MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 2003

Project IV

- Issued: Tuesday, November 18, 2003
- Due: Friday, December 5, by 6:00pm on the tutor website.
- Code: The following code (attached) should be studied as part of this problem set:
  - evalfa03.scm—basic evaluator for part 1.
  - desugar.scm—code to convert syntatic sugar for part 1.
  - oomeval.scm—evaluator for OOPS for part 2.
  - meval-part2.scm—evaluator to use for building OOPS evaluator in part 2.

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 for that week (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.

The purpose of this project is to familiarize you with evaluators, of various flavors, including one that is designed to directly support object oriented programming. There is a lot to read and understand; we recommend that you first skim through the project to familiarize yourself with the format, before tackling problems.
Part I

Word to the wise: This part of the project doesn’t require a lot of actual programming. It does require understanding a body of code, however, and thus it will require careful preparation. You will be working with evaluators such as those described in chapter 4 of the textbook, and variations on those evaluators. If you don’t have a good understanding of how the evaluator is structured, it is very easy to become confused between the programs that the evaluator is interpreting, and the procedures that implement the evaluator itself. For this project, therefore, we suggest that you do some careful preparation. Once you’ve done this, your work in the lab should be fairly straightforward.

Part 1: Extending an existing evaluator

Load the code (evalfa03.scm) for this part of the project. This file contains a version of the meta-circular evaluator described in lecture and in the textbook. Because this evaluator is built on top of the underlying Scheme evaluator, we have called the procedure that executes evaluation \texttt{m-eval} (with associated \texttt{m-apply}) to distinguish it from the normal \texttt{eval}.

You should look through this file to get a sense for how it implements a version of the evaluator discussed in lecture (especially the procedure \texttt{m-eval}).

Part 1.1: Understanding our evaluator

You will be both adding code to the evaluator, and using the evaluator. Be careful, because it is easy to get confused. Here are things to keep in mind:

- When adding code to be used as part of \texttt{evalfa03.scm}, you are writing in Scheme, and can use any and all of the procedures of Scheme. Changes you make to the evaluator are changes in defining the behavior you want your new evaluator to have.

- After loading the evaluator (i.e., loading the file \texttt{evalfa03.scm} and any additions or modifications you make), you start it by typing \texttt{(driver-loop)}. In order to help you avoid confusion, we’ve arranged that each driver loop will print prompts on input and output to identify which evaluator you are typing at. For example,

  \begin{verbatim}
  ;;; M-Eval input: 
  (+ 3 4) 
  ;;; M-Eval value: 
  7
  \end{verbatim}

  shows an interaction with the \texttt{m-eval} evaluator. To evaluate an expression, you type the expression and press \texttt{ctrl-x ctrl-e}. Don’t use \texttt{M-z} to evaluate the expression; the presence of the prompt confuses the \texttt{M-z} mechanism. Also notice that if you have started up the evaluator in a \texttt{*scheme*} buffer, you may go to any other buffer, write definitions or expressions and evaluate them from that buffer. This will cause the expressions to be evaluated in the new evaluator. On the other hand, if you have not started up the evaluator, evaluating expressions from a buffer will cause them to be evaluated in the normal Scheme evaluator.
The evaluator you are working with does not include an error system. If you hit an error you will bounce back into ordinary Scheme. You can restart the driver-loop by running the procedure `driver-loop`. If you get hopelessly fouled up, you may need to do this, but this initializes the environment, and you will lose any definitions you have made.

It is instructive to run the interpreter while tracing `m-eval`. (You will also probably need to do this while debugging your code for this assignment.) Since environments are, in general, complex circular list structures, we have set Scheme’s printer so that it will not go into an infinite loop when asked to print a circular list.\footnote{See the descriptions of `*unparser-list-depth-limit*` and `*unparser-list-breadth-limit*` in the Scheme reference manual.}

Notice that we have chosen to use `m-define` for our definitions in `m-eval`, instead of the normal `define`. This will make it easier for you to keep track of which evaluator you are using — `defines` are done in MIT scheme as part of the creation of the evaluator; `m-define`s are done in `m-eval` as part of the tests you are doing of your new evaluator.

