UNIVERSITY of CALIFORNIA at Berkeley
Department of Electrical Engineering and Computer Sciences
Computer Sciences Division

CS61A	**Kurt Meinz**
Structure and Interpretation of Computer Programs	Summer 2005

Programming Project 3: Adventure Game

This project is due at 11:59am on Tuesday, August 2. Please check the course website for updates and errata.

This project is designed to be done by two people, working in parallel, then combining your results into one finished product. (Hereafter the two partners are called Person A and Person B.) The project begins with two exercises that both partners should do; these exercises do not require new programming, but rather familiarize you with the overall structure of the program as we've provided it. After that, each person has separate exercises. There is one final exercise for everyone that requires the two sets of work to be combined. (Therefore, you should probably keep notes about all of the procedures that you've modified during the project, so you can notice the ones that both of you modified independently.)

Scoring: Each person works on eight problems. Three of these (numbers 1, 2, and 8) are common to the partnership; the others are separate. You hand in a single solution to each problem. Both of you get the points awarded for problems 1, 2, and 8; each person get the points for their own problems 3 through 7. This means that your score for the project is mostly based on your individual work but also relies partly on your partner. For the first two problems, you could get away with letting your partner do the work, but you shouldn't because those problems are necessary to help you understand the structure of the entire project. Problem 8 requires that both people have already done their separate work, and meet together to understand each other's solutions, so probably nobody will get credit for it unless both people have done their jobs.

This assignment is loosely based on an MIT homework assignment in their version of this course. (But since this is Berkeley, it was changed it to be politically correct; instead of killing each other, the characters go around eating gourmet food all the time.)

"In this laboratory assignment, we will be exploring two key 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."

Object-Oriented Programming (OOP) is becoming an extremely popular methodology for any application that involves interactions among computational entities. Examples:

- operating systems (processes as objects)
- window systems (windows as objects)
- distributed systems
- e-commerce systems (user processes as objects)

Getting Started: To start, copy the following five files into your directory:

~cs61a/lib/obj.scm The object-oriented system

~cs61a/lib/adv.scm The adventure game program

~cs61a/lib/tables.scm An ADT you'll need for part B4

~cs61a/lib/adv-world.scm The specific people, places, and things

~cs61a/lib/small-world.scm A smaller world you can use for debugging

To work on this project, you must load these files into Scheme in the correct order: `obj.scm` first, then `adv.scm` and `tables.scm` when you're using that, and finally the particular world you're using, either `adv-world.scm` or `small-world.scm`. The work you are asked to do refers to `adv-world.scm`; `small-world.scm` is provided in case you'd prefer to debug some of your procedures in a smaller world that may be less complicated to remember and also faster to load.

The reason the adventure game is divided into `adv.scm` (containing the definitions of the object classes) and `adv-world.scm` (containing the specific instances of those objects in Berkeley) is that when you change something in `adv.scm` you may need to reload the entire world in order for your changed version to take effect. Having two files means that you don't also have to reload the first batch of procedures.

In this program there are three classes:

- THING,
- PLACE, and
- PERSON

Here are some examples selected from adv-world.scm:

```
;;; construct the places in the world
(define Soda (instantiate place 'Soda))
(define BH-Office (instantiate place 'BH-Office))
(define 61A-Lab (instantiate place '61A-Lab))
(define art-gallery (instantiate place 'art-gallery))
(define Evans (instantiate place 'Evans))
(define Sproul-Plaza (instantiate place 'Sproul-Plaza))
(define Telegraph-Ave (instantiate place 'Telegraph-Ave))
(define Noahs (instantiate place 'Noahs))
(define Intermezzo (instantiate place 'Intermezzo))
(define s-h (instantiate place 'sproul-hall))

;;; make some things and put them at places
(define bagel (instantiate thing 'bagel))
(ask Noahs 'appear bagel)

(define coffee (instantiate thing 'coffee))
(ask Intermezzo 'appear coffee)

;;; make some people
(define Brian (instantiate person 'Brian BH-Office))
(define hacker (instantiate person 'hacker 61A-Lab))

;;; connect places in the world

(can-go Soda 'up art-gallery)
(can-go art-gallery 'west BH-Office)
(can-go Soda 'south Evans)
```

Having constructed this world, we can now interact with it by sending messages to objects. Here is a short example.

