MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1998

Problem Set 10

- Issued: Tuesday, April 10, 1998
- Tutorial preparation – NOT! Due to the holiday on April 20 and 21, there are no tutorials next week.
- Written solutions due: Friday, April 24 in recitation
- Reading: Study section 4.1 before starting on this assignment. Read chapter sections 4.2 and 4.3 before coming to lecture on April 23.
- Code: The following code (attached) should be studied as part of this problem set:
  - `tool.scm`—the TOOL interpreter

Languages for Object-oriented programming

This problem set is probably the most difficult one of the semester, but paradoxically, the one that asks you to write the least amount of code, and for which you should have to spend the least time programming, *provided that you prepare before you start.* In particular, you will need to gain a good mastery of the metacircular evaluator in section 4.1 of the notes—this material is critical, not only to this problem set, but to all of the material for the remainder of the semester.

This problem set has been designed so that the interpreter you will be dealing with is an extension of the metacircular evaluator. The implementation below is described with reference to the programs in the book. So in order to understand what is going on, it will be necessary to work through section 4.1 before starting on this assignment.

Instead of asking you to do a lot of implementation in this problem set, we are asking you to assume the role of language designer, and to think about and discuss some issues in the design of languages for object-oriented programming. Note especially that there is a significant part of this problem set to be completed *after* you have finished with the programming assignment.

Although Object-Oriented programming has become very popular, the design of languages to support Object-Oriented programming is still an area of active research. In this problem set, you will be dealing with issues that are not well-understood, and around which there are major disagreements among language designers. The questions in this problem set will not ask you to supply "right answers." Instead, they will ask you to make reasonable design choices, and to be able to defend these choices. We hope you will appreciate that, once you have come to grips with the notion of an interpreter, you are in a position to address major issues in language design, even issues that are at the forefront of current research.[1]

We've already seen two different approaches to implementing generic operations. One is *data-directed programming*, which relies on a table to dispatch based on the types of arguments. The second method, *message-passing*, represents objects as procedures with local state. As we saw in problem set 7, these objects can be arranged in complex *inheritance* relationships, such as "a robot is a kind of person."

One drawback with both of these approaches is that they make a distinction between generic operations and ordinary procedures, or between message-passing objects and ordinary data. This makes it awkward, for example, to extend an ordinary procedure so that it also works as a generic operation on new types of data. For instance, we might like to extend the addition operator `+` so that it can add two vectors, rather than having to define a separate `vector-add` procedure.

Object-oriented languages have attempted to integrate objects into the core of the language, rather than building an object system as an extension of a non-object language. In these object-oriented languages, every data type (object) belongs to a class, and there aren't really stand-alone procedures—but rather methods that are shared by various classes. This integrated approach was pioneered in the language *Smalltalk*, which was developed at Xerox PARC starting in 1971. This is also the approach used in *Java*.

The language we will implement in this problem set is called *MIT TOOL* (Tiny Object Oriented Language). It is most closely related to *Dylan*, a language developed at the at the (former) Apple Research Center in Cambridge. Dylan received a lot of attention when it was introduced in 1991, but its fortunes have waned (along with Apple's). MIT Tool is a very simplified version of Dylan, and it is designed to make the implementation an easy extension of the metacircular evaluator of chapter 4.

1. Classes, instances, and generic functions

The framework we'll be using in TOOL (which is the same as in many object-oriented systems) includes basically the same ideas as we've already seen, although with different terminology. An object's behavior is defined by its *class*—the object is said to be an *instance* of the class. All instances of a class have identical behavior, except for information held in a set of specified *slots*, which provides the local state for the instance. Following Dylan, we'll use the convention of naming classes with names that are enclosed in angle brackets, for example `<account>` or `<number>`.[2]

The `define-class` special form creates a new class. You specify the name of the class, the class's *superclass*, and the names for the slots. In TOOL, every class has a superclass, whose behavior (and slots) it inherits. There is a predefined class called `<object>` that is the most general kind of

[1]This problem set was developed by Hal Abelson, Greg McLaren, and David LaMacchia. It draws on a Scheme implementation of Oaklisp by McLaren and a Scheme implementation of Dylan by Jim Miller. The organization of the generic function code follows the presentation of the Common Lisp Object System (CLOS) in *The Art of the Metaobject Protocol*, by Gregor Kiczales, Jim des Rivières, and Dan Bobrow (MIT Press, 1991).

