MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1997

Problem Set 7

Issued: 4 November

Written solutions due: **FRIDAY**, 14 November

No Tutorial: Because of the Veterans Day holiday, there will be no Tutorial for the week of 10 November. There are several Tutorial problems in the problem set, but you need not prepare answers to them.

Reading for Lectures: *SICP* Sections 4.1–4.1.6
Reading for this Problem Set: the attached code files `object-system.scm`, `classes.scm`, `world.scm`, and `classes-expanded.scm`.

The Adventure Game: An Introduction to Object-Oriented Programming

This problem set explores three ideas. First, if you can represent a real or imagined world as a collection of distinct entities, then *object-oriented programming* will enable you to use the computer to simulate that world. Second, object-oriented programming can modularize a large system by bundling together procedures with the data that they operate on. Finally, objects can be implemented in Scheme very naturally as procedures with state. These ideas are presented in the context of a simple simulation game.

The system we present aims to capture some essential aspects of the "object" abstraction common to several object-oriented languages and to Java (without type declarations) in particular. But our system does not aim to be a precise simulation of Java in Scheme. There are some important differences, particularly in how method and variable lookup works, between our system and the corresponding mechanisms in Java. We believe that this problem set will help you understand some basics of object-oriented programming, and, when the time comes, will make it easier for you to learn languages like Java, but you will still have to learn the detailed mechanisms of whatever "real" object-oriented language you use.

In Section 1, we will show you how to use our system to create a world of characters and places, and how to simulate the passage of time.[1] We will be working mainly with the code in `world.scm`. In Section 2, we will ask you to add new classes of objects to the game, or to modify existing ones. We will be examining the code in `classes.scm`. Section 3 explores how we implemented objects in Scheme. We will look at the code that is in `classes-expanded.scm` and `object-system.scm`.

[1] The mechanism we use in our system to simulate the passage of time is not a standard aspect of object-oriented languages in general. In particular, our approach is contrasts with those of languages that support concurrency, or ones that let objects be distributed across a network of machines.

1. Creating and Simulating Worlds

In the MIT dungeon resides a former 6.001 student, Grendel the Troll. He has gotten loose, and is wandering the campus, eating anybody in his way. Will we be able to control him before he gets the 6.001 lecturers? Will the crucial Teleportation-to-Mars project successfully recover artifacts from the Red Planet in time? These are the technical questions we will explore in this problem set.

The basic idea of a simulation game 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 cause the character to move about and perform acts in the imaginary world, such as picking up things or interacting with other characters. 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 depicts the adventures of Sojourner the rover, and has places such as the Mars landing site, Technology Square, and transporter rooms.

Objects and Classes

Our system introduces two new kinds of first-class values: *objects* and *classes*. Because they are first class values, they can be given a name, passed as arguments into procedures, returned as values, and incorporated into data structures. The characters, places, and things in our world, like Sojourner, Rod, and Mars are objects. Each object is an instance of a class like the rover class, the person class, or the place class.

Classes are created using a procedure called `define-class`. The procedure `new` takes a class, and uses it to create (or *instantiate*) an object. Later, we will see that the procedure `new-part` can take a class, and use it to create a part of an object.[2] An object consists of data (or state), together with procedures (known as *methods*) that may operate on that data. For example, in our system, people have a list of possessions, and methods that add new possessions and delete possessions from the list. It is not possible to access or manipulate the list of possessions directly from outside of the object. We say that the object *encapsulates* the data. However, each method has a name, known as a *message*, and we can use a procedure called `ask` to invoke the method associated with a specific message. For example, we can use the `LOSE` message to have Rod remove a gun from his list of possessions: `(ask Rod 'LOSE gun)`.

For the rest of Section 1, we will see how to instantiate and manipulate objects. In Section 2, we will learn how to modify and write new classes.