; We start with the hacker in the 61A lab.

```
> (ask 61A_Lab 'exits)
(UP)
> (ask hacker 'go 'up)
HACKER moved from 61A_LAB to SODA
```

We can put objects in the different places, and the people can then take the objects:

```
> (define Jolt (instantiate thing 'Jolt))
JOLT
> (ask Soda 'appear Jolt)
APPEARED
> (ask hacker 'take Jolt)
HACKER took JOLT
TAKEN
```

You can take objects away from other people, but the management is not responsible for the consequences... (Too bad this is a fantasy game, and there aren't really vending machines in Soda that stock Jolt.)

PART I:

The first two exercises in this part should be done by both of you. It's okay to work in separately as long as you both really know what's going on by the time you're finished. (Nevertheless, you should only hand in one solution, that everyone agrees about.) The remaining exercises have numbers like "A3" which means exercise 3 for person A.

After you've done the individual work, you should meet together to make sure that everyone understands what the other person did, because Part II depends on all of Part I. You can do the explaining while you're merging the two sets of modifications into one adv.scm file to hand in.

1. Create a new person to represent yourself. Put yourself in a new place called Dormitory (or wherever you live) and connect it to campus so that you can get there from here. Create a place called Shin-Shin, north of Soda. (It's actually on Solano Avenue.) Put a thing called Potstickers there. Then give the necessary commands to move your character to Shin-Shin, take the Potstickers, then move yourself to where Brian is, put down the Potstickers, and have Brian take them. Then go back to the lab and get back to work. (There is no truth to the rumor that you'll get an A in the course for doing this in real life!) All this is just to ensure that you know how to speak the language of the adventure program.

   List all the messages that are sent during this episode. It's a good idea to see if you can work this out in your head, at least for some of the actions that take place, but you can also trace the `ask` procedure to get a complete list. You don't have to hand in this listing of messages. (Do hand in a transcript of the episode without the tracing.) The point is that you should have a good sense of the ways in which the different objects send messages back and forth as they do their work.

   [Tip: we have provided a `move-loop` procedure that you may find useful as an aid in debugging your work. You can use it to move a person repeatedly.]

2. It is very important that you think about and understand the kinds of objects involved in the adventure game. Please answer the following questions:

2A. What kind of thing is the value of variable `Brian`?

Hint: What is returned by scheme in the following situation: You type:

```
> Brian
```

2B. List all the messages that a `place` understands. (You might want to maintain such a list for your own use, for every type of object, to help in the debugging effort.)

2C. We have been defining a variable to hold each object in our world. For example, we defined bagel by saying:

```
(define bagel (instantiate thing 'bagel))
```

This is just for convenience. Every object does not have to have a top-level definition. Every object **does** have to be constructed and connected to the world. For instance, suppose we did this:

```
> (can-go Telegraph-Ave 'east (instantiate place 'Peoples-Park))
  ;;; assume BRIAN is at Telegraph
> (ask Brian 'go 'east)
```

What is returned by the following expressions and WHY?

```
> (ask Brian 'place)
> (let ((where (ask Brian 'place)))
   (ask where 'name))
>  (ask Peoples-park 'appear bagel)
```

2D. The implication of all this is that there can be multiple names for objects. One name is the value of the object's internal `name` variable. In addition, we can define a variable at the top-level to refer to an object. Moreover, one object can have a private name for another object. For example, `Brian` has a variable `place` which is currently bound to the object that represents People's Park. Some examples to think about:

```
> (eq? (ask Telegraph-Ave 'look-in 'east) (ask Brian 'place))

> (eq? (ask Brian 'place) 'Peoples-Park)

> (eq? (ask (ask Brian 'place) 'name) 'Peoples-Park)
```

OK. Suppose we type the following into scheme:

```
>  (define computer (instantiate thing 'Durer))
```

Which of the following is correct? Why?

