

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>	23
<i>Object methods</i>	25
<i>Robot methods</i>	26
<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 exported (external) symbols in one or more of the other packages you’re

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)`.

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., it's 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

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.

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

```
(second (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)
```

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 that allow for the robot to be placed at a particular location, where the robot specifies, or at a random location. If you pass the method your own robot instance via the **:robot** argument, by default, it places it in a random location. If you want it put a particular location that is *not* what specified by robot instance (in its **location** instance variable), then set the **:location** parameter; e.g.:³

```
(add-robot sim :robot my-robot :location '(3 4))
```

will put it at (3 4) and the robot's instance variable will be set accordingly. If you want the robot placed where its **location** instance variable says, then do not specify the **:location** parameter, but instead set the **:random-location** parameter to **nil**, e.g.:

```
(setq my-robot (make-instance 'reflex-agent :location '(3 4)))
(add-robot sim :robot my-robot :random-location nil)
```

The same thing happens with orientation (i.e., there are **:orientation** and **:random-orientation** parameters).

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

³ Note: Changed 2/13/21 to allow easier random placement of pre-instantiated robots.

- `: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 counter-clockwise 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 its 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     (:nicknames "SIM"))
5   )
6
7 (in-package sim)
8
9 (load "new-symbol")
10 (use-package 'sym)
11 (load "messages")
12 (shadowing-import 'msg:msg)
13 (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.

```

14 (defvar *directions* '(:north :south :east :west))
15
16 (defvar *robot-command-map*
17     '(:nop do-nop)
18       (:forward do-move-forward)
19       (:backward do-move-backward)
20       (:left do-move-left)
21       (:right do-move-right)
22       (:turn-right do-turn-clockwise)
23       (:turn-left do-turn-counterclockwise)))
24
25 (defvar *robot-percept-map*
26     '(:front-sensor forward-sensor)
27       (:front-bump front-bump-sensor)
28       (:rear-bump rear-bump-sensor)
29       (:right-bump right-bump-sensor)
30       (:left-bump left-bump-sensor)))
31
32 (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.

```

33 (defclass simulator ()
34   (
35     (world :initarg :world :initform nil)
36     (time :initarg :time :initform 0)
37   )
38 )
39
40 (export 'simulator)
41
42 (defclass world ()
43   (
44     (size :initarg :size :initform '(10 10))
45     (objects :initarg :objects :initform nil)
46   )
47 )
48
49 (export 'world)
50
51 (defclass object ()
52   (

```

```

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

```

Simulator methods

```

76 (defmethod clear ((self simulator))
77   (with-slots (world) self
78     (clear world)))
79
80 (export 'clear)
81
82 (defmethod reset-simulator ((self simulator) &key clear?)
83   (with-slots (time world) self
84     (setq time 0)
85     (when clear?
86       (clear world))))
87
88 (export 'reset-simulator)
89
90 (defmethod add-obstacles ((self simulator) locations)
91   (dolist (loc locations)
92     (add-obstacle self loc)))
93

```

```
94 (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
```

```
95 (defmethod add-obstacle ((self simulator) (object object))
96   (with-slots (world) self
97     (add-object world object)))
98
99 (defmethod add-obstacle ((self simulator) location)
100   (with-slots (world) self
101     (add-object world (make-instance 'object :name (new-symbol 'obj) :location location))))
102
103 (export 'add-obstacle)
104
105 (defmethod add-object ((self simulator) object)
106   (add-obstacle self object))
107
108 (export 'add-object)
109
110 (defmethod add-random-obstacles ((self simulator) &key number (max 20) (min 1))
111   (unless number
```



