

A Simple Robot Simulator: simulator.lisp

Roy M. Turner

Spring, 2021

Contents

<i>Description</i>	1
<i>Loading the simulator</i>	1
<i>If you get an error about package/symbol problems</i>	1
<i>Creating a simulator</i>	2
<i>Creating a new robot type</i>	3
<i>Percept format</i>	4
<i>Adding new percept components</i>	6
<i>Adding new actions</i>	6
<i>Adding your robot to the simulator</i>	7
<i>Changing the world</i>	7
<i>Simulating your work</i>	7
<i>Miscellaneous methods</i>	11
<i>Code</i>	12
<i>Package setup</i>	12
<i>Global variables</i>	12
<i>Classes</i>	13
<i>Simulator methods</i>	14
<i>World methods</i>	22
<i>Object methods</i>	25
<i>Robot methods</i>	25
<i>Example: random-robot</i>	26

A Simple Robot Simulator: simulator.lisp

Roy M. Turner

Spring, 2021

Description

This is a very simple “robot world” simulator for use by COS 470/570 students for the search assignment. It allows you to define a rectangular world and add some obstacles and one or more robots. It provides a base class for robots you will define: basically all you have to do is to define a robot class based on `robot` that has at least the method `agent-program` that you define to carry out the search program you’re interested in; this method will accept a percept in the form described below and provide an action from among the ones defined in `*commands*`. You can then use the `run` function of the simulator to test your agent.

There are also functions available give you all of the obstacle locations for when you implement your A^* search, as well as a very simple function (`world-sketch`) to show you an overview of the current world.

Loading the simulator

To load the simulator, make sure that the files `message.lisp`, `new-symbol.lisp`, and `simulator.lisp` are in Lisp’s current directory (usually the one you start Lisp in, and the one where your code lives). Then just do:

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

The simulator is in its own package, `simulator`, which has the nickname `sim`. Thus you either need to preface all of the simulator-related functions (below) with the package name or nickname, like:

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

or import the symbols you are interested in using, e.g.:

```
(import '(sim:create-simulator sim:run))
```

or import *all* exported symbols from the package:¹

```
(use-package 'sim)
```

If you get an error about package/symbol problems

Depending on your Lisp, you may already have a symbol in the current package you’re using that has the same name as one of the

¹ Note that although loading `simulator.lisp` will load the message handler and symbol-creation packages, importing from the `simulator` package *doesn’t* import from those packages. For that, you will have to do something like `(use-package 'message)` and `(use-package 'newsymbol)`.

exported (external) symbols in one or more of the other packages you're importing symbols from, which will result in an error. For example, with my setup (macOS, SBCL), if I load the "messages.lisp" file, then try to use-package, I get this error:



If this happens, after you load the file you can use `shadowing-import` to get around this problem. Suppose that you get an error when calling `use-package` for the simulator package that tells you the symbol name is in conflict with an existing one in your current package (usually `cl-user`). You can fix this by doing the following:

```
(shadowing-import 'sim:name)
(use-package sim)
```

You'll want to make sure that whatever the symbol name had been used for in your current package is not important, though, since you'll no longer be able to access it (except, perhaps, as `cl-user::name`). If it was important, you probably want to change to a different name for it.

Creating a simulator

In order to create a new simulator, you use the `create-simulator` function, which has the following format:

```
create-simulator &key (size '(10 10)) (num-obstacles 0)
                    (obstacle-locations nil)
```

That is, the default size is 10×10, and no obstacles are added by default when you do:

```
(create-simulator)
```

You can override these defaults, of course. To make a different-sized world, e.g.:

```
(create-simulator :size '(50 50))
```

or to add 10 obstacles:

```
(create-simulator :size '(50 50) :num-obstacles 10)
```

Obstacles created this way will be put in random locations. If you want to put obstacles in particular places, you can do something like:

```
(create-simulator :size '(50 50)
                  :obstacle-locations '((1 1) (3 4) (10 10)))
```

Note: The (x, y) coordinates for the world are 1-based, not 0-based.

You can combine these as well:

```
(create-simulator :size '(50 50) :num-obstacles 10
                  :obstacle-locations '((1 1) (3 4) (10 10)))
```

will add 10 random obstacles as well as at the three specified locations.

