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

Problem Set 7

- Issued: Thursday, October 29, 1998
- Tutorial preparation for: Week of November 9, 1998
- Written solutions due: Friday, November 13, 1998
- Reading: Read Section 3.4 by lecture on November 3, 1998, Section 4.1 by lecture on November 5, 1998, and Section 4.2 by lecture on November 10, 1998.
- 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 mansion world constructed using these classes
- Quiz 2 Reminder: November 4, 1998. 5–7PM xor 7–9PM, **room 3-270**.

Word to the wise: This problem set may be the most difficult one so far this semester. 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 the world (in `setup.scm`) that we've constructed for you. 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.

In this problem set we will try to master a powerful strategy for building simulations of possible worlds. 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 a murder mystery game) we may simplify and abstract this behavior.

Let's start with the simplest stuff first. We'll define some basic classes of objects. 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
```

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.

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

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 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 at a 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 create 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, objects, and people based on the Clue game (by Parker Brothers) 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. Also, 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`.

When you start the simulation, you will find yourself (the avatar) in one of the rooms of the Clue mansion. The Clue characters are also present somewhere in the mansion, and one of them is about to commit a murder by using one of the Clue weapons.[2] When a murder is committed, the victim screams and it's up to you to figure out who did it, in which room and with what weapon... but look out because you too (the avatar) can be killed. Note however, that once the murderer has found a victim, he becomes filled with remorse, repents and does not commit any more murders for the duration of the current simulation.

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 'ryan)
---Tick 0---
You are in a clean kitchen filled with delicious scents and aromas!
You see: lead-pipe refridgerator .
The exits are: north west secret-passage .
;Value: ready

(ask (ask me 'location) 'name)
;Value: kitchen

(ask me 'examine (thing-named 'refridgerator))
It is definitely GE!
;Value: done

(ask me 'take (thing-named 'lead-pipe))
ryan says -- I take lead-pipe
;Value: #t

(ask me 'go 'secret-passage)
ryan says -- Hi mrs-peacock
---Tick 1---
You are in a quiet study room. Shhh!
You see: lead-pipe mrs-peacock globe .
The exits are: secret-passage east south .
```

[2] If you are unfamiliar with the Clue game, just peruse the `setup.scm` code to become more familiar with this world.

```
;Value: ok

(ask (thing-named 'lead-pipe) 'owner)
;Value: #[compound-procedure 5]

(ask (ask (thing-named 'lead-pipe) 'owner) 'name)
;Value: ryan

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

(run-clock 3)
---Tick 2---
At foyer : miss-scarlet says -- I take wrench
---Tick 3---
mrs-white moves from lounge to dining-room
At billiard-room : professor-plum says -- I lose rope
miss-scarlet moves from foyer to ballroom
---Tick 4---
professor-plum moves from billiard-room to library
mrs-peacock moves from study to library
At library : mrs-peacock says -- Hi professor-plum
;Value: done
```

Remember to re-evaluate all definitions and re-run `setup` if you change anything just to make sure that all your definitions are up to date.

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`. Also look through the code in `setup.scm` to see what the world looks like. Draw a class diagram and a skeletal instance diagram like the ones presented in lecture. 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? Draw a map. You must be able to show the places and the exits that allow one to go from one place to a neighboring place.

Tutorial Exercise 4: Aside from you, the avatar, what other characters roam this world? What sorts of things are around? How is it determined which room each person and thing starts out in?

Tutorial Exercise 5: 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 more confusing than helpful!

5. Programming Assignment

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 world 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 should set `*deity-mode*` to false when you are ready to "play" the game and attempt to solve the murder.

Computer Exercise 1: Getting Acquainted with the System Walk the avatar to a room that has an unowned weapon. Have the avatar `take` this weapon, only to `lose` it somewhere else. Show a transcript of this session.

Computer Exercise 2: Understanding Installation Note how `install` is implemented as a method defined as part of `physical-object` and `autonomous-person`. Notice that the `autonomous-person` version puts the person on the clock list (this makes them "animated") then `delegate`s an `install` message from its `self` to its internal `physical-object`, which contains the `INSTALL` method responsible for adding the `person` to its `birthplace`. The relevant details of this situation are outlined in the code excerpts below:

```
(define (make-autonomous-person name birthplace laziness . characteristics)
  ;; Laziness determines how often the person will move.
  (let ((person (make-person name birthplace)))
    ...
      (case message
        ...
        ((INSTALL)
         (lambda (self)
           (add-to-clock-list self)
           (delegate person self 'INSTALL)))   ; **
        ...)))
```

```
(define (make-physical-object name location . characteristics)
  (let ((named-object (make-named-object name)))
    ...
      (case message
        ...
        ((INSTALL)
         (lambda (self) ; Install: synchronize thing and place
           (let ((my-place (ask self 'LOCATION)))
             ...
                (begin
                   (ask my-place 'ADD-THING self)
                   (delegate named-object self 'INSTALL))
                   ...)))))))
```

Louis Reasoner suggests that it would be simpler if we change the last line of the `make-autonomous-person` version of the `install` method to read:

```
                 (ask person 'INSTALL) ))   ; **
```

Alyssa points out that this would be a bug. "If you did that," she says, "then when you make and install an autonomous person, and this person moves to a new place, he'll be in two places at once!"

What does Alyssa mean? Specifically, what goes wrong? You may need to draw an appropriate environment diagram to explain carefully.

Computer Exercise 3: Who Just Died? Explore the world until "An earth-shattering, heart-wrenching, soul-piercing scream is heard...", which means that someone (hopefully not you) has just been murdered. Where does the victim go? If you know where the victim goes (and assuming you are not in `*deity-mode*`), what simple scheme expression can you evaluate to find out who just died?