Exercise 1.1 Exploring `m-eval`. Load the file `evalfa03.scm` into your Scheme environment. To begin using the Scheme interpreter defined by this file, evaluate `(driver-loop)`. Notice how it now gives you a new prompt, in order to alert you to the fact that you are now “talking” to this new interpreter. Try evaluating some simple expressions in this interpreter. Note that this interpreter has no debugging mechanism — that is, if you hit an error, you will be thrown into the debugger for the underlying Scheme. This can be valuable in helping you to debug your code for the new interpreter, but you will need to restart the interpreter.

You may quickly notice that some of the primitive procedures that Scheme normally knows about are missing in `m-eval`. These include some simple arithmetic procedures (such as `*`) and procedures such as `cadr`, `cdrr`, `newline`. Extend your evaluator by adding these new primitive procedures (and any others that you think might be useful). Check through the code to figure out where this is done.

Show your changes to the evaluator, and turn in a demonstration of your extended evaluator working correctly.

Exercise 1.2 Changing style. At present, what value is returned when you define a new variable? What value is returned when you change the binding of a variable?

Modify the evaluator so that when a new variable is bound for the first time (i.e. as a definition), the evaluator returns 0 if the value is a number, `#t` if the value is a boolean, the symbol `procedure` if the value is a lambda, and `#f` otherwise. Modify the evaluator so that when a variable is rebound (i.e. as a mutation), the evaluator returns the value of the old binding.

Show your changes to the evaluator, and turn in a demonstration of your extended evaluator working correctly.

Part 1.2: Adding new special forms to Scheme

Exercise 1.3 Adding a special form. We have seen that our evaluator treats any compound expression as an application of a procedure, unless that expression is a special form that has been
explicitly defined to obey a different set of rules of evaluation (e.g., \texttt{define}, \texttt{lambda}, \texttt{if}). We are going to add some special forms to our evaluator in the next few exercises.

A very simple special form is a \texttt{while} loop. This might have the following syntax:

\begin{verbatim}
(while (> i 0)
  (newline)
  (display i)
  (newline)
  (display (factorial i))
  (set! i (- i 1)))
\end{verbatim}

This is not necessarily a style of programming that we like, since it relies on explicit mutation of variables. In the above example, the idea is that \texttt{i} would have initially be defined to some value, and then we evaluate the \texttt{while} loop. This loop tests the first subexpression to see if it is true. If it is, the loop then evaluates all of the other subexpressions in order (note the mutation of \texttt{i} to change its value) and repeats the process, doing so until the test expression evaluates to false.

Your task is to add this special form to \texttt{m-eval}, showing your changes to the code, and demonstrating that it works by providing some test cases.

To do this, you should include the following:

- Create a data abstraction for handling \texttt{while} loops, that is, selectors for getting out the parts of the loop that will be needed in the evaluation.
- Add a dispatch clause to the right part of \texttt{m-eval}. We have actually included this clause, you simply need to uncomment it.
- Write the procedure(s) that handle the actual evaluation of the \texttt{while} loop.
- Decide what value your \texttt{while} loop should return when it is done.

Be sure to turn in a listing of your additions and changes, as well as examples of your code working on test cases.

\textbf{Exercise 1.4 \ The \texttt{let*} special form.}

Now let’s try something a bit tougher. We have already seen the \texttt{let} special form. A variation on this is the \texttt{let*} special form. This form acts much like \texttt{let}, however, the clauses of the \texttt{let*} are evaluated in order (rather than all at the same time as in a \texttt{let}), and any reference in a later clause to the variable of an earlier clause should use the new value of that variable. For example

\begin{verbatim}
(m-define i 1)
(let ((i 3)
  (j (fact i))
  (list i j)))
\end{verbatim}

would return the value \((3 \ 1)\) because the \texttt{let} variables \texttt{i}, \texttt{j} are bound in parallel, and thus the argument to \texttt{fact} is the value of \texttt{i} outside the expression, namely 1. On the other hand:
(m-define i 1)  
(let* ((i 3)  
       (j (fact i))  
       (list i j)))

would return the value (3 6) since the variable i is first bound to 3, and then the variable j is  
bound, and in this case the input to fact is the new value 3.

Add let* to the evaluator. To do this, you will need to

- create some syntax procedures to extract out the parts of a let* expression
- add a dispatch clause to m-eval to handle let* expressions
- create procedure(s) to handle the evaluation of the let*. We suggest that you use the  
  following idea. If the let* has no clauses, then you simply need to evaluate the body in the  
current environment. Otherwise you need to create a new environment, extending the current  
one, in which the variable of the first clause is bound to the value (obtained by evaluation)  
of the corresponding part of the first clause. Within that environment, you need to continue  
the process, either binding the next variable to its value, or if there are not more variables,  
evaluating the body.

