MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1998

Problem Set 7

- Issued: Tuesday, March 17
- Tutorial preparation for: Week of March 30
- Written solutions due: Friday, April 3 in recitation
- Reading: Read through section 3.3.3 before lecture on Thursday, 19 March. Read section 3.5 before lecture on Tuesday, 31 March.
- Code: The following code (attached) should be studied as part of this problem set:
  - `objsys.scm`—support for an elementary object system
  - `objtypes.scm`—a few nice object classes
  - `setup.scm`—a building NE43 (Tech Square) world constructed using these classes
  - `search-rooms.scm`—a searching program for a robot in this world

Word to the wise: This problem set is the most difficult one so far this semester. But, perhaps paradoxically, it involves only a small amount of programming. The trick lies in knowing *which* programs to write, and for that, you must understand the attached code, which is considerable. You'll need to understand the general ideas of object-oriented programming and the implementation provided of an object-oriented programming system (in `objsys.scm`). Then you'll need to understand the particular classes (in `objtypes.scm`) and world (in `setup.scm`) that we've constructed for you. Finally, you'll need to review the searching strategy (in `search-rooms.scm`), which is a variation of the code you used last week in problem set 6. In truth, this assignment in much more an exercise in *reading* and *understanding* code than in writing code, because reading significant amounts of code is a skill that you should master if you intend to go on in computer science. The tutorial exercises will require you to do considerable digesting of code before you can start on them. And we strongly urge you to study the code before you begin trying the programming exercises themselves. Diving in starting to program without understanding the code is a good way to get lost, will virtually guarantee that you'll be spending more time on this assignment than necessary.

A maze of twisty little passages, all alike

In this problem set we will try to master a powerful strategy for building simulations possible worlds, as discussed in lecture on March 19. The strategy will enable us to make modular simulations with enough flexibility to allow us to expand and elaborate the simulation as our conception of the world expands and becomes more elaborate.

One way to organize our thoughts about a possible world is to divide it up into discrete objects, where each object will have a behavior by itself, and it will interact with other objects in some lawful way. If it is useful to decompose a problem in this way then we can construct a computational world, analogous to the "real" world, with a computational object for each real object.

Each of our computational objects has some independent local state, and some rules (or programs) that determine its behavior. One computational object may influence another by sending it messages. The program associated with an object describes how the object reacts to messages, and how its state changes as a consequence.

You may have heard about this idea in the guise of "Object-Oriented Programming"(OOPs!). Languages such as C++ and Java are organized around OOP. Although OOP is helpful in many circumstances it has been oversold as a panacea for the software-engineering problem. What we will try to understand here is the essence of the idea, rather than the accidental details of their expression in particular languages.

1. The Object Simulation

Consider the problem of simulating the activity of a few interacting agents wandering around a simple world of rooms and corridors. Real people are very complicated; we do not know enough to simulate their behavior in any detail. But for some purposes (for example, to make an adventure game) we may simplify and abstract this behavior.

Let's start with the simplest stuff first. We'll define some basic classes of objects, using the simple object system described in lecture on March 19. The objects in our computational world all have names. An object will give you a method to find out its name if you send it the `name` message. We can make a named object using the procedure `make-named-object`. A named object is a procedure that takes a message and returns the method that will do the job you want.[1] For example, if we call the method obtained from a named object by the message `name` we will get the object's name.

```
(define (make-named-object name)
  (lambda (message)
    (case message
      ((NAMED-OBJECT?) (lambda (self) #T))
      ((NAME) (lambda (self) name))
      ((SAY)
       (lambda (self list-of-stuff)
         (if (not (null? list-of-stuff))
             (display-message list-of-stuff))
         'NUF-SAID))
      ((INSTALL) (lambda (self) 'INSTALLED))
      (else (no-method)))))

(define foo (make-named-object 'george))

((foo 'name) foo) ==> george
```

[1] We will use the special form `case` to do the dispatch. See the Scheme Reference Manual for details.

The first formal parameter of every method is `self`. The corresponding argument must be the object that needs the job done. This was explained in lecture, and we will see it again below.

A named object has a method for four different messages. It will give a method that confirms that it is indeed a `named-object`; it will give a method to return its `name`; it will give a method for the message `say` that will print out some stuff; and it will give a method for `install`ation that does nothing.

