MC28 Lab 5
Adventures in the Imaginary Land of Gack

Max Hailperin Karl Knight

Prelab due as HW5 November 3, 1995; Lab due November 13, 1995

In this lab, we will be exploring two ideas: the simulation of a world in which objects are characterized by a set of state variables, and the use of message passing as a programming technique for modularizing worlds in which objects interact. We will do this in the context of a simple simulation game like the ones available on many computers. Such games have provided an interesting waste of time for many computer lovers. In order to not waste too much of your own time, you should read the description of the system carefully, and plan your work in advance. You should be sure to look at the problems in the prelab section before coming to class on November 1^{st}; they are due as homework five in class on November 3^{rd}. *Note:* because of slight differences between the Scheme versions used, you should do this lab on the NeXTs. This lab is adapted from problem set 6 from MIT's course 6.001 in Fall of 1989, which is covered by the following copyright notice:

> Copyright © 1990 Massachusetts Institute of Technology
>
> This material was developed by the Scheme project at the Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science. Permission to copy this material, to redistribute it, and to use it for any non-commercial purpose is granted, subject to the following restrictions and understandings.
>
> 1. Any copy made of this material must include this copyright notice in full.
> 2. Users of this material agree to make their best efforts (a) to return to the MIT Scheme project any improvements or extensions that they make, so that these may be included in future releases; and (b) to inform MIT of noteworthy uses of this material.
> 3. All materials developed as a consequence of the use of this material shall duly acknowledge such use, in accordance with the usual standards of acknowledging credit in academic research.
> 4. MIT has made no warrantee or representation that this material (including the operation of software contained therein) will be error-free, and MIT is under no obligation to provide any services, by way of maintenance, update, or otherwise.
> 5. In conjunction with products arising from the use of this material, there shall be no use of the name of the Massachusetts Institute of Technology nor of any adaptation thereof in any advertising, promotional, or sales literature without prior written consent from MIT in each case.

Prelab

The essential system—places, persons, and things

The basic idea of simulation games is that the user plays a character in an imaginary world inhabited by other characters. The user plays the game by issuing commands to the computer that have the effect of moving the character about and performing acts 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 an an imaginary world called Gack, with imaginary places such as a computer lab atop the good ship Olin. Note well that this Land of Gack is entirely imaginary; we could have mixed in some real elements, but that would make it too complex. 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-thing name)
(make-place name)
(make-person name place threshold)
```

Each time we make a place or a thing, we give it a name. In addition, a person is created at some initial place, and with a threshold that determines how often the person moves. For example, the procedure `make-person` may be used to create the two imaginary characters, `max` and `karl`. (Note that persons have to be installed, to bring them to life. Also, other things need to be done to places in order that their position is known relative to other places. See the procedure `can-go` in `gack.scm`.)

```
(define computer-lab (make-place 'computer-lab))
(define offices (make-place 'offices))

(define max (make-person 'max offices 2))
(install-person max)

(define karl (make-person 'karl computer-lab 4))
(install-person karl)
```

All objects in the system are implemented as message-accepting procedures. Once you load the system in the laboratory, these and other objects will be created, and you will be able to control `max` and `karl` by sending them appropriate messages. As you enter each command, the computer reports what happens and where it is happening:

```
(ask max 'look-around)
At offices : max says -- I see nothing and can go north
;No value
```

```
(ask max 'go 'north)
max moves from offices to lounge
;No value

(ask karl 'look-around)
At computer-lab : karl says -- I see nothing and can go east
;No value

(ask karl 'go 'east)
karl moves from computer-lab to lounge
At lounge : karl says -- Hi max
;No value
```

In principle, you could run the system by issuing specific commands to each of the creatures in the world. But this defeats the notion of the game, since that would give you explicit control over all the characters. Instead, we will structure our system so that any character can be manipulated in some fashion by the computer. We do this by creating a list of all the characters to be moved by the computer, and by simulating the passage of time by a special procedure, `clock`, that sends a `maybe-act` message to each creature in the list. A `maybe-act` message does not automatically imply that the creature receiving it will perform an action. Rather, like all of us, a creature hangs about idly until he or she (or it) gets bored enough to do something. To account for this, the third argument to `make-person` specifies the number of clock intervals that the person will wait before doing something. We also can say `(run-clock` n`)` to step the clock n times, to more quickly see what happens.

Before we trigger the clock to simulate a game, let's explore the properties of our world a bit more.

First, let's create a `unix-programmers-manual` and place it in the `lounge` (where `max` and `karl` now are).

```
(define unix-programmers-manual (make-scroll 'unix-programmers-manual))
;Value: unix-programmers-manual

(ask lounge 'appear unix-programmers-manual)
;Value: appeared
```

Notice that the manual is a "scroll," which is a specialized kind of "thing." We'll return to this issue in a bit. Next, we'll have `karl` look around. He sees the manual and `max`. The manual is supposed to stay in the computer lab, so `karl` takes it and leaves.

