KOÇ UNIVERSITY
Department of Computer Engineering
COMP200 : Structure and Interpretation of Computer Programs
The Object-Oriented Adventure Game

- Code: The following code should be studied as part of this project:
  - `objsys.scm`—support for an elementary object system
  - `objtypes.scm`—some nice object classes
  - `setup.scm`—a bizarre world constructed using these classes
- Language: Select ***R5RS*** option in Dr. Racket as language. When you open the Dr. Racket, you need to select hte R5RS option from the bottom left part on the screen. **Projects that do not use R5RS will NOT be graded.**
- Submission: **Do NOT submit your transcript, written answers etc. in the files other than given. Please perform all your changes on project4.scm file. Please do not overwhelm your TA with huge volumes of material!!**
- Grading: Your work will be graded according to the following points:
  - `Run:` Your code should run without any syntactic errors. **Projects causing error will NOT be graded.**
  - `Correctness:` Your code should work correctly (No semantic errors).
  - `Readability:` Your code should be readable.
    "Software readability is a property that influences how easily a given piece of code can be read and understood. Since readability can affect maintainability, quality etc., programmers are very concerned about the readability of code." -D. Posnett, A. Hindle, P. Devanbu
  - `Clarity:` Your explanations, transcripts, and code should be simple, well-structured, well-presented, and clear.
  - `Completeness:` Your code should be complete and should not require any other dependencies apart from given ones.
- Transcript: The term "Transcript" refers to the output of your code. See `Installation` part for an example. Your transcript should be compact. **Do not run unnecessary clocks and paste their results!**

You should begin working on the assignment once you receive it. It is to your advantage to get work done early, rather than waiting until the night before it is due. You should also read over and think through each part of the assignment (as well as any project code) before you sit down at the computer. It is generally much more efficient to test, debug, and run a program that you have thought about beforehand, rather than doing the planning "online." Diving into program development without a clear idea of what you plan to do generally ensures that the assignments will take much longer than necessary. However you might want to get a feel for what our program will do by looking at and trying out the transcript in Section 3.6.

Word to the wise: This project is difficult. The trick lies in knowing *which* code to write, and for that you must understand the attached code, which is considerable. You'll need to understand the general ideas of object-oriented programming and the implementation provided of an object-oriented programming system (in `objsys.scm`). Then you'll need to understand the particular classes (in `objtypes.scm`) and the world (in `setup.scm`) that we've constructed for you. In truth, this assignment in much more an exercise in *reading* and *understanding* a software system than in writing programs, because reading significant amounts of code is an important skill that you must master. The warmup exercises will require you to do considerable digesting of code before you can start on them. And we strongly urge you to study the code before you try the programming exercises themselves. Starting to program without understanding the code is a good way to get lost, and will virtually guarantee that you will spend more time on this assignment than necessary.

In this project we will develop a powerful strategy for building simulations of possible worlds. The strategy will enable us to make modular simulations with enough flexibility to allow us to expand and elaborate the simulation as our conception of the world expands and becomes more detailed.

One way to organize our thoughts about a possible world is to divide it up into discrete objects, where each object will have a behavior by itself, and it will interact with other objects in some lawful way. If it is useful to decompose a problem in this way then we can construct a computational world, analogous to the "real" world, with a computational object for each real object.

Each of our computational objects has some independent local state, and some rules (or code) that determine its behavior. One computational object may influence another by sending it messages. The program associated with an object describes how the object reacts to messages and how its state changes as a consequence.

You may have heard of this idea in the guise of "Object-Oriented Programming systems"(OOPs!). Languages such as C++ and Java are organized around OOP. While OOP has received a lot of attention recently, it is only one of several powerful programming styles. What we will try to understand here is the essence of the idea, rather than the incidental details of their expression in particular languages.

2. An Object System

Consider the problem of simulating the activity of a few interacting agents wandering around different places in a simple world. Real people are very complicated; we do not know enough to simulate their behavior in any detail. But for some purposes (for example, to make an adventure game) we may simplify and abstract this behavior.

Let's start with the fundamental stuff first. We can think of our object oriented paradigm as consisting of *classes* and *instances*. Classes can be thought of as the "template" for how we want different kinds of objects to behave. The way we define the class of an object is with a basic "make object" procedure; when this procedure is applied, it makes for us a particular instance.

Our object instances are themselves procedures which accept messages. An object will give you a method if you send it a message; you can then invoke that method on the object (and possibly some arguments) to cause some action, state update, or other computation to occur.

2.1 Classes, Instances, and Methods

For example, our simulation world will consist of named objects. We can make a named object using the procedure `make-named-object`. A named object is a procedure that takes a message and returns the method that will do the job you want.[1] For example, if we call the method obtained from a named object by the message `NAME` we will get the object's name.

```
(define (make-named-object name)
  (let ((root-part (make-root-object)))
    (lambda (message)
      (case message
        ((NAMED-OBJECT?) (lambda (self) #T))
        ((NAME) (lambda (self) name))
        ((INSTALL) (lambda (self) 'INSTALLED))
        ((DESTROY) (lambda (self) 'DESTROYED))
        (else (find-method message root-part))))))

(define foo (make-named-object 'george))

((foo 'NAME) foo)
;Value: george
```

[1] We will use the special form `case` to do the dispatch. See the Scheme Reference Manual for details. In essence, this acts much like a `cond`, matching the first argument against the first clause of each subsequent term using `eq?`; when it finds one that matches, it evaluates and returns the subsequent part of that expression.