A `place` is another kind of computational object. A place has has a `neighbor-map` of exits to neighboring places, and it has `things` that are located at the place. Notice that it is implemented as a message acceptor that intercepts some messages. If it cannot handle a particular message itself, it passes the message along to a private, internal named-object that it has made for itself to deal with such messages. Thus, we may think of a place as a kind of named-object except that it also handles the messages that are special to places. This kind of arrangement is described in various ways in object-oriented jargon, e.g., "the `place` class `inherits` from the `named-object` class," or "`place` is a *subclass* of `named-object`," or `named-object` is a *superclass* of `place`."

```
(define (make-place name . characteristics)
  (let ((neighbor-map    '())
        (things       '())
        (named-obj (make-named-object name)))
    (lambda (message)
      (case message
        ((PLACE?) (lambda (self) #T))
        ((THINGS) (lambda (self) things))
        ((CHARACTERISTICS) (lambda (self) characteristics))
        ((NEIGHBOR-MAP)
         (lambda (self) neighbor-map))

        ...other message handlers...

        (else     ; Pass the unhandled message on.
         (get-method message named-obj))))))
```

Where:

```
(define (get-method message object)
  (object message))
```

2. Messages and Delegation

Another idea that was described in lecture on March 19th is that of *delegation*, which is the use of one object's method by another object. For example, when you read the code in `objtypes.scm`, you will see definitions of several different kinds of objects. Among these are mobile objects (made by `make-mobile-object`) that can `change-location`, and persons (made by `make-person`). When a mobile object moves from one place to another the lists of `things` at the old location and at the new location must be changed: The object is removed from the list at the old location and added to the list at the new location.

A person is a kind of mobile object. When a person is constructed, an internal mobile object is also constructed to handle messages such as `change-location`. The mobile object is bound to a variable that is visible only within the person object.

When a person moves from one place to another, it does so by using the `change-location` method from its internal mobile object. However, it is the person that moves. Thus, it is the person that must be added or removed from the lists of things, not the mobile object from which the method was obtained. To implement this behavior the `change-location` method needs to know the actual moving object, and this is what is passed to the method as `self`.

This mechanism implements an idea called *delegation.* Indeed, to send a message to an object in this system we use `ask`. For example, if `me` is an object, named Joe, and we want its name, we say:

```
(ask me 'name) ==> Joe
```

What `ask` does here is get the `name` method from `me` and then call it with `me` as the argument (so the value of `me` will be bound to `self` in the method body). The full `ask` procedure is defined in the file `objsys.scm`, but here is a simplified version that works for messages requiring no arguments:

```
(define (ask object message)
  ((get-method message object) object))

(define (get-method message object)
  (object message))
```

Sometimes it is necessary to exercise more detailed control over inheritance. For example, one kind of character you will encounter in this problem set is an `avatar`. The avatar is a kind of person who must be able to do the sorts of things a person can do, such as `move-to` a new location. However, the avatar must be able to intercept the `move-to` message, to do things that are special to the avatar, as well as to do what a person does when it receives a `move-to` message. This is accomplished by explicit delegation. The avatar does whatever it has to, and in addition, it delegates to its internal person the processing of the `move-to` message, with the avatar as `self`.

One final note about our system. If you look in `objtypes.scm`, you'll see that objects have an `install` method, which does some appropriate initialization for a newly created object. For instance, if you create a new mobile object a at place, the object must be added to the list of things at the place. As you'll see in the code, we define two procedures for each type of object—a maker and a constructor. The constructor makes the object and then installs it. When you create objects in our simulation, you should do this using the constructor. Thus, to make a new person, use `construct-person` rather than calling `make-person` directly. Also if you decide to extend the simulation by creating new classes and new kinds of objects, you'll find it convenient to maintain this discipline of defining separate maker and constructor procedures.

3. Our World

Our world is built by the `setup` procedure that you will find in the file `setup.scm`. You are the deity of this world. When you call `setup` with your name, you create the world. It has rooms, corridors, and people from building NE43, and it has an avatar (a manifestation of you, the deity, as a person in the world). The avatar is under your control. It goes under your name and is also the value of the globally-accessible variable `me`. Each time the avatar moves, simulated time passes in the world, and the various other creatures in the world take a time step. The way this works is that there is a clock that sends a `clock-tick` message to all autonomous persons. (The avatar is not an autonomous person; it is directly under your control.) In addition, you can cause time to pass by explicitly calling the clock. If the global variable `*deity-mode*` is true you will see everything that happens in the world; if `*deity-mode*` is false you will only see things happening in the same place as the avatar.