You will want to put the simulator instance returned by this into a variable, since you'll need it later to do anything:

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

Creating a new robot type

To run your agent code, you'll need to create a new kind of robot and add it to the simulator. I have provided a base class for you to use, `robot`. The base class has instance variables for the robot's name (`name`), current location (`location`), current orientation (`orientation`, one of `:north`, `:south`, `:east`, or `:west`), the last percept seen (`percept`), the next action the agent program has selected (`next-action`), the previous action (`prev-action`), and the success status of the previous action (`prev-action-success`, one of `t` or `nil`).

You should not in general, however, access these yourself from your agent program, since these are *simulation* values, not information the agent program knows. For example, you may want your agent program, for model-based and goal-based agents, to have and maintain its own idea of where it is. This may differ from the real location due to noise or other problems with sensors. However, for your goal-based agent assignment, where you will be using A^* and other search techniques, you may want to just assume no noise and use objects' and the robot's real positions.

You want your agent program—i.e., your AI code—to be run automatically by the simulator at each “clock tick”. The simulator is designed to call a `clock-tick` method of each object (obstacles, robots) for each of its own clock ticks after figuring out what that object should see of the world (i.e., its percept). For objects that are not active or are stationary, this is essentially a dummy method. For a robot class inheriting from the base robot class, the clock tick function calls the class’ `agent-program` method, giving it the current percept. The `agent-program` method determines what the next action should be and returns it, and the `clock-tick` both sets the robot’s `next-action` instance variable and returns the next action to its caller. The simulator’s own `clock-tick` method then continue by calling a method (`take-action`) to simulate the effect of the robot’s `next-action`.

To run your code, you will need to create another robot class based on `robot` and define its `agent-program` method to call your code. (In fact, you will create a different robot class for each of the parts of the assignment, most likely.) I have provided a sample robot class, `random-robot`, that you can look at (below or in `simulator.lisp`) to see how to do this.

For example, suppose you have written a reflex agent program, named `reflex` that takes a percept and returns an action to take. Then all you need to do is:

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

(defmethod agent-program ((self reflex-agent) percept)
  (reflex percept))
```

Note that for other kinds of agents, you may need to have a bit more code in `agent-program` to give your agent program code additional information about the world (e.g., the location of objects in the world).

Percept format

For the search assignment, the robots have a very limited repertoire of sensors: just a forward-looking sonar-type thing that can sense what is directly in front of the robot and four bump sensors, one on each side and in the front and rear, that can detect whether or not the robot bumped into something due to the *previous* command. This information is calculated by the simulator’s `clock-tick` method and put into the robot’s `percept` slot just prior to calling the robot’s own `clock-tick` method.

The format of the percept 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.²

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. 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:

```
(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) nil)
```

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)
```

² I know, this is a very verbose and redundant way to provide percepts (for example, no two bump sensors can be `t` at the same time, etc.), but it's easy for you to use.

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.

Adding new percept components

You can add new percept components to robots you define based on robot. The robot class has an instance variable, `percept-map`, that contains an association list with elements of the form:

```
(sensor-name method)
```

where `sensor-name` is a keyword that names the sensor—and that will show up in the percept—and `method` is the method to use to compute its value. The method, which is called by `calculate-percept` (see the code below), must take two arguments, a simulator instance and a robot (or your derived, robot-based class), and it needs to return the sensor's value. You can either specify the sensors you want directly in your robot class' `percept-map` variable, or you can just add it to the global variable `*robot-percept-map*`, since `robot` itself sets its `percept-map` to that value.

If you do the latter, though, *don't* list a value for `percept-map` in your class definition! That will override `robot`'s. You're better off, actually, not listing `percept-map` among the variables you define for your class unless you *do* want to override the default value.

Adding new actions

You may also want to add actions for the robot that are not provided by the standard robot class. Actions are carried out according to the `command-map` instance variable of the robot; as you can see from the code, this is set for `robot` to be the value of the global variable `*robot-command-map*`. A command map should be an association list (see above) whose elements are of the form:

```
(cmd method)
```

where `cmd` is the name of the action (or command) your agent program specifies when it returns and `method` is a method to carry out the command. This method needs to accept two arguments, an instance of `simulator` and an instance of `robot` (including your robot-derived class); it should return `t` if it succeeds and `nil` if not. These methods are called by the `take-action` method (see the code below).

