MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996

Problem Set 7

Issued: Tuesday, 22 October

Due: Friday, 1 November

Tutorial preparation for: Week of October 28

Reading:

- Course notes: through sectino 3.3.3
- code files `world.scm`, `game.scm` (attached to this handout)

Note: The overall object-oriented programming framework will be discussed in lecture on Thursday, October 24.

The programming assignment for this week explores two ideas: the simulation of a world in which objects are characterized by collections of state variables, and the use of *object-oriented programming* as a technique for modularizing worlds in which objects interact. These ideas are presented in the context of a simulation, much as might be used by economists, meteorologists, political scientists or physicists to analyze the complex interactions implied by mathematical models of the real world. Our system is a cross between these "serious" simulations and the popular simulation games (e.g, textual adventure games) available on many computers. While playing these games can be quite fun, programming them can be difficult if little or no effort is put into understanding the system's design and implementation. It is therefore very important that you adequately study the system and plan your work before coming to the lab.

Part 1 will help you to master the ideas involved. Part 2 consists of some tutorial exercises that you should complete before visiting the lab (you may wish to double check them using the machines in the lab). Part 3 contains a few warm-up exercises to do in the lab. You might do these and check your understanding of the answers before proceeding to the main lab assignment itself in Part 4.

Part 1: The 6.001 Election Game

The basic idea of a simulation is that the user creates a scenario in an imaginary world inhabited by various objects with special properties that govern how they interact. The user runs the simulation by creating appropriate (simulated) objects located in appropriate (simulated) locations with differing sets of parameters selected to test a model's ability to predict behavior under different conditions. In addition, some objects in the simulation may be under direct user control as the simulation proceeds. The user issues commands to the computer that have the effect of moving the objects or changing parameters in the imaginary world, such as picking up objects. The computer simulates the legal moves and rejects illegal ones. For example, it is illegal to move between places that are not connected (unless you have special powers). If a move is legal, the computer updates its model of the world and allows the next move to be considered.

Our game takes place in the strange world of U.S. politics. This world is inhabited by politicians, voters, reporters, and special investigators. In order to get going, we need to establish the structure of this imaginary world: the objects that exist and the ways in which they relate to each other.

Initially, there are three procedures for creating objects:

```
(make-city name)
(make-voter city how-influencable talkative?)
(make-politician name city risk-aversion restlessness)
```

In addition, there are procedures that make people and things, and procedures that install them in the simulated world. The reason that we need to be able to create people and things separately from installing them will be discussed in one of the exercises later. For now, we note the existence of the procedures

```
(make&install-city name)
(make&install-voter city how-influencable talkative?)
(make&install-politician name city risk-aversion restlessness)
```

All objects in the system are implemented as message-accepting procedures, as described in lecture on October 24. In addition to the main objects of the simulation, the file `game.scm` defines a number of other types of objects which are needed to construct cities, voters, and politicians. This is analogous to the "class library" that comes with most object-oriented development environments.

Our simulation initially has two kinds of people: voters and politicians. They have much in common (in fact, `person` is exactly what they have in common). We've built our world so that all people are located somewhere (in a `place`), but only politicians have names and can move about from place to place. The behavior of a politician is controlled by two variables (in our imaginary world, anyway): how much `risk-aversion` they have (0 means they don't like to take risks, 1 means they take a risk whenever they can) and their `restlessness` (a non-negative integer).

Voters are anonymous (they don't have names), but they do have a location. In addition, voters have two properties: they can be influenced to a certain extent (rated between 0 – not influencable at all and 1 – very easily persuaded) and a boolean that controls how much output they generate as the game proceeds.

The file `world.scm` has code to create a simulated world. It begins by creating objects to represent the major cities of the United States (Cambridge, Palo Alto, El Paso, Fairbanks, etc.) It then connects the cities to simulate airplane routes (we used the same techniques that the airlines apparently use: we selected locations at random and connected them with non-stop flights). In this simulation, we will just assume that politicians can teleport between cities.