To make it easier to use the simulation we have included a convenience procedure, `thing-named` for referring to an object at the location of the avatar. This procedure is defined at the end of the file `setup.scm`.

Here is a sample run of the system. Rather than describing what's happening, we'll leave it to you to examine the code that defines the behavior of this world and interpret what is going on.

```
(setup 'George-Spelvin)
---Tick 0---
You are in a wild long corridor
You see nothing in here.
The exits are: up in-408 in-430 in-429 in-428 .
;Value: ready

(ask (ask me 'location) 'name)
;Value: ne43-4th

(ask me 'go 'in-428)
george-spelvin says -- Hi gjs
---Tick 1---
You are in a cluttered laboratory office
You see: gjs sicp chalk .
The exits are: south out .
;Value: ok

(ask (thing-named 'chalk) 'owner)
;Value: nobody

(ask me 'take (thing-named 'chalk))
george-spelvin says -- I take chalk
;Value: #t

(ask me 'go 'out)
george-spelvin says -- Hi becky
---Tick 2---
```

```
You are in a wild long corridor
You see: chalk becky .
The exits are: up in-408 in-430 in-429 in-428 .
;Value: ok

(ask me 'go 'in-429)
george-spelvin says -- Hi hal
---Tick 3---
You are in a carefully arranged office
You see: chalk hal sicp chalk quiz2 .
The exits are: north out .
;Value: ok

(ask (thing-named 'quiz2) 'owner)
;Value: #[compound-procedure 12]

(ask (ask (thing-named 'quiz2) 'owner) 'name)
;Value: hal

(ask me 'take (thing-named 'quiz2))
hal says -- I lose quiz2
hal says -- Yaaaah! I am upset!
george-spelvin says -- I take quiz2
;Value: #t

(run-clock 5)
---Tick 4---
gjs says -- Hi george-spelvin hal
---Tick 5---
---Tick 6---
---Tick 7---
---Tick 8---
;Value: done

(set! *deity-mode* #t)
;Value: ()

(run-clock 3)
---Tick 9---
---Tick 10---
eric moves from ne43-7th to ne43-4th
At ne43-4th : eric says -- Hi gjs jim
gjs moves from ne43-4th to ne43-7th
---Tick 11---
becky moves from ne43-430 to ne43-4th
At ne43-4th : becky says -- Hi eric jim
jill moves from ne43-711 to ne43-7th
At ne43-7th : jill says -- Hi gjs
;Value: done
```

The Robot

We have provided you with an interesting kind of autonomous person, a robot. A robot can be created at any place that you have access to. The robot will sit around, and do nothing in particular, unless it is given a method to execute on clock ticks. In the file `search-rooms.scm` we provide the kernel of a program that will direct a robot to explore a world, looking for a thing with a given name. At the end of the file is an example where the robot is constructed at the location of the avatar, and it is programmed to look for an object whose name is `the-goal`.

4. Tutorial Exercises

You should prepare these exercises for oral presentation in tutorial.

Tutorial exercise 1: In the transcript above there is a line: `(ask (ask me 'location) 'name)`. What kind of value does `(ask me 'location)` return here? What other messages, besides `name`, can you send to this value?

Tutorial exercise 2: Look through the code in `objtypes.scm` to discover which classes are defined in this system and how the classes are related. For example, `place` is a subclass of `named-object`. Be prepared to draw the tree of subclass relationships and the messages that can be sent to each kind of object, using a diagram like the one presented in lecture on March 19. You will find such a diagram helpful (maybe indispensable) in doing the programming assignment.

Tutorial exercise 3: Look at the contents of the file `setup.scm`. What places are defined? How are they interconnected? Who is in each place in the initial state? Be prepared to draw a map. You must be able to show the places, the exits that allow one to go from one place to a neighboring place, the persons, the things, and the initial place for each person or thing.

Tutorial exercise 4: The avatar, as a person, may have possessions. How does the avatar handle the request `(ask me 'possessions)`? In particular, which method is used to respond to the request and which variable holds the list of possessions? Be prepared to sketch a skeletal environment diagram to explain your answer. Note that we are not asking you to draw a fully detailed environment diagram here—it is huge and useless!

Tutorial exercise 5: The file `search-rooms.scm` defines a search strategy maker that is similar to the search strategy maker you worked with in problem set 6. This week, you'll program a robot to follow this strategy to walk through a maze, searching for a valuable object. One difference between the robot version and the one for exercise 6 is the use of path procedures (implemented with `make-path-procedures`). What are these being used for? Why is this necessary for a robot walking through a maze but not for a spider crawling the Web as in exercise 6?

