CS 565 August 26, 1996
Jens Palsberg Readings: EOPL Chap. 7, OOTS Chap. 1-2.

Lecture Note 7:

Object-Oriented Programming

A key goal of object-oriented programming is to obtain software which is easy to

- Maintain
- Reuse

To ease maintenance, programs are organized as a collection of classes from which objects can be instantiated. Each object has an incapsulated state, and this state can only be accessed by invoking methods of the object. Thus, the representation of an object can be changed without affecting the behavior of other objects.

To ease reuse, a class can be derived from another class using inheritance.

An Object-Oriented Language

Let's build a small object-oriented language. An `object` will be a data structure that consists of a *class* (a symbol) and a rib of *instance variables.*

An object responds to *messages* (symbols); when a message is sent to an object, it consults its class to find out out what `method` to use to respond to the message. For example, here's a program in our little language:

```
class c1(i)
  var x
  method get() x;
  method foo() 2*x;
  init { x = i };
 class c2(i)
  var y
  method get() y;
  method foo() 3*y;
  init { y = i };
 var temp, o
 {read temp
  if temp = 1 {o new c1(1)}{o new c2(1)};
  print o<-get()
  print o<-foo()}")
```

Here we define two classes. Objects of each class have one instance variable, and respond to the `get` and `foo` messages. Here we build an object and assign it to `o`. The class of the object depends on the input. How `o` responds to the `foo` message is not determined until run time.

This is done in `classes.scm`. We won't go through all the changes here, but I'll give you a survey. We'll talk about design issues first, then implementation issues.

Design decisions

1. We will have only global classes. All classes will be declared at the beginning of the program, before the the `var`'s and `proc`'s. So we changed the grammar to say:

```
(pgm
  ((arbno class-decl optional-semicolon) block)
  pgm)
(class-decl
  (class identifier lparen identifier-list rparen
    var-decl
    (arbno method-decl optional-semicolon)
    init          ; a keyword before the initialization
    block)        ; the initializer
  class-decl)
(method-decl
  (method identifier lparen identifier-list
   rparen expression)
  method-decl)
```

The initializer gets to see the rib of instance variables and a rib of parameters. Hence we can write:

```
class c1(i)
  var x
  method get() x;
  method foo() 2*x;
  init { x = i };
...
  o new c1(33)
```

Note that here we've made object creation a new kind of statement; we might have made it an expression instead. [This may have been a misfeature].

For object creation, we specify the syntax by

```
    (primitive-tail
      (new identifier operand-list)
      new-statement)
```

2. Calls on objects will be expressions, returning values. I originally had them be side-effecting procedures, but that way it was too difficult to get information back from the objects. So we added:

```
(factor
  (identifier identifier-tail)
  identifier-exp)
(identifier-tail
  ()
  empty-identifier-tail)
(identifier-tail
  (message-send-sym identifier operand-list)
  method-call)
```

However, the bodies of methods often want to perform side-effects on the object. This requires a way of invoking a statement from inside an expression. Conditional expressions also turn out to be useful in this context. So we also added:

```
(expression
  (perform compound-statement returning expression)
  perform-expression)
(expression
  (if expression then expression else expression)
  if-expression)
```

3. We follow the standard practice in OOLs and say that methods in a class (unlike procedures in a block) are *not* mutually recursive. Instead, when a method is invoked, we bind the identifier `self` to the *entire* object. That way a method body can send messages to the object and thereby invoke the other methods. For example:

```
class c(i)
  var x
  method get() x;
  method set(y) perform {x = y} returning x;
  method even() if x = 0 then 1 else perform {x = x - 1}
                returning self<-odd();
  method odd () if x = 0 then 0 else perform {x = x - 1}
                returning self<-even();

  init {x = i};
 var o, temp
 {read temp;
  print temp;
  o new c (temp);
  print o<-get();
  print o<-even();
  print o<-get()}")
```

4. The `message-send-sym` lexical class is represented by the characters `<-`. This required another line in the scanner:

```
(lt-state
  (#\- message-send-sym)              ; this is a <- .
  (#\= le-sym)
  (else lt-sym))
(gt-state
  (#\= ge-sym)
  (else gt-sym))
```

Implementation

We have two things to do: first, we need to go through and update all the `eval-` procedures to match the changes to the grammar. We'll just show the pieces corresponding to the changes we've talked about. This is pretty easy, but somewhat tedious.

Second, we actually have to write the code for dealing with objects. This takes a little more effort.