We then create a very complicated type of person, the `*the-registrar-of-voters*`. This is by far the most complicated object in our system, but you don't need to worry about how it's built. Just notice the messages you can send to it: `REGISTER-CANDIDATE`, `REGISTER-VOTER`, `ELECTION`, plus the usual messages that any `person` will accept[1].

Finally, we create the voters and the politicians. The number of voters in each city is chosen somewhat randomly (between 10 and a selectable maximum number). As the voters are created, they are registered with the `*the-registrar-of-voters*`. The politicians are scattered around the cities, with randomly selected restlessness and risk-aversion factors.

After loading `game.scm` followed by `world.scm` the simulation is ready to go – we'll get back to that shortly.

Part 2: Tutorial exercises

Tutorial exercise 1: (a) Define a procedure `flip` (with no parameters) that returns 1 the first time it is called, 0 the second time it is called, 1 the third time, 0 the fourth time, and so on. (b) Now, define a procedure `make-flip` that can be used to generate procedures that have the same behavior as `flip` from part (a). That is, if we had `make-flip` for part (a), we could have implemented `flip` as `(define flip (make-flip))`. (c) Draw an environment diagram to illustrate the result of evaluating the following sequence of expressions:

```
(define (make-flip) ...)
(define flip1 (make-flip))
(define flip2 (make-flip))

(flip1) -->  value?
(flip2) -->  value?
```

Tutorial exercise 2: Assume that the following definitions are evaluated, using the procedure `make-flip` from the previous exercise:

```
(define flip (make-flip))
(define flap1 (flip))
(define (flap2) (flip))
(define flap3 flip)
(define (flap4) flip)
```

What is the value of each of the following expressions (evaluated in the order shown)?

[1] Actually, `*the-registrar-of-voters*` accepts three other messages: `tally`, `merge-results`, and `report-results` but these aren't part of the object's external interface.

```
flap1

flap2

flap3

flap4

(flap1)

(flap2)

(flap3)

(flap4)

flap1

(flap3)

(flap2)
```

Tutorial exercise 3: Consider the following code very carefully:

```
(define (get-method message preferred . others)
  ;; Get the "best" method, assuming objects are ordered from best to
  ;; worst.  GET-METHOD-FROM-OBJECT is in file game.scm.
  (define (loop objs)
    (let ((method (get-method-from-object message (car objs)))
          (rest (cdr objs)))
      (if (or (method? method) (null? rest))
          method
          (loop rest))))
  (loop (cons preferred others)))
```

Write a very short paragraph (no more than four sentences) describing how this code works. You might look at the notes or past problem sets for examples of good descriptive style; it should be correct, concise, and complete. Your description should, in particular, explain why it returns `method` when `(null? rest)` is true, even if `method` isn't really a method.

Tutorial exercise 4: The lab is much easier to do if you have a complete list of the types of objects and the class structure (which object inherits from which other). Create a sheet with this information on it, bring it with you, and make a copy for your tutor. There are two ways to prepare this information, and the choice is yours. The first is to draw a table with rows for data types and columns for messages, with a check mark if an object of the data type can handle the message. You can alternatively draw a directed graph with data types for nodes, arrows for showing which type inherits from which other, and with each data-type indicating the messages it can handle by itself. In the latter case, the root of the tree would be `named-object` (with messages `NAME`, `INSTALL`, and `SAY`).

Tutorial exercise 5: Suppose we evaluate the following expressions:

```
(define taxes (make-mobile-object 'student-money Cambridge))
(ask taxes 'CHANGE-LOCATION WashingtonDC)
```

At some point in the evaluation of the second expression, the expression

```
(set! location new-place)
```

will be evaluated in some environment. Draw an environment diagram, showing the full structure of `taxes` at the point where this expression is evaluated. Don't show the details of `Cambridge` or `WashingtonDC`—just assume that `Cambridge` and `WashingtonDC` are names defined in the global environment that point off to some objects that you draw as blobs.

Tutorial exercise 6: Suppose that, in addition to `taxes` in exercise 5, we define

```
(define local-taxes (make-mobile-object 'student-money cambridge))
```

Are `taxes` and `local-taxes` the same object (i.e., are they `eq?`)? What will result if we install `taxes` and `local-taxes` into the same location?

