MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1995

Lecture Notes – March 21

Object-Oriented Programming

Two different kinds of models

Functional models:

- Variables stand for values.
- A compound structure is determined by its parts: Two compound structures with the same parts are the same.

Object models:

- Variables stand for places whose contents can change.
- Compound structures are not determined by their parts, but rather by their "identity".

In Lisp, we say that two symbols are identical (`eq?`) if they print the same. But we stipulate that every call to `cons` returns a new cell with its unique identity.

Inventing a simple object-oriented programming language

These notes are more extensive than usual because this material is not covered in the textbook.

The beginning of this language is simply a set of conventions for using computational objects. (*Warning:* We will initially show you an implementation that has a bug and then show how to fix the bug below. The structure of this system follows that given by J. Rees and N. Adams, "Object-oriented programming in Scheme," in the *1988 ACM Conference on Lisp and Functional Programming.*)

An *object* is a procedure that, given a *message* as argument, returns another procedure called a method.

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

For example, the following procedure creates a simple object called a speaker whose only method is that it can *say* something (a list), which it does by printing the list:

```
(define (make-speaker)
  (define (self message)
    (cond ((eq? message 'say)
           (lambda (stuff) (display stuff)))
          (else (no-method "SPEAKER"))))
  self)
```

Notice that the object refers to itself as `self`. We'll make the convention that if an object does not recognize the message it will signal this by using `no-method`, which returns something that our object-oriented programming system will recognize.

```
(define (no-method name)
  (list 'no-method name))

(define (no-method? x)
  (if (pair? x)
      (eq? (car x) 'no-method)
      false))

(define (method? x)
  (not (no-method? x)))
```

To ask an object to apply a method to some arguments, we send the object a message asking for the appropriate method, and apply the method to the arguments. If the object has no method for this message, `ask` will produce an error. (The procedure `apply` takes a procedure and a list of arguments and applies the procedure to the arguments. Note the use of dot notation, so that `args` will be a list of the arguments.)

```
(define (ask object message . args)
  (let ((method (get-method object message)))
    (if (method? method)
        (apply method args)
        (error "No method" message (cadr method)))))
```

Now we can make a speaker object and ask it to say something:

```
(define george (make-speaker))
(ask george 'say '(the sky is blue))
(THE SKY IS BLUE)
```

One thing we may want to do is define an object type to be *a kind of* some other object type. For instance, we could say that a lecturer is a kind of speaker, expect that the lecturer also has a method called lecture. To lecture something, the lecturer says it and then says "You should be taking notes":

```
(define (make-lecturer)
  (let ((speaker (make-speaker)))
    (define (self message)
      (cond ((eq? message 'lecture)
             (lambda (stuff)
               (ask self 'say stuff)
               (ask self 'say '(you should be taking notes))))
            (else (get-method speaker message))))
    self))
```

Observe that we accomplish this by giving the lecturer an internal speaker of its own. If the message is not `lecture`, then lecturer passes the message on to the internal speaker. Thus, a lecturer can do anything a speaker can (i.e., say things) , and also lecture. In the object-oriented programming jargon, one says that `lecturer` *inherits* the `say` method from `speaker`, or that `speaker` is a *superclass* of `lecturer`.

```
(define Gerry (make-lecturer))
(ask Gerry 'say '(the sky is blue))
(THE SKY IS BLUE)

(ask Gerry 'lecture '(the sky is blue))
(THE SKY IS BLUE)
(YOU SHOULD BE TAKING NOTES)
```

Noticing a bug

The next example will reveal the bug in our simple implementation:

Let's define an `arrogant-lecturer` to be a kind of lecturer. But whenever an arrogant lecturer says anything, she or he prefaces it with "It is obvious that ..." and then says it as an ordinary lecturer would:

```
(define (make-arrogant-lecturer)
  (let ((lecturer (make-lecturer)))
    (define (self message)
      (cond ((eq? message 'say)
             (lambda (stuff)
               (ask lecturer 'say (append '(it is obvious that) stuff))))
            (else (get-method lecturer message))))
    self))

(define Albert (make-arrogant-lecturer))

(ask Albert 'say '(the sky is blue))
(IT IS OBVIOUS THAT THE SKY IS BLUE)
```

Albert here says things arrogantly, as he should. But he appears to have lost his arrogance when we ask him to lecture:

```
(ask Albert 'lecture '(the sky is blue))
(THE SKY IS BLUE)
(YOU SHOULD BE TAKING NOTES)
```

The bug here is that when Albert tells his internal `lecturer` to handle the `lecture` message, all information that this is being done as part of Albert is lost. We can see this from the following environment diagram. Note that Albert, his internal lecturer, and his internal lecturer's internal speaker, each have a `self`.