To implement classes, `eval-prog` will build a table of classes, called `class-env`, that will be passed as an additional parameter along with `env` everywhere.

```
;    (pgm
;        ((arbno class-decl optional-semicolon) block)
;        pgm)

(define eval-pgm
  (lambda (pgm)              ; no initial environment
    (record-case pgm
      (pgm (class-decl* block)
           (let ((env (empty-env))
                 (class-env (eval-class-decl* class-decl*)))
             (eval-block block env class-env)))
      (else (error 'eval-pgm "unknown pgm ~s" pgm)))))
```

```
;;; records for putting in class-env:

(define-record class-decl
  (name init-formals instance-var-decl methods initializer))

(define-record method-decl (name formals body))
```

The class environment will be a list of `class-decl`'s just like we get from sllgen, but we'd better strip out all the optional punctuation. I got tired of doing this and finally wrote a program to do it once and for all:

```
(define strip-optionals
  (lambda (alt-list)
    (if (null? alt-list) '()
      (cons (car alt-list)
            (strip-optionals (cddr alt-list))))))
```

Similarly, the `methods` should be a list of `method-decl`'s like the one we got from sllgen, but with the various optional punctuation removed.

Now we can write:

```
;     (class-decl
;       (class identifier lparen identifier-list rparen
;         var-decl
;         (arbno method-decl optional-semicolon)
;         init    ; a keyword before the initialization
;         block)
;       class-decl)

(define eval-class-decl*
  (lambda (class-decl*)
    ;; class-decl* is an alternating list
    (map eval-class-decl (strip-optionals class-decl*))))

(define eval-class-decl
  (lambda (class-decl)
    (record-case class-decl
      (class-decl (name init-formals instance-var-decl
                   methods initializer)
        (make-class-decl
          name
          (eval-identifier-list init-formals)
          instance-var-decl
          (map eval-method-decl (strip-optionals methods))
          initializer))
      (else (raise-error 'eval-class-decl class-decl)))))
```

```
;     (method-decl
;       (method identifier lparen identifier-list
;        rparen expression)
;       method-decl)

(define eval-method-decl
  (lambda (method-decl)
    (record-case method-decl
      (method-decl (name formals body)
        (make-method-decl
          name
          (eval-identifier-list formals)
          body))
      (else (raise-error 'eval-method-decl method-decl)))))
```

Object Creation

```
;     (primitive-tail
;       (new identifier operand-list)
;       new-statement)

(define eval-primitive-tail
  (lambda (identifier primitive-tail env class-env)
    (record-case primitive-tail
      (new-statement
        (class-identifier operand-list)
        (let ((cell (apply-env env identifier))
              (args (eval-operand-list operand-list
                     env class-env)))
          (set-cell!
            cell
            (make-new-object class-identifier args
                             class-env))))
      ...)))
```

We find the cell, evaluate the operands, call `make-new-object` to actually build the object, and then assign the object to the cell.

Now we've got to actually build an object. The object will consist of a class name and a rib of instance variables.

We have to look up the class declaration in the class environment to figure out just what to do. This tells us the formals corresponding to the creation arguments (`args`), the names of the instance variables (needed for the rib), and the initialization block. We'll worry about `apply-class-env` later.

The initialization block gets to see exactly two ribs: the rib of creation parameters and the rib of instance variables.

```
(define-record object (class-name instance-var-rib))

(define make-new-object
  (lambda (class-name args class-env)
    ;; first retrieve the class-decl from the class-env and pull out
    ;; the pieces.
    (let ((class-decl (apply-class-env class-env class-name)))
      (record-case class-decl
        (class-decl (name init-formals instance-var-decl methods initializer)
          ;; next make a rib for the instance variables
          (let ((instance-rib (eval-var-decl instance-var-decl))
                (init-rib
                  (make-var-rib
                    init-formals (map make-cell args))))
            ;; next run the initializer.
            (eval-block
              initializer
              (extend-env instance-rib (extend-env init-rib (empty-env)))
              class-env)
            ;; now return the object:
            (make-object class-name instance-rib)))
        (else (raise-error 'make-new-object class-decl))))))
```

Sending a Message to an Object

To get message-sends (method invocations) to return values, we had to hack the grammar for expressions:

```
;      (factor
;        (identifier identifier-tail)
;        identifier-exp)
```

```
(define eval-factor
  (lambda (factor env class-env)
    (record-case factor
      (identifier-exp (ident ident-tail)
        (eval-ident-tail
          (deref (apply-env env ident))
          ident-tail env class-env)
        ...))))
```

The identifier gets evaluated and dereferenced in either case, so we just do it now, and pass the result, along with the rest of the data to `eval-ident-tail`.