[2]Keep in mind that this use of brackets is a naming convention only—like naming predicates with names that end in question mark.

object. Every TOOL class has `<object>` as an ancestor. Once you have defined a class, you use the special form `make` to create instances of it. `Make` takes the class as argument, together with a list that specifies values for the slots. The order in which the slots and values are listed does not matter, since each slot is identified by name. For example, we can specify that a "cat" is a kind of object that has a size and a breed, and then create an instance of `<cat>`. Note the use of the `get-slot` procedure to obtain the value in a designated slot.

```
Tool==> (define-class <cat> <object> size breed)
;;; Tool value: (defined class: <cat>)

Tool==> (define garfield (make <cat> (size 6) (breed 'weird)))
;;; Tool value: ok

Tool==> (get-slot garfield 'breed)
;;; Tool value: weird
```

Tool doesn't have ordinary procedures. Rather, it has *generic-functions*, which do different things depending upon the classes of their arguments. A generic function is defined using the special form `define-generic-function`:

```
Tool==> (define-generic-function 4-legged?)
;;; Tool value: (defined generic function: 4-legged?)
```

You can think of a newly defined generic function as an empty table to be filled in with *methods*. You use `define-method` to specify methods for a generic function that determine its behavior on various classes.

```
Tool==> (define-method 4-legged? ((x <cat>))
           true)
;;; Tool value: (added method to generic function: 4-legged?)

Tool==> (define-method 4-legged? ((x <object>))
           'Who-knows?)
;;; Tool value: (added method to generic function: 4-legged?)

Tool==> (4-legged? garfield)
;;; Tool value: #t

Tool==> (4-legged? 'Hal)
;;; Tool value: who-knows?
```

The list in `define-method` following the generic function name is called the list of *specializers* for the method. This is like an argument list, except that it also specifies the class of each argument. In the first example above, we define a method for `4-legged?` that takes one argument named `x`, where `x` is a member of the class `<cat>`. In the second example, we define another method for `4-legged?` that takes one argument named `x`, where `x` is a member of the class `<object>`. Now

`4-legged?` will return true if the argument is a cat, and will return `who-knows?` if the argument is an object. Notice that `garfield` is an object as well as a cat (because `<object>` is the superclass of `<cat>`). Yet, when we call `4-legged?` with `garfield` as an input, TOOL uses the method for `<cat>`, and not the method for `<object>`. In general, TOOL uses the *most specific method* that applies to the inputs.[3]

In a similar way, we can define a new generic function `say` and give it a method for cats (and subclasses of cats):

```
Tool==> (define-generic-function say)
;;; Tool value: (defined generic function: say)

Tool==> (define-method say ((cat <cat>) (stuff <object>))
          (newline)
          (print 'meow) ;print is TOOL's procedure for printing things
          (newline)
          (print stuff)
          'done)
;;; Tool value: (added method to generic function: say)

Tool==> (define-class <house-cat> <cat> address)
;;; Tool value: (defined class: <house-cat>)

Tool==> (define socks                  ;note that a house cat is a cat, and therefore
          (make <house-cat>    ;has slots for breed and size, as well
                (size 'medium) ;as for address
                (address '(1600 Pennsylvania Avenue Washington DC))))
;;; Tool value: ok

Tool==> (get-slot socks 'breed)
;;; Tool value: *undefined*     ;we never initialized Socks's breed

Tool==> (say garfield '(feed me))
meow
(feed me)
;;; Tool value: done

Tool==> (say socks '(feed me))
meow
(feed me)
;;; Tool value: done

Tool==> (say 'hal '(feed me))
;No method found -- APPLY-GENERIC-FUNCTION
```

In the final example, TOOL signals an error when we apply `say` to the symbol `hal`. This is because `hal` is a symbol (not a cat) and there is no `say` method defined for symbols.

[3] See the code (below) for a definition of "most specific method." What "most specific method" ought to mean is something that language designers argue about.

We can go on to define more subclasses of `<cat>`:

```
Tool==> (define-class <show-cat> <cat> awards)
;;; Tool value: (defined class: <show-cat>)

Tool==> (define-method say ((cat <show-cat>) (stuff <object>))
          (newline)
          (print stuff)
          (newline)
          (print '(I am beautiful))
          'done)
;;; Tool value: (added method to generic function: say)

Tool==> (define Cornelius-Silverspoon-the-Third
           (make <show-cat>
                 (size 'large)
                 (breed '(Cornish Rex))
                 (awards '((prettiest skin)))))
;;; Tool value: ok

Tool==> (say cornelius-silverspoon-the-Third '(feed me))
(feed me)
(i am beautiful)
;;; Tool value: done


Tool==> (define-method say ((cat <cat>) (stuff <number>))
          (newline)
          (print '(cats never discuss numbers))
          'done)
;;; Tool value: (added method to generic function: say)

Tool==> (say socks 37)
(cats never discuss numbers)
;;; Tool value: done
```

As the final example illustrates, TOOL picks the appropriate method for a generic function by examining the classes of *all* the arguments to which the function is applied. This differs from the message-passing model, where the dispatch is done by a single object.