The first formal parameter of every method is `self`. The corresponding argument must be the object that needs the job done. This was explained in lecture, and we will see it again below.

Note that a named object inherits from a root object, which we treat as the most fundamental, and simplest, of classes.

```
(define (make-root-object)
  (lambda (message)
    (no-method)))
```

This object simply provides a basis for providing common behaviors to all classes, which for now is simply a way of indicating that no method is available for the desired message. We will by convention use this class as the base for all other classes.

A named object has a method for four different messages: `NAMED-OBJECT?`, `NAME`, `INSTALL` and `DESTROY`. Depending on the message, a named object will return a method that confirms that it is indeed a `named-object`; it will return a method to return its `name`; it will return a method for installation that does nothing; and it will return a method for destruction that does nothing.

In the above example, we created an instance `foo`, then sent it the message `NAME` to get its name method, and finally applied that method to the object itself to get the name. Our system provides a preferred short-hand way of putting together the method lookup and method application, using `ask`. What `ask` does here is get the `NAME` method from `foo` and then call it with `foo` as the argument (so the value of `foo` will be bound to `self` in the method body). The full `ask` procedure is defined in the file `objsys.scm`, but here is a simplified version that works for messages requiring no arguments:

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

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

(simple-ask foo 'NAME)
;Value: george
```

We see that our system also provides the procedure `get-method` to request a method from an object, which simply sends the message to the object. There is a special way for our objects to say there is no method: `(no-method)`, as shown in the `root-object` class definition above. This returns a special value that can be used later on in our system to detect when there is no method using the `method?` predicate, e.g.

```
(method? (foo 'NAME))
;Value: #T

(method? (foo 'SHAPE))
;Value:  #F
```

2.2 Inheritance and Subclasses

A `thing` is another kind of computational object which will be located somewhere in our world. In the code below we see that a thing is implemented as a message acceptor that intercepts some messages. If it cannot handle a particular message itself, it passes the message along to a private, internal named object (`named-object-part`) that it has made as part of itself to deal with such messages (see the last line in the definition of `make-thing`). Thus, we may think of a thing as a kind of named object except that it also handles the messages that are special to things. This arrangement is described in various ways in object-oriented jargon, e.g., "the `thing` class `inherits` from the `named-object` class," or "`thing` is a *subclass* of `named-object`," or "`named-object` is a *superclass* of `thing`."

```
(define (make-thing name location)
  (let ((named-object-part (make-named-object name)))
```

```
    (lambda (message)
      (case message
        ((THING?) (lambda (self) #T))
        ((LOCATION) (lambda (self) location))
        ((INSTALL)
         (lambda (self)         ; Install: synchronize thing and place
           (ask (ask self 'LOCATION) 'ADD-THING self)
           (delegate named-object-part self 'INSTALL)))
        ((DESTROY)
         (lambda (self)         ; Destroy: remove from place
           (ask (ask self 'LOCATION) 'DEL-THING self)
           (delegate named-object-part self 'DESTROY)))
        ((EMIT)
         (lambda (self text)         ; Output some text
           (ask screen 'TELL-ROOM (ask self 'LOCATION)
                (append (list "At" (ask (ask self 'LOCATION) 'NAME))
                        text))))
        (else (get-method message named-object-part))))))
```

There are several other interesting aspects of the `thing` class definition above. We see that a thing instance will respond to the `THING?` message with a procedure that, when applied to the instance, will return `#T`. But an object that is not a `thing` will not find the `THING?` message and an error will result. To get around this problem, and for improved convenience as well, our system provides a procedure `is-a` that can be used to check the class of an object.

```
(define (is-a object type-pred)
  (if (not (procedure? object))
      #f
      (let ((method (get-method type-pred object)))
        (if (method? method)
            (ask object type-pred)
            #F))))

(define my-book (make-thing 'great-gatsby library))

((get-method 'THING? my-book) my-book)
;Value: #T
((get-method 'NAMED-OBJECT? my-book) my-book)
;Value: #T

(is-a my-book 'THING?)
;Value: #T
(is-a my-book 'NAMED-OBJECT?)
;Value: #T
(is-a my-book 'EMOTION?)
;Value: #F
```

This enables us to ask an object if it is an instance of a particular class. For example, we can see that a thing we make is a `thing`, but also is a `named-object` (you can assume that `library` is a location previously made). How does the `is-a` procedure work? If we ask for the `THING?` method from a thing instance (`my-book`, in this case), `my-book` immediately gets and returns the method defined in `make-thing`. However, if we ask for the `NAMED-OBJECT?` method from `my-book`, the `my-book` object passes the message along to its internal `named-object-part`, where the `NAMED-OBJECT?` method is finally found and returned. The `is-a` utility procedure tries to find the appropriate type check method, and if found invokes it on the object, otherwise concluding that the object is not an instance of the requested type.

2.3 Delegation

Another idea shown in the "thing" class (which is specified by the `make-thing` procedure above) is that of *delegation*, which is the explicit use of an "internal" object's method by the object. In the `thing` class, we see that the `INSTALL` method "shadows" or intercepts the `INSTALL` method in the `named-object` class.

In `make-thing`, we want to first do some work to integrate the thing object into our simulation world (more on that later), but then we *also* want to invoke the superclass named-object `INSTALL` method in case something important happens there as well. But since the internal `named-object-part` is really not a "stand-alone" object all its own, we don't `ask` it to do something on its own, instead we `delegate` the task to the internal object. To delegate is to have the internal object do the requested work, but on *behalf* of the full `self` object.

The important difference is that if we `ask` an object to do something, then the `self` value passed to the method will be the object itself. Using `delegate`, on the other hand, we can explicitly control what the `self` value will be that is passed to the method, and can thus have a part (inherited superclass) of the object do something to the whole object. This is perhaps the single most subtle and difficult aspect of our system, and you will explore this idea and issue in more detail in the exercises.

3. Classes for a Simulated World

When you read the code in `objtypes.scm`, you will see definitions of several different classes of objects that define a host of interesting behaviors and capabilities using the OOP style discussed in the previous section. Here we give a brief "tour" of some of the important classes in our simulated world.

3.1 Container Class

Once we have `things`, it is easy to imagine that we might want containers for things. We can define a utility `container` class as shown below:

```
(define (make-container)
  (let ((root-part (make-root-object))
        (things '())) ; a list of THING objects in container
    (lambda (message)
      (case message
        ((CONTAINER?) (lambda (self) #T))
        ((THINGS) (lambda (self) things))
        ((HAVE-THING?)
         (lambda (self thing)  ; container, thing -> boolean
           (not (null? (memq thing things)))))
        ((ADD-THING)
         (lambda (self new-thing)
           (if (not (ask self 'HAVE-THING? new-thing))
               (set! things (cons new-thing things)))
           'DONE))
        ((DEL-THING)
         (lambda (self thing)
           (set! things (delq thing things))
           'DONE))
        (else (find-method message root-part))))))
```

Notice that a container does not inherit from `named-object`, so it does not support messages such as `NAME` or `INSTALL`. Containers are not meant to be stand-alone objects; rather, they are only meant to be used internally by other objects to gain the capability of adding things, deleting things, and checking if one has something.

3.1 Place Class

Our simulated world needs places (e.g. rooms or spaces) where interesting things will occur. The definition of the `place` class is shown below.

```
(define (make-place name)
  (let ((named-obj-part (make-named-object name))
        (container-part (make-container))
```

```
      (exits '()))    ; a list of exit
    (lambda (message)
      (case message
        ((PLACE?) (lambda (self) #T))
        ((MAKE-NOISE)
         (lambda (self who)
           (let ((interested (find-all self 'AWARE?)))
             (for-each (lambda (a) (ask a 'HEARD-NOISE who)) interested)
             'noise-made)))
        ((EXITS) (lambda (self) exits))
        ((EXIT-TOWARDS)
         (lambda (self direction)  ; place, symbol -> exit | #F
           (find-exit-in-direction exits direction)))
        ((ADD-EXIT)
         (lambda (self exit)
           (let ((direction (ask exit 'DIRECTION)))
             (cond ((ask self 'EXIT-TOWARDS direction)
                    (error (list name "already has exit" direction)))
                   (else
                    (set! exits (cons exit exits))
                    'DONE)))))
        (else
         (find-method message container-part named-obj-part))))))
```

If we look at the first and last lines of `make-place`, we notice that `place` inherits from two different classes: it has both an internal `named-object-part` and an internal `container-part`. Here we use the object oriented system procedure `find-method` (defined in `objsys.scm`) which will try to find the first matching method by looking (in order) in the provided internal objects. Thus, if we ask for the `NAME` method from a place instance, the method will be found in the internal `named-object-part`, while if we ask for the `HAVE-THING?` method from a place instance, the appropriate method will be found and returned from the internal `container-part` object. This idea is often termed "multiple inheritance".

You can also see that our `place` instances will each have their own internal variable `exits`, which will be a list of `exit` instances which lead from one place to another place. In our object-oriented terminology, we can say the place class establishes a "has-a" relationship with the exit class. You should examine the `objtypes.scm` file to understand the definition for `make-exit`.

3.2. Mobile-thing Class

Now that we have things that can be contained in some place, we might also want `mobile-thing`s (made by `make-mobile-thing`) that can `CHANGE-LOCATION`.

```
(define (make-mobile-thing name location)
  (let ((thing-part (make-thing name location)))
    (lambda (message)
      (case message
        ((MOBILE-THING?) (lambda (self) #T))
        ((LOCATION)     ; This shadows message to thing-part!
         (lambda (self) location))
        ((CHANGE-LOCATION)
         (lambda (self new-location)
           (ask location 'DEL-THING self)
           (ask new-location 'ADD-THING self)
           (set! location new-location)))
        ((ENTER-ROOM)
         (lambda (self exit) #t))
        ((LEAVE-ROOM)
         (lambda (self exit) #t))
        ((CREATION-SITE)
         (lambda (self)
           (delegate thing-part self 'location)))
        (else (get-method message thing-part))))))
```

When a mobile thing moves from one location to another it has to tell the old location to `DEL-THING` from its memory, and tell the new location to `ADD-THING`. Note that here we use the `ask` procedure, since we are sending a message to the specified location objects that exist external to the `mobile-thing`; it would be inappropriate to `delegate` in this situation.

3.3. Person Class

A person is a kind of mobile thing. When a person is made, an internal mobile thing is also made to handle messages such as `CHANGE-LOCATION`. The mobile thing is bound to a variable that is visible only within the person object – `mobile-thing-part`. When a person moves from one place to another, it does so by using the `CHANGE-LOCATION` method from its internal `mobile-thing-part`. However, it is the person that moves. Thus, it is the person that must be added or removed from the location, not the mobile thing from which the method was obtained. The internal `mobile-thing-part` is not a whole person – it is only a fragment of the person. To implement the desired behavior the `CHANGE-LOCATION` method needs to know the complete or whole moving object (the person), and this is what is passed to the method as `self`. This is crucial for you to understand if your objects are to maintain their integrity!

If we consider the (partial) definition of `make-person`, we also notice that a person is a container as well as a mobile thing. Again, this is an example of multiple inheritance. The idea here is that people can also "contain things" which they carry around with them when they move.

A person can `SAY` a list of phrases. A person can `TAKE` something, as well as `DROP` something. Some of the other messages a person can handle are briefly shown below; you should consult the full definition of `make-person` in `objtypes.scm` to understand the full set of capabilities a person instance has.

