By Joshua Cantrell
jjc@cory.berkeley.edu

Berkeley Scheme's OOP

Introduction to Mutation

If we want to directly change the value of a variable, we need a new special form, *set!* (pronounced "set BANG!").

```
(set! <variable> <new-value>)
(set-car! <pair> <new-value>)
(set-cdr! <pair> <new-value>)
```

To make the discussion more complete, notice that you can't use *set!* to change the elements inside a pair, but there are separate procedures (not special forms), *set-car!* and *set-cdr!*, to do that for you (pairs are a different data structure). These all return some unspecified value, but they change the current *state*[1] of the given variable or pair.

Evaluating Sequences of Expressions

Sometimes we have a list of expressions that need to be evaluted in a special order. Some expressions, like *cond*, *lambda*, or procedure *define*, automatically allow you to put multiple expressions in order (such as after the predicates in a cond and in the body of a procedure). Others, like *if*, are incapable of allowing multiple expressions to be listed in sequence. For these cases, you must use a Scheme special form, *begin*:

```
> (begin (display "I will square 5.")
         (square 25)
         (square 5))
I will square 5.
25
```

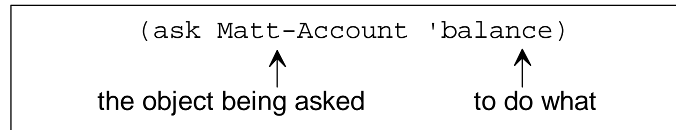
Notice that the result of evaluating the *begin* special form is the value returned from evaluating the last expression in the list of expressions.

[1] State has to do with the way a value looks. If you can mutate a value, then its form changes and we say its state changes.

By Joshua Cantrell
jjc@cory.berkeley.edu

Interactive Abstraction of Berkeley Scheme's OOP

The abstraction barrier is created in OOP by the user interfacing with the objects through a type of message passing. The programmer actually *asks* the object to do something and may provide needed arguments as well. One example taken from the reader is:



You can even use actual parameters to elaborate on what your are asking. The true form of ask looks like this:

(**ask** *<object> <message> <param$_1$> ... <param$_n$>*)

where you can have zero or more actual parameters. *Message* must be a symbol, so it will always be quoted.

Classes & Methods

Now we'd like to find out how to define objects and classes (the object oriented design is an example of how expandable Scheme is since it can be written using normal Scheme expressions). To define a class you have the following form:

(**define-class** (*<class-name> <obj-param$_1$>* ... *<obj-param$_n$>*) *<class-components>*)

The class is bound to the given symbol, *class-name*. The optional parameters are the *instantiation variables*, those variables that need specified values at the creation of an object. The body of the class, *class-components*, is a list of class special forms that define internal state and procedures.

The most important class component is the method. Methods are simply procedures defined inside the class. In order to use the methods, you must use the *ask* procedure shown above. Each method is referred to by its method name which is the *message*. Each method can also have corresponding formal parameters. To define a method within a class you have the following form:

(**method** (*<method-name> <param$_1$>* ... *<param$_n$>*) *<body>*)

To create an object, the *instantiate* procedure must be invoked given the class and the actual parameters needed to fill the instantiation variables. The *instantiate* procedure has the following form:

(**instantiate** *<class> <obj-param$_1$>* ... *<obj-param$_n$>*)

The following examples taken from the reader show how to form expressions using the previously covered forms:

```
(define-class (account balance)
  (method (deposit amount)
    (set! balance (+ amount balance))
    balance)
  (method (withdraw amount)
    (if (< balance amount)
        "Insufficient funds"
        (begin
          (set! balance (- balance amount))
          balance))) )

(define Matt-Account (instantiate account 1000))
```

Internal State

There are three types of variables that can be defined inside a class, *instantiation variables*, *instance variables*, and *class variables*. Each one has a special purpose in a class.

Instantiation Variables: These are what you define when you first make an object.

```
(define-class (<class-name> <var-name1> ... <var-namen>) ...)
```

Instance Variables: These are bound with the value resulting from the evaluation of the corresponding expression when the object is created. This is a class component.

```
(instance-vars (<var-name1> <exp1>) ... (<var-namen> <expn>))
```

Class Variables: These are bound with the value resulting from the evaluation of the corresponding expression when the class is created. This is a class component.

```
(class-vars (<var-name1> <exp1>) ... (<var-namen> <expn>))
```

Variables can only be accessed directly within their own class definition[2]. For each variable in an object, there is a method created with the same name which returns the variable's value. Class variables can be accessed from both the class and the object because they are shared by both. By using *ask*, class variables can be returned. This has the following form:

```
(ask <class or object> <var-name>)
```

Remember that *var-name* must be a quoted symbol since *ask* is a procedure and not a special form.