You can add your own action/method pairs to `*robot-command-map*` when you define your robot classes, if you like, since they will inherit from `robot`, which uses the value of the variable when instantiated as its own internal command map. You can also define your own in your robot class.

Adding your robot to the simulator

Suppose we have the `reflex-agent` as defined above. To add an instance of it to the world at a random location, we can just do this (assuming `sim` contains a simulator instance):

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

This will create a new instance of `reflex-agent` for you. You can instead specify an existing instance by:

```
(add-robot sim :robot my-robot)
```

The `add-robot` method has additional parameters to allow setting the location (`:location`), orientation (`:orientation`), and name (`:name`, which defaults to a new symbol based on `robot`).

Changing the world

There are various methods that you can use to change the world. For example, you can add an object (`add-object`), find an object (`find-object`), delete an object (`remove-object`), clear the entire world while leaving the simulator state alone (`clear`), and reset the simulator completely (`reset-simulator`, although why not just create a new instance?). See the definitions below.

Simulating your work

The major function to use to run your simulation is just `run`. Original, no? This has two parameters, both keyword (and thus optional):

- `:for` – how many clock-ticks to run for
- `:sketch-each` – show the state of the world after each clock tick

So if you want to run it for 10 seconds (if that's what you want clock-ticks to be):

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

With my random robot example, doing this will give:

```
SIM> (run s :for 10 :sketch-each t)
```

```
ROBOT0: Moving to (8 2).
```

```
+++++
```

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

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

```
+.....+
```

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

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

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

```
+.....+
```

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

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

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

```
+++++
```

```
ROBOT0: Moving to (9 2).
```

```
+++++
```

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

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

```
+.....+
```

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

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

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

```
+.....+
```

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

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

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

```
+++++
```

```
ROBOT0: Turning right, new orientation = :NORTH.
```

```
+++++
```

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

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

```
+.....+
```

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

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

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

```
+.....+
```

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

```
+.....^..+
```

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

```

+++++++
+++++++
+. . . . .@.@+
+. . . . .@+
+. . . . .+
+. . . . .@...+
+@. . . . .+
+. . . .@. . .+
+. . . . .+
+@. @. @. . .+
+. . . . .^.+
+..@. . . .+
+++++++

```

ROBOT0: Moving to (9 3).

```

+++++++
+. . . . .@.@+
+. . . . .@+
+. . . . .+
+. . . . .@...+
+@. . . . .+
+. . . .@. . .+
+. . . . .+
+@. @. @. . .^.+
+. . . . .+
+..@. . . .+
+++++++

```

ROBOT0: Moving to (8 3).

```

+++++++
+. . . . .@.@+
+. . . . .@+
+. . . . .+
+. . . . .@...+
+@. . . . .+
+. . . .@. . .+
+. . . . .+
+@. @. @. . .^.+
+. . . . .+
+..@. . . .+
+++++++

```

ROBOT0: Moving to (9 3).

```

+++++++
+. . . . .@.@+
+. . . . .@+
+. . . . .+

```

```

+.....@...+
+@.....+
+....@.....+
+.....+
+@.@.@...^.+
+.....+
+..@.....+
+++++++
ROBOT0: Moving to (9 2).
+++++++
+.....@.@+
+.....@+
+.....+
+.....@...+
+@.....+
+....@.....+
+.....+
+@.@.@...+
+.....^.+
+..@.....+
+++++++
ROBOT0: Moving to (8 2).
+++++++
+.....@.@+
+.....@+
+.....+
+.....@...+
+@.....+
+....@.....+
+.....+
+@.@.@...+
+.....^.+
+..@.....+
+++++++
+++++++
+.....@.@+
+.....@+
+.....+
+.....@...+
+@.....+
+....@.....+
+.....+
+@.@.@...+
+.....^.+

```

```

+..@.....+
+++++++
NIL
SIM>

```

I have provided a (very) simple way to show the world, examples of which were just shown. This is the `simulator` method `world-sketch`. It has keyword arguments that allow you to change what empty characters look like (`:empty-char`), what the side walls look like (`:side-wall-char`), and what the top and bottom look like (`:topo-bottom-char`).

The character output for each object is obtained by this method by calling each object's `icon` method, which should return a single character. The robot version of this outputs a pointer-like symbol to indicate its orientation.