```
(define (make-person name birthplace)
  (let ((mobile-thing-part (make-mobile-thing name birthplace))
        (container-part    (make-container))
        (health            3)
        (strength          1))
    (lambda (message)
      (case message
        ((PERSON?) (lambda (self) #T))
        ((STRENGTH) (lambda (self) strength))
        ((HEALTH) (lambda (self) health))
        ((SAY)
         (lambda (self list-of-stuff)
           (ask screen 'TELL-ROOM (ask self 'location)
                (append (list "At" (ask (ask self 'LOCATION) 'NAME)
                              (ask self 'NAME) "says --")
                        list-of-stuff))))
        ((HAVE-FIT)
         (lambda (self)
           (ask self 'SAY '("Yaaaah! I am upset!"))
           'I-feel-better-now))
        ((PEOPLE-AROUND) (lambda (self) ...))
        ...
        ((TAKE) (lambda (self thing) ...))
        ((LOSE)
         (lambda (self thing lose-to)
           (ask self 'SAY (list "I lose" (ask thing 'NAME)))
           (ask self 'HAVE-FIT)
           (ask thing 'CHANGE-LOCATION lose-to)))
        ((DROP)
         (lambda (self thing)
           (ask self 'SAY (list "I drop" (ask thing 'NAME)
                                "at" (ask (ask self 'LOCATION) 'NAME)))
           (ask thing 'CHANGE-LOCATION (ask self 'LOCATION))))
        ...
        (else (find-method message mobile-thing-part container-part))))))
```

3.4 Avatar Class

One kind of character you will use in this problem set is an `avatar`. The avatar is a kind of person who must be able to do the sorts of things a person can do, such as `TAKE` things or `GO` in some direction. However, the avatar must be able to intercept the `GO` message, to do things that are special to the avatar, as well as to do what a person does when it receives a `GO` message. This is again accomplished by explicit delegation. The avatar does whatever it has to, and in addition, it delegates to its internal person the processing of the `GO` message, with the avatar as `self`. Notice that we have a fairly fine degree of control over how inheritance and delegation are managed. In the case of the avatar, we first delegate to the internal person to handle the `GO` message, and then do something more after that (in this case, invoke the simulation clock).

```
(define (make-avatar name birthplace murder-details)
  (let ((person-part (make-person name birthplace)))
    (lambda (message)
      (case message
        ((AVATAR?) (lambda (self) #T))
        ((LOOK-AROUND)          ; report on world around you
         (lambda (self) ...))
        ((GO)
         (lambda (self direction)  ; Shadows person's GO
           (let ((success? (delegate person-part self 'GO direction)))
             (if success? (ask clock 'TICK))
             success?)))
        ...
        ((TAKE) (lambda (self thing) ...))
        (else (get-method message person-part))))))
```

The avatar also implements an additional message, `LOOK-AROUND`, that you will find very useful when running simulations to get a picture of what the world looks like around the avatar.

3.5 Autonomous-person Class

Our world would be a rather lifeless place unless we had objects that could somehow "act" on their own. We achieve this by further specializing the person class. An `autonomous-player` is a person who can move or take actions at regular intervals, as governed by the clock through a callback.

Our clock works by using what are known as "callbacks". This means that we create an instruction which we install in the clock, with the property that every time the clock iterates, it executes all the instructions it has stored up. Each of these instructions sends a message to an object, causing it to synchronously execute an action. In the example below, installing an autonomous person causes the clock object to add an instruction that will send this object a "move-and-take-stuff" message, which will then cause this object to select an action. See the discussion on the clock in the `objsys.scm` file for details on how the clock operates. However, the template used below for sending the clock a "callback" will be valuable to you in creating your own objects and methods. Also note how, when an autonomous player dies, we send a "remove-callback" message to the clock, so that we stop asking this character to act.

