Structure and Interpretation of Computer Programs
Second Edition
Sample Problem Set

Game Problem Set

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 coming to the lab.

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

The overall object-oriented programming framework was discussed in lecture on March 21 and in the lecture notes for that day.

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 MIT, with imaginary places such as a computer lab, Building 36, and Tech Square. 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 birthplace restlessness)
```

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

```
(make&install-thing  name birthplace)
(make&install-person name birthplace restlessness)
```

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, `albert` and `gerry`, and put them in their places, as it were.

```
(define albert-office (make-place 'albert-office))
(define gerry-office  (make-place 'gerry-office))

(define albert (make&install-person 'albert albert-office 3))
(define gerry  (make&install-person 'gerry  gerry-office  2))
```

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

Once you load the system in the laboratory, you will be able to control `albert` and `gerry` 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:

```
(ask albert 'look-around)
At albert-office : albert says -- I see nothing
;Value: #f

(ask (ask albert 'place) 'exits)
;Value: (west down)

(ask albert 'go 'down)
albert moves from albert-office to tech-square
;Value: #t

(ask albert 'go 'south)
albert moves from tech-square to building-36
;Value: #t

(ask albert 'go 'up)
albert moves from building-36 to computer-lab
;Value: #t

(ask gerry 'look-around)
at gerry-office : gerry says -- I see nothing
;Value: #f

(ask (ask gerry 'place) 'exits)
;Value: (down)
(ask gerry 'go 'down)
gerry moves from gerry-office to albert-office
;Value: #t

(ask gerry 'go 'down)
gerry moves from albert-office to tech-square
;Value: #t

(ask gerry 'go 'south)
gerry moves from tech-square to building-36
;Value: #t
(ask gerry 'go 'up)
gerry moves from building-36 to computer-lab
At computer-lab : gerry says -- hi albert
;Value: #t
```

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 `albert` and `gerry` now are).

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

Next, we'll have `albert` look around. He sees the manual and `gerry`. The manual looks useful, so we have `albert` take it and leave.

```
(ask albert 'look-around)
At computer-lab : albert says -- I see computer-manual gerry
;Value: (computer-manual gerry)

(ask albert 'take computer-manual)
At computer-lab : albert says -- I take computer-manual
;Value: #t

(ask albert 'go 'down)
albert moves from computer-lab to building-36
;Value: #t
```

`Gerry` had also noticed the manual; he follows `albert` and snatches the manual away. Angrily, `albert` sulks off to the EGG-Atrium:

```
(ask gerry 'go 'down)
gerry moves from computer-lab to building-36
At building-36 : gerry says -- Hi albert
;Value: #t

(ask gerry 'take computer-manual)
At building-36 : albert says -- I lose computer-manual
At building-36 : albert says -- yaaaah! I am upset!
At building-36 : gerry says -- I take computer-manual
;Value: #t

(ask albert 'go 'west)
albert moves from building-36 to egg-atrium
;Value: #t
```

Unfortunately for `albert`, beneath the EGG-Atrium 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 `albert`:

```
(ask grendel 'move)
grendel moves from dungeon to egg-atrium
At egg-atrium : grendel says -- Hi albert
;Value: #t
```

After a few more moves, `grendel` acts again:

```
(ask grendel 'move)
At egg-atrium : grendel says -- Growl.... I'm going to eat you, albert
At egg-atrium : albert says --
                   Dulce et decorum est
                   pro computatore mori!
albert moves from egg-atrium to heaven
At egg-atrium : grendel says -- Chomp chomp. albert tastes yummy!
;Value: *burp*
```

Implementation The simulator for the world is contained in two files, which are attached to 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 `albert`, `gerry`, and `grendel`.

Part 2: Tutorial exercises

Tutorial exercise 1: (a) Define a procedure `flip` (with no parameters) that returns 1 the first time it is called, 0 the second time it is called, 1 the third time, 0 the fourth time, and so on. (b) Define a procedure `make-flip` that can be used to generate `flip` procedures. That is, we should be able to write `(define flip (make-flip))`. (c) Draw an environment diagram to illustrate the result of evaluating the following sequence of expressions:

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

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

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

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

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