```

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

```

```

156 ; (with-slots (world) self
157 ; (unless (empty? world location)
158 ; (error "Can't add a robot to ~s: square is not empty." location))
159 ; (unless robot
160 ; (setq robot
161 ; (make-instance type :name name
162 ; :location location :orientation orientation)))
163 ; (add-object world robot)
164 ; robot))
165
166 (export 'add-robot)
167
168 (defmethod delete-object ((self simulator) object)
169 (with-slots (world) self
170 (delete-object world object)))
171
172 (export 'delete-object)
173
174 (defmethod random-location ((self simulator))
175 (with-slots (world) self
176 (list (+ (random (car (size world))) 1)
177 (+ (random (cadr (size world))) 1))))
178
179 (export 'random-location)
180
181 (defmethod random-empty-location ((self simulator))
182 (with-slots (world) self
183 (loop with loc
184 do (setq loc (list (+ (random (car (size world))) 1)
185 (+ (random (cadr (size world))) 1)))
186 until (empty? world loc)
187 finally (return loc))))
188
189 (export 'random-empty-location)
190
191 (defmethod next-location ((self simulator) location direction)
192 (case direction
193 (:north (list (car location) (1+ (cadr location))))
194 (:east (list (1+ (car location)) (cadr location)))
195 (:south (list (car location) (1- (cadr location))))
196 (:west (list (1- (car location)) (cadr location))))
197
198 (export 'next-location)
199

```

```

200 (defmethod opposite-direction ((self simulator) direction)
201   (case direction
202     (:north :south)
203     (:south :north)
204     (:east :west)
205     (:west :east)))
206
207 (export 'opposite-direction)
208
209 (defmethod clockwise-direction ((self simulator) direction)
210   (case direction
211     (:north :east)
212     (:south :west)
213     (:east :south)
214     (:west :north)))
215
216 (export 'clockwise-direction)
217
218 (defmethod counterclockwise-direction ((self simulator) direction)
219   (opposite-direction self (clockwise-direction self direction)))
220
221 (export 'counterclockwise-direction)
222
223 (defmethod run ((self simulator) &key (for 1) (sketch-each nil))
224   (dotimes (i for)
225     (clock-tick self)
226     (when sketch-each
227       (world-sketch self))))
228
229 (export 'run)
230
231 (defmethod clock-tick ((self simulator))
232   (with-slots (world time) self
233     (dmsg ".")
234     (dolist (object (objects world))
235       (calculate-percept self object)
236       (clock-tick object)
237       (take-action self object))
238     (incf time)))
239
240 (defmethod find-object ((self simulator) description)
241   (with-slots (world) self
242     (find-object world description)))
243

```

```

244 (export 'find-object)
245
246 (defmethod remove-object ((self simulator) description)
247   (with-slots (world) self
248     (remove-object world description)))
249
250 (export 'remove-object)
251
252 (defmethod world-sketch ((self simulator) &key (empty-char #\.) (side-wall-char #\+)
253   (top-bottom-char #\+))
254
255   (with-slots (world) self
256     (with-slots (size) world
257       (let ((w (world-array world)))
258         (write side-wall-char :escape nil)
259         (write (make-string (cadr size) :initial-element top-bottom-char) :escape nil)
260         (write side-wall-char :escape nil)
261         (fresh-line)
262         (loop for j from (1- (car size)) downto 0
263           do
264             (write side-wall-char :escape nil)
265             (dotimes (i (cadr size))
266               (if (null (aref w i j))
267                 (write empty-char :escape nil)
268                 (write (aref w i j) :escape nil))))
269             (write side-wall-char :escape nil)
270             (fresh-line))
271         (write side-wall-char :escape nil)
272         (write (make-string (cadr size) :initial-element top-bottom-char) :escape nil)
273         (write side-wall-char :escape nil)
274         (fresh-line))))))
275
276 (export 'world-sketch)
277
278 (defun create-simulator (&key (size '(10 10))
279   (num-obstacles 0)
280   (obstacle-locations nil)
281   )
282   (let* ((sim (make-instance 'simulator
283     :world (make-instance 'world :size size))))
284     (when obstacle-locations
285       (add-obstacles sim obstacle-locations))
286     (unless (zerop num-obstacles)
287       (add-random-obstacles sim :number num-obstacles))

```

```

288     sim))
289
290 (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.

```

291 (defmethod calculate-percept ((self simulator) (object object))
292   )
293
294 (defmethod calculate-percept ((self simulator) (object robot))
295   (with-slots (time) self
296     (with-slots (name percept-map percept) object
297       (dfmsg "[~s Calculating percept for ~s]" time name)
298       (setq percept
299         (loop for percept in percept-map
300           collect (list (car percept)
301             (funcall (cadr percept) self object))))))
302
303 (defmethod forward-sensor ((self simulator) object)
304   (with-slots (location orientation) object
305     (with-slots (world) self
306       (not (empty? world (next-location self location orientation)))))
307
308 (defmethod front-bump-sensor ((self simulator) (object robot))
309   (bump-sensor self object :forward))
310
311 (defmethod rear-bump-sensor ((self simulator) (object robot))
312   (bump-sensor self object :backward))
313
314 (defmethod left-bump-sensor ((self simulator) (object robot))
315   (bump-sensor self object :left))
316
317 (defmethod right-bump-sensor ((self simulator) (object robot))
318   (bump-sensor self object :right))
319
320 (defmethod bump-sensor ((self simulator) object which)
321   (with-slots (location orientation prev-action prev-action-success) object
322     (with-slots (world) self
323       (and
324         (eq1 prev-action which)