```
(define (make-autonomous-player name birthplace activity miserly)
  (let ((person-part (make-person name birthplace)))
    (lambda (message)
      (case message
        ((AUTONOMOUS-PLAYER?) (lambda (self) #T))
        ((INSTALL) (lambda (self)
                     (ask clock 'ADD-CALLBACK
                          (make-clock-callback 'move-and-take-stuff self
                                               'MOVE-AND-TAKE-STUFF))
                     (delegate person-part self 'INSTALL)))
        ((MOVE-AND-TAKE-STUFF)
         (lambda (self)
           ;; first move
```

```
    (let loop ((moves (random-number activity)))
      (if (= moves 0)
          'done-moving
          (begin
            (ask self 'MOVE-SOMEWHERE)
            (loop (- moves 1)))))
    ;; then take stuff
    (if (= (random miserly) 0)
           (ask self 'TAKE-SOMETHING))
    'done-for-this-tick))
((DIE)
 (lambda (self)
   (ask clock 'REMOVE-CALLBACK self 'move-and-take-stuff)
   (delegate person-part self 'DIE)))
((MOVE-SOMEWHERE)
 (lambda (self)
   (let ((exit (random-exit (ask self 'LOCATION))))
     (if (not (null? exit)) (ask self 'GO-EXIT exit)))))
((TAKE-SOMETHING)
 (lambda (self)
   (let* ((stuff-in-room (ask self 'STUFF-AROUND))
          (other-peoples-stuff (ask self 'PEEK-AROUND))
          (pick-from (append stuff-in-room other-peoples-stuff)))
     (if (not (null? pick-from))
         (ask self 'TAKE (pick-random pick-from))
         #F))))
...
(else (get-method message person-part))))))
```

3.6 Installation

One final note about our system. If you look in `objtypes.scm`, you'll see that objects have an `INSTALL` method which does some appropriate initialization for a newly made object. For example, if you *create* a new mobile thing at a place, the object must be added to the place. As you'll see in the code, we define two procedures for each type of object: `make-` and a `create-` procedure. The make procedure (e.g. `make-person`) simply makes a new instance of the object, while the create procedure (e.g. `create-person`) both (1) makes the object *and* (2) installs it. When you create objects in our simulation world, you should do this using the appropriate create procedure. Thus, to create a new person, use `create-person` rather than calling `make-person` directly.

The following distinction should also help you think about `make-object` versus `create-object` procedures. The `make-object` procedure should only be used "inside" our object oriented programming code: e.g., in `objtypes.scm` you "make" a stand-alone person or part of an person using, for example `make-person` or `make-named-object` or whatever. But this only gives you an object that is not yet connected up with our world. To get a fully functioning object in a particular world, you need to "create" that object. Thus you should use the `create-object` variant when you actually want to make and install an object in a simulation world, as we do in `setup.scm`.

Our world is built by the `setup` procedure that you will find in the file `setup.scm`. You are the deity of this world. When you call `setup` with your name, you create the world. It has rooms, objects, and people based on a small liberal arts school on the hills of a mighty city; and it has an avatar (a manifestation of you, the deity, as a person in the world). The avatar is under your control. It goes under your name and is also the value of the globally-accessible variable `me`. Each time the avatar moves, simulated time passes in the world, and the various other creatures in the world take a time step. The way this works is that there is a clock that sends an `activate` message to all callbacks that have been created. This causes certain objects to perform specific actions. In addition, you can cause time to pass by explicitly calling the clock, e.g. using `(run-clock 20)`.

If you want to see everything that is happening in the world, do

```
(ask screen 'DEITY-MODE #t)
```

which causes the system to let you act as an all-seeing god. To turn this mode off, do

```
(ask screen 'DEITY-MODE #f)
```

in which case you will only see or hear those things that take place in the same place as your avatar is. To check the status of this mode, do

```
(ask screen 'DEITY-MODE?)
```

To make it easier to use the simulation we have included a convenient procedure, `thing-named` for referring to an object at the location of the avatar. This procedure is defined at the end of the file `objsys.scm`.

When you start the simulation, you will find yourself (the avatar) in one of the locations of the world. There are various other characters present somewhere in the world. You can explore this world, but the real goal is to get a diploma from Prof. Yuret on the graduation stage, and get out into the "real" world.

Here is a sample run of a variant of the system (we have added a few new objects to this version but it gives you an idea of what will happen). Rather than describing what's happening, we'll leave it to you to examine the code that defines the behavior of this world and interpret what is going on.

```
(setup 'george)
;Value: ready

(ask (ask me 'location) 'name)
;Value: soccer-field

(ask me 'look-around)

You are in soccer-field
You are not holding anything.
You see stuff in the room: football
You see other people: lambda-man
The exits are in directions: south
;Value: ok

(ask me 'take (thing-named 'football))

At soccer-field george says -- I take football from soccer-field
;Value: #[unspecified-return-value]

(run-clock 3)

suzy moves from suzy-cafe to student-center
lambda-man moves from soccer-field to eng-building
lambda-man moves from eng-building to soccer-field
At soccer-field lambda-man says -- Hi george
At soccer-field lambda-man says -- I take football from george
At soccer-field george says -- I lose football
At soccer-field george says -- Yaaaah! I am upset!
comp200-student moves from library to gym
comp200-student moves from gym to library
At library comp200-student says -- I take engineering-book from library
prof-yuret moves from migros to cici-bufe
At cici-bufe prof-yuret says -- Hi cici
At cici-bufe prof-yuret says -- I take kofte from cici-bufe
alyssa-p-hacker moves from eng-auditorium to eng-z21
alyssa-p-hacker moves from eng-z21 to eng-auditorium
ben-bitdiddle moves from graduation-stage to great-court
ben-bitdiddle moves from great-court to cas-building
--- the-clock Tick 0 ---
At cici-bufe cici says -- Prepare to suffer, prof-yuret !
At cici-bufe prof-yuret says -- Ouch! 3 hits is more than I want!
At cici-bufe prof-yuret says -- SHREEEEK!  I, uh, suddenly feel very faint...
At cici-bufe prof-yuret says -- I lose kofte
```