Notice also that TOOL knows that 37 is a member of the class `<number>`. In TOOL, *every* data object is a member of some class. The classes `<number>`, `<symbol>`, `<pair>`, and `<procedure>` are predefined, with `<object>` as their superclass. Also, *every* procedure is a generic procedure, to which you can add new methods. The following generic procedures are predefined, each initially with a single method as indicated by the specializer:

```
+         (<number> <number>)
-         (<number> <number>)
*         (<number> <number>)
/         (<number> <number>)
```

```
=         (<number> <number>)
>         (<number> <number>)
<         (<number> <number>)
sqrt      (<number>)
cons      (<object> <object>)
car       (<pair>)
cdr       (<pair>)
true?     (<object>)
false?    (<object>)
not       (<object>)
null?     (<object>)
print     (<object>)
get-slot  (<object> <symbol>)
set-slot! (<object> <symbol> <object>)
display   (<object>)
newline   ()
```

2. The TOOL Interpreter

A complete listing of the TOOL interpreter is appended to this problem set. This section leads you through the most important parts, describing how they differ from the Scheme evaluator in section 4.1.

EVAL and APPLY

We've named the eval procedure `tool-eval` so as not to confuse it with Scheme's ordinary `eval`. The only difference between `tool-eval` and the `eval` in chapter 4 are the new cases added to handle the new special forms: `define-generic-function`, `define-method`, `define-class`, and `make`. Each clause dispatches to the appropriate handler for that form. Note that we have deleted `lambda`; all TOOL functions are defined with `define-generic-function`.[4]

```
(define (tool-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ;;we've omitted lambda
        ;;((lambda? exp)
        ;;   (make-procedure (lambda-parameters exp) (lambda-body exp) env))
        ((generic-function-definition? exp)
         (eval-generic-function-definition exp env))
        ((method-definition? exp) (eval-define-method exp env))
```

[4] Omitting `lambda` takes away our ability to have unnamed procedures, as we do in Scheme. You might want to think about how to add such a feature to TOOL.

```
        ((class-definition? exp) (eval-define-class exp env))
        ((instance-creation? exp) (eval-make exp env))

        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (tool-eval (cond->if exp) env))
        ((application? exp)
         (tool-apply (tool-eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL >> " exp))))
```

`Apply` also gets an extra clause that dispatches to a procedure that handles applications of generic functions.

```
(define (tool-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment (procedure-parameters procedure)
                              arguments
                              (procedure-environment procedure))))
        ((generic-function? procedure)
         (apply-generic-function procedure arguments))
        (else (error "Unknown procedure type -- APPLY"))))
```

New data structures

A *class* is represented by a data structure that contains the class name, a list of slots for that class, and a list of all the ancestors of the class. For instance, in our cat example above, we would have a class with the name `<house-cat>`, slots `(address size breed)`, and superclasses `(<cat> <object>)`. Note that the slot names include *all* the slots for that class (i.e., including the slots for the superclass). Similarly, the list of ancestors of a class includes the superclass and all of its ancestors.

A *generic function* is a data structure that contains the name of the function and a list of the methods defined for that function. Each method is a pair—the specializers and the resulting procedure to use. The specializers are a list of classes to which the arguments must belong in order for the method to be applicable. The procedure is represented as an ordinary Scheme procedure.

An *instance* is a structure that contains the class of the instance and the list of values for the slots.

See the attached code for details of the selectors and constructors for these data structures.

Defining generic functions and methods

The special form `(define-generic-function` *name*`)` is handled by the following procedure:

```
(define (eval-generic-function-definition exp env)
  (let ((name (generic-function-definition-name exp)))
    (let ((val (make-generic-function name)))
      (define-variable! name val env)
      (list 'defined 'generic 'function: name))))
```

This procedure extracts the *name* portion of the expression and calls `make-generic-function` to create a new generic function. Then it binds *name* to the new generic function in the given environment. The value returned is a message to the user, which will be printed by the read-eval-print loop.

