A Simple Robot Simulator: simulator.lisp

Roy M. Turner

Spring, 2021

Contents

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)
```

[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)`.

NOTE: Loading the simulator *also* loads the files `messages.lisp` and `=new-symbol.lisp`. You do not need to load them yourself. It does *not*, however, import the symbols (functions, methods, etc.) into the `CL-USER` package the REPL starts in. To do that, you would need to do, in the REPL or your own files:

```
(use-package 'sym)   ;; for the new-symbols file
(use-package 'message) ;; for the messages file
```

NOTE: As of 2023-10-19, the simulator and associated files no longer spit out the annoying warnings that they did previously. If you want to get rid of similar warnings and style warnings from your own code, you can put the following at the top of your file(s):

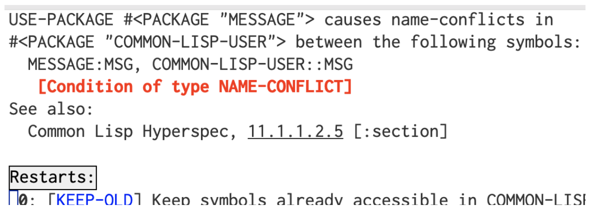
```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (declaim (sb-ext:muffle-conditions warning))
  (declaim (sb-ext:muffle-conditions style-warning)))
```

and then put this at the bottom to turn them back on after loading:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (declaim (sb-ext:unmuffle-conditions warning))
  (declaim (sb-ext:unmuffle-conditions style-warning)))
```

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 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:



```
USE-PACKAGE #<PACKAGE "MESSAGE"> causes name-conflicts in
#<PACKAGE "COMMON-LISP-USER"> between the following symbols:
  MESSAGE:MSG, COMMON-LISP-USER::MSG
   [Condition of type NAME-CONFLICT]
See also:
  Common Lisp Hyperspec, 11.1.1.2.5 [:section]

Restarts:
 0: [KEEP-OLD] Keep symbols already accessible in COMMON-LIS
```

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]

[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 easy for you to use.

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) 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.

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 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-objectremove-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 ...)
```

Muffle warnings and style-warnings

This will get rid of the warnings and style warnings from this file and the others it loads.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (declaim (sb-ext:muffle-conditions warning))
  (declaim (sb-ext:muffle-conditions style-warning)))
```

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.

```
(unless (find-package "SIM")
  (defpackage "SIMULATOR"
    (:use "COMMON-LISP")
 ;   (:shadowing-import-from "COMMON-LISP" "NAME")
    (:nicknames "SIM"))
    )

(in-package sim)

(shadowing-import '(NAME) :cl-user)

(load "new-symbol")
(use-package 'sym)
(load "messages")
(shadowing-import 'msg:msg)
(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.

```
(defvar *directions* '(:north :south :east :west))

(defvar *robot-command-map*
    '((:nop do-nop)
      (:forward do-move-forward)
      (:backward do-move-backward)
      (:left do-move-left)
      (:right do-move-right)
      (:turn-right do-turn-clockwise)
      (:turn-left do-turn-counterclockwise)))

(defvar *robot-percept-map*
    '((:front-sensor forward-sensor)
      (:front-bump front-bump-sensor)
      (:rear-bump rear-bump-sensor)
      (:right-bump right-bump-sensor)
      (:left-bump left-bump-sensor)))

(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.

```
(defclass simulator ()
  (
   (world :initarg :world :initform nil)
   (time :initarg :time :initform 0)
   )
  )

(export 'simulator)

(defclass world ()
  (
   (size :initarg :size :initform '(10 10))
   (objects :initarg :objects :initform nil)
   )
  )

(export 'world)

(defclass object ()
  (
   (name :initarg :name :initform (new-symbol 'o))
   (location :initarg :location :initform '(1 1))
   (orientation :initarg :orientation :initform :north)
    )
  )

(export 'object)

(defclass robot (object)
  (
   (name :initarg :name :initform (new-symbol 'robot))
   (percept :initarg :percept :initform nil)
   (next-action :initarg :next-action :initform :nop)
   (prev-action :initarg :prev-action :initform nil)
   (prev-action-success :initarg :prev-action-success :initform nil)
   (command-map :initarg :command-map
 :initform *robot-command-map*)
   (percept-map :initarg :percept-map
 :initform *robot-percept-map*)
   )
  )

(export 'robot)
```