Creating Objects: `new`

We start out with three classes: people, places, and things. We use `new` to create objects:

[2] Our terminology may be slightly confusing to those who have seen specific object oriented languages already. When we say that an object has a part, we mean that the class that the object belongs to is a subclass of the class that the part belongs to.

```
(new person-class name birthplace restlessness)
(new place-class  name)
(new thing-class  name birthplace)
```

As you can see, when they are created, people and things are given a name and an initial place. In addition, a person has a restlessness factor[3] that determines how often the person moves. For example, we can create two imaginary characters, Albert and Rod, and put them in their offices:

```
(define albert-office (new place-class 'albert-office))
(define rod-office    (new place-class 'rod-office))

(define albert (new person-class 'albert albert-office 2))
(define rod    (new person-class 'rod rod-office 3))

(define diamond (new thing-class 'diamond rod-office))
```

According to this code, Rod will try to go to a new location about one third of the time, while Albert will try to walk somewhere about half of the time.[4]

Manipulating the Objects you Have Created: `ask`

The classes we have described so far implement notions of ownership and location. An instance of the thing class can be owned by an instance of the person class. Because the thing class and the person class are *subclasses* of a class called the mobile class, things and people have locations.

Once an object has been created, we can manipulate it by using `ask` to send it *messages*. For example, people objects can respond to `TAKE` and `GO` messages, and place objects can respond to `ADD-NEIGHBOR` messages:

```
(ask rod 'TAKE diamond)
(ask albert-office 'ADD-NEIGHBOR 'UP rod-office)
(ask rod-office 'ADD-NEIGHBOR 'DOWN albert-office)
(ask albert 'GO 'UP)
(ask albert 'TAKE diamond)
(ask albert 'GO 'DOWN)
```

If any object is sent a `NAME` message, it will return its name. If a person or thing is sent a `PLACE` message, it will return the object that represents its current location. If a place is sent a `THINGS` message, it will return a list of all of the people and things that are at that place.

Lab Exercise 1: Use `load-problem-set` to start the adventure game.[5] Then, as described above, create Albert, Rod, the offices and the diamond, and have Albert take the diamond.

We have written a utility procedure called `print-state` that prints out the names of the objects in the world, their locations, possesions, etc. We expect that you may find this procedure useful to test your solutions. Use `print-state` to print out the state of the system.

[3]This might better be dubbed a *lethargy factor*, since the larger the number, the less frequent the moves.
[4]We say "try" because some places have no exits.
[5]`load-problem-set` will evaluate the code in `object-system.scm` and `classes.scm`.

The Passage of Time

Rather than having to tell each object what to do explicitly, we want them to interact with each other on their own.

Send the ACT message to each person (Albert and Rod) 10 times. Send the CLOCK-TICK message to each person 10 times. Now, try the following:

```
(define a-clock (new clock-class 'a-clock))
(ask a-clock 'ADD-TO-LIST albert)
(ask a-clock 'ADD-TO-LIST rod)
(ask a-clock 'RUN 10)
```

Tutorial Question 1: (Remember there is **no tutorial**, and these tutorial questions are included as food for thought; **you need not prepare answers to them**.)

What do you think the ACT message does? What do you think the CLOCK-TICK message does? What does the RUN message do?

The Sample World

In the file world.scm, there are definitions for a more involved world. Look at the file, and use M-o to evaluate the contents of the world.scm file. Use run-clock to run the world for 50 clock ticks. What happened to the characters?

Lab Exercise 2: Add a new character, Sojourner (an instance of the rover-class) to the world. To create a rover, you need to specify a name, a birthplace, a restlessness threshold, and a weight limit (that specifies how many pounds it can carry). You should be able to do this with one line of code.

Now check that Sojourner moves around when you run the world.

2. Modifying and Creating Classes

In this section, we will take a closer look at how classes, like people and trolls, are designed. The classes that we have looked at so far can be found in the file classes.scm. Each class describes a way to create objects. Each object consists of data, together with the procedures that operate on that data. The class must specify what data the object keeps track of, it must define the set of procedures that manipulate the state, and it must have a way to initialize the state of the object.