Add this special form to your evaluator. Turn in a listing of your changes, and examples of your  
procedures working on test cases.

Part 1.3: Adding new derived expressions to Scheme

Early in the semester, we introduced the idea of “syntactic sugar,” that is, the notion that some  
of the special forms in our language can be expressed as simple syntactic transformations of other  
language elements. Examples are cond, which can be implemented in terms of if; and let, which  
can be implemented in terms of lambda. Such expressions are also call derived expressions. Your  
job in this part will be to design and implement a new derived expression for Scheme.

Section 4.1.2 of the textbook demonstrates how to implement cond as a derived expression: to  
evaluate a cond expression, we transform it to an equivalent if expression using the procedure  
cond->if; then we evaluate the transformed expression. Let can also be implemented as a derived  
expression, as explained in exercise 4.6 on page 375.

Exercise 1.5 Transforming a while.

Implement a syntactic transformation to convert a while loop expression into an if expression.  
Then change your evaluator so that when the special form while is recognized, the evaluator  
converts this expression into an if expression and evaluates that.

Remember that in a syntactic transformation, you are converting one list structure into another  
(which will subsequently be evaluated). Thus, you should start with a simple while-loop example,  
and write down what the equivalent if expression would be.

For example,
(while (> i 0)
  (display i)
  (display (factorial i))
  (set! i (- i 1)))

can be rewritten as

(if (> i 0)
  (begin
    (display i)
    (display (factorial i))
    (set! i (- i 1))
    (while (> i 0)
      (display i)
      (display (factorial i))
      (set! i (- i 1)))
  'done)

Thus we need to generalize this transformation, and implement this transformation as a procedure, that takes the `while` expression and returns the transformed expression. This procedure can now be added to the syntax procedures of the evaluator, and an appropriate clause added to `m-eval` to handle the new type of expression.

The easiest way to start is to directly use `list`, `append`, `cons` and `quote` to create the list structure. You will see, however, that this way of doing this is a bit cumbersome, because we have to do so much direct list manipulation. An alternative method is to use what is known as `quasiquote`. If you are feeling brave, check out the appendix on quasiquote, and use that if you prefer to create this transformation.

Desugaring expressions We could continue to add special forms in this manner, but a nicer way of handling special forms is to separate the new special forms from a base set. The file `desugar.scm` contains procedures that transform a Scheme expression with various special forms such as `and`, `cond`, and `let` into an expression containing at most the five special forms `if`, `define`, `lambda`, `set!`, and `begin`.

You will note that after the interface to `m-eval` has read in a new expression, that expression is first “desugared”, and then evaluated. In this way, an interpreter that directly evaluates only these five special forms still winds up being able to handle nearly all the Scheme special forms.

Exercise 1.6 Transforming boolean combinations. Look up the official specification of the special form `or` and the special form `and` (they are very similar). Modify the procedures in `desugar.scm` to desugar `or` into an expression that `m-eval` can evaluate, and to desugar `and`. Remember that the individual expressions in the body of `or` are supposed to be `evaluated at most once`:

(or
  (begin (display "once ") #f)
  (begin (display "and ") #f)
(begin (display "only") 'done))
once and only
;Value: done

(One way to desugar \texttt{or} involves using \texttt{let}—which can itself be desugared later. In this case there can be “false capture” errors if the \texttt{let} variable already appears in the \texttt{or} expression being desugared. For this problem, it is OK to assume that any variables you use in desugaring do not already appear. Note that there is a better way to desugar an \texttt{or} expression that avoids this problem.)

**Exercise 1.7 Desugaring \texttt{let*}s.** Extend \texttt{desugar} to handle \texttt{let*} expressions, by desugaring them into some other form, such as a \texttt{let}.

