MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.037—Structure and Interpretation of Computer Programs
IAP 2019

Project 4

Release date: 24 January, 2019
Due date: 1 February, 2019 at 2300h

Code to load for this project:

- The code for the evaluator: `oo-eval.scm`
- There are also some files which your finished evaluator will be able to run: `oo-util.scm`, `oo-types.scm`, and `oo-world.scm`. You will not need these files at first.
- All of these files can be retrieved from the course web site.

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 causes assignments to take much longer than necessary.

Word to the wise: This project can take a lot of time. You'll need to understand the general ideas of object-oriented programming and the implementation provided of an object-oriented programming system as discussed in lecture. You'll also need to have a firm grasp of the metacircular evaluator from the previous project, of which `oo-eval.scm` is a derivative. In truth, this assignment is 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. 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 apply the concepts of object orientation to the evaluator which we built in the previous project. Rather than construct an object system on top of the language, like the code handout from lecture 6 did, we will instead build it *into* the language itself, adding additional special forms and data structures to make the concepts of "classes" and "objects" into first-class types and objects in the language. That is, while `m-eval`'s goal was to mimic Scheme's semantics as closely as possible so that it could run itself, `oo-eval` intentionally adds entirely new syntaxes which will allow us to write simple code for `oo-eval` which would be considerably more complex in ordinary Scheme.

Once you have finished implementing this Scheme variant, you will be able to run, and improve upon, the provided object-oriented text adventure game which uses the new syntax and semantics.

One key thing to be careful of – symbols in DrRacket are case sensitive. For readability, we have tried to use upper case for names of object types and for names of methods.

The story so far. . . Cons of the Dead

Later reports could never clearly nail down the exact source of the plague, but they all agree that late Saturday night of the 2018 Mystery Hunt puzzle-solving competition[1], something snapped in some poor hunter. Instead of solving puzzles, he blindly went in search of more brains. The disease spread quickly through the ranks of the hunters, with one biting another until they were all shambling mockeries of their former selves.

You were working late in lab, unplugged from the world and tooling away, blissfully unaware that anything was amiss, until Alyssa P. Hacker burst in and slammed the door shut. "Zombies!" she shouted. You looked through the window in the door and saw a slightly decomposing Ben Bitdiddle trying to figure out how to operate the numeric keypad on the combo lock. "That won't hold him for long, and there are more of them coming! Quick, help me barricade the door!"

You both pile up printers, desks, and extra workstations against the door. "Now what?" you ask. Alyssa pulls a USB flash drive our of her pocket. "We hope the trouble I went through to get this was worth it. I hope you've got some food; this is going to be a long night." You nod, looking at the pile of snacks and soda you had brought with you. You were planning for an all-nighter, but not one like this. . .

Can software save your mortal soul?

Alyssa sits down at a computer and plugs in the drive. "Ben and I had been working for the last few hours on trying to determine the cause of the outbreak, with hopes we could solve it before it spread further. We hoped we might get lucky and find a way to restore these people before the National Guard shows up and starts roasting zombies.

"We worked in separate locations (for improved fault tolerance) and the zombies found him first. The timing was particularly bad– he had just Zephyred me that he'd had a break-through in the simulation he had written! And since he hadn't committed his changes yet to our git repository, I had no choice but to sneak through the zombie-infested halls over to where he was working and grab what I could off his computer."

Alyssa looks through the contents of the drive and sighs, "Most of his code has been overwritten by nonsense words, probably as his mind faded away. I'll extract what I can..."

After a few minutes, Alyssa looks at you, worried. "Ben wrote an object-oriented world simulator, so he could model various scenarios. I've recovered most of the code for his game world. But, this code isn't normal Scheme. It appears to have syntax for creating classes, instantiating objects, and so on. I know Ben had been working on adding an object system to his own personal scheme evaluator (`b-eval`), but that's in the part of his files that were corrupted." She sits back, dejected.