5. Programming Assignment

For this assignment you will need the ID number printed on your MIT ID card. Be sure to have it with you when you do the problems.

To warm up, load the code for problem set 7 and start the simulation by typing `(setup <`*your name*`>)`. Play with the world a bit. One simple thing to do is to stay where you are and run the clock for a while with `(run-clock <`*ticks*`>)`. Since the characters in our simulated building NE43 have a certain amount of restlessness, people should come walking by and say hi to you. Try running the clock with `*deity-mode*` set to both true and false. When it is set to true, you see everything that happens everywhere in the simulation. When it is set to false, you see only what happen in the room you are in. You'll probably want to set `*deity-mode*` it false most of the time.

Computer exercise 1: Walk the avatar to a room that has a thing called `the-goal` in it. Have the avatar `take the-goal`, and `lose` it somewhere else. Show a transcript of this session.

Searching the labyrinth

To solve the previous problem you did not really have to search for the goal: You had a map (from tutorial exercise 3) and you knew where the goal was, so you could plan a route and follow it directly. But suppose that you are in a world with no map, as in the World Wide Web. As you saw in problem set 6, you can search such a graph if you can keep track of where you have already been. You also need a bit more–see tutorial exercise 5. Hint: You may want to look at the procedure `reverse-exit` in the file `search-rooms.scm`.

Rather than searching the humdrum world of building NE43, you are to explore a magic labyrinth. If you enter the labyrinth unprepared, you are very likely to get very lost, because all the rooms have the same name and the same description.

You can explore the labyrinth using the same method you learned about in problem set 6. Namely, you can do a graph traversal, marking where you've been. Recall that exercise 6 implemented the marking by means of `make-mark-procedures`, which we used to construct two procedures, one for testing whether we've already visited a place, and one to mark the place as visited:

```
(define mark-procedures (make-mark-procedures))
(define deja-vu? (car mark-procedures))
(define note-visited! (cadr mark-procedures))
```

You'll find `make-mark-procedures` supplied for you in the file `search-rooms.scm`. By applying `deja-vu?` and `note-visited!` to your location as you walk around, you should be able to search the labyrinth without getting terribly lost.

Computer exercise 2: Reinitialize the world for this exercise by running `(start-exercise-2 <`*your name*`>)`. It makes the same NE43 world, with the addition of a randomly constructed labyrinth of twisty little passages, all alike. This labyrinth is accessible through an exit, constructed at the location of the avatar, called `Dis`. Somewhere in the labyrinth there is a thing named `diamond`. You should be able to use your mark procedures to help you find the diamond and bring it back into the world of NE43. In point of fact, the maze we construct here is very small. We don't want you to get lost—yet! So it is likely you'll quickly stumble across the diamond, even without a search strategy. But use the marks anyway, because you'll need to understand how this works for the rest of this problem set. Turn in a transcript showing a record of your activity in exploring the labyrinth.

Dropping pebbles

The mark procedures we provided for you in exercise 2 maintain a list of (pointers to) the rooms you've visited. But this is unrealistic, even in our fantasy magic world: What is the physical interpretation of carrying around a list of rooms to compare with your current location? (Does `eq?` make sense in the "real world"?) To be more realistic, we should mark the rooms in some other way. For example, when we enter a room, we can drop some object that we can later recognize if we ever return to the room.

Computer exercise 3: Define a new class of object called a `pebble`, and define a new version of `make-mark-procedures` such that `note-visited!` works by constructing a pebble and dropping it at the place being visited. The associated procedure `deja-vu?` checks to see if a pebble is present in the room visited. The pebble should be a new subclass of `thing`.

Notice that two searches will be confused if we cannot distinguish the pebbles used in the first search from the pebbles used in the second search. A simple way to deal with this is to name all the pebbles used in a given search with a fixed number, but to use different numbers for different searches (like using different colored pebbles for each search). The number must be incremented each time a search is begun (when `make-mark-procedures` allocates the name). Walk the avatar around NE43, dropping pebbles and recognizing them when you encounter them. (Hint: Look through `objtypes.scm` to see how new classes of objects are defined. Observe that for each class there is a `make` procedure, and an associated `construct` procedure, which is the procedure you call to actually construct the object. Also note that you should make your objects respond to a `PEBBLE?` message, so that you can use `is-a` to identify these as pebbles.)