**Exercise 1.8 Recursive \texttt{let}s.**

Try evaluating the following expression in MITScheme:

\begin{verbatim}
(let ((fact
    (lambda (n)
      (if (\= n 0)
        1
        (* n (fact (- n 1)))))
    (fact 10))

What happens? You might try drawing an environment diagram to figure it out. Scheme has a special form which works for this case called \texttt{letrec}:

\begin{verbatim}
(letrec ((fact
    (lambda (n)
      (if (\= n 0)
        1
        (* n (fact (- n 1)))))
    (fact 10))

Your task for this question is to add the \texttt{letrec} special form to our evaluator by desugaring it. Do this by first dropping a new frame with the variables bound to some dummy value, then changing the bindings to their actual values. However, you'll have to figure out how to express this in terms of expressions already implemented by \texttt{m-eval}. Also note that you should do this without explicitly using a \texttt{m-define}.

After implementing \texttt{letrec}, test it with a variety of expressions. Does it work the same as \texttt{let}, except for allowing recursive bindings? Are there any cases where the implementation strategy above yields answers that don't seem right?

**Appendix on transformers**

Here is an example of using quasiquote to create a transformer. Suppose we have a special form called \texttt{until}, for example:
The idea is that `until` checks to see if the first subexpression is true. If it is, it evaluates and returns the next expression, otherwise it skips to the subsequent expression, evaluates it, and repeats the process.

This can be generalized to:

```scheme
(until test
  exp1
  exp2
  ...
  expn)
```

where `test, exp1, ..., expn` are expressions. The description is that `until` evaluates the `test` expression. If the result is true, the value returned by `until` is `#t`. If the result is false, `until` evaluates each of the expressions `exp_i` in sequence, then goes back to repeat the `test`, and keeps doing this over and over until the result of the test is true.

Thus, one way to implement an `until` is to rewrite it in terms of more primitive Scheme expressions. For instance, in the above example,

```scheme
(until (> x n)
  (write-line x)
  (set! x (+ x 1)))
```

can be rewritten as

```scheme
(let ()
  (define (loop)
    (if (> x n)
        #t
        (begin (write-line x)
            (set! x (+ x 1))
            (loop))))
  (loop))
```

and thus

```scheme
(until test
  exp1
  exp2
  ...
  expn)
```

can be rewritten as
(let ()
  (define (loop)
    (if test
      #t
      (begin exp1
        exp2
        ...
        expn
        (loop)))))

\(\) where test, exp1, exp2, ..., expn can be arbitrary expressions.

Hence one way to implement the transformation is as a procedure, which takes the \texttt{until} expression and returns the transformed expression.

\[
\text{(define (until->transformed exp)}
\begin{align*}
&\text{(let ((test (cadr exp))}
&\text{(other-exps (cddr exp))})
&\text{(list}
&\text{'let}
&\text'()
&\text{(list 'define}
&\text{'(loop)
&\text{(list 'if}
&\text{test}
&\text'#t
&\text{(cons 'begin}
&\text{other-exps (list '(loop))))}
&\text'(loop))})
\end{align*}
\]

This is the direct way of creating the transform. You should check it through carefully to see how the use of \texttt{list} and \texttt{quote} will create an appropriate expression. This form is also a bit cumbersome, because we have to do so much direct list manipulation.

Here is an alternative method called quasiquote – (note that this is much hairier – don’t feel you have to use this approach as the version above is perfectly fine!! However, you can check out the Scheme reference manual for more details on quasiquote.)

\[
\text{(define (until->transformed exp)}
\begin{align*}
&\text{(let ((test (cadr exp))}
&\text{(other-exps (cddr exp))})
&\text{'(let ()}
&\text{(define (loop)
&\text{(if ,test}
&\text'#t
&\text{(cons 'begin}
&\text{other-exps (list 'loop))))}
&\text'(loop)))}
\end{align*}
\]

As this example illustrates, the special syntax characters backquote ('), comma (,), and at-sign (@) are extremely useful in writing syntactic transformations. Placing a backquote before an expression
is similar to placing an ordinary quote before the expression, except that any subexpression preceded by a comma is evaluated. If a subexpression is preceded by comma at-sign, it is evaluated (and must produce a list) and the list is appended into the result being formed. For example:

(define x '(1 2 3))
(define y '(4 5 6))

'(a b c ,x ,@y 7 8)
;Value: (a b c (1 2 3) 4 5 6 7 8)

Note that that the evaluated subexpressions of a backquote form can be actual expressions, not just symbols. Thus \texttt{until->transformed} can also be defined as

(define (until->transformed exp)
  '(let ()
    (define (loop)
      (if ,(cadr exp)
          #t
          (begin ,@(cddr exp)
            (loop))))
    (loop)))
Part II

In the previous part of this project, we have seen a basic Scheme evaluator, and in the previous project, we saw some basic ideas in Object Oriented Programming. In that first implementation of OOPS, we introduced our system as a software system, that was evaluated on top of Scheme. This is a perfectly fine way to incorporate OOPS style programming, but it exposes the implementation an OOPS system to the programmer. This means he or she can directly alter that implementation, which we may not want to allow. It also may mean that the system is less efficient (for reasons we will see in a few lectures). An alternative is to build object oriented classes, instances and other characteristics of OOPs directly into a Scheme evaluator. We are going to explore this idea in this part of the project.