In addition, it is possible to use *inheritance* to create a class that is similar to another class. For example, a troll is very similar to a person, except that it has an extra method "EAT-PERSON", and its "ACT" method does some extra stuff before doing what a person usually does when it receives an ACT message. By specifying that the troll class inherits from the person class, we can reuse the person class code when we implement the troll class. One way to think about inheritance is to say that a troll object consists of parts – namely, it consists of a person part, together with the extra stuff that makes it a troll. The person part, in turn, happens to have a mobile part. Part of the definition of a class is the list of parts that it consists of. The immediate significance of this is that a troll can give its `NAME`, `SAY` things, `LOOK-AROUND`, etc., even though these methods are not explicitly part of its definition, because it inherits all these methods from it named-part, person-part, etc.

To summarize, each class has the same structure. It has a class-name, a list of parts, a list of slots and their default values, a constructor, and a list of methods and their associated messages. The constructor creates parts, initializes slots, and establishes consistency in the world when an object is instantiated.

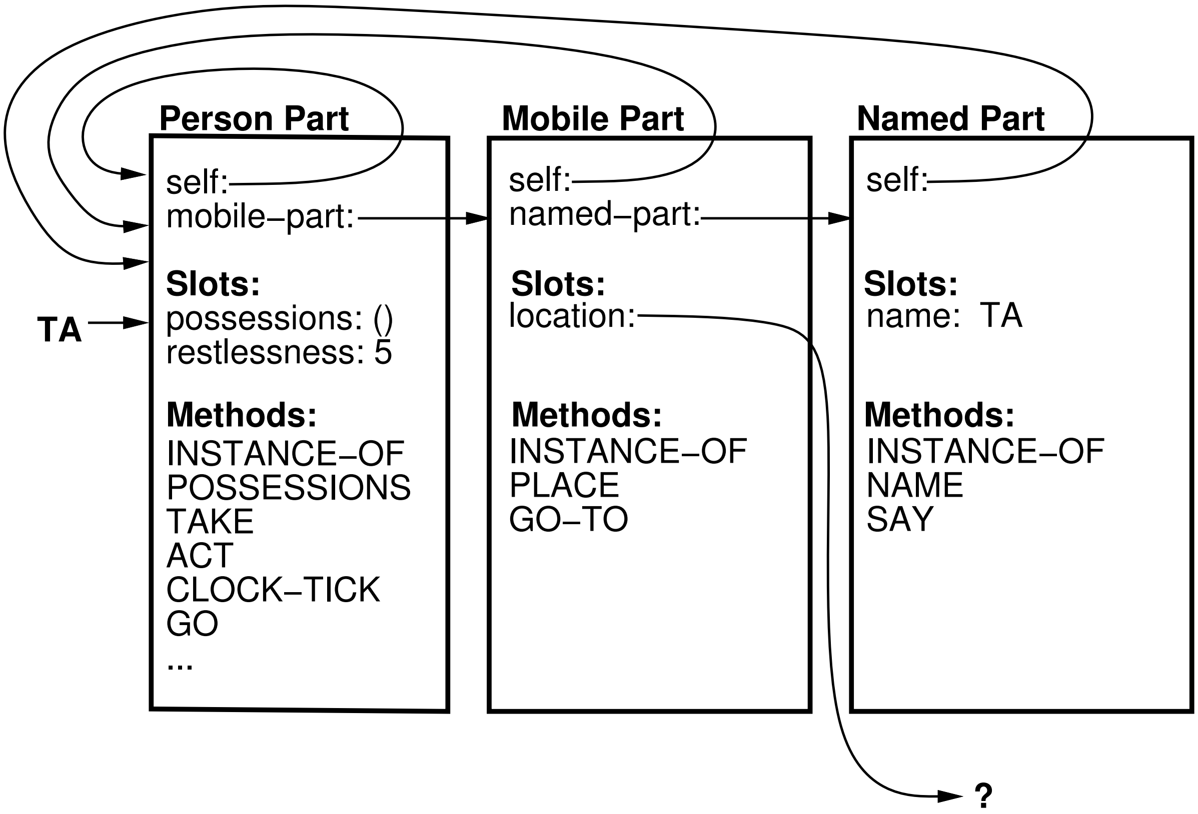

Figure 1: The object representing a TA.

