest something like a paired t-test, which Excel should be able to do for you.[4] You should report your results along with a the confidence you have in them, expressed as a p-value. For example, "BFS produced more nodes than A^* ($p < 0.05$)," which means, roughly, that there is only a 1 in 20 chance that the results are due to chance alone. Do this for run times, too. How can you explain the results?

   (b) Compare the number of nodes created and the run times for A^* when different heuristics are used. You will want to look at both the average number of nodes created for all 20 runs above, and you may need to look at the number created for each individual run. Compare run times as well. Again, use statistical analysis to bound the confidence in your conclusion. How can you explain the results?

   (c) Are there other variables that you would like to compare? If so, what are they and how would you run tests to compare them?

5. How did your search agents compare to your reflex agents?

6. What have you learned, or demonstrated, that might be useful when implementing AI systems in the real world?

A good organization for your write-up is something like the general format of scientific papers:

- Introduction/Background – What are you trying to do? Etc.
- Methods – How did you do it? Include here a discussion of your programs, any discussion of your statistical techniques, programs, etc., that will be useful. (You must tell me the statistical basis for your conclusions.)
- Results – Describe the data obtained.
- Discussion – Conclusions, etc.

[4] If it can't, and you don't have access to any other statistical software, then see me—we have a copy of a Lisp-based statistics package locally.

There should also be sufficient comments/documentation in your code that I can understand how your program works. Alternatively, you can use a literate programming system to provide a detailed description of your code in lieu of comments in the source directly. If you do that, make that part of your write-up (e.g., an appendix).

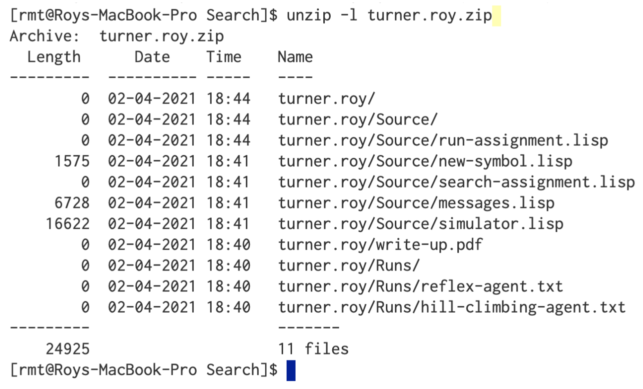
Grammar and spelling, as well as tone (i.e., this is a more or less formal paper, so no colloquialisms, etc.), will count toward your grade.

6 What to turn in

A zipfile containing:

1. Your write up, in **PDF form only**. No Word, plain text, OpenOffice, etc., will be accepted. This should be named `write-up.pdf`.
2. Example runs for each agent performing each task; these should be in files in a subdirectory (i.e., a sub-folder) named `Runs`, and each run should be named so that it is immediately clear which agent and domain it pertains to.
3. Your code, in a subdirectory named `Source`. Put your `run-assignment.lisp` file *and* the copies of `simulator.lisp`, `messages.lisp`, and `new-symbol.lisp` you used in this subdirectory.

So if I were to call `unzip` to look at the contents of your zipfile, it should look something like this:



```
[rmt@Roys-MacBook-Pro Search]$ unzip -l turner.roy.zip
Archive:  turner.roy.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  02-04-2021 18:44   turner.roy/
        0  02-04-2021 18:44   turner.roy/Source/
        0  02-04-2021 18:44   turner.roy/Source/run-assignment.lisp
     1575  02-04-2021 18:41   turner.roy/Source/new-symbol.lisp
        0  02-04-2021 18:41   turner.roy/Source/search-assignment.lisp
     6728  02-04-2021 18:41   turner.roy/Source/messages.lisp
    16622  02-04-2021 18:41   turner.roy/Source/simulator.lisp
        0  02-04-2021 18:40   turner.roy/write-up.pdf
        0  02-04-2021 18:40   turner.roy/Runs/
        0  02-04-2021 18:40   turner.roy/Runs/reflex-agent.txt
        0  02-04-2021 18:40   turner.roy/Runs/hill-climbing-agent.txt
---------                     -------
    24925                     11 files
[rmt@Roys-MacBook-Pro Search]$
```

Note: All `.lisp` files must be in plain text.