If we are going to build an OOPs system directly into the evaluator, just what does that mean? Primarily it means that we don’t need to make the object representation something that was previously present in the language. We can start from scratch. Thus, we can introduce two new types into our system: the class and the instance, which we are free to design to meet our needs. The class will be the “recipe” from which instances are constructed. An instance is what we would term something that resulted from a `(make-...)` call in our old system, but here we are going to choose a different representation.

So, we will need some “built in” expressions to create classes of objects and instances of classes. We will also need a “built in” way of evaluating expressions that cause instances of classes to use methods to change the instance or to interact with other instances. By “built in”, we mean that these expressions are recognized by the evaluator as special forms or as primitive procedures, with a particular method of evaluating that form.

First, let’s deal with classes. Since we are creating our OOPs system, we have complete syntactic freedom in designing an expression to create a class. Just as `lambda` creates a procedure in normal Scheme, we will define a `kappa` expression to define or create an object class. So what needs to be included in a class? We will need

- a name (which tells us what kind of class we are creating);
- a parent (which tells us the class from which this class inherits methods and instance variables, note that a class may have a null parent);
- instance variables (parameters that define the state of each instance);
- list of methods (which identify each method by name, and provide a procedure to use in executing the method)
- the environment in which it was evaluated, so that methods will have access to variables present in the environment in which the class was constructed.

Here is our choice for the syntax of a `kappa` expression:

```
(kappa
  name
  parent
  (ivars)
  METHODS)
```
Note that `name` will be some symbol, `parent` will be an actual class object, `(ivars)` will be a list of variable names, and `METHODS` will be a list of method names paired with the procedure used to invoke the method.

An example of what gets created when we use the `kappa` special form is shown above.

For example, here are `kappa` expressions that define a couple of simple classes:

```
(define root
  (kappa
    root
    #f
    ()
    ())))

(define named-object
  (kappa
    named-object
    root
    (name)
    ((INSTALL
      (lambda (n)
        (set! name n)))
      (NAME
      Class: foo
      Ivars: ...
      Methods: ...)})

Result: (kappa foo bar (...) (...))
```

Object: 
- Class name
- Ivars: ...
- Methods: ...

Object Text: Parent pointer

Environment Pointer
The first expression creates a root class, with that name, that has no parent class, no instance variables, and no methods (a pretty boring class!). The second expression creates a named-object class that has a root class as its parent, one instance variable (name), and two methods.

Since we get to define the behavior of our system, we decree that methods have access to all the state of the class, including the state of any superclass (or parent) above that class.

One of the things we will need to do to our evaluator is add a special form for handling the evaluation of kappa expressions, with the effect that a structure representing a class is built (just as evaluating a lambda expression results in a structure representing a procedure). We will get to the details of this in the exercises.

Now, what about instances? An instance consists of some local state, plus the methods that act on it. Under our old system, the state was stored in the environment above the message-accepting procedure. Storing the state in an environment is a good choice because it will interact well with the rest of the system. While previously an instance could only store the methods, if it also stores the entire class then we can figure out which calls the instance was created from. This functionality will remove the need for all those WEAPON?, THING?, etc. methods you saw in the earlier project. Thus an instance will be created from an environment and the class from which the instance was constructed. An instance's state will be the instance variables declared in it's class, plus all the instance variables of it's superclasses.