```
(ask karl 'look-around)
At lounge : karl says -- I see unix-programmers-manual max and can go
south west
;No value
```

```
(ask karl 'take unix-programmers-manual)
At lounge : karl says -- I take unix-programmers-manual
;No value

(ask karl 'go 'west)
karl moves from lounge to computer-lab
;No value
```

Max, however, had wanted to use the manual, so he follows Karl and snatches the manual away. Angrily Karl sulks off to the offices:

```
(ask max 'go 'west)
max moves from lounge to computer-lab
At computer-lab : max says -- Hi karl
;No value

(ask max 'take unix-programmers-manual)
At computer-lab : karl says -- I lose unix-programmers-manual
At computer-lab : karl says -- Yaaaah! I am upset!
At computer-lab : max says -- I take unix-programmers-manual
;No value

(ask karl 'go 'east)
karl moves from computer-lab to lounge
;No value

(ask karl 'go 'south)
karl moves from lounge to offices
At offices : karl says -- Hi barbara
;No value
```

Unfortunately for Karl, he has run into the wicked witch, Barbara. A witch is a kind of person; it can move around, take things, and so on. When a witch gets a `maybe-act` message from the clock, it acts just like an ordinary person—unless someone else is in the room. When Barbara decides to act, it's bad news for Karl:

```
(ask barbara 'maybe-act)  ; note she doesn't act until the third time
;Value: done

(ask barbara 'maybe-act)
;Value: done

(ask barbara 'maybe-act)
At offices : barbara says -- Hah hah hah, I'm going to turn you into a
frog karl
At offices : karl says -- Ribbitt!
karl moves from offices to pond
At offices : barbara says -- Hee hee karl looks better in green!
;Value: done
```

The object system—inheritance

Because a witch is a kind of person, she has the capabilities of a person. However, she may have additional capabilities which are explicitly part of her witch-ness. In order to support this kind of description, we need to allow objects to inherit properties of the larger classes they are part of.

In this problem set, we expand on the idea of a message-passing implementation of an object system so we can say things like, "A witch (object) is a special type of person (object)." In other words, we would like one type of object to be able to "inherit" the "methods" used by the objects which are the more general form of the same object.

The beginning of our language for describing these concepts is simply a set of conventions for using computational objects. (The structure of this system follows that given by J. Rees and N. Adams, "Object-oriented programming in Scheme," in the 1988 *ACM Conference on Lisp and Functional Programming.*)

An object in our system is a procedure which, given a message as an argument, returns another procedure called a method. A method is a procedure which, given the appropriate arguments, will perform the action requested by the message. For example, the following procedure creates a simple object called a speaker whose only method is that it can say something (a list), which it does by displaying the list on a new line:

```
(define (make-speaker)
  (lambda (message)
    (cond ((eq? message 'say)
           (lambda (self stuff)
             (newline)
             (display stuff)))
          (else (non-method "SPEAKER")))))
```

The speaker object, like all objects in our system, is represented as a procedure which takes a message. It returns a method that takes two arguments, the identity of the object that is using the method (this argument, named `SELF`, is not used here, but it will be important later), and a second argument, `STUFF`, which is what is to be said.

We will get a method from an object by asking for it:

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

We make the convention that if an object does not recognize the message it is given, it will signal this fact by using the procedure `non-method`, which returns something that our object system will recognize as not a method.

```
(define (non-method name)
  (list 'non-method name))

(define (method? x)
  (procedure? x))
```

To `ask` an object to apply one of its methods to some arguments, we pass the object a message asking for the appropriate method, and apply the method to the object and its arguments. If the object has no method for this message, `ask` will produce an error. (The procedure `apply` takes a procedure and a list of arguments and applies the procedure to the arguments. Note: We use dot notation in the formal parameters of `ask`, so that `args` will be a list of the arguments we wish to pass along. This is a feature of scheme you have not seen before. In the below, `ask` can be used with two or more arguments. The first gets called `object`, the second gets called `message`, and a list of the zero or more remaining ones is formed and called `args`.)

```
(define (ask object message . args)
  (let ((method (get-method object message)))
    (if (method? method)
        (apply method (cons object args))
        (error "No method" message (cadr method)))))
```

Now we can make a speaker object and ask it to say something:

```
(define george (make-speaker))
;Value: george

(ask george 'say '(the sky is blue))
(the sky is blue)
;No value
```

One thing we may want to do is define an object type to be a specialization of some other object type. For instance, we could say that a lecturer is a kind of speaker, expecting that the lecturer also has a method called lecture. To lecture something, the lecturer says it and then says "You should not be taking notes":

```
(define (make-lecturer)
  (let ((speaker (make-speaker)))
    (lambda (message)
      (cond ((eq? message 'lecture)
             (lambda (self stuff)
               (ask self 'say stuff)
               (ask self 'say '(you should not be taking notes))))
            (else (get-method speaker message))))))
```

Observe that we accomplish this by giving the lecturer an internal speaker of its own. If the message is not recognized, `lecturer` passes it on to the internal speaker. Thus, a lecturer can do anything a speaker can (i.e., say things), and also lecture. In the object-oriented programming jargon, one says that `lecturer` inherits the `say` method from `speaker`, or that `speaker` is a superclass of `lecturer`.