You exclaim, "But I just finished writing a Scheme evaluator for 6.037! I can extend that to add an implementation of an object-oriented system in no time. Then we can run Ben's simulation and see what he discovered!"

Test for success

Alyssa tells you that Ben's object system has some complexity to it, but if you start with the basics and test each piece as you go, you should be able to build up to what you need, a layer at a time.

[1] http://web.mit.edu/puzzle/www/

She says, "I'll help guide you as much as I can, including producing some test cases for you, but I need to focus on seeing if there are any patterns in this gibberish." She points at her screen. "Maybe Ben's zombie-addled subconscious had something to say– I'll see if I can decipher any more code."

If you just load and run `oo-eval.scm`, you'll find that it runs some test cases and produces an error message. This is because the file loads a series of test definitions, and then applies the `run-all-tests` procedure. You can comment-out that call if you like and run the individual tests as you go, but do occasionally run all the tests to make sure you don't have any regressions.

"Think of this as test-driven development... except with the threat of zombies. That's not usually part of this methodology."

She adds that though her tests will help, you still need to consider the requirements at each step and do your own testing in addition to what she cooks up.

Getting started

You start thinking about what you'll need to add to your evaluator: data abstractions internal to `oo-eval` for managing classes and instances, as well as the special forms or primitive procedures you'll expose to the `oo-eval` user to create classes, instantiate them, and invoke methods on the resulting instances.

Alyssa interrupts your train of thought. "Hey, look at this. I was able to recover part of an older version of Ben's evaluator with his object system implementation. I should warn you, though– he'd been feeling a little *meta* lately."

Ben had decided on a simple implementation for instances: a tagged list of the symbol `instance` and an association list[2] of all the instance's local state (slot names and current values). He would always include a special slot named `:class` which would refer to the class the instance was made from.

A snippet of Ben's code looks like this:

```
(define (instance? obj)
  (tagged-list? obj 'instance))

(define (instance-state inst)
  (second inst))

(define (instance-class inst)
  (read-slot inst ':class))

; Given an object instance and slot name, find the slot's current value
(define (read-slot instance slotname)
  (let ((result (assq slotname (instance-state instance))))
    (if result
        (cadr result)
        (error "no slot named" slotname))))
```

[2] A list of two-element lists, each containing a name and a value.

```
; Create an object instance from a class
; Store list of slots' data (including class as :class) in a tagged-list
; All slots other than :class start out with the value 'uninitialized
; Need to call the constructor if there is one
(define (make-instance class . args)
  (let ((instance
         (list 'instance
               (cons (list ':class class)
                     (map
                      (lambda (x) (list x 'uninitialized))
                      (invoke class 'GET-SLOTS))))))

    (if (class-has-method? class 'CONSTRUCTOR)
        (method-call instance 'CONSTRUCTOR class args))

    ; return the constructed instance
    instance))
```

Study this code trying to make sure you understand it. Alyssa provides you with some test cases, based on a class representation she hacked together just for this test. Her test cases are defined in these procedures: `test-problem1-0slots` and `test-problem1-3slots`. To run these tests, run `oo-eval.scm` as provided, which runs all tests.

If you want to run tests individually, you can do so from the REPL (i.e. the DrRacket Interactions window) like this:

```
> (test-instances-0slots)
RUNNING TEST: Getting started: make-instance with no extra args.
test-passed

> (test-instances-3slots)
RUNNING TEST: Getting started: make-instance with arguments.
test-passed
```

Right now, these tests should pass, if Alyssa got them right. Hopefully that will continue to be the case as you work on the following exercises.

Computer Exercises

Problem 1: It's objects all the way down

Next you'll need to consider how classes are to be represented. Alyssa made a few guesses about how Ben's system represented them for the previous test cases, but that wasn't quite right.

It turns out that Ben had decided to keep track of 4 things: its name, its parent class, the names of its slots, and its methods. But rather than just toss these all into a list with a tag on the front, he decided he already had a perfectly suitable data abstraction he could use: an instance.