For example, suppose we create a *TA*:

```
(define TA-office (new place-class 'TA-office))
(define TA (new person-class 'TA TA-office 5))
```

The structure of the resulting object is illustrated in Figure 1. If you look at the definition of the person-class in the file `classes.scm`, you will see that it specifies that a person has a slot for his or her possessions, and one for his or her restlessness, bundled together with methods that handle the messages `INSTANCE-OF?`, `POSSESSIONS`, `TAKE`, `ACT`, etc. The methods can refer to `self`, to the parts, and to the slots directly, but, because objects enforce encapsulation, no procedures outside of the object can do so[6].

Notice that the values of the `self`'s of the named part, the mobile part, and the person part of TA all equal the person part. This has important implications for how object methods behave. For example, if we ask a TA to `GO-TO albert-office`, the TA `GO-TO` method in turn asks the TA's mobile-part to `GO-TO albert-office`. Now the mobile part's `GO-TO` method asks `TA-office` (assuming that is still where the TA is) to `DEL-THING self`. The value of this `self` is the TA's *person* part, *not* its mobile part, even though it is the `GO-TO` method of the mobile part which is asking `TA-office` to do the deletion. And of course, that's just what is desired: the deletion would fail if `TA-office` was asked to delete TA's mobile part, because mobile part isn't there: only TA's *person* part is at `TA-office`[7].

Pre-Lab Exercise 3: Look at the `classes.scm` file, and find where the constructor, parts, slots, messages, and methods for the TA (which is an instance of the person class) described in Figure 1 are implemented. Now, expand Figure 1 by showing a representation of the object that the location slot of mobile part is bound to. Your answer to this question should be a graphical diagram similar to that in Figure 1 of the object that the location slot of the mobile object points to.

Suppose that Rod owns a diamond. This fact is stored in two places:

1. The diamond object is in Rod's list of possessions, which is stored in the possessions slot of the rod object.
2. Rod is the owner of the diamond, which is stored in the owner slot of the diamond object.

Any time a system stores a fact in more than one place, there is a potential consistency problem — the system must make sure that the multiple places store the same fact. In the case of people owning things, the following properties must be true for the system to be consistent:

- Each thing is owned either by a person or by nobody.
- Each thing occurs at most once in the possessions list of at most one person.
- If a thing occurs in the possessions list of a given person, the owner of that thing must be that person.
- If the owner of a thing is a given person, the thing must be in that person's possessions list.

[6]A feature of our implementation is that an object does not even have direct access to the slots of its parts: it has to `ask` its parts for their contents. In other object-oriented languages, an object typically does have direct access to the instance variables of it parts

[7]Informal language can be misleading here. You might want to say that since TA's person part is at `TA-office`, and since TA's mobile part is "part" of TA, then the mobile part must be there too. This sounds plausible, but is wrong: being *at* `TA-office` has a technical meaning, and we would almost never expect to have the private, internal parts of an object "at" *any* location in the system.

Periodically things change owners. In our system, these ownership changes take place in response to TAKE and LOSE messages sent to people. The system maintains the consistency of the ownership information by sending messages to all of the objects that must update their internal state to reflect the ownership change. Here is the sequence of messages that the system sends when a person takes something:

- A person receives a TAKE message. That person's TAKE method cons's the thing on to the person's list of possessions. Note that at this moment the global state of the system is inconsistent: the thing may be in the possessions list of more than one person, and the owner of the thing is not the person that is taking it.
- If the thing was owned by another person, the TAKE method then sends a LOSE message to the original owner. The original owner's LOSE method removes the thing from the original owner's list of possessions.
- The original owner's LOSE message sends a SET-OWNER message to the thing, and the thing's SET-OWNER method sets its owner to be nobody.
- Finally, the TAKE method sends a SET-OWNER message to the thing, and the thing's SET-OWNER message sets the owner of the thing to be the new owner. At this point the system is consistent again.