```
At cici-bufe prof-yuret says -- Yaaaah! I am upset!
An earth-shattering, soul-piercing scream is heard...
prof-yuret moves from cici-bufe to heaven
lambda-man moves from soccer-field to eng-building
lambda-man moves from eng-building to eng-z21
comp200-student moves from library to great-court
comp200-student moves from great-court to student-center
At student-center comp200-student says -- Hi suzy
alyssa-p-hacker moves from eng-auditorium to eng-z21
At eng-z21 alyssa-p-hacker says -- Hi lambda-man
At eng-z21 alyssa-p-hacker says -- I take problem-set from eng-z21
ben-bitdiddle moves from cas-building to sos-building
--- the-clock Tick 1 ---
cici moves from cici-bufe to computer-club
At student-center suzy says -- Prepare to suffer, comp200-student !
At student-center comp200-student says -- Ouch! 3 hits is more than I want!
At student-center comp200-student says -- SHREEEEK!  I, uh, suddenly feel very faint...
At student-center comp200-student says -- I lose engineering-book
At student-center comp200-student says -- Yaaaah! I am upset!
An earth-shattering, soul-piercing scream is heard...
comp200-student moves from student-center to heaven
At heaven comp200-student says -- Hi prof-yuret
lambda-man moves from eng-z21 to eng-auditorium
lambda-man moves from eng-auditorium to eng-z21
At eng-z21 lambda-man says -- Hi alyssa-p-hacker
alyssa-p-hacker moves from eng-z21 to eng-b30
At eng-b30 alyssa-p-hacker says -- I try but cannot take white-board
ben-bitdiddle moves from sos-building to cas-building
--- the-clock Tick 2 ---
;Value: done

(ask screen 'deity-mode #f)
;Value: #t

(ask me 'go 'south)

george moves from soccer-field to eng-building
--- the-clock Tick 3 ---
;Value: #t

(ask me 'go 'in)

george moves from eng-building to eng-z21
At eng-z21 lambda-man says -- Hi george
lambda-man moves from eng-z21 to eng-auditorium
At eng-z21 alyssa-p-hacker says -- Hi george
alyssa-p-hacker moves from eng-z21 to eng-building
--- the-clock Tick 4 ---
;Value: #t

(run-clock 3)

At eng-z21 lambda-man says -- Hi george
lambda-man moves from eng-z21 to eng-b30
--- the-clock Tick 5 ---
At eng-z21 lambda-man says -- Hi george
lambda-man moves from eng-z21 to eng-building
--- the-clock Tick 6 ---
--- the-clock Tick 7 ---
;Value: done
```

3.7 Changing the World

In parts of this project, you will be asked to elaborate or enhance the world (e.g. create new objects like the ones in `setup.scm`), as well as add to the behaviors or kinds of objects in the system (e.g. add modified classes to your submission file like the ones in `objtypes.scm`). If you do make such changes, you must remember to re-evaluate all definitions and re-run `(setup 'your-name)`, just to make sure that all your definitions are up to date. An easy way to do this is to reload all the files (be sure to save your files to disk before reloading), and then re-evaluate `(setup 'your-name)`.

4. Warm Up Exercises

Warm up exercises below will not be graded and we are not asking you to submit them. However, It is highly recomended that you do these exercises to capture the structure of the system. You should prepare these exercises early, in order to get a sense for the world you will be exploring.

Exercise 1: In the transcript above there is a line: `(ask (ask me 'location) 'name)`. What kind of value does `(ask me 'location)` return here? What other messages, besides `name`, can you send to this value?

Exercise 2: Look through the code in `objtypes.scm` to discover which classes are defined in this system and how the classes are related. For example, `place` is a subclass of `named-object`. Also look through the code in `setup.scm` to see what the world looks like. Draw a class diagram and a skeletal instance diagram like the ones presented in lecture. You will find such a diagram helpful (maybe indispensable) in doing the programming assignment.

Exercise 3: Look at the contents of the file `setup.scm`. What places are defined? How are they interconnected? Draw a map. You must be able to show the places and the exits that allow one to go from one place to a neighboring place.

Exercise 4: Aside from you, the avatar, what other characters roam this world? What sorts of things are around? How is it determined which room each person and thing starts out in?

Exercise 5: The avatar, as a person, may have possessions. How does the avatar handle the request `(ask me 'things)`? In particular, which method is used to respond to the request and which variable holds the list of possessions? Sketch a skeletal environment diagram to help. Note that we are not asking you to draw a fully detailed environment diagram here—it is huge and more confusing than helpful!

Exercise 6: Start the the simulation by typing `(setup '<your name>)`.

Walk the avatar to a room that has an unowned object. Have the avatar `take` this object, only to `drop` it somewhere else.

Exercise 7: You may find it useful to draw an environment diagram, in order to understand how objects inherit methods from other objects. For example, you might draw an environment diagram showing the state of the environment after evaluating:

```
(define foo (make-mobile-object 'george student-center))
```

Assume that `student-center` is bound to some procedure, but don't worry about the details of that procedure.

Further, show the state of the environment after evaluating