Part 3: Lab Warm-up Exercise

Lab Warm-up 1: Ask the politician `test-pol` and the city `Cambridge` for their names (both are defined in `world.scm`). Write a procedure that takes an object that inherits from (delegates to) `physical-object` and returns the name of that object's location. Test it by showing the name of the city in which the `test-pol` is located. Turn in the code and a transcript demonstrating that it works.

Lab Warm-up 2: Before trying out the simulation, let's take a look at it a bit. Notice that a politician, if it can't understand a message, delegates it to a `traveller`; and a `traveller` has a message `TELEPORT` which makes it choose a city at random and move there. So, we should be able to move our politicians around by sending them a `TELEPORT` message. In our current implementation, however, politicians are very close-mouthed about their movements. Let's fix that.

Change the code for `make-politician` so that it will print a single message when it receives a `TELEPORT` message. The message should inform us of the politician's location before and after teleportation. Turn in a listing of the `method` that implements the change, along with a short transcript showing that it works. Here's some possible output[2]:

[2] If you want to make the exact changes shown here, you will have to think very carefully about how `politicians`, `travellers`, and `persons` interact. In particular, think about how to get the politician to act like a `person` when (s)he speaks but as a `mobile-object` otherwise. Of course, it's hardly a surprise that it's hard to make a politician act like a person ...

```
(ask test-pol 'teleport)
At fairbanks : test says -- Teleported from fairbanks to cambridge
;Value: nuf-said
(ask test-pol 'teleport)
At cambridge : test says -- Teleported from cambridge to washingtondc
;Value: nuf-said
```

In principle, you could run the system by issuing specific commands to each of the objects in the world, but this defeats the intent of the game since that would give you explicit control over all the objects[3]. Instead, we will structure our system so that any object can be manipulated automatically in some fashion by the computer. We do this by creating a list of all the objects to be moved by the computer and by simulating the passage of time by a special procedure, `clock`, that sends a `clock-tick` message to each object in the list.

Different objects in our simulation react differently to the passage of time. In our simulation up to this point, only politicians and cities react to a clock tick.

Lab Warm-up 3: Louis Reasoner claims that having a separate `INSTALL` method for objects is unnecessary. He claims that the `INSTALL` method should be called automatically every time an object is created. For example, Louis proposes that `make-politician` be modified as follows:

```
(define (make-politician
          name initial-location thrill-seeking restlessness)
  (let ((traveller (make-traveller name initial-location))
        (ticks-to-go restlessness))
    (define politician                ;Louis added this line
      (lambda (message)
        (case message
          ...
          (else (get-method message traveller)))))
    (ask politician 'INSTALL)         ;Louis added this line
    politician))                      ;Louis added this line
```

Other make procedures would be modified the same way. Alyssa P. Hacker, however, disagrees. She says that Louis's implementation would leave "ghost" objects behind when politicians left cities.

Who is right? Provide a brief explanation of your answer.

Lab Warm-up 4: Write a very brief description (less than 3 sentences) describing what a politician does on each clock tick; be sure to explain what the `restlessness` argument to `make-politician` does. Do the same for a city.

Lab Warm-up 5: Now it's time to run the simulation at full speed for a while. The procedure `run-clock` will run the simulation for a specified number of clock ticks. Try running the simulation for, say, 5 ticks. Then hold an election by evaluating the expression `(ask *the-registrar-of-voters*`

[3] Besides, who types faster: you or the computer?

'ELECTION). The registrar will try to declare a winner, if possible, by polling the voters and attempting to get undecided voters to select a candidate. If the election is a tie, the registrar will re-run the vote with just the tied learders. For how many ticks did you have to run the simulation in order to have a clear winner after only one round of voting? Turn in a transcript of your interaction with the system[4].

Part 4: Lab exercises

When you load the code for problem set 7, the system will load `game.scm`. We do not expect you to have to make significant changes in this code, though you may do so if you want to.