Simulator methods

```
(defmethod clear ((self simulator))
  (with-slots (world) self
    (clear world)))

(export 'clear)

(defmethod reset-simulator ((self simulator) &key clear?)
  (with-slots (time world) self
    (setq time 0)
    (when clear?
      (clear world))))

(export 'reset-simulator)

(defmethod add-obstacles ((self simulator) locations)
  (dolist (loc locations)
    (add-obstacle self loc)))

(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
```

```
(defmethod add-obstacle ((self simulator) (object object))
  (with-slots (world) self
    (add-object world object)))

(defmethod add-obstacle ((self simulator) location)
  (with-slots (world) self
    (add-object world (make-instance ’object :name (new-symbol ’obj) :location location))))

(export ’add-obstacle)

(defmethod add-object ((self simulator) object)
  (add-obstacle self object))

(export ’add-object)

(defmethod add-random-obstacles ((self simulator) &key number (max 20) (min 1))
  (unless number
    (setq number (random (+ (- max min) 1))))
  (dotimes (i number)
    (add-random-obstacle self)))

(export ’add-random-obstacles)

(defmethod add-random-obstacle ((self simulator))
  (with-slots (world) self
    (add-object world (make-instance ’object :location (random-empty-location self)))))

(export ’add-random-obstacle)

(defmethod add-robot ((self simulator) &key (robot nil)
     (name (new-symbol ’robot))
     (location (random-empty-location self))
     (orientation (nth (random 4) *directions*))
     (type ’robot))
  (with-slots (world) self
    (unless (empty? world location)
      (error "Can’t add a robot to ~s: square is not empty." location))
    (unless robot
      (setq robot
 (make-instance type :name name
        :location location :orientation orientation)))
    (add-object world robot)
    robot))

(export 'add-robot)

(defmethod delete-object ((self simulator) object)
  (with-slots (world) self
    (delete-object world object)))

(export 'delete-object)

(defmethod random-location ((self simulator))
  (with-slots (world) self
    (list (+ (random (car (size world))) 1)
   (+ (random (cadr (size world))) 1))))

(export 'random-location)

(defmethod random-empty-location ((self simulator))
  (with-slots (world) self
    (loop with loc
 do (setq loc (list (+ (random (car (size world))) 1)
    (+ (random (cadr (size world))) 1)))
 until (empty? world loc)
 finally (return loc))))

(export 'random-empty-location)

(defmethod next-location ((self simulator) location direction)
  (case direction
    (:north (list (car location) (1+ (cadr location))))
    (:east (list (1+ (car location)) (cadr location)))
    (:south (list (car location) (1- (cadr location))))
    (:west (list (1- (car location)) (cadr location)))))

(export 'next-location)

(defmethod opposite-direction ((self simulator) direction)
  (case direction
    (:north :south)
    (:south :north)
    (:east :west)
    (:west :east)))

```

```
(export 'opposite-direction)

(defmethod clockwise-direction ((self simulator) direction)
  (case direction
    (:north :east)
    (:south :west)
    (:east :south)
    (:west :north)))

(export 'clockwise-direction)

(defmethod counterclockwise-direction ((self simulator) direction)
  (opposite-direction self (clockwise-direction self direction)))

(export 'counterclockwise-direction)

(defmethod run ((self simulator) &key (for 1) (sketch-each nil))
  (dotimes (i for)
    (clock-tick self)
    (when sketch-each
      (world-sketch self))))

(export 'run)

(defmethod clock-tick ((self simulator))
  (with-slots (world time) self
    (dmsg ".")
    (dolist (object (objects world))
      (calculate-percept self object)
      (clock-tick object)
      (take-action self object))
    (incf time)))

(defmethod find-object ((self simulator) description)
  (with-slots (world) self
    (find-object world description)))

(export 'find-object)

(defmethod remove-object ((self simulator) description)
  (with-slots (world) self
    (remove-object world description)))

(export 'remove-object)
```

```

(defmethod world-sketch ((self simulator) &key (empty-char #\.) (side-wall-char #\+)
 (top-bottom-char #\+))

  (with-slots (world) self
    (with-slots (size) world
      (let ((w (world-array world)))
 (write side-wall-char :escape nil)
 (write (make-string (cadr size) :initial-element top-bottom-char) :escape nil)
 (write side-wall-char :escape nil)
 (fresh-line)
 (loop for j from (1- (car size)) downto 0
     do
       (write side-wall-char :escape nil)
       (dotimes (i (cadr size))
 (if (null (aref w i j))
   (write empty-char :escape nil)
   (write (aref w i j) :escape nil)))
       (write side-wall-char :escape nil)
       (fresh-line))
 (write side-wall-char :escape nil)
 (write (make-string (cadr size) :initial-element top-bottom-char) :escape nil)
 (write side-wall-char :escape nil)
 (fresh-line)))))

(export 'world-sketch)

(defun create-simulator (&key (size '(10 10))
       (num-obstacles 0)
       (obstacle-locations nil)
       )
  (let* ((sim (make-instance 'simulator
 :world (make-instance 'world :size size))))
    (when obstacle-locations
      (add-obstacles sim obstacle-locations))
    (unless (zerop num-obstacles)
      (add-random-obstacles sim :number num-obstacles))
    sim))

(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.