`(ask 61a-lab 'appear computer)` or

`(ask 61a-lab 'appear Durer)` or

`(ask 61a-lab 'appear 'Durer)`

What is returned by `(computer 'name)` ? Why?

2E. We have provided a definition of the `thing` class that does not use the object-oriented programming syntax described in the handout. Translate it into the new notation.

2F. Sometimes it's inconvenient to debug an object interactively because its methods return objects and we want to see the names of the objects. You can create auxiliary procedures for interactive use (as opposed to use inside object methods) that provide the desired information in printable form. For example:

```
(define (name obj) (ask obj 'name))
 (define (inventory obj)
  (if (person? obj)
    (map name (ask obj 'possessions))
         (map name (ask obj 'things))))
```

Write a procedure `whereis` that takes a person as its argument and returns the name of the place where that person is.

Write a procedure `owner` that takes a thing as its argument and returns the name of the person who owns it. (Make sure it works for things that aren't owned by anyone.)

Procedures like this can be very helpful in debugging the later parts of the project, so feel free to write more of them for your own use.

Now it's time for you to make your first modifications to the adventure game. This is where you split into subgroups.

PART I – Person A:

A3. You will notice that whenever a person goes to a new place, the place gets an `'enter` message. In addition, the place the person previously inhabited gets an `'exit` message. When the place gets the message, it calls each procedure on its list of `entry-procedures` or `exit-procedures` as appropriate. Places have the following methods defined for manipulating these lists of procedures: `add-entry-proc`, `add-exit-proc`, `remove-entry-proc`, `remove-exit-proc`, `clear-all-proc`. You can read their definitions in the code.

Sproul Hall has a particularly obnoxious exit procedure attached to it. Fix `sproul-hall-exit` so that it counts how many times it gets called, and stops being obnoxious after the third time.

Remember that the `exit-procedures` list contains procedures, not names of procedures! It's not good enough to redefine `sproul-hall-exit` , since Sproul Hall's list of exit procedures still contains the old procedure. The best thing to do is just to load `adv-world.scm` again, which will define a new sproul hall and add the new exit procedure.

A4. We've provided people with the ability to say something using the messages `'talk` and `'set-talk` . As you may have noticed, some people around this campus start talking whenever anyone walks by. We want to simulate this behavior. In any such interaction there are two people involved: the one who was already at the place (hereafter called the *talker*) and the one who is just entering the place (the *listener*). We have already provided a mechanism so that the listener sends an `enter` message to the place when entering. Also, each person is ready to accept a `notice` message, meaning that the person should notice that someone new has come. The talker should get a `notice` message, and will then talk, because we've made a person's `notice` method send itself a `talk` message. (Later we'll see that some special kinds of people have different `notice` methods.)

Your job is to modify the `enter` method for places, so that in addition to what that method already does, it sends a `notice` message to each person in that place other than the person who is entering. In order to make this work, the place has to know who sent the `enter` message. Modify that method so that it takes the sender as an argument, just as the `appear` method does. (Make sure that you find everywhere in the program where an `enter` message is sent!)

Test your implementation with the following:

```
(define singer (instantiate person 'rick sproul-plaza))

(ask singer 'set-talk ``My funny valentine, sweet comic valentine")

(define preacher (instantiate person 'preacher sproul-plaza))

(ask preacher 'set-talk ``Praise the Lord")

(define street-person (instantiate person 'harry telegraph-ave))

(ask street-person 'set-talk ``Brother, can you spare a buck")
```

YOU MUST INCLUDE A TRANSCRIPT IN WHICH YOUR CHARACTER WALKS AROUND AND TRIGGERS THESE MESSAGES.

