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 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 not to waste too much of your own time, it is important to study the system and plan your work before starting to write any code.

This problem set begins by describing the overall structure of the simulation. The warm-up exercises in Part 2 will help you to master the ideas involved. Part 3 contains the assignment itself.

Part 1: The SICP Adventure Game

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 a strange, imaginary world called UMass Lowell, with imaginary places such as a computer lab, a robot lab, and a department office. 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:

\[
\begin{align*}
&\text{(make-thing name)} \\
&\text{(make-place name)} \\
&\text{(make-person name birthplace restlessness)}
\end{align*}
\]

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:

\[
\begin{align*}
&\text{(make-install-thing name birthplace)} \\
&\text{(make-install-person name birthplace restlessness)}
\end{align*}
\]
Each time we make or make and install a person or a thing, we give it a name. People and things also are created at some initial place. In addition, a person has a restlessness factor that determines how often the person moves. For example, the procedure `make-install-person` may be used to create the two imaginary characters, holly and samip, and put them in their places, as it were.

```lisp
(define computer-lab (make-place 'computer-lab))
(define robot-lab (make-place 'robot-lab))
(define holly (make-install-person 'holly robot-lab 3))
(define samip (make-install-person 'samip computer-lab 2))
```

All objects in the system are implemented as message-accepting procedures.

Once you load the system on your machine, you will be able to control holly and samip by sending them appropriate messages. As you enter each command, the computer reports what happens and where it is happening. For instance, imagine we had interconnected a few places so that the following scenario is feasible:

```lisp
(ask holly 'look-around)
At robot-lab : holly says -- I see nothing
;Value: ()
(ask (ask holly 'place) 'exits)
;Value 12: (south)
(ask holly 'go 'south)
holly moves from robot-lab to west-hall
;Value: #t
(ask holly 'go 'east)
holly moves from west-hall to elevator-lobby
;Value: #t
(ask holly 'go 'north)
holly moves from elevator-lobby to computer-lab
At computer-lab : holly says -- Hi samip
;Value: #t
(ask samip 'look-around)
At computer-lab : samip says -- I see holly
;Value 13: (holly)
```

In principle, you could run the system by issuing specific commands to each of the creatures in the world, but this defeats the intent 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 automatically 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 `move` message to each creature in the list. A `move` 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 average number of clock intervals that the person will wait before doing something (the restlessness factor).

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 `computer-manual` and place it in the `computer-lab` (where holly and samip now are).

```lisp
(define computer-manual (make-install-thing 'computer-manual computer-lab))
```

Next, we'll have holly look around. She sees the manual and samip. The manual looks useful, so we have holly take it and leave.
Samip had also noticed the manual; he follows holly and snatches the manual away. Angry, holly sulks off to the network-closet:

Unfortunately for holly, beneath the network closet is an inaccessible dungeon, inhabited by a troll named grendel. A troll is a kind of person; it can move around, take things, and so on. When a troll gets a move message from the clock, it acts just like an ordinary person—unless someone else is in the room. When grendel decides to act, it’s game over for holly:

Implementation  The simulator for the world is contained in two files, which are included at the end of the problem set. The first file, game.scm, contains the basic object system, procedures to create people, places, things and trolls, together with various other useful procedures. The second file, world.scm, contains code that initializes our particular imaginary world and installs holly, samip, and grendel.

You can find these files on the web site. Use \texttt{M-x eval-current-buffer} when in each buffer to evaluate the code into your scheme environment. Evaluate the \texttt{game.scm} file, then the \texttt{world.scm} file.
Part 2: Warm-up Exercises

Warm-up Exercise 1: Draw a simple inheritance diagram showing all the kinds of objects (classes) defined in the adventure game system, the inheritance relations between them, and the methods defined for each class.

Warm-up Exercise 2: Draw a simple map showing all the places created by evaluating world.scm, and how they interconnect. You will probably find this map useful in dealing with the rest of the problem set.