```

```

325         (eq1 nil prev-action-success)
326         (not
327   (empty? world
328   (next-location self
329     location
330     (case which
331       (:forward orientation)
332       (:backward
333         (opposite-direction self orientation))
334       (:left
335         (counterclockwise-direction self orientation))
336       (:right
337         (clockwise-direction self orientation)))))))))
338
339 (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:

```

340 (defmethod take-action ((self simulator) (object object))
341   (vdfmsg "[~s: ignoring take-action method]" (slot-value object 'name))
342   )
343
344 (defmethod take-action ((self simulator) (object robot))
345   (with-slots (time) self
346     (with-slots (prev-action prev-action-success next-action
347       name command-map) object
348       (let ((command (cadr (assoc next-action command-map))))
349         (cond
350           ((null command)
351            (warn "~s Command ~s isn't implemented for ~s; ignoring."
352              time next-action name)
353            (setq prev-action-success nil))
354           (t
355            (fmsg "~s ~s: Performing action ~s." time name next-action)
356            (dfmsg "[~s: calling method ~s]" name command)
357            (setq prev-action-success (funcall command self object))
358            ))
359         (setq prev-action next-action)
360         (setq next-action nil)
361         prev-action-success))))

```

```

362
363 (defmethod do-nop ((self simulator) (object object))
364   (declare (ignore self object))
365   t)
366
367 (defmethod do-move-forward ((self simulator) (object object))
368   (with-slots (name location orientation) object
369     (move-object self object (next-location self location orientation))))
370
371 (defmethod do-move-backward ((self simulator) (object object))
372   (with-slots (name location orientation) object
373     (move-object self object
374       (next-location self
375         location (opposite-direction self orientation)))))
376
377 (defmethod do-move-left ((self simulator) (object object))
378   (with-slots (name location orientation) object
379     (move-object self object
380       (next-location self
381         location (counterclockwise-direction
382           self orientation)))))
383
384 (defmethod do-move-right ((self simulator) (object object))
385   (with-slots (name location orientation) object
386     (move-object self object
387       (next-location self location (clockwise-direction
388         self orientation)))))
389
390 (defmethod do-turn-clockwise ((self simulator) (object object))
391   (turn-object self object :clockwise))
392
393 (defmethod do-turn-counterclockwise ((self simulator) (object object))
394   (turn-object self object :counterclockwise))
395
396
397 (defmethod turn-object ((self simulator) (object object) direction)
398   (declare (ignore direction))
399   t)
400
401 (defmethod turn-object ((self simulator) (object robot) direction)
402   (with-slots (orientation name) object
403     (setq orientation (if (eql direction :clockwise)
404       (clockwise-direction self orientation)
405       (counterclockwise-direction self orientation))))

```

```

406      (fmsg "~s: Turning right, new orientation = ~s."
407      name orientation)
408      t))
409
410 (defmethod move-object ((self simulator) object new-loc)
411   (with-slots (name location) object
412     (with-slots (world) self
413       (cond
414         ((empty? world new-loc)
415          (setq location new-loc)
416          (fmsg "~s: Moving to ~s." name location)
417          t)
418         (t
419          (fmsg "~s: Tried and failed to move to ~s." name location)
420          nil))))))
421
422 (export '(do-nop do-move-forward do-move-backward do-move-left
423          do-move-right do-turn-clockwise do-turn-counterclockwise
424          turn-object move-object ))

```

World methods