```
(ask foo 'location)
```

Don't worry about showing the frames created by calling `ask` or `ask-helper`.

Though it is more work, you may find it useful to think about what happens when other methods, such as `install` or `name` are called.

5. Programming Assignment

See that the three files `objsys.scm, objtypes.scm` and `setup.scm` are loaded for you in the `project4.scm`. Make sure that `project4.scm` loads correctly. Then start the simulation by typing `(setup '<your name>)`. Play with the world a bit. One simple thing to do is to stay where you are and run the clock for a while with `(run-clock <ticks>)`. Since the characters in our simulated world have a certain amount of restlessness, people should come walking by and say "Hi" to you. Try running the clock with the screen's `deity-mode` parameter set to both true and false. When it is set to true, you see almost everything that happens everywhere in the simulation. When it is set to false, you see only what happens in the room you are in. You should set `deity-mode` to false when you are ready to "play" the game (that is to get Prof. Yuret and a diploma on the graduation stage at the same time so that you can graduate).

What to turn in: When preparing your answers to the questions below, please **just** turn in the procedures that you have either written or changed **(highlighting the actual portions changed)** for each problem, a brief description of your changes, and a **brief** transcript indicating how you tested the procedure. Put each of your solutions and associated transcript into your submission file. Do not submit your transcript, written answers etc. in the files other than given. **Please do not overwhelm your TA with huge volumes of material!!**

Computer Exercise 0: Setting up the world After setting up, do the following actions using appropriate commands:

- Check where your avatar is.
- Make your avatar say its name.
- Make your avatar say "Hello World".
- Make your avatar go to another room.
- Make your avatar take stuff from the place it is in.
- Make your avatar toss the stuff it has.
- Intentionally kill your avatar.

Turn in your commands and the transcript.

Computer Exercise 1: Understanding installation Note how `install` is implemented as a method defined as part of `thing` and `autonomous-person`. Notice that the `autonomous-person` version sends a callback to the clock that will "animate" the person, then `delegate`s an `install` message from its `self` to its internal `thing`, which contains the `INSTALL` method responsible for adding the `person` to its `birthplace`. The relevant details of this situation are outlined in the code excerpts below:

```
(define (make-autonomous-person name birthplace laziness)
  ;; Laziness determines how often the person will move.
  (let ((person-part (make-person name birthplace)))
    ...
      (case message
        ...
        ((INSTALL)
         (lambda (self)
           (ask clock 'ADD-CALLBACK
```

```
                (make-clock-callback 'move-and-take-stuff self
                                     'MOVE-AND-TAKE-STUFF))
           (delegate person-part self 'INSTALL)))   ; **
        ...)))


(define (make-thing name location)
  (let ((named-object-part (make-named-object name)))
    ...
      (case message
        ...
        ((INSTALL)
         (lambda (self) ; Install: synchronize thing and place
             ...
             (ask (ask self 'LOCATION) 'ADD-THING self)
             (delegate named-object-part self 'INSTALL))
                   ...))))))
```

Louis Reasoner suggests that it would be simpler if we change the last line of the `make-autonomous-person` version of the `install` method (marked `; **`) to read:

```
                  (ask person-part 'INSTALL) ))   ; **
```

Alyssa points out that this would be a bug. "If you did that," she says, "then when you make and install an autonomous person, and this person moves to a new place, he'll be in two places at once!"

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

Computer Exercise 2: Who just died? Explore the world until "An earth-shattering, soul-piercing scream is heard...", which means that someone (hopefully not you) has just been murdered. Where does the victim go? If you know where the victim goes (and assuming you are not in `deity-mode`), what simple scheme expression can you evaluate to find out who just died?

Computer exercise 3: Having a quick look

Change the behavior of the avatar, to LOOK-AROUND whenever it successfully moves to a new location. Shows the change to your code, explain it, and demonstrate it working in an example scenario.

Computer exercise 4: But I'm too young to die!!

In the current setting, once a person's health becomes negative, they die. Implement a simple kind of reincarnation, with the following behavior. When a person dies, they still will lose all of their possessions (check the code for `person` to see how this happens). However, a person will initially start with 3 lives. If upon time of death they still have lives left, they lose a life, but reappear in their birthplace. Thus, a person can continue through several death cycles. Implement this change, by modifying the `make-person` procedure. Then reinstall the world and test out the idea (e.g. by asking `(ask me 'die)`).

Show the change to your code, explain it, and demonstrate it working in an example scenario.

NOW, FOR SOME REAL CHANGES! In the next several exercises you will extend the system to add additional behaviors and nuances.

Computer exercise 5: Perhaps to arm oneself against a sea of ....

If you have been wandering around the world, you will have discovered that Suzy and Cici, two characters that live in the student center, have a penchant for eating people. So it might help if you can defend yourself against them.

Implement a new class of object, a **weapon**. Since a weapon should be something that can be transported from place to place, you should think about the class of objects from which it should inherit. The procedure that makes instances of a weapon should take as inputs a name, a location, and a maximal amount of damage that it can inflict (see `setup.scm` for the order of these arguments). A weapon should support these methods:

- **WEAPON?**: return `#t`, to indicate this is a weapon,
- **DAMAGE**: return the maximal amount of damage the weapon can inflict,
- **HIT**: given the person using the weapon, and a target for the weapon, this method should "emit" some information about who is hitting whom with what, and then should cause the target to `SUFFER` an amount of damage. In particular, the amount of damage suffered should be a random integer no more than the `DAMAGE` of the weapon, and at least 1. An example of this in use might be