A robotic assistant

You can now use your pebbles to search the labyrinth. But you do not want to do this for large labyrinth. Instead, you can program your robot to do the searching for you. Of course, you have to construct a robot before you can use it. Look at the definition of the `robot` object in `objtypes.scm`, to see how robots are constructed in general. Then, look at the procedure `start-robot-searching!` in `search-rooms.scm` to see how to the robot's program gets coupled to the search strategy.

Look at the procedure `setup-robot` at the end of the file `search-rooms.scm` to see how to construct a robot and start it up. The existing `setup-robot` program will find the object called `the-goal`, just as you did in computer exercise 1.

Computer exercise 4: Play with the robot a bit. Reinitialize the simulation with `(setup <`*your name*`>)` and run the robot until it finds `the-goal`. Do this with `*deity-mode*` true and also false so you can see how the robot is working. Now try `(start-exercise-4 <`*your name*`>)`, set up the robot to find the `diamond` (which is somewhere in the labyrinth) than `the-goal`) and observe how this works. There is nothing to turn in for this exercise.

Finding something of real value

For the next part of your adventure, you will search the maze for a truly valuable object: a magic scroll. If you can get your hands on the scroll, you can learn a magic word that you can use to get extra points on Quiz 2. (We're not kidding!) This magic word is personalized to you: Each student will get a different magic word.[2]

The magic scroll will be hidden inside a new labyrinth, which is a lot larger than the one for exercises 3 and 4. There are also a few characters that wander aimlessly through the labyrinth, and do nothing in particular. You *might* be able to search this labyrinth by hand, but we don't recommend trying. Instead, you should use a robot to perform the search.

There is, however, a catch.[3] The magic scroll will work only in Jill's office. So it's not enough to just send the robot searching for the scroll. You'll have to modify the robot's program so that it not only finds the scroll, but brings it back to you so you can take it to Jill's office to read it.

Computer exercise 5: Reinitialize the the world for this exercise by running `(start-exercise-5 <`*your name*`>)`. This creates a large labyrinth with the magic scroll hidden in it. The magic scroll is a new kind of object of class `scroll` (so that it responds to the `SCROLL?` message), and whose name is `magic-scroll`. Reprogram the robot to venture into the labyrinth and retrieve the scroll for you. Take the scroll to Jill's office and ask the scroll to `read`, and follow the instructions. If all works well, you'll obtain the magic word. You'll need to have your MIT ID number handy. Caution: You can read the scroll only once. After that, its magic will be used up. Keep your magic word for the quiz, but turn in your modified robot program for this exercise.

Warning: Like all things magical, the scroll must be handled with great care. It is content to be handled by a robot or an avatar. But if you try to "cheat" by ordering the scroll to move directly, the scroll's magic will be destroyed and you will have to restart the problem.

Exercise 6: Suppose we had a poltergeist, who randomly wanders around the labyrinth. The poltergeist picks up objects that are not `owned` and drops them in other places. Would your search robot work if the world included a poltergeist? If you think it would, explain. If you think it would not, explain. Does it make a difference to your answer whether pebbles are constructed as a subclass of `thing` vs. as a subclass of `named-object`? You need not provide a running example to back up your explanation.

[2] If you are collaborating with other students in doing this problem set, you should each do this part individually. To get points on the quiz, you will each need your personal magic word.

[3] There's always a catch, isn't there?

Optional Contest

Having read through to code for this problem set, you've seen that there are lots of possibilities to extend the world with new kinds of objects. (See, for example, the `troll` implementation in `objtypes.scm`.) Create a world of your own, with some novel characters and an interesting story line. Include a short narrative description of your work. We will award prizes for the most interesting stories combined with the cleverest technical ideas. Note that it is more impressive to implement a simple, elegant idea than to amass a pile of characters and places.

Turn in answers to the following questions along with your answers to the questions in the problem set:

1. About how much time did you spend on this homework assignment? (Reading and preparing the assignment plus computer work.)
2. Which scheme system(s) did you use to do this assignment (for example: 6.001 lab, your own NT machine, your own Win95 machine, your own Linux machine)?
3. We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout).
   - If you cooperated with other students, LA's, or others, or found portions of your answers for this problem set in references other than the text (such as some of the archives), please indicate your consultants' names and your references. Also, explicitly label all text and code you are submitting which is the same as that being submitted by one of your collaborators.
   - Otherwise, write "I worked alone using only the reference materials," and sign your statement.