End of Part I for Person A

PART I, Person B:

B3. Define a method `take-all` for people. If given that message, a person should `take` all the things at the current location that are not already owned by someone.

B4A. It's unrealistic that anyone can take anything from anyone. We want to give our characters a `strength`, and then one person can take something from another only if the first has greater `strength` than the second.

However, we aren't going to clutter up the person class by adding a local `strength` variable. That's because we can anticipate wanting to add lots more attributes as we develop the program further. People can have `charisma` or `wisdom` ; things can be `food` or not; places can be `locked` or not. Therefore, you will create a class called `basic-object` that keeps a local variable called `properties` containing an attribute-value table like the one that we used with `get` and `put` in 2.3.3. However, `get` and `put` refer to a single, fixed table for all operations; in this situation we need a separate table for every object. The file tables.scm contains an implementation of the table Abstract Data Type:

constructor: `(make-table)` returns a new, empty table.

mutator: `(insert! key value table)` adds a new key-value pair to a table.

selector: `(lookup key table)` returns the corresponding value, or `#f` if the key is not in the table.

You'll learn how tables are implemented in 3.3.3 (pp. 214-215). For now, just take them as primitive.

You'll modify the person, place and thing classes so that they will inherit from basic-object. This object will accept a message `put` so that

```
> (ask Brian 'put 'strength 100)
```

does the right thing. Also, the basic-object should treat any message not otherwise recognized as a request for the attribute of that name, so

```
> (ask Brian 'strength)
  100
```

should work **without** having to write an explicit `strength` method in the class definition.

Don't forget that the property list mechanism in 3.3.3 returns `#f` if you ask for a property that isn't in the list. This means that

```
> (ask Brian 'charisma)
```

should never give an error message, even if we haven't `put` that property in that object. This is important for true-or-false properties, which will automatically be false (but not an error) unless we explicitly `put` a true value for them.

Give people some reasonable (same for everyone) initial strength. Next week they'll be able to get stronger by eating.

B4B. You'll notice that the type predicate `person?` checks to see if the type of the argument is a member of the list '(person place thief). This means that the `person?` procedure has to keep a list of all the classes that inherit from `person` , which is a pain if we make a new subclass.

We'll take advantage of the property list to implement a better system for type checking. If we add a method named `person?` to the person class, and have it always return `#t` , then any object that's a type of person will automatically inherit this method. Objects that don't inherit from person won't find a `person?` method and won't find an entry for `person?` in their property table, so they'll return `#f` .

Similarly, places should have a `place?` method, and things a `thing?` method.

Add these type methods and change the implementation of the type predicate procedures to this new implementation.

End of Part I, Person B

PART II:

This part of the project includes three exercises for each person, followed by a final exercise that requires the individual work to be combined. You will have to create a version of adv.scm that includes the changes made by both of you. This may take some thinking! If you both modify the same method in the same object class, you'll have to write a version of the method that incorporates both modifications.

PART II, Person A:

A5. The way we're having people take food from restaurants is unrealistic in several ways. Our overall goal this week is to fix that. As a first step, you are going to create a `food` class. We will give things that are food two properties, an `edible?` property and a `calories` property. `edible?` will have the value `#t` if the object is a food. If a `person` eats some food, the food's `calories` are added to the person's `strength` .

(Remember that the `edible?` property will automatically be false for objects other than food, because of the way properties were implemented in question B4. You don't have to go around telling all the other stuff not to be edible explicitly.)

Write a definition of the `food` class that uses `thing` as the parent class. It should return `#t` when you send it an `edible?` message, and it should correctly respond to a `calories` message.

Replace the procedure named `edible?` in the original adv.scm with a new version that takes advantage of the mechanism you've created, instead of relying on a built-in list of types of food.

Now that you have the `food` class, invent some child classes for particular kinds of food. For example, make a bagel class that inherits from `food` . Give the bagel class a class-variable called `name` whose value is the word bagel. (We'll need this later when we invent `restaurant` objects.)