That's right, Ben's decided that classes are object instances, too. But, if a class is an instance, what class is it an instance of? What's the class of a class? That's a *metaclass*. Ben had started adding a *Metaobject Protocol (MOP)* to his evaluator, to allow users of his new language lots of control over the internals of the object system. But more on that later...

Ben had rigged up a definition of a class called `default-metaclass` to bootstrap things. It looks just like an object instance, but was not created by calling `make-instance`. All other classes can be created by instantiating this class. It contains a `CONSTRUCTOR` method, which will setup the slots for a new class, a `GET-SLOTS` method which is used to assemble the instance's state when this new class is itself instantiated, and a `FIND-METHOD` method, which controls how method look-up works.

Oh, and don't be spooked by the leading colon in the slot names. They're not special to Scheme. Ben probably just thought it'd be cool to establish a consistent convention for how slots were named.

Fix the following code and pass Alyssa's tests:

```
(define (create-class name parent-class slots methods)
  'BRAINSBRAINSBRAINSBRAINS)
```

You will want to refer to the `default-metaclass`'s `CONSTRUCTOR` method to see the order the arguments are expected in. You do not need to supply `self`.

Alyssa offers you these test cases: `test-problem1-simple`, `test-problem1-subclass`.

Problem 2: We need to go deeper

Alyssa says, "Ok, you've got enough of the basic data structures down. It's time to start adding object support to your evaluator. Start with class creation. Here's the full syntax you need to support to run Ben's simulation:"

```
(make-class <name>
            <parent-class>
            (<slotname1> <slotname2> ...)
            ((<methodname1> (lambda (<args>...) <method1 body>))
             (<methodname2> (lambda (<args>...) <method2 body>))
             . . .))
```

Here's an example:

```
(define named-object
  (make-class 'NAMED-OBJECT root-object (name)
    ((NAME      (lambda () name))
     (SET-NAME! (lambda (new-name) (set! name new-name))))))
```

Note that not every sub-expression is evaluated. The name and parent-class are, but the list of slots is taken verbatim. There is also an association list of method names and procedures. Only the lambda expressions in those need to be evaluated. Since the normal rules for evaluating a combination do not apply, `make-class` must be a special form.

Create a syntactic abstraction for `make-class` (`make-class?` and selectors to extract the various parts of such an expression). Add a clause to `oo-eval` to test for `make-class` and dispatch to `eval-make-class`. Complete the implementation of `eval-make-class` and run the tests.

Alyssa has provided `test-problem2-simple` and `test-problem2-subclass`. These tests will run `oo-eval` on test expressions and check the results.

Problem 3: This is new

To instantiate classes, users of Ben's evaluator apparently use the `new` special form, which looks like this:

```
(new <class> <expression1> <expression2> ...)
```

Everything after the class is optional and should be passed to the class's `CONSTRUCTOR` method. Two examples:

```
(define sicp (new named-object 'Structure-and-Interpretation-of-Computer-Programs))
(define our-clock (new clock))
```

Handily, `new` doesn't need to be a special form, as it evaluates all of its arguments, so we can simply implement it as a primitive procedure in `oo-eval`.

Add a primitive named `new` to the list of primitive procedures, which should refer to a function you have already seen.

The tests for this problem are `test-problem3` and `test-problem3-no-args`.

Problem 4: Method to his madness

Now you need to be able to actually invoke an object's methods. Users of Ben's evaluator can just apply instances as if they were procedures, like this:

```
(define sicp (new named-object 'Structure-and-Interpretation-of-Computer-Programs))
(sicp 'NAME)
(sicp 'SET-NAME! 'The-venerable-SICP)
```

Right now that won't work, because instances are not procedures. But, you can support these semantics in your evaluator, because you have control over `oo-apply`. When an instance is applied, the evaluator needs to first locate the method requested, using the current inheritance rules, and then actually apply it.