Miscellaneous methods

Here are some additional simulator methods are provided that you may find useful. I've listed them like you would call them, assuming `sim` contains a simulator instance.

- `(random-location sim)` → a random location (`x y`) in the world
- `(random-empty-location sim)` → a random location that happens to be empty
- `(next-location sim loc dir)` → the adjacent location to `loc` in the direction `dir`
- `(opposite-location sim dir)` → the opposite direction from `dir`
- `(clockwise-direction sim dir)` → the direction clockwise from direction `dir`
- `(counterclockwise-direction sim dir)` → the direction counterclockwise from direction `dir`

And here are some world methods you may find useful; the following assumes `w` contains an instance of `world`:

- `(objects w)` → list of object instances in the world
- `(object-locations w)` → list of all locations occupied by an object
- `(empty? w loc)` → `t` if the location is empty, `nil` otherwise
- `(in-bounds? w loc)` → `t` if location is inside the world, `nil` otherwise
- `(add-object w object)` → adds the object (or robot or ...) instance to the world
- `(clear w)` → removes all objects from world
- `(size w)` → size of the world (as two-element list)
- `(delete-object w object)`, `(remove-object w object)` → (synonyms) remove the object from the world

- (find-object w x) → returns the object if found, nil otherwise; x can be an object (and so will return non-nil if the object is in the world), a location (returns the object at that location), or the name of an object (a symbol)
- (world-array w) → returns an array representing the world, with icons for objects (using the objects' icon methods) and nil everywhere else; used by world-sketch
 ((export '(objects empty? in-bounds? add-object clear object-locations size delete-object find-object remove-object world-array))

Code

In the code below, I have split up the action of exporting symbols so that you can better see which ones are available to you to import; look for lines that look like:

```
(export ...)
```

Package setup

Here is the package setup; see above for how to load the package and use it's exported symbols. As mentioned, this package uses a couple of others, and the shadowing-import function's use is also explained above.

```

1 (unless (find-package "SIM")
2   (defpackage "SIMULATOR"
3     (:use "COMMON-LISP")
4     ; (:shadowing-import-from "COMMON-LISP" "NAME")
5     (:nicknames "SIM"))
6   )
7
8 (in-package sim)
9
10 (shadowing-import '(NAME) :cl-user)
11
12 (load "new-symbol")
13 (use-package 'sym)
14 (load "messages")
15 (shadowing-import 'msg:msg)
16 (use-package 'message)
```

Global variables

The first of these just lists the directions the simulator/world deals with. The second is a map (well, an association list) that maps from

robot actions (e.g., :right) to methods that carry out those actions (e.g., do-move-right). The third is a similar map for percepts. See above for more information about both of them.

```

17 (defvar *directions* '(:north :south :east :west))
18
19 (defvar *robot-command-map*
20     '(:nop do-nop)
21       (:forward do-move-forward)
22       (:backward do-move-backward)
23       (:left do-move-left)
24       (:right do-move-right)
25       (:turn-right do-turn-clockwise)
26       (:turn-left do-turn-counterclockwise)))
27
28 (defvar *robot-percept-map*
29     '(:front-sensor forward-sensor)
30       (:front-bump front-bump-sensor)
31       (:rear-bump rear-bump-sensor)
32       (:right-bump right-bump-sensor)
33       (:left-bump left-bump-sensor)))
34
35 (export '(*robot-command-map* *robot-percept-map* *directions*))

```

Classes

Since some classes are referenced by methods of other classes, the classes should be created first.

```

36 (defclass simulator ()
37   (
38     (world :initarg :world :initform nil)
39     (time :initarg :time :initform 0)
40   )
41 )
42
43 (export 'simulator)
44
45 (defclass world ()
46   (
47     (size :initarg :size :initform '(10 10))
48     (objects :initarg :objects :initform nil)
49   )
50 )
51

```

```

52 (export 'world)
53
54 (defclass object ()
55   (
56     (name :initarg :name :initform (new-symbol 'o))
57     (location :initarg :location :initform '(1 1))
58     (orientation :initarg :orientation :initform :north)
59   )
60 )
61
62 (export 'object)
63
64 (defclass robot (object)
65   (
66     (name :initarg :name :initform (new-symbol 'robot))
67     (percept :initarg :percept :initform nil)
68     (next-action :initarg :next-action :initform :nop)
69     (prev-action :initarg :prev-action :initform nil)
70     (prev-action-success :initarg :prev-action-success :initform nil)
71     (command-map :initarg :command-map
72 :initform *robot-command-map*)
73     (percept-map :initarg :percept-map
74 :initform *robot-percept-map*)
75   )
76 )
77
78 (export 'robot)