Name your zipfile `last-first.zip`, where "last" is your last name and "first" is your first name.

You will turn the zipfile in via BrightSpace.

7 Grading

Your programs will account for 75% of your grade, with the rest being the write-up. I will post rubrics for grading both.

8 Simulator

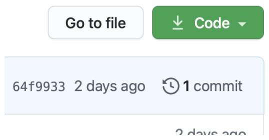
The simulator code is in the class Github repository (github.com/rmt59/cos470). Documentation is provided in the Github repository for using the simulator (as `simulator.pdf` for the simulator, `messages.pdf` for some output utilities, and `new-symbol.pdf` for a utility to generate custom new symbols, such as robot names).

The simulator defines a base CLOS (Common Lisp Object System) class, `robot`, that you will need to create a subclass of to have it run your agent programs. You should create a subclass of `robot` for each part of the assignment, each with its `agent-program` method that takes a percept and returns an action when called.

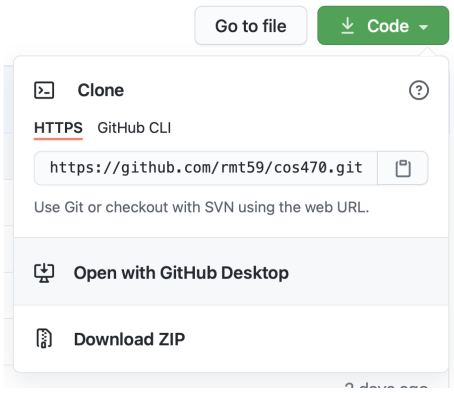
8.1 Downloading the simulator files

If you are familiar with GitHub, just download the repository in the normal way. The branch you want is "master".

For those of you who aren't familiar with it, go to the site github.com/rmt59/cos470. You'll see a list of the files present in the repository along with (if you scroll down) a README file's contents. You will see this a bit down from the top of the screen:



Click the "Code" button. You will then see:



https://github.com/rmt59/cos470.git

Use Git or checkout with SVN using the web URL.

Click on "Download ZIP", unzip the downloaded file, and you're good.

8.2 Loading the simulator

Copy the `messages.lisp`, `new-symbol.lisp`, and `simulator.lisp` files into the directory containing the code you are writing. If you just want to use the simulator, you can load it (at the Lisp prompt or in your code) by:

```
(load "simulator")
```

You can now access all the externally-visible symbols, methods, etc., in the `simulator` package defined in that file. This means that if you want to make a new simulator using the `create-simulator` function, you would need to tell it what package that function is in:

```
(sim:create-simulator)
```

where `sim` is a nickname you can use for the `simulator` package. This is very similar to how packages/modules are loaded and used in Python.

If you don't want to always have to preface everything with `sim:`, then you can do:[5]

```
(use-package 'simulator)
```

and you can then just do, e.g.,

```
(create-simulator)
```

Note that there are quite a few methods in `simulator.lisp` that you may want to use, so you may want to read through `simulator.pdf`.

If you want to use the messages facility so you can set different levels of verbosity for your code (e.g., one setting for normal messages, one for debugging messages, etc.), you can do:

```
(load "messages")
```

to have access to things like:

```
(msg:fmsg "regular message")
(msg:dfmsg "only shows up when verbosity is :debugging or greater")
```

or, if you don't want to preface everything with the package nickname, after loading it do:

```
(shadowing-import 'msg:msg)
(use-package 'message)
```

The first line is needed for SBCL and probably some other Lisps that may already have the symbol `msg` defined for some reason; then you can do:

```
(fmsg "regular message")
(dfmsg "only shows up when verbosity is :debugging or greater")
```

See the documentation in `messages.pdf` for more information.