Modify `oo-apply` to recognize the application of an instance and call `oo-apply-instance` in that case. Complete the implementation of oo-apply-instance.[3]

You may have noticed that the method bodies you have seen so far refer to an instance's slots as if they were normal variables. You will need to make sure the environment that the body of the

[3] We recommend you avoid calling `invoke` directly so you don't get caught up by Scheme's variable-args syntax.

method is evaluated in contains bindings for these, so that the normal `set!` and variable-lookup handlers will work. Currently the code in `apply-method` doesn't do this.

The way m-eval created bindings and frame in project 3 always copied the values into fresh cons cells; this would not work here because the bindings therefore aren't shared with the instance's list of internal state. Thus changes made via `set!` in a method would not actually update the internal state of the object. Ben had started working on this issue by extending the bindings and frame abstraction to add `make-frame-from-bindings` and `make-binding-shared` which might be of use to you.

Modify `apply-method` to handle slots as normal variables in the environment that a method runs in.

Alyssa has these test cases for you: `test-problem4-trivial`, `test-problem4-inheritance`, `test-problem4-slot-read`, `test-problem4-slot-write`, `test-problem4-self-and-super`.

Note that `self` and `super` are always defined when a method is applied (thanks to `apply-method`), and they are used just like instances: `(self 'METHOD ...)` and `(super 'METHOD ...)`.

Problem 5: Do you have a type?

Alyssa points out that there are many different classes in the simulation, so having a convenient way of identifying the type of an instance would be handy. Specifically she asks that you make it so that every class has a GET-TYPES method, which returns a list of the names of itself and all its parent classes. Alyssa's test case: `test-problem-5`.

Modify default-metaclass to add the GET-TYPES method.

OPTIONAL ADVANCED VARIANT:

With a little work, you can actually redefine `default-metaclass` using `oo-eval` and an appropriate `make-class` expression. You could create a subclass, which will let you can inherit the methods of the exising `default-metaclass`, but you would need to explicitly provide `FIND-METHOD` again since the object system expects the class-of-a-class to always have this method. (It would simply need to be `(super 'FIND-METHOD methodname)`.)

Or you could define a new `default-metaclass` entirely, rewriting all the existing methods into simpler variants that can access slot variables directly instead of though `read-slot` and `write-slot` and then adding you own. (Hint: instantiate the current `default-metaclass` to produce your new one, and then `set!` it into place.) Take this far enough, and expose the basic `default-metaclass` to the user and you can have a definition of `default-metaclass` entirely as user code!

Problem 6: What's in the box? (OPTIONAL)

Alyssa checks on your progress. "Wow, I think you might be able to load at least the first part of the simulation now. Wait... what's this?" She's noticed that when you ask `oo-eval` for the value of an instance, it prints out a giant blob of list structure. "This. . . is not ideal. You could manipulate the internal representation of this instance abstraction and wreak who knows what kind of havoc. Look, I know time's short, and strictly speaking this is **optional**, but I think your life will be easier if you hide the representation."

Conceptually, she tells you, if the implementation of `oo-eval` kept a table of all instances created, indexed by some unique id, it could hand out the unique ids as instances. So, anytime you create an

instance (e.g. via `make-instance`), you need to hide it in this table and return the unique identifier. Then, you only need to worry about retrieving the actual instance list structure when trying to manipulate the instance internals. Thanks to data abstraction, there are only two procedures in the instance abstraction that need to be altered, and one of them is a predicate.

It looks like Ben had this idea, too. He'd already written some helper procedures relevant to this: `hide-instance`, `make-label`, `lookup-instance`, and `instance-uid?`.

Implement this idea. Don't forget about `default-metaclass`. Make sure all prior tests still pass.

Alyssa's test cases for this are `test-problem6-basic` and `test-problem6-meval`. But you should also evaluate `(run-all-tests)` just to make sure the prior ones still work.