```

Simulator methods

```

79 (defmethod clear ((self simulator))
80   (with-slots (world) self
81     (clear world)))
82
83 (export 'clear)
84
85 (defmethod reset-simulator ((self simulator) &key clear?)
86   (with-slots (time world) self
87     (setq time 0)
88     (when clear?
89       (clear world))))
90
91 (export 'reset-simulator)
92

```

```

93 (defmethod add-obstacles ((self simulator) locations)
94   (dolist (loc locations)
95     (add-obstacle self loc)))
96
97 (export 'add-obstacles)

```

This next pair of methods demonstrate CLOS' function polymorphism. CLOS is a *generic function*-based object-oriented system, unlike, say, in Python or Java, where methods are tightly associated with the classes themselves as part of their definitions. In CLOS, all methods are instances of some "generic function" that when called, checks to see which method is appropriate for its arguments. The first method below, for example, would be used if:

```
(add-obstacle sim foo)
```

is called and `sim` is a simulator instance and `foo` is an instance of object. The second would be called otherwise.

These restrictions aren't limited to user-defined objects, either; for example, you can specify that an argument must be a symbol, number, cons cell, etc.:

```

SIM> (defmethod foo ((a number)) nil)
#<STANDARD-METHOD SIMULATOR::FOO (NUMBER) {10047F9B93}>
SIM> (defmethod foo ((a number)) nil)
#<STANDARD-METHOD SIMULATOR::FOO (NUMBER) {10048391F3}>
SIM> (defmethod foo (a) t)
#<STANDARD-METHOD SIMULATOR::FOO (T) {100486CC93}>
SIM> (foo 3)
NIL
SIM> (foo 'a)
T

```

```

98 (defmethod add-obstacle ((self simulator) (object object))
99   (with-slots (world) self
100     (add-object world object)))
101
102 (defmethod add-obstacle ((self simulator) location)
103   (with-slots (world) self
104     (add-object world (make-instance 'object :name (new-symbol 'obj) :location location))))
105
106 (export 'add-obstacle)
107
108 (defmethod add-object ((self simulator) object)
109   (add-obstacle self object))
110

```



```

111 (export 'add-object)
112
113 (defmethod add-random-obstacles ((self simulator) &key number (max 20) (min 1))
114   (unless number
115     (setq number (random (+ (- max min) 1))))
116   (dotimes (i number)
117     (add-random-obstacle self)))
118
119 (export 'add-random-obstacles)
120
121 (defmethod add-random-obstacle ((self simulator))
122   (with-slots (world) self
123     (add-object world (make-instance 'object :location (random-empty-location self)))))
124
125 (export 'add-random-obstacle)
126
127 (defmethod add-robot ((self simulator) &key (robot nil)
128   (name (new-symbol 'robot))
129   (location (random-empty-location self))
130   (orientation (nth (random 4) *directions*))
131   (type 'robot))
132   (with-slots (world) self
133     (unless (empty? world location)
134       (error "Can't add a robot to ~s: square is not empty." location))
135     (unless robot
136       (setq robot
137         (make-instance type :name name
138           :location location :orientation orientation))))
139     (add-object world robot)
140     robot))
141
142 (export 'add-robot)
143
144 (defmethod delete-object ((self simulator) object)
145   (with-slots (world) self
146     (delete-object world object)))
147
148 (export 'delete-object)
149
150 (defmethod random-location ((self simulator))
151   (with-slots (world) self
152     (list (+ (random (car (size world))) 1)
153       (+ (random (cadr (size world))) 1))))
154