`Eval-define-method` handles the special form `(define-method` *generic-function* `(`*params-and-classes*`) .` *body*`)`

For example

```
(define-method say ((cat <cat>) (stuff <number>))
          (newline)
          (print '(cats never discuss numbers))
          'done)
```

In general here, *generic-function* is the generic function to which the method will be added, *params-and-classes* is a list of parameters for this method and the classes to which they must belong, and *body* is a procedure body, just as for an ordinary Scheme procedure.[5] The syntax procedures for this form include appropriate procedures to select out these pieces (see the code).

`Eval-define-method` first finds the generic function. Notice that the *generic-function* piece of the expression must be evaluated to obtain the actual generic function. `Eval-define-method` disassembles the list of *params-and-classes* into separate lists of parameters and classes. The parameters, the *body*, and the environment are combined to form a procedure, just as in Scheme. The classes become the specializers for this method. Finally, the method is installed into the generic function.

```
(define (eval-define-method exp env)
  (let ((gf (tool-eval (method-definition-generic-function exp) env)))
    (if (not (generic-function? gf))
        (error "Unrecognized generic function -- DEFINE-METHOD >> "
               (method-definition-generic-function exp))
        (let ((params (method-definition-parameters exp)))
          (install-method-in-generic-function
           gf
           (map (lambda (p) (paramlist-element-class p env))
                params)
           (make-procedure (map paramlist-element-name params)
                           (method-definition-body exp)
                           env))
          (list 'added 'method 'to 'generic 'function:
                (generic-function-name gf))))))
```

[5]The dot before the word "body" signifies that we can put more than one expression in the body—just as with ordinary Scheme procedures.

Defining classes and instances

The special form `(define-class` *name superclass* . *slots*`)` is handled by

```
(define (eval-define-class exp env)
  (let ((superclass (tool-eval (class-definition-superclass exp)
                               env)))
    (if (not (class? superclass))
        (error "Unrecognized superclass -- MAKE-CLASS >> "
               (class-definition-superclass exp))
        (let ((name (class-definition-name exp))
              (all-slots (collect-slots
                          (class-definition-slot-names exp)
                          superclass)))
          (let ((new-class
                 (make-class name superclass all-slots)))
            (define-variable! name new-class env)
            (list 'defined 'class: name))))))
```

The only tricky part here is that we have to collect all the slots from all the ancestor classes to combine with the slots declared for this particular class. This is accomplished by the procedure `collect-slots` (see the code).

The final special form `(make` *class slot-names-and-values*`)` is handled by the procedure `eval-make`. This constructs an instance for the specified class, with the designated slot values. See the attached code for details.

REST STOP
This is a lot to absorb, isn't it?

Applying generic functions

Here is where the fun starts, and what all the preceding machinery was for. When we apply a generic function to some arguments, we first find all the methods that are applicable, given the classes of the arguments. This gives us a list of methods, of which we will use the first one. (We'll see why the first one in a minute.) We extract the procedure for that method and apply that procedure to the arguments. Note the subtle recursion here: `apply-generic-function` (below) calls `tool-apply` with the procedure part of the method.

```
(define (apply-generic-function generic-function arguments)
  (let ((methods (compute-applicable-methods-using-classes
                  generic-function
                  (map class-of arguments))))
    (if (null? methods)
        (error "No method found -- APPLY-GENERIC-FUNCTION")
        (tool-apply (method-procedure (car methods)) arguments))))
```

To compute the list of "applicable methods" we first find all methods for that generic function that can be applied, given the list of classes for the arguments. We then sort these according to an ordering called `method-more-specific`. The idea is that the first method in the sorted list will be the most specific one, which is the the best method to apply for those arguments.

```
(define (compute-applicable-methods-using-classes generic-function classes)
  (sort
   (filter
    (lambda (method)
      (method-applies-to-classes? method classes))
    (generic-function-methods generic-function))
   method-more-specific?))
```

To test if a method is applicable, given a list of classes of the supplied arguments, we examine the method specializers and see whether, for each supplied argument, the class of the argument is a subclass of the class required by the specializer:

```
(define (method-applies-to-classes? method classes)
  (define (check-classes supplied required)
    (cond ((and (null? supplied) (null? required)) true)
          ;;something left over, so number of arguments does not match
          ((or (null? supplied) (null? required)) false)
          ((subclass? (car supplied) (car required))
           (check-classes (cdr supplied) (cdr required)))
          (else false)
          ))
    (check-classes classes (method-specializers method)))
```

To determine subclasses, we use the class ancestor list: `class1` is a subclass of `class2` if `class2` is a member of the class ancestor list of `class1`:

```
(define (subclass? class1 class2)
  (or (eq? class1 class2)
      (memq class2 (class-ancestors class1))))
```

Finally, we need a way to compare two methods to see which one is "more specific." We do this by looking at the method specializers. `Method1` is considered to be more specific than `method2` if, for each class in the list of specializers, the class for `method1` is a subclass of the class for `method2`. (See the procedure `method-more-specific?` in the attached code.)