To create an instance of a class, we decide (arbitrarily) to use the new expression, as in

(define chair (new named-object))

New should be a primitive procedure that takes a class (the actual representation for a class, not just the name for it) as argument, and creates a specific instance of that class. An instance will be a structure that includes the class object (since we will need access to its parts) and an environment. This environment should extend the environment stored in the class with a new frame, in which all the class variables of the class are initially bound to nil. The figure below demonstrates what the primitive procedure new builds when used.
We are also going to include two special names when dealing with instances: `self` and `super`. Their role requires a bit of explanation, and relates to how we are going to find a method, when classes have inheritance. The basic idea behind finding a method for an instance is to start with a class, and ascend through the linear sequence of superclasses looking for a matching method (i.e. walking up the inheritance chain until a method of the desired name is found). When a method name is found, the associated part of the method is a `lambda` that will be evaluated to create the method invocation procedure. If a method invocation wants to inherit methods from points further up the chain of superclasses, we want `super` to start that lookup above the point at which the previous lookup stopped. This is the superclass of the original object only if the method was found the base class. Thus the binding for `super` will remember the information on where the lookup stopped, in order to get future calls to `super` correct. Note that since this is done on a “per lookup” basis, the binding for `super` must be rebound on each method call, not at instance construction time.

The variable `self` should point to the particular instance the method was invoked on (so that methods can refer back to the instance, just as we used `self` in our earlier OOPs system). It will be bound in the same environment as `super`. Note that as a consequence the types of these two variables are different: `self` points to an instance, `super` points to a class. We will see details of this in the exercises.

Once we can make instances of classes, we will want to use them, that is, ask them to use methods (either their own, or those they inherit by following up the parent chain) to change state variables or interact with other instances. Again, we are free to design our syntax, and we arbitrarily opt to
use the “:” character (much as “.” is used in Java). For example,

```scheme
(define chair (new named-object))
(chair : 'install 'rocker)
(chair : 'name)
```

When we invoke a method on an instance, as in the above examples, we will hang a new frame under the instance environment containing the bindings for self and super, then we will retrieve the method text from the class corresponding to the method name passed in (the first argument after the “:”), evaluate the method procedure definition associated with the name, and finally apply that procedure to the supplied arguments (those after the method name).

Given these ideas about classes, instances and method invocations, let’s see if we can create an OOPs evaluator out of them.

1. Creating an OOPs evaluator

In this part of the project, we are going to install the descriptions above into an existing Scheme evaluator. Thus, you should load `meval-part2.scm` into a Scheme environment. This will create an evaluator that you can manipulate, just as you did in Part I of this project.

In addition, check out the file `oomeval.scm`. As you create parts of your system, you will add to this file. Loading this file on top of `meval-part2.scm` will create your new evaluator. If you then evaluate `(start-meval)` you will be able to create classes, and instances and use those instances (or at least you will once we have added pieces to complete this file).

**Exercise 2.1** First, let’s implement classes. You will see that our definition of `meval` in `oomeval.scm` (which overrides the version in `meval-part2.scm`) includes a special form for recognizing kappa expressions. To complete this, you need to provide a definition for `eval-kappa`. This should return a class abstraction with the class name, the actual parent class of the kappa expression, the list of instance variables, the list of methods, and the environment in which the kappa is evaluated. Use the `make-class` data abstraction, and the `kappa` syntax procedures. Once you have done this, reload your `oomeval.scm` file into Scheme, start up the evaluator using `(start-meval)`, and show an example of some class definitions, and tests of those definitions using `class?`.

For example, try evaluating these class definitions:

```scheme
(define root
  (kappa
   root
   #f
   ()
   ()
))

(define named-object
  (kappa
   named-object
   root
   (name)
```

(((INSTALL
  (lambda (n)
    (set! name n)))
  (NAME
    (lambda ()
      name)))))

(define food
  (kappa
    food
    named-object
    (nutrition)
    (((INSTALL
      (lambda (n nut)
        (super : 'INSTALL n)
        (set! nutrition nut)))
      (NUTRITION
        (lambda ()
          nutrition))))))

and test with (class? food).

Exercise 2.2 When we want to create an instance of a class, we will need to get all of the instance variables of that class. Since we are designing our own system, we choose to include in this list all the instance variables of the specific class, plus any instance variables of an superclasses. If there are duplicates among this list, they should be removed (actually they should probably be an error, but for now, let’s just go with removing them). Thus, we will need a way of collecting all the instance variables into a list. Complete the definition of collect-class-vars.