Make an `eat` method for people. Your `eat` method should look at your possessions and filter for all the ones that are edible. It should then add the calorie value of the foods to your strength. Then it should make the foods disappear (no longer be your possessions and no longer be at your location).

A6A. Eventually, we are going to invent restaurant objects. People will interact with the restaurants by buying food there. First we have to make it possible for people to buy stuff. Give `person` objects a `money` property, which is a number, saying how many dollars they have. Note that money is not an object. We implement it as a number because, unlike the case of objects such as chairs and potstickers, a person needs to be able to spend `some` money without giving up all of it. In principle we could have objects like `quarter` and `dollar-bill` , but this would make the change-making process complicated for no good reason.

Now, in order to get money, people will need to go to the bank. Create a new place, a `bank`. (That is, `bank` is a subclass of `place`. The bank contains a procedure `make-account` which returns a bank-account object, and contains a method `withdraw` for withdrawing money.

The method `withdraw` takes two arguments, the person who wants to withdraw money from their account, and the amount. The `withdraw` method should check that a person has that amount in their account, and if so, it returns a number corresponding the the amount of the withdrawal or reports #f.

Now, change `person` so that on instantiation, they have a bank account containing $100, (but their money variable is 0.) (We should really start people with no money, and invent jobs and so on, but we won't.)

Create a method for people, `pay-money` , which takes a number as argument and returns true or false depending on whether the person had enough money, and updates the person's money value.

Now, create a method `get-money` which only works at a bank. It should check that the bank completed the withdrawal, and if so, it should update the person's money value appropriately.

A6B. Another problem with the adventure game is that Noah's only has one bagel. Once someone has taken that bagel, they're out of business.

To fix this, we're going to invent a new kind of place, called a `restaurant` . (That is, `restaurant` is a subclass of `place` .) Each restaurant serves only one kind of food. (This is a simplification, of course,

and it's easy to see how we might extend the project to allow lists of kinds of food.) When a restaurant is instantiated, it should have two extra arguments, besides the ones that all places have: the class of food objects that this restaurant sells, and the price of one item of this type:

```
> (define-class (bagel) (parent (food ...)) ...)
> (define Noahs (instantiate restaurant 'Noahs bagel 0.50))
```

Notice that the argument to the restaurant is a **class** , not a particular bagel (instance).

Restaurants should have two methods. The `menu` method returns a list containing the name and price of the food that the restaurant sells. The `sell` method takes two arguments, the person who wants to buy something and the name of the food that the person wants. The `sell` method must first check that the restaurant actually sells the right kind of food. If so, it should `ask` the buyer to `pay-money` in the appropriate amount. If that succeeds, the method should instantiate the food class and return the new food object. The method should return `#f` if the person can't buy the food.

A7. Now we need a `buy` method for people. It should take as argument the name of the food we want to buy: `(ask Brian 'buy 'bagel)` . The method must send a `sell` message to the restaurant. If this succeeds (that is, if the value returned from the `sell` method is an object rather than `#f`) the new food should be added to the person's possessions.

Person A skip to question 8 below

PART II, Person B:

`adv.scm` includes a definition of the class `thief`, a subclass of person. A thief is a character who tries to steal food from other people. Of course, Berkeley can not tolerate this behavior for long. Your job is to define a `police` class; police objects catch thieves and send them directly to jail. To do this you will need to understand how theives work.

Since a thief is a kind of person, whenever another person enters the place where the thief is, the thief gets a `notice` message from the place. When the thief notices a new person, he does one of two things, depending on the state of his internal `behavior` variable. If this variable is set to `steal` , the thief looks around to see if there is any food at the place. If there is food, the thief takes the food from its current possessor and sets his behavior to `run` . When the thief's behavior is `run` , he moves to a new random place whenever he `notice` s someone entering his current location. The `run` behavior makes it hard to catch a thief.