---

[2] In the case of inheritance discussed later, a parent class is not considered a part of the child class, so the inherited variables are not directly accessable.

By Joshua Cantrell
jjc@cory.berkeley.edu

The *Self* Object

Each object needs a way to access itself to use its own methods. While its variables can be accessed by evaluating Scheme symbols in the normal fashion, methods are not bound to symbols in the same way and must be *ask*ed to perform an operation[3]. To facilitate this need, a local instance variable, *self*, is created in each object and is bound to each corresponding object. Within an object, the *self* variable can be used as shown:

```
(ask self <message> <param1> ... <paramn>)
```

Inheritance

An important feature of our OOP paradigm is inheritance. We can reuse class structures as the basis for new, more specific classes. The new class shares the methods of the base classes[4], and can be expanded on to have additional state variables and procedures. This is done by using the *parent* class component. The *parent* special form can take as many classes as you want the new class to inherit from. Any conflicting names of methods will be resolved by using the value from the parent closest to the front of the list. This has the following form:

```
(parent (<class-name1> <param1> ... <paramk>)
                     ...
        (<class-namen> <param1> ... <param1>))
```

In order to access variables related to a parent class, the method must be used because the child only shares method definitions, not variable definitions. The inheritance example from the reader is:

```
(define-class (checking-account init-balance)
  (parent (account init-balance))
  (method (write-check amount)
    (ask self 'withdraw (+ amount 0.10)) ))
```

---

[3] You need *ask* to use methods because they are not bound to symbols in Scheme's environment like local variables. Instead, they are stored in a separate data structure, and must be looked up there.

[4] This is an important distinction from other OOP languages. In other languages, variables are also shared by the child class. This implementation of OOP only shares methods.

By Joshua Cantrell
jjc@cory.berkeley.edu

Asking The Usual Method

There are times when you have a child class and you need to make some of its methods do something slightly different than its parent's methods. If you want the child class's interface to match the parent's, then you'll need to use the same method names as those used in the parent class. Now if you also need to call the parent's methods instead of the newly redefined methods, you're in trouble if you only have *ask* because it checks for the child class's methods before checking for the parent class's methods.

The *usual* method call is just like using *ask* with *self* except it acts as though the method name provided could not be found in the child class, so it automatically starts checking the parent classes. The example given in the reader is:

```
(define-class (TA)
  (parent (worker))
  (method (work)
    (usual 'work)
    '(Let me help you with that box and pointer diagram))
  (method (grade-exam) 'A+) )
```

Notice how the *work* method was defined in the *TA* class, yet within the class we needed to access the parent's *work* method. In this case, if *usual* hadn't been used, an infinite loop would have been formed.

Initialize Procedure

The class's *initialize* procedure is always called at the creation of an object. It can be used to perform operations that need to be done whenever an object is created. A common use of the *initialize* procedure is to make a list of all the objects of a given class that are in existence (usually stored in a class variable). The example given in the reader is:

```
(define-class (worker)
  (instance-vars (hunger 0))
  (class-vars (all-workers '())
              (work-done 0))
  (initialize (set! all-workers (cons self all-workers)))
  (method (work)
    (set! hunger (1+ hunger))
    (set! work-done (1+ work-done))
    'whistle-while-you-work))
```

By Joshua Cantrell
jjc@cory.berkeley.edu

The Default Method

The *default-method* in a class defines a method that is called when the user *ask*s the object to perform an undefined method. There are two implicit formal parameters to this method, *message* and *args*. The *message* symbol is bound to the method name that wasn't found. The *args* symbol is bound to those actual parameters (values) that follow after the method name. The example given in the reader is:

```
(define-class (echo-previous)
  (instance-vars (previous-message 'first-time))
  (default-method
    (let ((result previous-message))
      (set! previous-message message)
      result)))
```

The downfall of *default-method* is that it is only called when all other cases are exhausted, so if you have a parent class, the class's default-method is ignored and skipped over when the evaluator decides to look for the missing methods inside the parent class(es). The side-effect is that any default-method in a child class is never used.