The system will also set up a buffer with `world.scm` and load it into SCHEME. Since the simulation model works by data mutation, it is possible to get your SCHEME-simulated world into an inconsistent state while debugging. To help you avoid this problem, we suggest the following discipline: any procedures you change or define should be placed in your answer file; any new characters or objects you make and install should be added to `world.scm`. This way whenever you change some procedure you can make sure your world reflects these changes by simply re-evaluating the entire `world.scm` file. Finally, to save you from retyping the same scenarios repeatedly—for example, when debugging you may want to create a new character, move it to some interesting place, then ask it to act—we suggest you define little test "script" procedures at the end of `world.scm` which you can invoke to act out the scenarios when testing your code. See the comments in `world.scm` for details.

Lab exercise 1: Meta-adventure You can inspect an environment structure using the `show` procedure from `game.scm`. `Show` is a bit like the `pp` procedure you have used in the past for printing out procedures, but it prints things out so that they look more like parts of an environment diagram. It can be used like this:

```
(show cambridge)
#[compound-procedure 40]
Frame:
  #[environment 41]
Body:
  (lambda (message)
    (cond ((eq? message 'clock-tick) (lambda ... ...))
          ((eq? message 'install) (lambda ... ... ...))
          ((eq? message 'sponsor-debate) (lambda ... ...))
          ...))
```

Now you can inspect the environment of the procedure by calling `show` with the 'hash number' of the environment (41, in this example). The hash number is the number after '`compound-procedure`' or '`environment`' in the usual printed representations of these objects. The system guarantees that all different (i.e. non-`eq?`) objects have different hash numbers so you can tell if you get back to the same place.

[4] This is intended to be easy – just run the simulation and see what happens. Don't try to analyze your result or find a repeatable answer to this question!

```
(show 41)
#[environment 41]
Parent frame: #[environment 42]
place:        #[compound-procedure 43]
```

Here we see that the environment frame that is part of the procedure `cambridge` has one variable defined in it (`place`) and, of course, has a parent frame (number 42).

This exercise is called meta-adventure because you are going to use the `show` procedure to explore and 'map' the environment structure for `cambridge` and produce an environment diagram.

Start with `cambridge` and follow all the hash numbers *except* those of other cities, politicians, or voters (just show their names). There should be about 10 things to show. Print out the results and cut out the individual results. Arrange the pieces on a large blank piece of paper so that they are in the correct positions to make an environment diagram. Glue the pieces in place and draw in the arrows to make a complete environment diagram. Turn in your diagram.

Warning: The environment in this game can get very large in a hurry. Thus we strongly suggest that you start with a clean Scheme system, and just load `game.scm` and `world.scm`, then answer this question.

Adding to our world

We are now ready to start modifying our simulated world, in order to explore the interactions between different kinds of objects. We are first going to add a type of person called a reporter to our world. The idea behind a reporter is that he/she can interview a candidate to get a sound bite, which can then be broadcast to the voters. If you look at the code for `make-politician` you will see that the connection is already there to have a reporter do the interview, so let's look into creating the reporter.

The file `report.scm` contains the following template for creating a reporter.

```
(define make-reporter
  (lambda (voting-location noisy?)
      (define (reporter message)
        (case message
          ((REPORTER?) (lambda (self) true))
          ((INSTALL)
           (lambda (self)
             ...))
          ((INTERVIEW)
           ...)
           (else (get-method message person))))
      reporter)))

(define (make&install-reporter voting-location noisy?)
  (make&install-object make-reporter voting-location noisy?))
```

Lab exercise 2: Write the code for `INSTALL`. In principle, this should just involve asking the location of the reporter to add the reporter object to the list of things in that location. However, we can be more careful, by observing that a reporter is just a special kind of person. Thus, you should be able to modify the template to create a person object, perhaps with the name `anonymous-reporter`, in the same location, then use delegation to the person object to install the reporter. Turn in your implementation of the `INSTALL` method. Be sure to complete the default case of the template to handle inheritance of other behaviors.

Lab exercise 3: Write and turn in the code for `INTERVIEW`. This should connect to the method within a politician for conducting an interview. The method should allow the reporter and the politician to engage in a brief dialogue, creating a campaign "sound bite". One way to do this is to create a global list of witty sayings, and have the politician select one of those sayings at random.