Notice that a thief object delegates many messages to its person object.

B5A. To help the police do their work, you will need to create a place called jail. Jail has no exits. Moreover, you will need to create a method for persons and thieves called `go-directly-to`. `go-directly-to` does not require that the new-place be adjacent to the current-place. So by calling (`ask thief 'go-directly-to jail`) the police can send the thief to jail no matter where the thief currently is located, assuming the variable thief is bound to the thief being apprehended.

B5B. Lucky for us, at least in the adventure world, we can stop thieves from stealing bicycles. To do this, we first need to invent a class `bicycle.` Instantiate one, and add it to yourself. Later, we'll update some other code so that bicycles can't be taken.

B6. Your job is to define the police class. A police officer is to have the following behavior:

The police officer stays at one location. When the officer notices a new person entering the location, the officer checks to see if that person is a thief. If the person is a thief the officer says "Crime Does Not Pay," then takes away all the thief's possessions and sends the thief directly to jail. If the person is not a thief, but has a bicycle, the officer says "No bike riding on Campus". (At this point, the policeman doesn't DO anything else, since we haven't implemented riding a bicycle yet, but it'll have to do.)

Give thieves and police default strengths. Thieves should start out stronger than persons, but police should be stronger than thieves. Of course, if you eat lots you should be able to build up enough

`strength` (mass?) to take food away from a thief. (Only a character with a lot of `chutzpah` would take food away from the police.)

Please test your code and turn in a transcript that shows the thief stealing your food, you chasing the thief and the police catching the thief. In case you haven't noticed, we've put a thief in Sproul Plaza. Then, turn in a transcript showing that policeman telling you not to ride a bicycle on campus.

B7. Now we want to reorganize `take` so that it looks to see who previously possesses the desired object. If its possessor is `'no-one` , go ahead and take it as always. Otherwise, invoke

```
(ask (ask thing 'possessor) 'may-take? receiver thing)
```

That is, a person must be able to process a message `may-take?` by calling a procedure that accepts two additional arguments: the person who wants to take the thing, and the thing itself. The associated method should return #F if the person may not take the thing, or the thing itself if the person may take it.

Note the flurry of message-passing going on here. We send a message to the taker. It sends a message to the thing, which sends messages to two people to find out their strengths.

Now, ensure that bikes cannot be taken by thieves.

End of Part II, Person B (but both of you do question 8 below)

8. Combine your work. For example, both of you have created new methods for the `person` class. Both of you have done work involving strengths of kinds of people; make sure they work together.

   Now make it so that when a `police` officer asks to `buy` some food the restaurant doesn't charge him any money. (This makes the game more realistic...)

******** OPTIONAL **********

As you can imagine, this is a truly open-ended project. If you have the time and inclination, you can populate your world with new kinds of people (e.g., punk-rockers), places (Gilman-St), and especially things (magic wands, beer, gold pieces, cars looking for parking places...).

When your instructor took this class, he and his partner added inventories, hit points, weapons, and a statistical combat simulator. In addition, we used the picture language of project 2 to add a graphical interface so that locations would draw with all of the people inside of them and people would draw with graphical depictions of the objects in their possession.

Students who are looking for a challenge are invited to make similar improvements to the adventure game. Extra credit may be given for particularly novel and interesting additions, commensurate with the difficulty of the work done.

For your enjoyment we have developed a procedure that creates a labyrinth (a maze) that you can explore. To do so, load the file ~`cs61a/lib/labyrinth.scm`.

Legend has it that there is a vast series of rooms underneath Sproul Plaza. These rooms are littered with food of bygone days and quite a few theives. You can find the secret passage down in Sproul Plaza.

[Note: `labyrinth.scm` may need some modification to work with the procedures you developed in part two of the project.]

You may want to modify `fancy-move-loop` so that you can look around in nearby rooms before entering so that you can avoid thieves. You might also want your character to maintain a list of rooms visited on its property list so you can find your way back to the earth's surface.