Note that some methods are not intended to be invoked directly from outside the system. The SET-OWNER method, for example, should only be invoked during the sequence of messages sent as a result of a person taking or losing something.

Pre-Lab Exercise 4: Suppose that Albert is in his office. This fact is stored in multiple slots; give the slots in which it is stored. Give the consistency properties regarding these slots that must be true for the system to be consistent. Give the sequence of messages that result when a person is sent a GO-TO message. Indicate when the system becomes inconsistent and when it becomes consistent again.

Inheritance

In addition to the slots and methods explicitly described in the definition of a class, a class can use *parts* or *superclasses* to *inherit* additional state and behavior. When a message is sent to the TA, the system will first see if the person-part can handle the message. If it can not, then the mobile-part is checked. Finally, if the mobile part can not handle the message, then the named-part is checked. Note that in our system, the parts are encapsulated. For example, the PLACE method can directly access the location slot, and it can send messages to the named-part and to self, but it can not directly manipulate the list of possessions; it would have to send a TAKE message to self if it wanted to add a thing to the list of possessions, just as any procedure external to the object would.

PreLab Exercise 5: Draw the inheritance hierarchy for the classes in `classes.scm`.

Lab Exercise 6: Ben Bitdiddle left his computer unattended, and his brother, Little Bitdiddle, made a small change to his `classes.scm` file.

A. Little B. changed the line following the one marked "*See Problem 6A*" to

```
(ask self 'ACT)
```

Try to figure out why the system will now behave incorrectly and what the observable incorrect behavior will be. Now modify the line and test for the incorrect behavior. Briefly explain what's going on.

(Note: it is a good idea to repeat `M-X load-problem-set` and reload `world.scm` and any additional code you used to change modify the system state after making any such change to a class definition. **Tutorial Question 2:** Why?)

B. (Optional Problem) Suppose instead of 6A, Little B. changed the line following the one marked "*See Problem 6B*" to

```
(new person-class name birthplace threshold)
```

Explain what would happen.

Lab Exercise 7: We have given you a class of rocks, whose objects behave like things, except that they have weight. Put some rocks on Mars, including Barnacle-Bill, Scooby-Doo, and Yogi. Your answer should be the three lines of code you wrote to create the rocks.

Lab Exercise 8: Finish the implementation of the rover class.

First, modify the class so that rovers refuse to carry more than their weight limit allows. Rovers should also refuse to carry anything that is not a rock. You will probably have to override the inherited `TAKE` method and add a customized one for rovers; you may also have to override the `LOSE` method. Then, override the `ACT` method so that if there are any rocks in the area that the rover does not already possess, then the rover will pick one at random and try to take it. (Be careful when you modify the `ACT` method, and make sure that your rovers still wander.)

You will have to make some design decisions as part of this question. For example, you will have to decide if rovers can take rocks from other rovers or other people, or if they can only pick up rocks that nobody currently possesses. You should document such decisions you make.

Tutorial Question 3: Our implementation makes rovers a subclass of people. Does this class hierarchy seem intuitive to you? Do you think the code was developed with this intention? Is this organization of the system an abuse of the inheritance mechanism?

Lab Exercise 9: Define a `rover-maker-class` and Pat, a rover-maker object. Pat sits in the rover-lab, and creates a rover when sent the `ACT` message. Pat never wanders. Be sure that the new rovers are an active part of the world and wander around. Also, make sure that each rover has a different name. We have given you a routine called `attach-suffix` (in `object-system.scm`) for attaching numerical suffixes to symbols that should help you create the new symbols you need to give each rover a new name.

Again, you will have to make some design decisions which you should briefly indicate as comments in your code. This problem will be easier if you create your new class by having it inherit from one or more previous classes, copying in methods from other classes, and then hacking them.