```

```

155 (export 'random-location)
156
157 (defmethod random-empty-location ((self simulator))
158   (with-slots (world) self
159     (loop with loc
160       do (setq loc (list (+ (random (car (size world))) 1)
161         (+ (random (cadr (size world))) 1)))
162       until (empty? world loc)
163       finally (return loc))))
164
165 (export 'random-empty-location)
166
167 (defmethod next-location ((self simulator) location direction)
168   (case direction
169     (:north (list (car location) (1+ (cadr location))))
170     (:east (list (1+ (car location)) (cadr location)))
171     (:south (list (car location) (1- (cadr location))))
172     (:west (list (1- (car location)) (cadr location))))
173
174 (export 'next-location)
175
176 (defmethod opposite-direction ((self simulator) direction)
177   (case direction
178     (:north :south)
179     (:south :north)
180     (:east :west)
181     (:west :east)))
182
183 (export 'opposite-direction)
184
185 (defmethod clockwise-direction ((self simulator) direction)
186   (case direction
187     (:north :east)
188     (:south :west)
189     (:east :south)
190     (:west :north)))
191
192 (export 'clockwise-direction)
193
194 (defmethod counterclockwise-direction ((self simulator) direction)
195   (opposite-direction self (clockwise-direction self direction)))
196
197 (export 'counterclockwise-direction)
198

```

```

199 (defmethod run ((self simulator) &key (for 1) (sketch-each nil))
200   (dotimes (i for)
201     (clock-tick self)
202     (when sketch-each
203       (world-sketch self))))
204
205 (export 'run)
206
207 (defmethod clock-tick ((self simulator))
208   (with-slots (world time) self
209     (dmsg ".")
210     (dolist (object (objects world))
211       (calculate-percept self object)
212       (clock-tick object)
213       (take-action self object))
214     (incf time)))
215
216 (defmethod find-object ((self simulator) description)
217   (with-slots (world) self
218     (find-object world description)))
219
220 (export 'find-object)
221
222 (defmethod remove-object ((self simulator) description)
223   (with-slots (world) self
224     (remove-object world description)))
225
226 (export 'remove-object)
227
228 (defmethod world-sketch ((self simulator) &key (empty-char #\.) (side-wall-char #\+)
229   (top-bottom-char #\+))
230
231   (with-slots (world) self
232     (with-slots (size) world
233       (let ((w (world-array world)))
234         (write side-wall-char :escape nil)
235         (write (make-string (cadr size) :initial-element top-bottom-char) :escape nil)
236         (write side-wall-char :escape nil)
237         (fresh-line)
238         (loop for j from (1- (car size)) downto 0
239           do
240             (write side-wall-char :escape nil)
241             (dotimes (i (cadr size))
242               (if (null (aref w i j))

```

```

243 (write empty-char :escape nil)
244 (write (aref w i j) :escape nil)))
245 (write side-wall-char :escape nil)
246 (fresh-line))
247 (write side-wall-char :escape nil)
248 (write (make-string (cadr size) :initial-element top-bottom-char) :escape nil)
249 (write side-wall-char :escape nil)
250 (fresh-line))))))
251
252 (export 'world-sketch)
253
254 (defun create-simulator (&key (size '(10 10))
255 (num-obstacles 0)
256 (obstacle-locations nil)
257 )
258 (let* ((sim (make-instance 'simulator
259 :world (make-instance 'world :size size))))
260 (when obstacle-locations
261 (add-obstacles sim obstacle-locations))
262 (unless (zerop num-obstacles)
263 (add-random-obstacles sim :number num-obstacles))
264 sim))
265
266 (export 'create-simulator)

```

1. Sensor methods

Percepts are created by the method(s) `calculate-percept`. Even though I have put these methods here, as you can see, they are just as much “methods of” objects as the simulator. See the discussion of percepts above for more information.

```

267 (defmethod calculate-percept ((self simulator) (object object))
268 )
269
270 (defmethod calculate-percept ((self simulator) (object robot))
271 (with-slots (time) self
272 (with-slots (name percept-map percept) object
273 (dfmsg "[~s Calculating percept for ~s]" time name)
274 (setq percept
275 (loop for percept in percept-map
276 collect (list (car percept)
277 (funcall (cadr percept) self object)))))))
278
279 (defmethod forward-sensor ((self simulator) object)