```
(defmethod calculate-percept ((self simulator) (object object))
  )

(defmethod calculate-percept ((self simulator) (object robot))
  (with-slots (time) self
    (with-slots (name percept-map percept) object
      (dfmsg "[~s  Calculating percept for ~s]" time name)
      (setq percept
 (loop for percept in percept-map
     collect (list (car percept)
   (funcall (cadr percept) self object)))))))

(defmethod forward-sensor ((self simulator) object)
  (with-slots (location orientation) object
    (with-slots (world) self
      (not (empty? world (next-location self location orientation))))))

(defmethod front-bump-sensor ((self simulator) (object robot))
  (bump-sensor self object :forward))

(defmethod rear-bump-sensor ((self simulator) (object robot))
  (bump-sensor self object :backward))

(defmethod left-bump-sensor ((self simulator) (object robot))
  (bump-sensor self object :left))

(defmethod right-bump-sensor ((self simulator) (object robot))
  (bump-sensor self object :right))

(defmethod bump-sensor ((self simulator) object which)
  (with-slots (location orientation prev-action prev-action-success) object
    (with-slots (world) self
      (and
       (eql prev-action which)
       (eql nil prev-action-success)
       (not
 (empty? world
 (next-location self
        location
        (case which
  (:forward orientation)
  (:backward
   (opposite-direction self orientation))
  (:left
   (counterclockwise-direction self orientation))
  (:right
   (clockwise-direction self orientation)))))))))

(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:

```
(defmethod take-action ((self simulator) (object object))
  (vdfmsg "[~s: ignoring take-action method]" (slot-value object 'name))
  )

(defmethod take-action ((self simulator) (object robot))
  (with-slots (time) self
    (with-slots (prev-action prev-action-success next-action
  name command-map) object
      (let ((command (cadr (assoc next-action command-map))))
 (cond
  ((null command)
   (warn "~s  Command ~s isn't implemented for ~s; ignoring."
 time next-action name)
   (setq prev-action-success nil))
  (t
   (fmsg "~s  ~s: Performing action ~s." time name next-action)
   (dfmsg "[~s: calling method ~s]" name command)
   (setq prev-action-success (funcall command self object))
   ))
 (setq prev-action next-action)
 (setq next-action nil)
 prev-action-success))))

(defmethod do-nop ((self simulator) (object object))
  (declare (ignore self object))
  t)

(defmethod do-move-forward ((self simulator) (object object))
  (with-slots (name location orientation) object
    (move-object self object (next-location self location orientation))))

(defmethod do-move-backward ((self simulator) (object object))
  (with-slots (name location orientation) object
    (move-object self object
  (next-location self
 location (opposite-direction self orientation)))))

(defmethod do-move-left ((self simulator) (object object))
  (with-slots (name location orientation) object
    (move-object self object
  (next-location self
 location (counterclockwise-direction
   self orientation)))))

(defmethod do-move-right ((self simulator) (object object))
  (with-slots (name location orientation) object
    (move-object self object
  (next-location self location (clockwise-direction
        self orientation)))))

(defmethod do-turn-clockwise ((self simulator) (object object))
  (turn-object self object :clockwise))

(defmethod do-turn-counterclockwise ((self simulator) (object object))
  (turn-object self object :counterclockwise))


(defmethod turn-object ((self simulator) (object object) direction)
  (declare (ignore direction))
  t)

(defmethod turn-object ((self simulator) (object robot) direction)
  (with-slots (orientation name) object
    (setq orientation (if (eql direction :clockwise)
 (clockwise-direction self orientation)
 (counterclockwise-direction self orientation)))
    (fmsg "~s: Turning right, new orientation = ~s."
   name orientation)
    t))

(defmethod move-object ((self simulator) object new-loc)
  (with-slots (name location) object
    (with-slots (world) self
      (cond
       ((empty? world new-loc)
 (setq location new-loc)
 (fmsg "~s: Moving to ~s." name location)
 t)
       (t
 (fmsg "~s: Tried and failed to move to ~s." name location)
 nil)))))

(export '(do-nop do-move-forward do-move-backward do-move-left
   do-move-right do-turn-clockwise do-turn-counterclockwise
   turn-object move-object ))
```