Warm-up Exercise 3: Suppose we evaluate the following expressions:

```
(define ice-cream (make-thing 'ice-cream robot-lab))
(ask ice-cream 'set-owner holly)
```

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 ice-cream at the point where this expression is evaluated. Don’t show the details of holly or robot-lab—just assume that holly and robot-lab are names defined in the global environment that point off to some objects that you draw as blobs.

Warm-up Exercise 4: Suppose that, in addition to ice-cream in Exercise 3, we define

```
(define rum-and-raisin (make-named-object 'ice-cream))
```

Are ice-cream and rum-and-raisin the same object (i.e., are they eq?)? If holly wanders to a place where they both are and looks around, what message will she print?

Exercises

The first exercise here should be written up and turned in with your problem set, but we suggest that you do it before you start coding the assignment, because it illustrates a bug that is easy to fall into when working with the adventure game.

Exercise 1: Note how install is implemented as a method defined as part of both mobile-object and person. Notice that the person version puts the person on the clock list (this makes them “animated”) then invokes the mobile-object version on self, which makes the birthplace where self is being installed aware that self thinks it is in that place. That is, it makes the self and birthplace consistent in their belief of where self is. The relevant details of this situation are outlined in the code excerpts below:
(define (make-person name birthplace threshold)
  (let ((mobile-obj (make-mobile-object name birthplace))
    ...)
  (lambda (message)
    (cond ...)
    ((eq? message 'install)
      (lambda (self)
        (add-to-clock-list self)
        ((get-method mobile-obj 'install) self) )) ; **
    ...)
))

(define (make-mobile-object name place)
  (let ((named-obj (make-named-object name)))
    (lambda (message)
      (cond ...
        ((eq? message 'install)
          (lambda (self)
            (ask place 'add-thing self)))
        ...))))

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

  (ask mobile-obj 'install) )) ; **

Alyssa P. Hacker points out that this would be a bug. “If you did that,” she says, “then when you make\install-person holly and holly moves to a new place, she’ll thereafter be in two places at once! The new place will claim that holly is there, and holly’s place of birth will also claim that holly is there.”

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

Exercise 2: We do not expect you to have to make significant changes in the game.scm code, though you may do so if you want to.

You will also have a world.scm buffer. 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.

After loading the files, make holly and samip move around by repeatedly calling clock (with no arguments).

(a) Which person is more restless? (b) How often do both of them move at the same time?

Exercise 3: Make and install a new character, yourself, with a high enough threshold (say, 100) so that you have “free will” and are not likely to be moved by the clock. Place yourself initially in the computer-lab. Also make and install a thing called late-homework, so that it starts in the computer-lab. Pick up the late-homework, find out where holly is, go there, and try to get holly to take the homework even though he is notoriously
adamant in his stand against accepting tardy problem sets. 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 calls to the clock to make things more interesting. (Watch out for grendel!)

Exercise 4: Meta-adventure You can inspect an environment structure using the show procedure from game.scm. Show is a bit like the pp procedure you should have used in intro problem set 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 samip)
#\[compound-procedure 25]\nFrame:
#\[environment 26]\nBody:
(lambda (message)
  (cond ((eq? message ...) (lambda ... true))
        ((eq? message ...) (lambda ... possessions))
        ((eq? message ...) (lambda ... ... possessions))
          ...
))
;Unspecified return value

Now you can inspect the environment of the procedure by calling show with the 'hash number' of the environment. 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.

(show 26)
#\[environment 26]\nParent frame: #\[environment 27]\npossessions: (#\[compound-procedure 28]\nmobile-obj: #\[compound-procedure 29]\n;Unspecified return value

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

Start with holly and follow all the hash numbers except those associated with direction names. There should be between 10 and 20 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.

Student Disservice Cards

UML has asked us for help in expanding the features offered by the pass-card system. Luckily, our object-oriented simulation is just what’s needed for trying out new ideas.