Ben's Simulation of an Imperfect World

Alyssa says, "I think you've got your evaluator working well enough to try running Ben's simulation!" Now it's time to turn your attention to the files which contain the simulation program: `oo-world.scm`, `oo-types.scm`, and `oo-utils.scm`

Classes for a Simulated World

When you read the code in `oo-types.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 classes in our simulated world.

Place class

Our simulated world needs places (e.g. rooms or spaces) where interesting things will occur. A `place` is a subclass of `container`. You can see that our `place` instances will each have their own `:exits` state, which will be a list of `exit` instances which lead from one place to another place. Using object-oriented terminology, we can say the place class establishes a "has-a" relationship with the exit class (as opposed to the "is-a" relationship denoting inheritance).

Mobile-thing class

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

Person class

A `person` is a kind of `mobile-thing`. Because `container` is an ancestor of `person`, people can "contain things" which they carry around with them when they move. A person can `SAY` a list of phrases. A person can `TAKE` and `DROP` things. You should consult the full definition of the `person` class in `oo-types.scm` to understand the full set of capabilities a `person` instance has.

Clocks and Callbacks

In order to provide for the passage of time in our system, we have a global clock object, whose implementation may be found in `oo-util.scm`. This class has exactly one instance, which is created when the simulation is started and is bound to the globally accessible variable `our-clock`. Unlike the real world, time passes only when we want it to, by invoking the clock's `TICK` method. The rest of the system finds out that time has passed because the clock informs them by invoking a method on them. However, not every object cares about time, so the clock only informs objects that have indicated to the clock that they care.

In order to hear about the passage of time, an object registers a *callback* with the clock. The clock object promises to activate this *callback* when time passes. As with everything else in our system, a callback is an instance, in this case of the class `clock-callback`. `Clock-callback`s are created with a name, an object, and a method name. When a `clock-callback` is `ACTIVATE`d, it invokes the requested method on the object with data supplied to the callback when it was instantiated.

To register a callback with the clock, use its `ADD-CALLBACK` method. When the clock `TICK`s, it `ACTIVATE`s every callback on its list.

Autonomous Person class

Our world would be a rather lifeless place unless we had objects that could somehow "act" on their own – i.e, computer-controlled characters. We achieve this by extending the person class with a subclass called `autonomous-person`. The `autonomous-person` can both move and take things, done at a frequency governed by its `:restlessness` and `:miserly` values (0-10 corresponding to 0-100% of the time). An instance of this class indicates that it wants to know when the clock ticks by registering a callback with the clock in its `CONSTRUCTOR` method. Finally, when an autonomous-person dies, we invoke the clock's `REMOVE-CALLBACK` method to the clock, so that we stop asking this character to act.

Avatar class

One other person class you will use in this project is an `avatar`. An avatar represents you, the player of the game. 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. Note that the `avatar` class overrides and extends the functionality of the `GO` method in the `person` class.

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

Running the Game

To run the game, evaluate `(run-game <yourname>)` in either base racket or from `oo-eval`'s driver loop. (`oo-eval`'s REPL will be started if necessary.)

This will run `oo-world.scm` through `oo-eval`, which will in turn evaluate everything in `oo-types.scm` and `oo-util.scm`. And finally the the `setup` procedure that you will find in the file `oo-world.scm` is applied.

You are the deity of this world. When you call `setup` with your name (directly or via `run-game`), you create the world. It has rooms, objects, and people based on a minor technical college on the banks of the Mighty Chuck River; 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 age by a time step, possibly with a change in state (where they are, how healthy they are, etc.). This works by using the clock object described earlier. You can cause time to pass by explicitly running the clock, e.g. using `(run-clock 20)`.

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

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

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

```
(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. To check the status of this mode, do:

```
(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 in the file `oo-util.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. This is the system Ben Bitdiddle constructed to model the zombie menace and to help him and Alyssa find a solution.