`eval-identifier-tail` gets to figure out whether this is just a variable reference or method-call.

```
;      (identifier-tail
;        ()
;        empty-identifier-tail)
;      (identifier-tail
;        (message-send-sym identifier operand-list)
;        method-call)

(define eval-ident-tail
  (lambda (val-of-ident ident-tail env class-env)
    (record-case ident-tail
      (empty-identifier-tail ()
        ;; ah, this was really just an identifier
        val-of-ident)
      (method-call (message operand-list)
        ;; this is a message send.
        ;; Evaluate the operands and send
        ;; everything off to the object.
        (let ((args (eval-operand-list operand-list
                      env class-env))
              (object val-of-ident))
          (send-message object message args class-env)))
      (else (raise-error 'eval-ident-tail ident-tail)))))
```

To send the message, we retrieve the class name from the object. We then use `lookup-method` to find the correct method to use for that class and that message. Methods are not scoped. They always run with a 2-rib environment: the first rib contains the parameters to this call, and the second is the rib of instance variables from the object.

Also, we bind the identifier `self` to a cell containing `object`. This could be a separate rib, but we just put it in as part of the first rib.

```
(define send-message
  (lambda (object message args class-env)
    (let ((class-name (object->class-name object)))
      (let ((method-decl (lookup-method class-name message
                           class-env)))
        (record-case method-decl
          (method-decl (name formals body)
            ;; methods are NOT recursive.  They have to use self.
            ;; Also, methods are not scoped.  They run with a 2-rib
            ;; environment, always:
            (eval-expression body
                        (extend-env
                          (make-var-rib
                            (cons 'self formals)
                            (map make-cell
                                 (cons object args)))
                          (extend-env
                            (object->instance-var-rib object)
                            (empty-env)))
                        class-env))
          (else (raise-error 'send-message method-decl)))))))
```

Now we get to `class-env`. This is just a list of class declarations, preprocessed by `eval-class-decl` to remove the assorted optional punctuation. So we can look things up by a simple linear search:

```
(define apply-class-env
  (lambda (class-env class-name)
    (cond
      ((null? class-env)
       (error 'apply-class-env "undefined class ~s" class-name))
      ((eq? class-name (class-decl->name (car class-env)))
       (car class-env))
      (else (apply-class-env (cdr class-env) class-name)))))
```

Similarly, the `methods` field of a `class-decl` is just a list of method declarations (minus the punctuation). So we just look up the method by linear search again:

```
(define lookup-method
  (lambda (class-name method-name class-env)
    (letrec
      ((loop (lambda (method-decls)
               (cond
                 ((null? method-decls)
                  (error 'lookup-method
                    "unknown message ~s for class ~s"
                    method-name class-name))
                 ((eq? method-name
                       (method-decl->name (car method-decls)))
                  (car method-decls))
                 (else (loop (cdr method-decls)))))))
      (loop (class-decl->methods
             (apply-class-env class-env class-name))))))
```

Test Runs

```
;; creating an object and retrieving the value via a method:

(define pgm5
"class c1()
  var x
  method get() x;
  init {x = 33} ;
  var o,a
  {o new c1() ;
   a = o <- get ();
   print a }")

; (run pgm5)
; 33
; #t
```

```
;; creating an object and side-effecting it through a method:

(define pgm6
"class c1()
  var x
  method get() x;
  method set(y) perform {x = y} returning x;
  init {x = 33} ;
 var o,a
 {o new c1() ;
  a = o <- set (34)
  a = o <- get ();
  print a }")

; > (run pgm6)
; 34
; #t
```

```
;; getting recursion via SELF:

(define pgm7
"class c(i)
  var x
  method get() x;
  method set(y) perform {x = y} returning x;
  method even() if x = 0 then 1 else perform {x = x - 1}
                returning self<-odd();
  method odd () if x = 0 then 0 else perform {x = x - 1}
                returning self<-even();

  init {x = i};
 var o, temp
 {read temp;
  print temp;
  o new c (temp);
  print o<-get();
  print o<-even();
  print o<-get()}")

; > (run pgm7)
; input> 3
; 3
; 3
; 0
; 0
```

```
; #t
; > (run pgm7)
; input> 8
; 8
; 8
; 1
; 0
; #t
```

```
(define pgm8   ;; dynamic method dispatch
"class c1(i)
  var x
  method get() x;
  method foo() 2*x;
  init { x = i };
 class c2(i)
  var y
  method get() y;
  method foo() 3*y;
  init { y = i };
 var temp, o
 {read temp
  if temp = 1 {o new c1(1)}{o new c2(1)};
  print o<-get()
  print o<-foo()}")

; > (run pgm8)
; input> 1
; 1
; 2
; #t
; > (run pgm8)
; input> 2
; 1
; 3
; #t
```