```
(define barbara (make-lecturer))
;Value: barbara
```

```
(ask barbara 'say '(the sky is blue))
(the sky is blue)
;No value

(ask barbara 'lecture '(the sky is blue))
(the sky is blue)
(you should not be taking notes)
;No value
```

Every method takes `self`, the actual object, as its first parameter. To understand why, consider this:

Suppose we wanted to define an arrogant-lecturer to be a kind of lecturer. But whenever an arrogant lecturer says anything, he prefaces his statement with "It is obvious that ...":

```
(define (make-arrogant-lecturer)
  (let ((lecturer (make-lecturer)))
    (lambda (message)
      (cond ((eq? message 'say)
             (lambda (self stuff)
               (ask lecturer 'say (append '(it is obvious that) stuff))))
            (else (get-method lecturer message))))))

(define max (make-arrogant-lecturer))

(ask max 'say '(the sky is blue))
(it is obvious that the sky is blue)
```

Now consider what happens when the following `ask` happens:

```
(ask max 'lecture '(the sky is blue))
(it is obvious that the sky is blue)
(it is obvious that you should not be taking notes)
```

Because an arrogant-lecturer has no direct method for handling the `lecture` message, Max asks his internal lecturer to handle the `lecture` message; he does this by returning `(get-method lecturer message)`. When `ask` applies the method, it passes `max` and the arguments to the method. Thus, `self` will be bound to `max` in the body of the method. This ensures that, even though we are using the `lecture` method from `lecturer`, the `say` method from `max`, not from `lecturer`, will be used to do the actual saying.

Actually, the above definition of `make-arrogant-lecturer` is still not quite perfect, in a way that we'll explore in class, probably on November 3^{rd}. Prelab problem 3 below shows how it can be improved, and asks you to start thinking about why it would be an improvement.

Implementation

The simulator for the world is contained in the files `adventure.scm` and `gack.scm`, in the directory `~karl/Public/MC28`; printouts are included separately. The files contain procedures to create people, places, things, scrolls, wizards, and witches, and code to initialize the Imaginary Land of Gack.

Problem 1 Suppose we execute the following expressions.

```
(define pizza (make-thing 'pizza))
(ask pizza 'change-owner max)
```

At some point in the evaluation of the second expression, the expression `(set! owner new-owner)` will be evaluated in some environment. Draw an environment diagram, showing the full structure of `pizza` at the point where this expression is evaluated. Don't show the details of `max`—just assume that `max` is a symbol defined in the global environment that points off to some object that you draw as a blob.

Problem 2 Suppose that, in addition to `pizza` in problem 1, we define

```
(define deluxe-veggie-combo (make-thing 'pizza))
```

Are `pizza` and `deluxe-veggie-combo` the same object (i.e., are they `eq?`)? If `max` wanders to a place where they both are and looks around, what message will he display?

MAYBE-ACT and ACT

Note how `maybe-act` is implemented as a method defined as part of `person`. If the number of clock ticks elapsed since the previous action has not passed the person's threshold, nothing happens. But if enough time has gone by, the response to `maybe-act` is to send oneself an `act` message. Special characters such as witches have a threshold like ordinary people. Thus they use the same `maybe-act` method. But when they actually do something, they do something special because their `act` method is different from ordinary people's. For example, here is the witch's `act` method, reprinted from the code:

```
(lambda (self)
  (let ((others (other-people-at-place self (ask self 'place))))
    (if (not (null? others))
        (ask self 'curse (pick-random others))
        ((get-method p 'act) self))))
```

The effect of the last line is that if there are no other people where the witch is located, the witch will behave like an ordinary person. She does this by getting and using the `act` method from `p`, the internal person associated with this particular witch (i.e., this witch "as a person").

Problem 3 Louis Reasoner suggests that it would be simpler if we changed the last line of the method to read:

```
(ask p 'act)
```

"After all," he says, "that is analogous to what was done with the `say` method of arrogant-lecturer when it delegated to the lecturer's say method."

Allysa P. Hacker replies that Louis's change would be a bug, and that in fact it was the arrogant-lecturer method that should be changed; it should use `get-method` to get the `lecturer`'s method and then pass it `self` and the message to say.

Write out the improved `say` method for `make-arrogant-lecturer`. Think a little about why this more long-winded means of delegating to the super-class object might be preferable. I don't expect you to necessarily figure it out—we'll talk about it in class.

In Lab

You should open both of the files from `~karl/Public//MC28`, namely `adventure.scm` and `gack.scm`. You should do a "Save As" command to save `gack.scm` into your own directory. That way, you can add your own creatures, objects or places into this file. Whenever you want to re-initialize your world, you just have to do an "Evaluate All" in the `gack.scm` window.

Remember that the current state of the objects in the world may still reflect past, buggy versions of your procedures even after you have fixed the procedures themselves. Therefore, if you are confused it is definitely worthwhile to try re-initializing the world and seeing if the problems go away.

It would be clearest if you kept all your procedure definitions (including any modified procedures from `adventure.scm`) in yet a third file.

After loading the system, make the various characters move around by repeatedly calling `clock` (with no arguments). Which person is most restless? How often do both Max and Barbara move at the same time?

Problem 1 Install a new character, yourself, with a high enough threshold (say, 100) so that you have "free will" and will not be moved by the clock. Place yourself initially in the `dormitory`. Create a new thing, `late-lab-report`, and also place it in the `dormitory`. (You can make it a scroll rather than a vanilla "thing" if you think it more appropriate.) Pick up the `late-lab-report`, find out where `max` is, go there, and try to get `max` to take the homework even though it's late. Can you find a way to do this that does not leave you upset? Turn in a list of your definitions and actions. If you wish, you can intersperse your moves with call to the clock to make things more interesting. (Watch out for Barbara!) Rather than having to manually invoke the clock between each move, you may find it easier to use a definition such as

```
(define difficulty 1) ; you can increase this if you are bored

(define (play . args)
  (let ((result (apply ask (cons me args)))) ; change me as appropriate
```

```
    (run-clock difficulty)
    result))
```

A more sophisticated solution to this problem would be to make yourself be an instance of a subclass of person that causes the clock to be advanced.

Problem 2 Now you have the elements of a simple game: try to get from the dormitory to the computer lab without getting turned into a frog along the way. To make it more challenging, try to get to the lab possessing the scroll of enlightenment.

Problem 3 Of course, you can make the game much more interesting, by adding additional twists. Implement some subset of the following features which interests you, or come up with some of your own. Turn in a listing of your new and modified procedures. For modified procedures, mark the sections you changed. Describe in English whatever new ideas you introduce.

- Add a kind of thing `chocolate`. They should have a `chocolate?` method that returns `true` and they should be able to `be-eaten`. When a chocolate is eaten the owner loses it and it disappears from the place the owner is at. You can optionally add cute sound effects or make some crumbs appear at the place. Make some chocolates appear at `food-service`.
- Modify `make-person` so people have an `eat` method that causes a chocolate they own to be eaten, much like the `read` method. Note that the necessity of adding such methods is one of the decided shortcomings to our implementation; we may discuss this in class.
- Now that we have `chocolate`s, we can modify `make-witch` to reflect a little-known property of witches: they can be bought off with chocolates. Change the `act` procedure so that if Barbara's victim has any chocolates, Barbara will take and eat one, rather than turning the victim into a frog. Only if the victim possesses no chocolates will he/she become a frog. This makes the game more interesting, in that you have to decide whether to take the extra time to make a detour to food service for chocolates or go for speed but risk being caught unprepared by Barbara.
- If you want to make the game harder yet, you can make it difficult to hoard protective chocolates by creating a troll who wanders around eating chocolates. Perhaps you should change Max from being a normal person to being the troll—it was probably just the fact that he wrote the assignment that kept him from being cast as a troll in the first place.
- Introduce magic scrolls into the game, which are a kind of scroll that has a limited number of "charges," (call it n) and when read the first n times invokes its `do-magic` method. Define one or more specific kinds of magic scrolls with interesting `do-magic` methods; for example, you might make a `scroll-of-teleportation`.

- It makes the game rather easy if the title of a magic scroll tells what it does. You could add a special `name` method to magic scrolls such that until the first time they are read the name is just `a-mysterious-scroll` and thereafter is the actual title. This feature is particularly effective if you include scrolls with undesirable effects as well as desirable ones. Of course, you could counter by including a scroll-of-identification that can be used to decode the titles.
- Currently Elvee always takes the first scroll at a place. Since there are some scrolls that are more valuable than others, this can make the game boring. Change the definition of `make-great-white-wizard` so he chooses a scroll at random to take.
- Currently it is an error to read anything other than a scroll. Make an appropriate modification so that instead you get the message "Nothing is written on *whatever*." Be sure to do this in a way that allows you to later add other kinds of things that can be read.
- As remarked above, please feel free to add you own interesting kinds of people, places and things. (Maybe two munchkins called `erick` and `little-karl`? What is special about munchkins?)