```
(ask (thing-named 'serious-weapon) 'hit me (thing-named
'ben-bitdiddle))
```

Turn in a listing of your procedure, and a transcript of your testing of it.

Once you have tested out your code, uncommenting the corresponding line in `setup.scm` to populate the world with weapons. Fell free to introduce your own weapons to the world. After that, try running the game, in which you look for a weapon, then use it to attack a target. Turn in the changes in the code, your explanation, and the transcript.

Computer exercise 6: Good thing I'm armed and dangerous

To even up the game a bit, let's allow other autonomous players also to use weapons. Create a new kind of object, a `violent-person`. This object should inherit the behaviors of an autonomous player, but with a few changes:

- Creating an instance of a violent person should include a parameter that specifies how frequently the person is violent;
- Upon installation, a callback should be sent to the clock, that will ask this object potentially to engage in a violent act on each clock tick;
- The actual method for a violent act should first decide at random whether to actually do something violent (e.g. if the frequency parameter was 4, then with a 1 in 4 probability, the person should act violently). If the decision is affirmative, then the person should pick another person in this location at random, select a weapon from among its possessions at random, and assuming there is both a victim and a weapon it should hit the target victim with the weapon.

Turn in a listing of your procedure(s), your explanation, and a transcript of your testing of it.

Once you have tested out your code, do not forget to change the code for the `setup` procedure to add violent players in the world (or change autonomous players to violent players) , and try running the game, in which you look for a weapon, then use it to attack a target, while hoping that others don't attack you first.

Computer exercise 7: A good hacker could defuse this situation

Just to make life a bit more difficult, let's add some explosive devices to our world. In particular, we want you to create a class of object called a `bomb`. A bomb should inherit properties of `mobile` things (since they can be moved) and of `aware` things. If you look at the code for this project, you will see that an `aware` thing is simply an object that can sense noises, though initially it does nothing about it.

However, when a person enters a room, he or she MAKES-NOISE, which causes the location to have HEARD-NOISE, and this in turn causes every AWARE object in that location to HEARD-NOISE. As a consequence, any AWARE object in a room is now able to react to the noise, provided they have an appropriate method. Building on this idea, a bomb should have the following properties:

- ARM: a bomb can be armed by sending it an `ARM` message. This should set some appropriate internal state variable in the bomb.
- DISARM: a bomb can be disarmed by sending it a `DISARM` message. This should set some appropriate internal state variable in the bomb.
- TRIGGER: a bomb can be sent a `TRIGGER` message. If it is armed, this message will `ACTIVATE` the bomb, otherwise nothing will happen.
- HEARD-NOISE: If a bomb hears a noise, it attempts to `TRIGGER` itself.
- ACTIVATE: If a bomb successfully activates, it finds all the people at its location, does some prespecified amount of damage (or suffering) to each, and destroys itself. It should probably also `EMIT` some information about what has happened.

Implement this idea of a bomb, then modify your `setup` procedure to populate the world with some bombs. Have your character wander around, arm some bombs, and demonstrate that they detonate as expected.

Computer exercise 8: Well, maybe only if they have enough time

Create a new kind of bomb, that has the property that when successfully triggered, it starts a timer. When the timer runs out, the bomb activates.

Add an instance of this bomb to the world, and demonstrate its use.

Computer Exercise 9: Even you can change the world! Now that you have had an opportunity to play with our "world" of characters, places, and things, we want you to extend this world in some substantial way. The last part of this project gives you an opportunity to do this.

Here, we want you to plan out the design for some extensions to your world.

Designing changes to the world – a new class We want you to design some new elements to our world. The first thing we want you to do is design a new class of objects to incorporate into the world. To do this, you should plan each of the following elements.

1. **Object class:** First, define the new class you are going to build. What kind of object is it? What are the general behaviors that you want the class to capture?
2. **Class hierarchy:** How does your new class relate to the class hierarchy of the existing world? Is it a subclass of an existing class? Is it a superclass of one or more existing classes?
3. **Class state information:** What internal state information does each instance of the class need to know?
4. **Class methods:** What are the methods of the new class? What methods will it inherit from other classes? What methods will shadow methods of other classes?
5. **Demonstration plan:** How will you demonstrate the behavior of instances of your new class within the existing simulation world?

Here are some examples of a possible new class of objects:

- A university professor. Currently Prof. Yuret is just an autonomous player. It would be better if he were actually on stage when you graduate. To do this, you need a way of getting him there. Here is one suggestion. Create a new class of object, a `university-professor`. Instances of this object will stay in the same location if there is an instance of another new class present – `alumni-donation`. Thus, if you create such an object and move it to the graduation stage, once Prof. Yuret is there, he will stay there.

- A dog. This is a very loyal dog, so it always wants to stay with its owner. Thus, if the owner is in some room together with the dog, the dog should stay in that room. If, however, the owner changes locations, the dog will want to follow. If it doesn't know which direction the owner moved, then it will need to move randomly until it finds its owner. Clearly this needs to be a specialization of a `mobile-object`. You might even consider it as a kind of `person`, though you will then need to think about what methods of a person will need to be shadowed by this kind of object.
- A bomb defuser. This is an autonomous player who tries to defuse bombs. He is very quiet and so doesn't MAKE-NOISE. He also looks for bombs, and DISARMs them.

What to turn in for this exercise

Design of your improvements You should work out a design of some new object. Use your imagination, and invent something intriguing (i.e., you don't have to make professors or dogs!). Write up a **BRIEF** description in your code file of your design, addressing each of the issues raised above.

Note your description as part of your solution in the code file.

Making it work Now, implement your new class of objects, and test them out.

For this part of the project, you will paste a transcript of your system in action in your code file. Be sure to document appropriately!