(The empty frames are from `make-arrogant-lecturer` and `make-lecturer` which have no parameters.) When Albert's lecturer returns its `lecture` method, namely,

```
(lambda (stuff)
  (ask self 'say stuff)
  (ask self 'say '(you should be taking notes)))
```

it is the lecturer's `self`, not the Albert `self` that is asked to `say` things.

Fixing the bug

We can fix this by changing the implementation so that all the methods keep track of self, by taking self as an extra input. For example, make-speaker now becomes:

```
(define (make-speaker)
  (define (self message)
    (cond ((eq? message 'say)
           (lambda (self stuff) (display stuff)))
          (else (no-method message))))
  self)
```

To make this implementation work, we change the definition of `ask` so that the object is also passed to the method (to serve as the `self` argument):

```
(define (ask object message . args)
  (let ((method (get-method object message)))
    (if (method? method)
        (apply method (cons object args))
        (error "No method" message (cadr method)))))
```

Here are the new corresponding definitions of `make-lecturer` and `make-arrogant-lecturer`. Other than the extra `self` arguments to the methods, everything works as before, except this time the bug has been fixed:

```
(define (make-lecturer)
  (let ((speaker (make-speaker)))
    (define (self message)
      (cond ((eq? message 'lecture)
             (lambda (self stuff)
               (ask self 'say stuff)
               (ask self 'say '(you should be taking notes))))
            (else (get-method speaker message))))
    self))

(define (make-arrogant-lecturer)
  (let ((lecturer (make-lecturer)))
    (define (self message)
      (cond ((eq? message 'say)
             (lambda (self stuff)
               (ask lecturer 'say (append '(it is obvious that) stuff))))
            (else (get-method lecturer message))))
    self))

(define Albert (make-arrogant-lecturer))

(ask Albert 'say '(the sky is blue))
(IT IS OBVIOUS THAT THE SKY IS BLUE)

(ask Albert 'lecture '(the sky is blue))
(IT IS OBVIOUS THAT THE SKY IS BLUE)
(IT IS OBVIOUS THAT YOU SHOULD BE TAKING NOTES)
```

Multiple superclasses

We can have object types that inherit methods from more than one type. Here, for example, is a `singer`, that can sing and also say things:

```
(define (make-singer)
  (lambda (message)
    (cond ((eq? message 'say)
           (lambda (self stuff)
             (display (append '(tra-la-la --) stuff))))
          ((eq? message 'sing)
           (lambda (self)
             (display '(tra-la-la))))
          (else (no-method "SINGER")))))

(define anna (make-singer))

(ask anna 'sing)
(TRA-LA-LA)

(ask anna 'say '(the sky is blue))
(TRA-LA-LA -- THE SKY IS BLUE)
```

We'll make Ben be both a singer and a lecturer:

```
(define ben
  (let ((singer (make-singer))
        (lecturer (make-lecturer)))
    (lambda (message)
      (let ((sing (get-method singer message))
            (lect (get-method lecturer message)))
        (if (method? sing)
            sing
            lect)))))

(ask ben 'sing)
(TRA-LA-LA)

(ask ben 'lecture '(the sky is blue))
(TRA-LA-LA -- THE SKY IS BLUE)
(TRA-LA-LA -- YOU SHOULD BE TAKING NOTES)
```

Ben is both a singer and a lecturer, but primarily a singer. It is only if his internal singer has no method that he passes the message on to his internal lecturer. We could also have done it the other way round. Alyssa here is also both a singer and a lecturer, but primarily a lecturer:

```
(define alyssa
  (let ((singer (make-singer))
        (lecturer (make-lecturer)))
    (lambda (message)
      (let ((sing (get-method singer message))
            (lect (get-method lecturer message)))
        (if (method? lect)
            lect
            sing)))))

(ask alyssa 'sing)
(TRA-LA-LA)

(ask alyssa 'lecture '(the sky is blue))
(THE SKY IS BLUE)
(YOU SHOULD BE TAKING NOTES)
```

Much of the complexity of contemporary object-oriented programming languages has to do with specifying ways to control the order of inheritance in situations like this.