```
flap1

flap2

flap3

flap4

(flap1)

(flap2)

(flap3)

(flap4)

flap1

(flap3)

(flap2)
```

Tutorial exercise 3: 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.

Tutorial exercise 4: 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.

Tutorial exercise 5: Suppose we evaluate the following expressions:

```
(define ice-cream (make-thing 'ice-cream dormitory))
(ask ice-cream 'set-owner gerry)
```

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 `gerry` or `dormitory`—just assume that `gerry` and `dormitory` are names defined in the global environment that point off to some objects that you draw as blobs.

Tutorial exercise 6: Suppose that, in addition to `ice-cream` in exercise 5, 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 `gerry` wanders to a place where they both are and looks around, what message will he print?

Exercises

The first exercise here should be written up and turned in at recitation, but we suggest that you do it *before* coming to the lab, because it illustrates a bug that is easy to fall into when working with the adventure game.

Pre-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 gerry` and `gerry` moves to a new place, he'll thereafter be in two places at once! The new place will claim that `gerry` is there, and `gerry`'s place of birth will also claim that `gerry` is there."

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

Exercises

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

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

Exercise 2: After loading the system, make `albert` and `gerry` 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 `dormitory`. Also make and install a thing called `late-homework`, so that it starts in the `dormitory`. Pick up the `late-homework`, find out where `gerry` is, go there, and try to get `gerry` 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 gerry)
#[compound-procedure 48]
Frame:
  #[environment 49]
Body:
  (lambda (message)
    (cond ((eq? message 'person?) (lambda ... true))
          ((eq? message 'possessions) (lambda ... possessions))
          ((eq? message 'list-possessions) (lambda ... ... possessions))
          ...))
```

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 49)
#[environment 49]
Parent frame: #[environment 50]
possessions:  ()
mobile-obj:   #[compound-procedure 52]
```

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

Start with `albert` 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

The MIT Office against Student Affairs has asked us for help in expanding the features offered by the MIT "Student Disservice Card" system. Luckily, our object-oriented simulation is just what's needed for trying out new ideas.

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

The procedure `make&install-sd-card` (shown below) can be used to make a card and install it. It uses the procedure `make-sd-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-sd-card name birthplace id)
  (let ((card (make-sd-card name birthplace id)))
    (ask card 'install)
    card))

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

Note the presence of the `sd-card?` method, which identifies the object as an sd-card. In general, our system is structured so that a recognizable `foo` must have a `foo?` method that answers `true`. Objects that aren't `foo`s don't have a `foo?` method. We've supplied a procedure called `is-a` that

Figure 1: Housing Director Lem E. Quagmire

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 sd-card by evaluating `(is-a object 'sd-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 gerry-office 'accept-person? gerry)  => #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 `sd-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 `sd-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: The Director of Housing has ordered that student residences can be entered only by students living at that residence. He has decided to enforce his policy by securing each student residence with a card lock that opens only for cards with a registered id.

Create a new class of place, a `student-residence`, which implements this policy. A student-residence should keep a list of card id-numbers (not the cards themselves in case a student loses their card and has to get a replacement). In addition to the `accept-person?` method we need a `register-card` method which adds the card's id to the list. `Register-card` should register the card only if the card is already inside the residence. This way we can create cards with the residence as their 'birth place' and register those cards. Only those cards will let us back in.

Turn in a listing of your procedure(s). To demonstrate that your implementation works, create two student residences and an sd-card for you character. Start at your residence, register your card, go to the other residence, try to get in, and return home.

Exercise 7: There has been a spate of card thefts on campus 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 8: Ace hackers Ben Bitdiddle and Alyssa P. Hacker find the new card locks on the student residences 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 friends.

To discourage the use of duplicate cards the Director of Housing 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 student-residences 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 9: 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 your residence, gain access to all the other residences, 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 sd-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 10: 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.

Post-exercise 11: When our characters go to heaven, they make a brief proclamation. Who is the source of these (misquoted) last words, what is the actual quotation, and what is the English translation?