```

425 (defmethod objects ((self world))
426   (with-slots (objects) self
427     objects))
428
429 (defmethod empty? ((self world) location)
430   (with-slots (objects size) self
431     (and (> (car location) 0)
432          (<= (car location) (car size))
433          (> (cadr location) 0)
434          (<= (cadr location) (cadr size))
435          (loop for obj in objects
436                when (equal (slot-value obj 'location) location)
437                return nil
438                finally (return t)))))
439
440 (defmethod in-bounds? ((self world) loc)
441   (with-slots (size) self
442     (and (>= (car loc) 1) (<= (car loc) (car size))
443          (>= (cadr loc) 1) (<= (cadr loc) (cadr size)))))
444
445 (defmethod add-object ((self world) object)
446   (with-slots (size objects) self
447     (with-slots (location name) object

```



```

448      (cond
449        ((not (in-bounds? self location))
450         (cerror "Continue" "Can't add object ~s at ~s -- out of bounds."
451                name location)
452         nil)
453        ((not (empty? self location))
454         (cerror "Continue" "Can't add object ~s at ~s -- location isn't empty"
455                name location)
456         nil)
457        (t (push object objects))))))
458
459 (defmethod clear ((self world))
460   (with-slots (objects) self
461     (setq objects nil)))
462
463 (defmethod object-locations ((self world))
464   (with-slots (objects) self
465     (mapcar #'(lambda (o) (copy-list (slot-value o 'location)))
466             objects)))
467
468 (defmethod size ((self world))
469   (with-slots (size) self
470     size))
471
472 (defmethod delete-object ((self world) object)
473   (remove-object self object))
474
475
476
477 (defmethod remove-object ((self world) description)
478   (with-slots (objects) self
479     (let ((obj (find-object self description)))
480       (when obj
481         (with-slots (name) obj
482           (dfmsg "[Removing object ~s from world]" name)
483           (setq objects (remove obj objects)))))))
484
485
486 (defmethod find-object ((self world) (location cons))
487   (with-slots (objects) self
488     (car (member location objects :test #'(lambda (a b)
489       (equal a (location b)))))))
490
491

```

```

492 (defmethod find-object ((self world) (location symbol))
493   (with-slots (objects) self
494     (car (member location objects :test #'(lambda (a b)
495       (eql a (name b)))))))
496
497 (defmethod find-object ((self world) (object object))
498   (with-slots (objects) self
499     (car (member object objects))))
500
501
502
503
504 (defmethod world-array ((self world))
505   (with-slots (size objects) self
506     (let ((a (make-array size :initial-element nil)))
507       (dolist (obj objects)
508         (setf (aref a (1- (car (slot-value obj 'location)))
509           (1- (cadr (slot-value obj 'location)))))
510         (icon obj)))
511     a)))
512 (export '(objects empty? in-bounds? add-object clear object-locations size delete-object find-object))

```

Object methods

```

513 (defmethod clock-tick ((self object))
514   :nop)
515
516 (defmethod name ((self object))
517   (with-slots (name) self
518     name))
519
520 (export 'name)
521
522 (defmethod location ((self object))
523   (with-slots (location) self
524     location))
525
526 (export 'location)
527
528 (defmethod orientation ((self object))
529   (with-slots (orientation) self
530     orientation))
531
532 (export 'orientation)

```

```

533
534 (defmethod icon ((self object))
535   #\@)
536
537 (export 'icon)

```

Robot methods

```

538 (defmethod clock-tick ((self robot))
539   (with-slots (percept next-action name agent-program) self
540     (setq next-action (agent-program self percept))
541     (dfmsg "[~s: ~s -> ~s]" name percept next-action)
542     next-action
543   ))
544
545 (defmethod agent-program ((self robot) percept)
546   (with-slots (name percept next-action) self
547     (dfmsg "[~s: current percept = ~s, next action = ~s]"
548       name percept next-action)
549     (setq next-action :nop)
550   ))
551
552 (export 'agent-program)
553
554
555 (defmethod icon ((self robot))
556   (with-slots (orientation) self
557     (case orientation
558       (:north #\^)
559       (:south #\v)
560       (:east #\>)
561       (:west #\<)
562       (otherwise #\R))))

```

Example: random-robot

```

563 (defclass random-robot (robot) ())
564
565 (export 'random-robot)
566
567 (defmethod agent-program ((self random-robot) percept)
568   (with-slots (name) self
569     (let ((next-action (car (nth (random (length *robot-command-map*))
570       *robot-command-map*))))
571       (dfmsg "[~s: percept = ~s]" name percept)

```

```
572      (dfmsg "[~s: choosing ~s as next action]" name next-action)
573      next-action)))
```