Computer Exercise 4: On Closer Inspection In the next several exercises you will extend the system to add additional behaviors and nuances.

When the avatar enters a room, the current system reports `things` that are in that room. (You can do the same with `(ask (ask self 'LOCATION) 'THINGS)`). This is like a quick glance around to see what's there, but like any quick glance, it is possible to miss objects if they are hidden. We would like to make it possible for an avatar to `SEARCH` an object and thus discover a hidden object (if there is one) that wasn't visible before. For example, if you search the bookcase in the Library, you might find the 6.001 text in mint condition; if you search the Study, you might find a rare coin. Your job is to implement the capability to specify a hidden object as well as the ability to `SEARCH` an object.

Note that there may be several ways to achieve the desired behavior described above. Here is a very broad outline for one possible implementation to help guide you. Even with this one choice for an implementation, there are a few design decisions you will need to make:

(a) Think about what types of objects should be `SEARCHABLE` – which class would it make the most sense to add this capability to? Do you want to allow more than one hidden object?

(b) Create a local variable, `hidden-object`, that is local to each instance of the appropriate class and stores what the hidden object is.

(c) Create a method that is a means for you to specify what the hidden object is (e.g. a method that you call during `setup` that allows you to set the `hidden-object` local variable). Note that some more design decisions lurk here: What could you set it with? A symbol representing the name of the hidden object? The hidden object itself (e.g. as returned from `make-thing`)?

(d) Create a method `SEARCH` such that searching will reveal the `hidden-object` (if there is one). If there is no hidden object, it will say something to the extent that "You find nothing."

(e) (Optional) Other questions to consider: Once you've searched and found a hidden object, is it now always visible (if you were to `LOOK-AROUND` would you see it?) or is the object returned only when you search for it (see example session below)? What happens if you have taken the object and perform another `SEARCH`?

Hopefully, you have gotten a feel for all the intricacies and possibilities of our simulation world ("What happens if...??"), and enjoyed some freedom in your implementation design (so long as it is consistent and conforms to the behavior specifications we wanted).

Test your code and submit your code together with a transcript containing an example that shows that your code is working. Also include in your writeup the design decisions that you made. Remember to write neat and commented code.

Here is a sample of what the system could output:

```
(ask self 'LOOK-AROUND)
;Value: (rope knife mrs-white miss-scarlet piano)

(ask (thing-named 'piano) 'SEARCH)
You find nothing of interest
;Value: done

;; search the conservatory
(ask (ask self 'LOCATION) 'SEARCH)
You find what looks like a whistle
;Value: done

;; implemented so that it doesn't turn up if you look-around
(ask self 'LOOK-AROUND)
;Value: (rope knife mrs-white miss-scarlet piano)

(ask self 'TAKE (ask  (ask self 'LOCATION) 'SEARCH))
ryan says -- I take whistle
;Value: #t

(ask (ask self 'LOCATION) 'SEARCH)
You find nothing of interest
;Value: done
```

```
;; Ryan is now the proud owner of a whistle
```

Computer Exercise 5: Fingerprinting Weapons Next, extend the system so that when someone picks up a weapon (or any `thing` object), his/her fingerprints are left behind and recorded on the object.

(a) modify `make-thing` so that it can keep track of its previous owners.

(b) extend `make-thing` with a method `FINGERPRINT` that returns a list of all its previous owners.

Submit your code and include a short transcript (only relevant parts) that shows that this code is working properly.

Computer Exercise 6: Reading Fingerprints Carefully Now create a magnifying glass, and modify the system so that the fingerprints on a weapon are visible only to someone who is holding the magnifying glass.

Submit your code and detail any design decisions you made. Also include a short transcript (only the relevant parts in your excerpt) that shows that this code is working properly.

Computer Exercise 7: Solving the Crime Once a murder has been committed, it is your task to solve the mystery. You can figure out where the murder occurred (examine the code and explain how). Armed with the magnifying glass, you can examine the fingerprints on each weapon you find to figure out which people it has come in contact with. Lastly, you can ask each character for his/her `ALIBI`. An alibi is simply a report from a Clue character (now turned suspect) that says (1) where they were when the murder occurred, (2) what was in their possession at that time and (3) who else was in the room. These three pieces of information are returned as a list, and the suspect will interpret them for you as a side effect. For example:

```
(ask (thing-named 'colonel-mustard) 'ALIBI)
colonel-mustard says -- Me?  I was in the conservatory
colonel-mustard says -- I had in my possession: (knife)
colonel-mustard says -- Oh, and  (miss-scarlet) was in the room with me
;Value: (conservatory (knife) (miss-scarlet))
```

If you think you have solved the crime, you can then hazard a `GUESS` as to the room, weapon and murderer (in this order). For example:

```
(ask me 'GUESS 'library 'wrench 'mr-green)
```

Turn in a sample excerpt (not the whole transcript) showing that everything works appropriately and that you can indeed figure out who did it, where and with what. Note that in some cases, you may not be able to deduce the answer uniquely so you may have to make a few guesses.

Optional Computer Exercise: An Automated Detective Now that you have some experience with the Clue world and have manually done some detective work, describe and implement a detective which automates the process of solving the crime. This problem is challenging, and is completely optional.

Optional Contest

Having read through the code for this problem set, you've seen that there are lots of possibilities to extend the world with new kinds of objects. Create some new types of novel objects/characters with special properties, extend the current objects with new methods, or add some interesting twists to the storyline. Include a short narrative description of your work. We will award prizes for the most interesting modifications 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.