Here is a sample run of the system. 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.

```
> (run-game 'mpp)

At bldg-32-cp-hq zombie-of-george says -- uuuuUUUUuuuuh.. brains...
You are in lab-supply-room
You are not holding anything.
You see stuff in the room: coin
There are no other people around you.
The exits are in directions: up north

;;; OO-Eval input
(me 'TAKE (thing-named 'coin))

At lab-supply-room mpp says -- I take coin from lab-supply-room
;;; OO-Eval value:
taken


;;; OO-Eval input
(me 'GO 'north)

mpp moves from lab-supply-room to bldg-32-cp-hq
At bldg-32-cp-hq mpp says -- Hi zombie-of-george

--- Clock tick 0 ---
ben-bitdiddle moves from barker-library to great-dome
At great-dome ben-bitdiddle says -- Hi gjs
```

```
At great-dome ben-bitdiddle says -- I take 6001-quiz from great-dome
gjs moves from great-dome to barker-library
zombie-of-george moves from bldg-32-cp-hq to student-street
You are in bldg-32-cp-hq
You are holding: coin
There is no stuff in the room.
There are no other people around you.
The exits are in directions: south up
;;; OO-Eval value:
OK

;;; OO-Eval input
```

Problem 7: How's this thing work?

Try running the game and performing actions with the avatar. Please don't feel the need to overwhelm us with pages of output, though hilarious excerpts are always welcome. For a list of methods available to the avatar, you can look through `oo-types.scm` or evaluate

```
((me 'GET-CLASS) 'GET-METHODS)
```

Why does that work? Why are there duplicates? What code would you need to change to fix this?

Problem 8: It's just like "Off," but for Zombies (OPTIONAL)

Alyssa looks up from her workstation and says, "I think I found something! Ben came up with the notion that brandishing a fiendishly difficult puzzle might repel zombies, at least for a time."

Implement a new type of `mobile-thing`, a `hunt-puzzle`. Hunt puzzles do not have any new methods, being the simplest extension of a `mobile-thing`. Then change zombies so that if they attempt to bite a person carrying a `hunt-puzzle` they will be repulsed and unable to turn the person into a zombie. When thwarted, zombies should wax poetic about their inability to assist with the puzzle due to a lack of brains. Modify the code in `oo-world.scm` to populate the world with some `hunt-puzzle`s.

Demonstrate the effectiveness of your `hunt-puzzle`s with test cases.

And then....

Alyssa says, "Great, you've got the basic simulation working. Now all we need to do is figure out what Ben's breakthrough was. I'm still working for more clues in Ben's gibberish. Maybe you can–"

Alyssa is drowned out by banging on the door. A crowd of zombies has amassed in the hall and have started trying to force their way in. "There's not much time!" Alyssa exclaims.

The zombies continue banging on the door. Thinking quickly, Alyssa prints out a collection of puzzles from the Mystery Hunt and slides them under the door. The commotion dies down, and the zombies start shuffling about in the hallway asking for brains.

Alyssa shouts, "This won't last. Back to work!"

Problem 9: Examining your Methodology (OPTIONAL)

One powerful feature Ben's system has is *introspection*, which is the ability to programmatically examine the structure of classes and instances. This capabilty can be quite powerful. Every instance of `root-object` has the `GET-CLASS` method. Once you've obtained the class as an object, you can invoke any of the methods defined in `default-metaclass`, including `GET-METHODS`, as you saw earlier.

Having the `ACT` method in `autonomous-person` individually call `AUTO-MOVE`, `AUTO-TAKE`, etc. is cumbersome. Also, if you ever added other `AUTO-` methods, you'd have to manually add their invocations. **Alter `ACT` to invoke all methods that start with `AUTO-` that the object has.** You'll find `symbol-prefix?` will come handy. What changes do you need to make in the `zombie` class as a result so that their behavior is not changed by your alteration of `autonomous-person`?

Problem 10: Mixin it up

Another powerful property of an object-oriented system is *intercession*, which allows dynamic changing of the properties of classes, objects, methods, slots, and so on.