World methods

```
(defmethod objects ((self world))
  (with-slots (objects) self
    objects))

(defmethod empty? ((self world) location)
  (with-slots (objects size) self
      (and (> (car location) 0)
    (<= (car location) (car size))
    (> (cadr location) 0)
    (<= (cadr location) (cadr size))
    (loop for obj in objects
        when (equal (slot-value obj 'location) location)
        return nil
        finally (return t)))))

(defmethod in-bounds? ((self world) loc)
  (with-slots (size) self
    (and (>= (car loc) 1) (<= (car loc) (car size))
  (>= (cadr loc) 1) (<= (cadr loc) (cadr size)))))

(defmethod add-object ((self world) object)
  (with-slots (size objects) self
    (with-slots (location name) object
      (cond
       ((not (in-bounds? self location))
 (cerror "Continue" "Can't add object ~s at ~s -- out of bounds."
        name location)
 nil)
       ((not (empty? self location))
 (cerror "Continue" "Can’t add object ~s at ~s -- location isn’t empty"
         name location)
 nil)
      (t (push object objects))))))

(defmethod clear ((self world))
  (with-slots (objects) self
    (setq objects nil)))

(defmethod object-locations ((self world))
  (with-slots (objects) self
    (mapcar #’(lambda (o) (copy-list (slot-value o ’location)))
     objects)))

(defmethod size ((self world))
  (with-slots (size) self
    size))

(defmethod delete-object ((self world) object)
  (remove-object self object))



(defmethod remove-object ((self world) description)
  (with-slots (objects) self
    (let ((obj (find-object self description)))
      (when obj
 (with-slots (name) obj
   (dfmsg "[Removing object ~s from world]" name)
   (setq objects (remove obj objects))))))))


(defmethod find-object ((self world) (location cons))
  (with-slots (objects) self
    (car (member location objects :test #’(lambda (a b)
     (equal a (location b)))))))


(defmethod find-object ((self world) (location symbol))
  (with-slots (objects) self
    (car (member location objects :test #’(lambda (a b)
     (eql a (name b)))))))

(defmethod find-object ((self world) (object object))
  (with-slots (objects) self
    (car (member object objects))))




(defmethod world-array ((self world))
  (with-slots (size objects) self
    (let ((a (make-array size :initial-element nil)))
      (dolist (obj objects)
 (setf (aref a (1- (car (slot-value obj 'location)))
     (1- (cadr (slot-value obj 'location))))
   (icon obj)))
      a)))
(export '(objects empty? in-bounds? add-object clear object-locations size delete-object find-objectr
```

Object methods

```
(defmethod clock-tick ((self object))
  :nop)

(defmethod name ((self object))
  (with-slots (name) self
    name))

(export 'name)

(defmethod location ((self object))
  (with-slots (location) self
    location))

(export 'location)

(defmethod orientation ((self object))
  (with-slots (orientation) self
    orientation))

(export 'orientation)

(defmethod icon ((self object))
  #\@)

(export 'icon)
```

Robot methods

```
(defmethod clock-tick ((self robot))
  (with-slots (percept next-action name agent-program) self
    (setq next-action (agent-program self percept))
    (dfmsg "[~s: ~s -> ~s]" name percept next-action)
    next-action
    ))

(defmethod agent-program ((self robot) percept)
  (with-slots (name percept next-action) self
    (dfmsg "[~s: current percept = ~s, next action = ~s]"
    name percept next-action)
    (setq next-action :nop)
    ))

(export 'agent-program)


(defmethod icon ((self robot))
  (with-slots (orientation) self
    (case orientation
      (:north #\^)
      (:south #\v)
      (:east #\>)
      (:west #\<)
      (otherwise #\R))))
```

Example: `random-robot`

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

(export 'random-robot)

(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)))
```

Restore the warnings and style warnings:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
(declaim (sb-ext:unmuffle-conditions warning))
(declaim (sb-ext:unmuffle-conditions style-warning)))
```