Classes for Scheme data

TOOL is arranged so that ordinary Scheme data objects—numbers, symbols, and so on—appear as TOOL objects. For example, any number is an instance of a predefined class called `<number>`, which is a class with no slots, whose superclass is `<object>`. The TOOL interpreter accomplishes this by having a special set of classes, called `scheme-object-classes`. If a TOOL object is not an ordinary instance (i.e., an instance data structure as described above), the interpreter checks whether it belongs to one of the Scheme object classes by applying an appropriate test. For example, anything that satisfies the Scheme predicate `number?` is considered to be an instance of `<number>`. See the code for details.

Initial environment and driver loop

When the interpreter is initialized, it builds a global environment that has bindings for `true`, `false`, `nil`, the pre-defined classes, and the initial set of generic functions listed at the end of section 1. The driver loop is essentially the same as the `driver-loop` procedure in chapter 4 of the book. One cute difference is that this driver loop prints values using the TOOL generic function `print`. By defining new methods for `print`, you can change the way the interpreter prints data objects.

3. Programming assignment

When you load the code for this problem set, the entire TOOL interpreter code (attached) will be loaded into Scheme. However, in order to do the programming assignment, you will need to modify only a tiny bit of the interpreter. This code has been separated out in the file `ps10-mod.scm`, so you can edit it conveniently. (You may also need to define some auxiliary procedures.)

To start the TOOL interpreter, type `(initialize-tool)`. This initializes the global environment and starts the read-eval-print loop. To evaluate a TOOL expression, type the expression after the prompt, followed by CTRL-`x` CTRL-`e`.

In order to keep the TOOL interpreter simple, we have not provided any mechanism for handling errors. Any error (such as an unbound variable) will bounce you back into Scheme's error handler. To get back to TOOL, quit out of the error and restart the driver loop by typing `(driver-loop)`. If you make an error that requires reinitializing the environment, you can rerun `initialize-tool`, but this will make you lose any new classes, generic functions, or methods you have defined.

Computer exercise 1: Show how to implement two-dimensional vector arithmetic in TOOL by extending the generic functions `+` and `*`, which are already predefined to work on numbers. Define a class `<vector>` with slots `xcor` and `ycor`. Arithmetic should be defined so that adding two vectors produces the vector sum, and multiplying two vectors produces the dot product

$$(x_1, y_1) \cdot (x_2, y_2) \mapsto x_1 x_2 + y_1 y_2$$

Multiplying a number times a vector, or a vector times a number, should scale the vector by the number. Adding a vector plus a number is not defined. Also define a generic function length, such that the length of a vector is its length and the length of a number is its absolute value. Also, add a `print` method for vectors so that TOOL will print vectors showing their xcor and ycor. Turn in your definitions and a brief interaction showing that they work. Here are some useful definitions (not included in the problem set code) that illustrate one way to get started:

```
(define-method - ((x <number>))
  (- 0 x))

(define-generic-function square)
(define-method square ((x <object>))
  (* x x))
```

Computer exercise 2: (This is really an exercise in reading code, not programming.) Towards the end of section 1 of this problem set, there was an example showing the result of evaluating the expression `(say socks 37)`. Explain how `apply-generic-function` finds the correct `say` method when we ask the cat `socks` to say a number. In particular, what are all the applicable methods? In what order will they appear after they are sorted according to `method-more-specific`?

Computer exercise 3: One annoying thing about TOOL is that if you define a method before you've defined a generic function for that method, you will get an error. For example, in the first example in section 1, we had to explicitly evaluate

```
(define-generic-function 4-legged?)
```

before we could evaluate

```
(define-method 4-legged? ((thing <object>))
  'Who-knows?)
```

If we hadn't done this, the second expression would have given the error that `4-legged?` is undefined. Modify the TOOL interpreter so that, if the user attempts to define a method for a generic function that does not yet exist, TOOL will first automatically define the generic function. One thing to consider: In which environment should the name of the generic function be bound: the global environment, the environment of the evaluation? some other environment? There is no "right answer" to this question—*you* are the language designer. But whatever choice you make, write a brief paragraph justifying your choice. In particular, include an example of a program for which the choice of environment matters, i.e., where the program would have a different behavior (or perhaps give an error) if the choice were different. (Hint: The only procedure you should need to modify for this exercise is `eval-define-method`.) Turn in, along with your design justification, your modified code together with a brief interaction showing that the modified interpreter works as intended.)