```

```

280   (with-slots (location orientation) object
281     (with-slots (world) self
282       (not (empty? world (next-location self location orientation))))))
283
284 (defmethod front-bump-sensor ((self simulator) (object robot))
285   (bump-sensor self object :forward))
286
287 (defmethod rear-bump-sensor ((self simulator) (object robot))
288   (bump-sensor self object :backward))
289
290 (defmethod left-bump-sensor ((self simulator) (object robot))
291   (bump-sensor self object :left))
292
293 (defmethod right-bump-sensor ((self simulator) (object robot))
294   (bump-sensor self object :right))
295
296 (defmethod bump-sensor ((self simulator) object which)
297   (with-slots (location orientation prev-action prev-action-success) object
298     (with-slots (world) self
299       (and
300         (eql prev-action which)
301         (eql nil prev-action-success)
302         (not
303          (empty? world
304           (next-location self
305            location
306            (case which
307              (:forward orientation)
308              (:backward
309               (opposite-direction self orientation))
310              (:left
311               (counterclockwise-direction self orientation))
312              (:right
313               (clockwise-direction self orientation))))))))))
314
315 (export '(forward-sensor front-bump rear-bump left-bump right-bump bump-sensor))

```

2. Effector (actuator) methods

The method `take-action`, which is specialized for each kind of object, does whatever the next-action of the robot is. See above for how to add new actions.

Here are the supplied take-action methods:

```

316 (defmethod take-action ((self simulator) (object object))

```

```

317 (vdfmsg "[~s: ignoring take-action method]" (slot-value object 'name))
318 )
319
320 (defmethod take-action ((self simulator) (object robot))
321   (with-slots (time) self
322     (with-slots (prev-action prev-action-success next-action
323       name command-map) object
324       (let ((command (cadr (assoc next-action command-map))))
325         (cond
326           ((null command)
327            (warn "~s Command ~s isn't implemented for ~s; ignoring."
328              time next-action name)
329            (setq prev-action-success nil))
330           (t
331            (fmsg "~s ~s: Performing action ~s." time name next-action)
332            (dfmsg "[~s: calling method ~s]" name command)
333            (setq prev-action-success (funcall command self object))
334            ))
335         (setq prev-action next-action)
336         (setq next-action nil)
337         prev-action-success))))
338
339 (defmethod do-nop ((self simulator) (object object))
340   (declare (ignore self object))
341   t)
342
343 (defmethod do-move-forward ((self simulator) (object object))
344   (with-slots (name location orientation) object
345     (move-object self object (next-location self location orientation))))
346
347 (defmethod do-move-backward ((self simulator) (object object))
348   (with-slots (name location orientation) object
349     (move-object self object
350       (next-location self
351         location (opposite-direction self orientation)))))
352
353 (defmethod do-move-left ((self simulator) (object object))
354   (with-slots (name location orientation) object
355     (move-object self object
356       (next-location self
357         location (counterclockwise-direction
358           self orientation)))))
359
360 (defmethod do-move-right ((self simulator) (object object))

```

```

361 (with-slots (name location orientation) object
362   (move-object self object
363   (next-location self location (clockwise-direction
364     self orientation))))))
365
366 (defmethod do-turn-clockwise ((self simulator) (object object))
367   (turn-object self object :clockwise))
368
369 (defmethod do-turn-counterclockwise ((self simulator) (object object))
370   (turn-object self object :counterclockwise))
371
372
373 (defmethod turn-object ((self simulator) (object object) direction)
374   (declare (ignore direction))
375   t)
376
377 (defmethod turn-object ((self simulator) (object robot) direction)
378   (with-slots (orientation name) object
379     (setq orientation (if (eql direction :clockwise)
380       (clockwise-direction self orientation)
381       (counterclockwise-direction self orientation)))
382     (fmsg "~s: Turning right, new orientation = ~s."
383       name orientation)
384     t))
385
386 (defmethod move-object ((self simulator) object new-loc)
387   (with-slots (name location) object
388     (with-slots (world) self
389       (cond
390         ((empty? world new-loc)
391          (setq location new-loc)
392          (fmsg "~s: Moving to ~s." name location)
393          t)
394         (t
395          (fmsg "~s: Tried and failed to move to ~s." name location)
396          nil))))))
397
398 (export '(do-nop do-move-forward do-move-backward do-move-left
399   do-move-right do-turn-clockwise do-turn-counterclockwise
400   turn-object move-object ))

```

World methods

```

401 (defmethod objects ((self world))