Lab Exercise 10: Finish the implementation of the transporter rooms: when a rover enters the room, it should be teleported to the destination. Your answer to this question should be one additional method added to the transporter class. The important requirement here is that the system state be consistent and reflect the transportation before the next clock tick. In particular, if no class other than the transporter class is changed, we expect your solution will print out the movement information in a counterintuitive order. Also, if the rover has possessions, the possessions will probably follow the rover to the new location two times, not just once. It is OK if your solution has these properties as long as the final system state is the desired one.

Tutorial Exercise 4: Can you make the movement information print out in the correct order without modifying the person class? We did not anticipate this difficulty when we designed the object system. Can you see how we might have? Is there anything wrong with the original design of the object system? How would you change the design of the system to make it more amenable to these kinds of extensions?

3. Implementation of Classes in Scheme

If you look at the definition of `define-class` in `object-system.scm`, you will see that it takes a class definition, transforms it into a scheme expression, and evaluates that expression.

Objects are implemented as message-passing procedures. The message-passing objects have only the slots, methods, and parts they should: classes ingeniously construct objects in a way that avoids entangling unnecessary pieces of environment inside the object procedures.

You are not expected to master the code by which classes construct objects, but you should understand how objects are finally represented as procedures. Classes are implemented as thunks returning message-passing procedures that can respond to `NEW` and `NEW-PART` messages.

We took the classes in `classes.scm`, transformed them into Scheme expressions, and put the result in `classes-expanded.scm`. Evaluating a `define-class` expression in `classes.scm` should be equivalent to evaluating the associated scheme expression in `classes-expanded.scm`.

Lab Exercise 11: Apply `transform-class-definition` to one of your new class definitions, and pretty-print the result using the `pp` procedure. You can look in the file `object-system.scm` for a version of `define-class` that is commented out. You can use this version to pretty-print your transformed class definition. Turn in the print-out.

Tutorial Exercise 5: The code you generated in Exercise 11 almost surely can be simplified using specific features of the code which the general `define-class` procedure did not try to detect. How much simpler can you make the code?