Computer exercise 4: Another inconvenience in TOOL is that we need to use `get-slot` in order to obtain slot values. It would be more convenient to have TOOL automatically define selectors for slots. For example, it would be nice to be able to get the x and y coordinates of a vector by typing `(xcor v)` and `(ycor v)` rather than `(get-slot v 'xcor)` and `(get-slot v 'ycor)`. Modify the interpreter to do this. Namely, whenever a class is defined, TOOL should automatically define a generic function for each of its slot names, together with a method that returns the corresponding slot value for arguments of that class. Turn in a listing of your code and an example showing that it works. (Hint: The only part of interpreter you need to modify for this exercise is `eval-define-class`.)

Lab exercise 5: Give some simple example of defining some objects and methods (besides cats and vectors) that involve subclasses, superclasses, and methods, and which illustrate the modifications you made in exercises 3 and 4.

4. Multiple Superclasses: To do AFTER you are done programming

This final question asks you to consider a tricky issue in language design. We are not requiring you to actually implement your design. Nevertheless, we do expect you to think carefully about the issues involved and to give a careful description of the solution you come up with. Don't think that this is a straightforward exercise—designers of object-oriented languages are still arguing about it.

The major way in which TOOL is simpler than other object-oriented languages is that each class has only *one* immediate superclass. As illustrated with message-passing systems (lecture on March 19), there are cases where it is convenient to have a class inherit behavior from more than one kind of class.[6]

This will involve some changes to TOOL. As a start, the syntax for `define-class` must be modified to accept a list of superclasses rather than a single superclass. Let's assume that `define-class` now takes a list of superclasses. For instance, going back to our original example about cats, we might have:

```
(define-class <fancy-house-cat> (<house-cat> <show-cat>))
```

This new class should inherit from both `<house-cat>` and `<show-cat>`. In general, when the new class is constructed, it should inherit methods and slots from *all* its superclasses (and their ancestors).

However, it's not obvious what inheritance should mean. For example, suppose we have a generic function `eat` and we define methods as follows:

```
(define-method eat ((c <house-cat>))
   (print '(yum: I'm hungry)))

(define-method eat ((c <show-cat>))
   (print '(I eat only caviar)))
```

What should happen when we ask a fancy-house-cat (which is both a show-cat and a house-cat) to eat? More generally, what is the "most specific method" that should be used when a generic function is applied to its arguments, given that some of the arguments may have multiple superclasses? What are the new kinds of choices that arise? How should the language give the user the ability to control these choices? (Or maybe it *shouldn't* give the user this level of control.)

[6]Java also enforces the restriction that a class has at most one superclass. On the other hand, Java includes a different mechanism called *interfaces* that provides some of the capabilities associated with multiple superclasses.

Post-computer exercise 6: You are now a language designer. Your task is to design an extension to TOOL so that it handles classes with multiple superclasses. Three of the issues you have to deal with are: (a) What should be the syntax for defining classes? (b) What slots does a class get when it is defined? (c) How is a method chosen when a generic function is applied to its arguments? Prepare a design writeup that has three parts:

1. Write a clear 2–3 page description of your language extension. This description should be geared toward the *user* of the language. It should include a simple, but realistic and non-trivial example of a program that involves multiple superclasses. The example should illustrate how your language handles each of the three issues (a), (b), and (c). You should also explain how the language deals with each of these issues in general.

2. For each of design choices you illustrated in part 1, give an *alternative* choice you could have made, and explain briefly why you think your choice is better. If you can't think of any other choice you might have made, then say so.

3. As carefully as you can (but without actually writing any code) specify the procedure that the evaluator should follow in choosing which method to select when applying a generic function to a given set of arguments. Your description should be clear enough so that someone could implement this procedure based upon your specification.

Optional extra credit Implement your design for multiple superclasses in TOOL and demonstrate that it works. The TOOL interpreter was designed to make this not too difficult, but it will involve a considerable number of small changes to the code and is likely to be time-consuming.