Once you have done this, you can install a set of reporters by using some code that we have provided, in particular by evaluating:

```
(populate-reporters 2 *all-real-places*)
```

Inheriting Behavior by Delegating Messages

With the basic code for creating a reporter in place, we can turn to questions of how reporters fit into the object framework. If you think about it carefully you will realize that a reporter can also be a `voter`.

Lab exercise 4: Modify the reporter code you created above so that it can handle all of the messages sent to either a `person` or to a `voter`. Don't forget to modify the `INSTALL` handler (method) you wrote so that it installs any new objects that it creates. Turn in a listing of your changes.

Lab exercise 5: You have probably realized by now that just creating a reporter and having him/her interact with a politician has not effect on the voters. We need to connect their behavior with the voters. This means that we need to modify a voter object to handle the `WATCH-SOUNDBITE` message. Add such a method. You can select the details, but the behavior of the method should include the following:

- If the voter is undecided, then he should reconsider the politician who generated the sound bite with some probability.
- If the voter is already supporting the politician, then he should become less influencable.
- If the voter is supporting someone else, then he should become more influencable, and should reconsider this politician with some probability.

Be sure to have the voter engage in some dialogue as part of this.

Once you have made the changes to the `make-voter` procedure, reload the world, and test your code.

Lab exercise 6: The implementation of reporters is now complete. You should test it by writing a script which will create a set of reporters, and then just let the clock tick for some number of trials, before trying an election. Turn in your script and a transcript that shows it working. Run a few tests of your own, too, just to make sure that everything is working OK.

Adding other kinds of objects

Now, let's try adding a slightly more complicated object, a PACster, or Political Action Committee activist. The file `pacster.scm` contains a template for creating such objects.

```
(define (make-pacster name initial-location restlessness candidate)
  (let ((traveller (make-traveller name initial-location))
        (ticks-to-go restlessness))
    (lambda (message)
      (case message
        ((PACSTER?) (lambda (self) true))
        ((CLOCK-TICK)
         (lambda (self) ...
                 ))
        ((TRAVEL)
         (lambda (self)
           (ask self 'TELEPORT)))
        ((INSTALL)
         (lambda (self)
           (delegate traveller self 'INSTALL)
           (add-to-clock-list self)))
        ((GREASE)
         (lambda (self)
           ...))
        (else (get-method message traveller))))))

(define (make&install-pacster name initial-location restlessness candidate)
  (make&install-object
   make-pacster name initial-location restlessness candidate))
```

The behavior of the PACster should be as follows. The PACster is a mobile object, that can move from place to place. Thus, when the PACster gets restless enough, he TRAVELS. When he is not travelling, the PACSter engages in GREASE. This means that a random sampling of voters at the PACster's current location should be selected and asked to take a BRIBE.

Lab Exercise 7: Complete the above described methods in the template for a PACster. Turn in a listing of your code.

Lab Exercise 8: Modify the code to make `voter` objects, to handle the taking of a BRIBE. The kinds of behaviors to support are similar to those for watching a sound bite. Turn in a listing of your code.

Lab Exercise 9: To install some PACsters, we need to connect them with their candidates. Try evaluating the following code to do this.

```
(for-each make&install-pacster
          '(a-pac b-pac c-pac d-pac e-pac f-pac g-pac h-pac i-pac j-pac k-pac)
          (list Cambridge Cambridge Cambridge PaloAlto PaloAlto
                Denver Denver Denver Denver Kalamazoo)
          (list 5 2 3 1 5 6 2 5 3 4 6)
          *all-politicians*)
```

Reload your world, and try running a simulation involving reporters, voters, PACsters and politicians. Turn in a transcript showing the behavior of your objects.

Lab Exercise 10: Design and add some other object to this world. You can be as inventive as you like in doing this. Some examples might include a special investigator (we've actually left some hooks for this object in the code) who investigates candidates, with some effect on the voters; or a pollster who samples the current voters and reports on the latest trends, or something else. Turn in a listing of your code, and an example transcript of the system interacting with your additions.

Have fun. It's a presidential election year!

Part 5: Contest

Prizes will be awarded for the cleverest ideas turned in for this last problem.