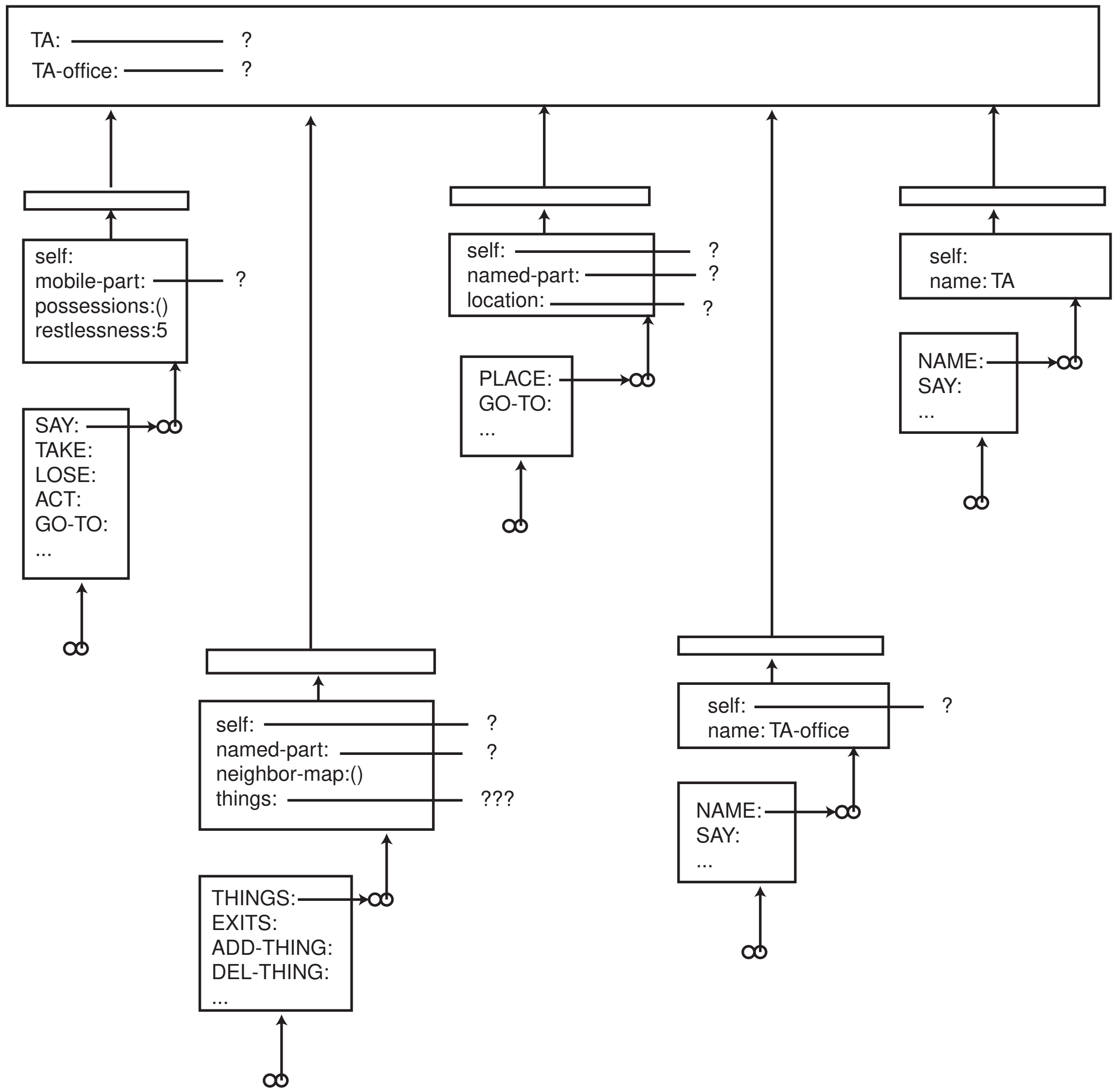

Figure 2: The environment diagram for Exercise 12.

PostLab Exercise 12: Figure 2 shows an incomplete environment diagram that we get if we evaluate the expressions:

```
(define TA-office (new place-class 'TA-office))
(define TA (new person-class 'TA TA-office 5))
```

The diagram has several lines that terminate in a ?. Your job is to extend each line to point to the correct entity in the diagram.

Note that one of the lines terminates in a ???, for this line you should draw a box-and-pointer diagram for what it points to. Be sure to draw the lines going from the cars of the cons cells in the box-and-pointer diagram.

Optional Contest

If you would like to play with the system some more, feel free to choose a theme or a story line, and expand on the world with your own ideas. Include a short narrative description of your work, and we will award prizes for the cleverest ideas. Note that it is more impressive to implement a simple, elegant idea, than to amass tons of new characters and places. We will consider both interesting story lines, and technically interesting ideas.

For example, you might create a thief class whose instances cannot go to Mars themselves, but who take rocks from any rovers that wander by. Or you might create a `rocket-class` which inherits from *both* the `mobile-class` and the `place-class` — because rockets can move, but a person could be in the rocket (note that our system handles such *multiple inheritance* by allowing an object to have a sequence of objects in its `parts` field). And you might want to allow for removal of objects from the system by adding `DIE` methods to the classes.

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

This problem set is substantially different from versions that have been given out in previous years, and we would appreciate your feedback. Was it interesting, too hard/easy, long/short, well/poorly written, ...?

How much time did you spend on this homework assignment?

Indicate all of the following facilities you in doing this problem set:

- The 6.001 lab in building 36.
- MIT Scheme on Project Athena.
- MIT Scheme on a Windows PC.
- EdScheme on a PC.
- Some other Scheme system (which one?).

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 course text (such as some of the course archives), please indicate your consultants' names and your references in the space below. 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 course reference materials." and sign your statement.