Now, how do we demonstrate its use? The problem is that this definition is in the underlying Scheme evaluator, but our OOPs classes get defined in the meval evaluator. So here is a trick that will let you test your code. As in the previous problem, reload your oomeval.scm file into Scheme, start up the evaluator using (start-meval), and define the same classes you did for exercise 2.1. Now evaluate the expression **quit-meval**. This will cause the meval evaluator to terminate and return control to the basic Scheme evaluator, but leaving the special variable the-global-environment bound to an environment that represents the state of the environment in meval just before we terminated. You can then test the following expressions:

((collect-class-vars
  (meval 'root the-global-environment))

((collect-class-vars
  (meval 'food the-global-environment))

The explicit call to meval with this environment will return exactly the structure we need to pass into these procedures.

Exercise 2.3 Now we can complete our method for creating instances. Complete the definition of builtin-new (once you have done this, uncomment the use of new in the list of primitive procedures
so that your evaluator will recognize and use this. **Built-in new** should take a class as argument and return an instance. An instance will be constructed using the instance data abstraction, which will include the class data abstraction and a new frame extended from the environment stored in the class, which binds the class variables (i.e. all the instance variables of the class as in exercise 2.2) to nil. See `make-list` in the reference manual for a helpful procedure for creating a list of nil’s of some length.

Note that by completing this procedure, we are adding a new primitive procedure to our evaluator. Thus to test this out, reload your `oomeval.scm` file, and start a fresh evaluator (`start-meval`). Then redefine your class definitions from exercise 2.1 (we need to do this again since we have a brand new evaluator), and create an instance of a class, e.g.

```
(define f (new food))
(instance? f)
```

**Exercise 2.4** Once we have classes and instances, we will need to tell if an object is an instance of a class. Implement `instance-of?`, which takes an instance and a class as arguments, and checks to see if the name of the instance’s class or superclasses matches the name of the class supplied. Said differently, this procedure should match the class’s name against the name of the instance’s class, then the instance’s superclass, and so on, until a match is found, or one reaches the end of the chain of superclasses. If a match is found, return `#t`, else return `#f`.

Note that `instance-of?` replaces the explicit type check methods such as `NAMED-OBJECT?`, `THING?`, `MOBILE-THING?` that we used in the earlier project.

Once you have completed your definition, use the same testing method as in exercise 2.3 – reload `oomeval.scm`, restart the evaluator, create your class definitions, and an instance of food, then test the following expressions:

```
(instance-of? f food)
(instance-of? f root)
(instance-of? (new root) food)
```

**Exercise 2.5** So now we can define classes and create instances thereof. We still need a way of invoking an instance’s method. Our goal is to support syntax of the form:

```
(chair : 'install 'rocker)
(chair : 'name)
(self : 'methodname)
(super : 'methodname args)
```

The first thing we need is a way of recognizing that an expression (which after all is written in Scheme syntax) is actually a method invocation. Since all invocations use the character `:`“, we just need a way of spotting when an expression is a list that contains the symbol `:`“. Complete the definition for `object-ref?` to do this. Note that `( : 'foo)` is a valid invocation, since this will be treated like `(self : 'foo).`
Since this procedure is just manipulating syntax, to test it, quit out of the `meval` (by evaluating `**quit-meval**`), evaluate your definition for `object-ref?` in the base evaluator, and then try some test cases:

```scheme
(object-ref? '(foo bar baz))
(object-ref? '(foo : 'bar baz))
(object-ref? '(: 'bar baz))
```

**Exercise 2.6** Once we know that we have a method invocation, we will need to get the method out of the class associated with an instance. Complete `method-lookup`. This should take a class and a method-name as input, and return a pair whose **car** is the method text itself (the procedure description associated with the method name) and whose **cdr** is the parent of the class on which the method was found. Note that if the class does not have a method for this name, then it should inherit from the superclass (or parent class).

The figure below illustrates what `method-lookup` should return.

---

**Method Lookup**

```
(method-lookup A foo)
Result: (bar . Foo's A)
(method-lookup B foo)
Result: (baz . Bar's B)
```

First element (bar, baz above) is class to start “super” lookups on.

If in Bar’s B, says (super : B), should find Baz’s B.

---

Since your definition is in the underlying Scheme, but we want to evaluate it on objects created in our new Scheme, we play the same trick as before. Specifically, having created some classes and
instances in `meval`, quit out of it (evaluating **`quit-meval`**, then evaluate your new definition for `method-lookup` in this Scheme, and test using:

```
(method-lookup (meval 'food the-global-environment) 'INSTALL)
(method-lookup (meval 'food the-global-environment) 'NAME)
```

**Exercise 2.7** Now we can complete the code needed to support method invocation. The procedure `eval-object-ref` will be used whenever an expression that invokes a method is found (see the addition to `meval` in `oomeval.scm`). We have provided a partial definition of this procedure, which you are to complete. Note that `new-self` should point to the actual object, unless `obj` is the symbol `super`, in which case it should point to the binding for `self` in the environment (since that points to the superclass). The symbol `new-super` should point to the superclass associated with the method.