Generically, a *mixin* is a set of methods that can be applied to augment an existing class or instance. Every computer language out there with mixins seems to define them slightly differently. The design of Ben's object system allows for the possibility of mixing in additional methods to an individual instance, so let's implement that.

Specifically, a `mixin` has a `name` and an association list of `methods`. When mixed into an instance, the instance gains the type and the methods of the `mixin`. This is best illustrated with an example:

```
(define name-changer
  (make-mixin 'NAME-CHANGER
    (list
      (list 'CHANGE-NAME!
            (lambda (n) (set! :name n))))))

(define book (new named-object 'sicp))
(book 'NAME)
;=> sicp
((is-a 'NAMED-OBJECT) book)
;=> #t
((is-a 'NAME-CHANGER) book)
;=> #f
(instance-add-mixin! book name-changer)
(book 'CHANGE-NAME! 'way-of-kings)
(book 'NAME)
;=> way-of-kings
((is-a 'NAME-CHANGER) book)
;=> #t
```

Note how the type of the object changes, and it suddenly has an additional method. Furthermore, the method from the mixin has access to the private state of the instance!

First, in `oo-util.scm`, which is run by `oo-eval`, **implement a simple data abstraction for mixins**: `make-mixin`, `mixin?`, `mixin-name`, and `mixin-methods`. A simple tagged list would be a reasonable representation, but it's up to you.

You're going to implement mixins by sneaking a new class into the class hierarchy for the instance. This new class will have the name and methods from the mixin, but its super class will be the original class of the instance. Mixins bring no additional state to the instance.

In `oo-util.scm` again, **implement `instance-add-mixin!` to behave as described.** You will find that using `GET-CLASS` from `root-object` will be needed. You'll also find that you're going to want to be able to mutate the class of an instance, so add a `SET-CLASS!` method to `root-object`.

Hint: `instance-add-mixin!` will need to create a new class. However, the special syntax of `make-class` is going to get in your way. You may want to expose `create-class` or `default-metaclass` from the underlying evaluator in `oo-eval.scm`. It is, however, also technically possible to solve this problem without any changes to `oo-eval.scm`.

Test out your behaviour with the above example, or others that you come up with.

Problem 11: Re-Zombification!

The current implementation of `create-zombie` is unfortunate because it destroys the victim and re-creates a new instance. Any state associated with the old instance is lost (`health`, for example). Alyssa still holds out hope that the zombification is temporary. Use your mixin implementation to improve zombification.

Create a `zombie-mixin` mixin with copies of the `NAME` and `BITE` methods from the `zombie` class. Change `create-zombie` to use this mixin, instead of instantiating a `zombie` object. The `zombie` class will no longer be needed at all. You'll find that `create-zombie` will now only require a single argument, `victim`.

When a zombie is created, it should announce the transformation and stop trying to take stuff. So when someone becomes a zombie, `create-zombie` should invoke a method, `BECOME-ZOMBIE` to do these things. You'll find that you can prevent the zombie from taking stuff by modifying a certain piece of object state inherited from `autonomous-person`, or by overriding a method from `autonomous-person`. Which approach seems better to you and why? Include these changes in your `zombie-mixin`.

When you've implemented zombies, remove the code in `setup` in `oo-world.scm` to create `george` and replace it with code to make a random person the zombie.

Problem 12: The obvious solution (OPTIONAL)

Alyssa looks up from Ben's code and yawns. "This is so frustrating! Nothing else in this pile of parentheses makes any sense! This pile of Cs, Ns, Hs, and Os probably means something important– a chemical formula?– but it's getting harder and harder to concentrate....unh.."

You offer her some of your soda.

"Thanks! That certainly should keep me fully awake a little longer."

At the same time you both exclaim, "CAFFEINE!?!?"

Alyssa says, "Suddenly this rambling comment in Ben's code about 'weaponized RedBull' makes a lot more sense. Maybe extreme doses of caffeine can reverse the transformation! Quickly, model the possibility of using caffeinated weaponry! I'll see what I can scrounge up in the lab here..."