```

```

402   (with-slots (objects) self
403     objects))
404
405 (defmethod empty? ((self world) location)
406   (with-slots (objects size) self
407     (and (> (car location) 0)
408           (<= (car location) (car size))
409           (> (cadr location) 0)
410           (<= (cadr location) (cadr size))
411           (loop for obj in objects
412                 when (equal (slot-value obj 'location) location)
413                 return nil
414                 finally (return t))))))
415
416 (defmethod in-bounds? ((self world) loc)
417   (with-slots (size) self
418     (and (>= (car loc) 1) (<= (car loc) (car size))
419          (>= (cadr loc) 1) (<= (cadr loc) (cadr size)))))
420
421 (defmethod add-object ((self world) object)
422   (with-slots (size objects) self
423     (with-slots (location name) object
424       (cond
425         ((not (in-bounds? self location))
426          (cerror "Continue" "Can't add object ~s at ~s -- out of bounds."
427                  name location)
428          nil)
429         ((not (empty? self location))
430          (cerror "Continue" "Can't add object ~s at ~s -- location isn't empty"
431                  name location)
432          nil)
433         (t (push object objects))))))
434
435 (defmethod clear ((self world))
436   (with-slots (objects) self
437     (setq objects nil)))
438
439 (defmethod object-locations ((self world))
440   (with-slots (objects) self
441     (mapcar #'(lambda (o) (copy-list (slot-value o 'location)))
442             objects)))
443
444 (defmethod size ((self world))
445   (with-slots (size) self

```



```

446     size))
447
448 (defmethod delete-object ((self world) object)
449   (remove-object self object))
450
451
452
453 (defmethod remove-object ((self world) description)
454   (with-slots (objects) self
455     (let ((obj (find-object self description)))
456       (when obj
457         (with-slots (name) obj
458           (dfmsg "[Removing object ~s from world]" name)
459           (setq objects (remove obj objects)))))))
460
461
462 (defmethod find-object ((self world) (location cons))
463   (with-slots (objects) self
464     (car (member location objects :test #'(lambda (a b)
465       (equal a (location b)))))))
466
467
468 (defmethod find-object ((self world) (location symbol))
469   (with-slots (objects) self
470     (car (member location objects :test #'(lambda (a b)
471       (eql a (name b)))))))
472
473 (defmethod find-object ((self world) (object object))
474   (with-slots (objects) self
475     (car (member object objects))))
476
477
478
479
480 (defmethod world-array ((self world))
481   (with-slots (size objects) self
482     (let ((a (make-array size :initial-element nil)))
483       (dolist (obj objects)
484         (setf (aref a (1- (car (slot-value obj 'location))))
485           (1- (cadr (slot-value obj 'location))))
486         (icon obj)))
487     a)))
488 (export '(objects empty? in-bounds? add-object clear object-locations size delete-object find-object

```

Object methods

```

489 (defmethod clock-tick ((self object))
490   :nop)
491
492 (defmethod name ((self object))
493   (with-slots (name) self
494     name))
495
496 (export 'name)
497
498 (defmethod location ((self object))
499   (with-slots (location) self
500     location))
501
502 (export 'location)
503
504 (defmethod orientation ((self object))
505   (with-slots (orientation) self
506     orientation))
507
508 (export 'orientation)
509
510 (defmethod icon ((self object))
511   #\@)
512
513 (export 'icon)

```

Robot methods

```

514 (defmethod clock-tick ((self robot))
515   (with-slots (percept next-action name agent-program) self
516     (setq next-action (agent-program self percept))
517     (dfmsg "[~s: ~s -> ~s]" name percept next-action)
518     next-action
519     ))
520
521 (defmethod agent-program ((self robot) percept)
522   (with-slots (name percept next-action) self
523     (dfmsg "[~s: current percept = ~s, next action = ~s]"
524       name percept next-action)
525     (setq next-action :nop)
526     ))
527
528 (export 'agent-program)

```

```

529
530
531 (defmethod icon ((self robot))
532   (with-slots (orientation) self
533     (case orientation
534       (:north #\^)
535       (:south #\v)
536       (:east #\>)
537       (:west #\<)
538       (otherwise #\R)))))

```

Example: random-robot

```

539 (defclass random-robot (robot) ())
540
541 (export 'random-robot)
542
543 (defmethod agent-program ((self random-robot) percept)
544   (with-slots (name) self
545     (let ((next-action (car (nth (random (length *robot-command-map*))
546       *robot-command-map*))))
547       (dfmsg "[~s: percept = ~s]" name percept)
548       (dfmsg "[~s: choosing ~s as next action]" name next-action)
549       next-action)))

```