[5] If you get an error about packages and symbols, see the documentation file `simulator.pdf` for information on how to continue.

8.3 Running the simulator

In order to run the simulator, you need to create a new simulator instance; assuming that you have done the `use-package` thing above, you can do this by:

```
(setq sim (create-simulator))
```

where `sim` is whatever variable you want to hold the simulator instance.

`create-simulator` takes three optional, keyword arguments:

- `:size` – what size simulated world you want
- `:num-obstacles` – how many randomly-placed obstacles you want in the world
- `:obstacle-locations` – where you want obstacles placed

The size is specified as a list of the form `(r c)`, for `r` rows and `c` columns. The world's southwest corner is `(1 1)` and its northeast corner is `(r c)`. The obstacle arguments can be used together to create some number of random obstacles in addition to some number for which you specify the locations. For example, to create a 10×10 world with 10 random obstacles and another at location `(3 4)`, you would do:

```
(setq sim (create-simulator :size '(10 10)
    :num-obstacles 10
    :obstacle-locations '((3 4))))
```

To see a rudimentary view of what the simulated world looks like, do:

```
(world-sketch sim)
```

assuming `sim` contains a simulator instance. This would result in something like this:

```
++++++++++++
+..........+
+..@.......+
+.@........+
+..........+
+....@.....+
+.........@+
+.@........+
+..........+
+@..@...@.@+
+....@.....+
++++++++++++
```

where each `@` is an obstacle. (See the documentation for how to change the symbol used.)

To run the simulator, use the `run` method:

```
(run sim)
```

This runs the simulator for 1 "clock-tick", i.e., giving each robot a chance to move once. To run for more clock ticks, do:

```
(run sim :for 10)
```

for example. If you want to see what the world looks like after each clock tick, do:

```
(run sim :for 10 :sketch-each t)
```

8.4 Adding new robot types for this assignment

To add a new robot type, you will create a subclass of `robot`. For example, suppose you are creating your reflex agent; you might do:

```
(defclass reflex-agent (robot)
    ...any new instance variables you need...
 )
```

The simulator will call the `agent-program` method of any instance of this object in the simulator each "clock tick"—that is, each simulator cycle. You will need to define this method for your new class, e.g.,

```
(defmethod agent-program ((self reflex-agent) percept)
   ...your code...
   ...suppose it decides to go foward...
   :forward)
```

As an example to follow if you need to, I've included in `simulator.lisp` a robot that moves randomly. The `robot` subclass for it is defined as:

```
(defclass random-robot (robot) ())
```

and the agent program is:

```
(defmethod agent-program ((self random-robot) percept)
  (with-slots (name) self
    (let ((next-action
           (car (nth
                 (random (length *robot-command-map*))
                 *robot-command-map*))))
      (dfmsg "[~s: percept = ~s]" name percept)
      (dfmsg "[~s: choosing ~s as next action]" name next-action)
      next-action)))
```

To add an instance of the robot you've created to the simulator, you would do something like either:

```
(setq r (make-instance 'reflex-agent))
(add-robot sim :robot r)
```

or:

```
(add-robot sim :type 'reflex-agent)
```

where `reflex-agent` is the subclass of `robot` and `sim` contains a simulator instance.

The `add-robot` method has the optional, keyword arguments:

- `:orientation` – which direction the robot is initially facing (one of `:north`, `:south`, `:east`, or `:west`)
- `:location` – where to place the robot; otherwise it is placed randomly in the world
- `:name` – if you want to name the robot (the name shows up in messages from the simulator); otherwise, the robot is named something like `ROBOT3`

If you call `world-sketch` after adding a robot, you'll see something like:

```
++++++++++++
+..........+
+..@.......+
+.@........+
+..........+
+....@.....+
+.........@+
+.@........+
+..........+
+@..@...@.@+
+>...@.....+
++++++++++++
```

The robot appears at (1 1) in this example, as shown by its default icon `>`, indicating it is pointed `:east`.

For additional information about methods that may be useful when debugging, etc., see the `simulator.pdf` documentation file.