Add a new class to `oo-types.scm` called `weaponized-caffeine` which subclasses `weapon`. This class should override the `HIT` method such that when hitting something which is a zombie, it transforms them back into a person. To do that, write a procedure called `pop-class!` which takes an instance and undoes the effect of `instance-add-mixin!`. The weapon should `DESTROY` itself after being used.

Modify `setup` in oo-world.scm to add several instances of this class to the world.

Add a method to the autonomous-person class named `AUTO-CURE` which will look to see if there is a `weaponized-caffeine` object in its inventory and a `zombie` at the current location. If so, it will invoke the weapon's `HIT` method on the zombie. If you did not do problem 9, then there will be an extra step to perform to cause autonomous people to try this.

Run the simulation and see if this stems the zombie menace. Try differing quantities of people and caffeine to see what sort of ratio would work best.

It's all in the delivery

You finish up your simulation and tell Alyssa, "I think it can work, if we can find a reasonable means of delivery!" when the zombies return in force. In a matter of moments they've bashed an opening in the door and are trying to climb through and over the barricade.

You try to e-mail and Zephyr your results to everyone you can think of only to find that there's a problem with the lab's network connection. You say, "Well, Alyssa, if you've got any bright ideas, now would be the time..."

"How's your aim?!" she yells, and throws a liquid-filled balloon at the zombies. You see that she found some party balloons somewhere and had been filling them up with soda! After several volleys, the zombies start to relent and then finally collapse.

A few minutes later, Ben Bitdiddle wakes up. "Ow, my aching head. What happened? Why am I lying in a heap of students? Why do I smell strongly of Dr. Pepper?"

You start to try to explain, when Ben interrupts, "Hey! I just had the greatest idea! Since I have an evaluator which supports object-oriented programming, I should be able to rewrite it to use objects to store all of its state, including the environment, and then I could run it inside of itself!"

Alyssa just hits him with an inflatable lambda.

Problem 13: Your turn (OPTIONAL)

Now that you've had an opportunity to play with both an implementation of an object system and our "world" of characters, places, and things, we want you to extend the object system and/or the world in some substantial way. The last part of this project gives you an opportunity to do this. As you haven't had much freedom to design your own code up until now, this exercise gives you the opportunity to demonstrate your knowledge of design principles.

You could dive into the object system further. Or you could just have fun with the game! You don't have to stick with the zombie theme or the MIT locale if you don't want to. Here are a few ideas that we've come up with for extensions, but you certainly aren't limited to these:

Sleep-deprivation Some hypothesize that sleep deprivation caused the zombie plague. Make students go to sleep every once on a while. If they don't sleep for long enough, they turn into a zombie!

Scared-masses Make people who run from zombies.

Game Actually implement some sort of game with a win-condition. For instance, turn in your last `pset` to a `professor` so that you can go to Killian Court and get your `diploma`.

REPL Add a simple Read-Eval-Print-Loop so that you can play the game with simpler commands like `(GO NORTH)` and `(GET LAMP)`.

Multiple inheritance Add multiple inheritance to the system. How many different pieces of code would be touched by this change?

MOP Improve the `Metaobject Protocol`. Add the ability to add/change methods and slots dynamically at runtime. Add the ability to subclass the `default-metaclass` and use the result when creating new classes. Implement a metaclass which supports multiple inheritance.

Garbage collection Instances are added to the global table of instances by `make-instance`, but they are never removed. When a new instance is requested and the table has reached a certain size, go through it and discard items that are no longer reachable from any other objects. What's the root set from which to start your search?

Crazier stuff Racket has lots of libraries for things like graphics, networking, a web server, etc. Go nuts.

Submission

Your submission should be one (or more) text files containing your solutions to the project– code modified, added, etc. Please make sure it is easy for the course staff to spot what you've done for each problem. Be sure to include test cases that show your code works.

We encourage you to work with others on projects as long as you acknowledge it. If you cooperated with other people, please indicate your consultants' names and how they collaborated.