To model a student pass-card, we can make a new kind of object, called an id-card, which is a special kind of thing. Besides inheriting the standard properties of a thing, each id-card has some local state: an id, which identifies the person to whom the card was issued. An id-card supports a message id-card?, indicating that it is an id-card. The card also accepts message that returns the id.

The procedure make&install-id-card (shown below) can be used to make a card and install it. It uses the procedure make-id-card to actually create the card. Note that both procedures take a name, an initial place and an id (which should be a symbol).
(define (make-install-id-card name birthplace id)
  (let ((card (make-id-card name birthplace id)))
    (ask card 'install)
    card))

(define (make-id-card name birthplace idnumber)
  (let ((id idnumber)
         (thing (make-thing name birthplace)))
    (lambda (message)
      (cond ((eq? message 'id-card?) (lambda (self) true))
            ((eq? message 'id) (lambda (self) id))
            (else (get-method thing message))))))

Note the presence of the id-card? method, which identifies the object as an id-card. In general, our system is structured so that a recognizable foo must have a foo? method that answers true. Objects that aren’t foons don’t have a foo? method. We’ve supplied a procedure called is-a that can be used to test whether an object is of some particular type. is-a works like ask except that if the message doesn’t correspond to a method it returns false rather than causing an error. For example, you can test whether an object is an id-card by evaluating (is-a object 'id-card?).

Exercise 5: A person may move to a new place only if the place returns true in response to the message accept-person?, for example

(ask robot-lab 'accept-person? holly) => #t

This is trivially true of ordinary places. Make a new kind of place that will accept a person only if they are carrying an id-card by completing the procedure below:

(define (make-card-locked-place name)
  (let ((place (make-place name)))
    (lambda (message)
      (cond ((eq? message 'accept-person?) ...
             ...
             (else (get-method place message)))))))

Change some of the places on campus to be card-locked-places, and make some id-cards. Demonstrate that a person may enter a card-locked-place only if they are carrying a card.

Turn in a listing of your make-card-locked-place definition and a transcript of your demonstration.

Exercise 6: There has been a spate of card thefts recently. Obviously these criminals need to be sorted out. Create a new kind of person called an ogre, which is like a troll but only eats people who are carrying a card which has been reported stolen. You can use grendel as an example of a being that does special things when asked to act. To make the ogres especially effective they should have a low restlessness factor.

Write a procedure (report-stolen-card id) which creates and dispatches a new ogre to hunt down the felon. Naturally, the ogre should start its hunt from the dungeon.

Exercise 7: Ace hackers Ben Bitdiddle and Alyssa P. Hacker find the new card locks on the protected labs a nuisance. It is difficult to get together to do problem sets and their friends who used to drop by to chat never do so anymore because they can’t get in without a valid card. Luckily, the cards are easy to duplicate and distribute.

To discourage the use of duplicate cards the Chairman of the Board has decided to monitor the use of cards. If a card is used in two places at the same time then one of the copies must be forged.

Implement a new object, big-brother, which accepts a message inform which takes a card-id and a place. Big-brother should monitor all the information to detect forged cards and report them by calling report-stolen-card.
The time is available from the procedure (current-time). Modify the code for the card-locked-places and protected-student-lab so that they inform big-brother when someone gains access with a card.

Turn in a listing of your new and modified code, with a transcript showing it in action.

**Exercise 8:** Now you have the elements of a simple game that you play by interspersing your own moves with calls to the clock. Your goal is to leave the computer-lab, gain access to the robot-lab, and return, without being eaten.

To make the game more interesting, you should also create some student(s) besides yourself and set up the student act method so that a student will try to move around campus, collecting id-cards and other interesting things that they find and occasionally leaving some of their possessions behind when they move on. Be sure to leave a few forged cards around just for fun.

Turn in your new student act method and a demonstration that it works.

**Exercise 9:** Design and implement some new student behaviors. Turn in a description (in English) of your ideas, the code you wrote to implement them, and a demonstration scenario.

*This problem is not meant to be a major project. Don’t feel that you have to do something elaborate.*