The figure below illustrates what `object-ref` should produce as part of the process of applying a method.

![Object-Ref Diagram](image)

You'll also need to complete the definition for `method-call`, which actually applies the method's procedure. We want the method procedure to be applied to the supplied arguments, but we want
to do this with the procedure dropping it’s frame under a particular environment. Specifically, the procedure body will need access to the instance environment. It will also need access to a binding of super to the new value object-ref came up with. The binding for super should be in a new frame created just for this purpose (don’t use set! or define).

Once you have completed your code for method invocation, you are set to test the full system. Restart your evaluator, by reloading your completed version of oomeval.scm and invoking (start-meval). Create classes and instances (as above, for example) and show some examples of method invocation using those classes, or any other classes you choose to create.

Exercise 2.8  Create versions of the following classes from our previous project: named-object, thing, mobile-thing, container, place. You won’t be able to directly map all of these into our new system, but try to keep the same functionality. What are the major differences in this version versus the previous version? How do these changes affect the functionality of objects? Which approach works better?

Exercise 2.9  With our new implementation of OOPs, some things actually become much easier to do. One of these is the idea of “introspection”, which is the idea of examining the code at run time, in order to deduce information about it. This idea is referred to in Java as “reflection” and is used in Java Beans. The basic idea is that the system can determine how to use the objects it presently has, by deducing information about them. We can very simply implement this idea by creating three “builtin” procedures:

- (get-class instance) should return the class from which the instance was made;
- (get-method-names class) should return all the method names that a class accepts;
- (get-instance-variables class) should return all the instance variables available to a method of that class.

Implement these three procedures and install them as primitive procedures in your system.

Once you have do so, you should be able to do things like create copies of instances, for example:

(define (memq item lst)
  (cond ((null? lst) '())
        ((eq? item (car lst))
         lst)
        (else
         (memq item (cdr lst)))))

(define (for-each proc lst)
  (if (null? lst)
      'done
      (begin
       (proc (car lst))
       (for-each proc (cdr lst))))))

;; tries to create an accurate copy of an object. Creates a
;; new instance. Then tries to find a "getter" method for
;;; each instance variable. If one exists, it uses it to get
;;; the value from the original. Then it looks for a setter
;;; to use on the copy. If it exists, it uses it to set the
;;; value of the instance variable to the value extracted from
;;; the original.
(define (clone instance)
  (let ((class (get-class instance)))
    (let ((obj (new class))
      (ivars (get-instance-variables class))
        (methods (get-method-names class)))
          (for-each
            (lambda (ivar)
              (if (memq ivar methods)
                (let ((val (instance : ivar))
                  (setter-name (symbol-append 'set- ivar '!)))
                    (if (memq setter-name methods)
                      (obj : setter-name val))))
            ivars)
          obj)))

(define sheep
  (kappa
    sheep
    #f
    ((INSTALL (lambda (n c)
      (set! name n)
      (set! color c)))
      (NAME (lambda () name))
      (SET-NAME! (lambda (n) (set! name n)))
      (COLOR (lambda () color))
      (SET-COLOR! (lambda (c) (set! color c)))
      (MAKE-NOISE (lambda ()
        (newline)
        (display (self : 'NAME))
        (display " the ")
        (display (self : 'COLOR))
        (display " sheep: Baa.")))))

(define dolly (new sheep))
(dolly : 'INSTALL 'dolly 'white)
(dolly : 'MAKE-NOISE)
;dolly the white sheep: Baa.
(define dolly2 (clone dolly))
(dolly2 : 'MAKE-NOISE)
;dolly the white sheep: Baa.
(eq? dolly dolly2)
;Value: #f

Try this yourself.

We encourage you to work with others on projects so long as you acknowledge it
(see the 6.001 General INformation handout) and so long as you observe the rules on
collaboration and on using “bibles”.

If you cooperated with other students, LA’s, or others, please indicate your consultants’ names. Otherwise, write “I worked alone using only the course reference materials.”