Reference Manual for the OOP Language

There are only three procedures that you need to use: `define-class`, which defines a class; `instantiate`, which takes a class as its argument and returns an instance of the class; and `ask`, which asks an object to do something. Here are the explanations of the procedures:

ASK: (ask *object message* . *args*)

`Ask` gets a method from *object* corresponding to *message*. If the object has such a method, invoke it with the given *args*; otherwise it's an error.

INSTANTIATE: (instantiate *class* . *arguments*)

`Instantiate` creates a new instance of the given *class*, initializes it, and returns it. To initialize a class, `instantiate` runs the `initialize` clauses of all the parent classes of the object and then runs the `initialize` clause of this class.

The extra arguments to `instantiate` give the values of the new object's instantiation variables. So if you say

```
(define-class (account balance) ...)
```

then saying

```
(define my-acct (instantiate account 100))
```

will cause `my-acct`'s `balance` variable to be bound to 100.

DEFINE-CLASS:

(define-class (*class-name args*...) *clauses*...)

This defines a new class named *class-name*. The instantiation arguments for this class are *args*. (See the explanation of `instantiate` above.)

The rest of the arguments to `define-class` are various *clauses* of the following types. All clauses are optional. You can have any number of `method` clauses, in any order.

(METHOD (*message arguments*...) *body*)

A `method` clause gives the class a method corresponding to the *message*, with the given *arguments* and *body*. A class definition may contain any number of `method` clauses. You invoke methods with `ask`. For example, say there's an object with a

```
(method (add x y) (+ x y))
```

clause. Then `(ask object 'add 2 5)` returns 7.

Inside a method, the variable `self` is bound to the object whose method this is. (Note that `self` might be an instance of a child class of the class in which the method is defined.) A method defined within a particular class has access to the instantiation variables, instance variables, and class variables that are defined within the same class, but does *not* have access to variables defined in parent or child classes. (This is similar to the scope rules for variables within procedures outside of the OOP system.)

Any method that is usable within a given object can invoke any other such method by invoking `(ask self` *message*`)`. However, if a method wants to invoke the method of the same name within a parent class, it must instead ask for that explicitly by saying

`(usual` *message* *args...*`)`

where *message* is the name of the method you want and *args...* are the arguments to the method.

(INSTANCE-VARS (*var1 value1*) (*var2 value2*) ...)

`Instance-vars` sets up local state variables `var1`, `var2`, etc. Each instance of the class will have its own private set of variables with these names. These are visible inside the bodies of the methods and the initialization code within the same class definition. The initial values of the variables are calculated when an instance is created by evaluating the expressions `value1`, `value2`, etc. There can be any number of variables. Also, a method is automatically created for each variable that returns its value. If there is no `instance-vars` clause then the instances of this class won't have any instance variables. It is an error for a class definition to contain more than one `instance-vars` clause.

(CLASS-VARS (*var1 value1*) (*var2 value2*) ...)

`Class-vars` sets up local state variables `var1`, `var2`, etc. The class has only one set of variables with these names, shared by every instance of the class. (Compare the `instance-vars` clause described above.) These variables are visible inside the bodies of the methods and the initialization code within the same class definition. The initial values of the variables are calculated when the class is defined by evaluating the expressions `value1`, `value2`, etc. There can be any number of variables. Also, a method is automatically created for each variable that returns its value. If there is no `class-vars` clause then the class won't have any class variables. It is an error for a class definition to contain more than one `class-vars` clause.

(PARENT (*parent1 args...*) (*parent2 args...*))

`Parent` defines the parents of a class. The *args* are the arguments used to instantiate the parent objects. For example, let's say that the `rectangle` class has two arguments: `height` and `width`:

```
(define-class (rectangle height width) ...)
```

A `square` is a kind of `rectangle`; the `height` and `width` of the `square`'s `rectangle` are both the `side-length` of the `square`:

```
(define-class (square side-length)
  (parent (rectangle side-length side-length))
  ...)
```

When an object class doesn't have an explicit method for a message it receives, it looks for methods of that name (or default methods, as explained below) in the definitions of the parent classes, in the order they appear in the `parent` clause. The method that gets invoked is from the first parent class that recognizes the message.

A method can invoke a parent's method of the same name with `usual`; see the notes on the `method` clause above.

(DEFAULT-METHOD *body*)

A `default-method` clause specifies the code that an object should execute if it receives an unrecognized message (i.e., a message that does not name a method in this class or any of its superclasses). When the body is executed, the variable `message` is bound to the message, and the variable `args` is bound to a list of the additional arguments to `ask`.

(INITIALIZE *body*)

The body of the `initialize` clause contains code that is executed whenever an instance of this class is created.

If the class has parents, their `initialize` code gets executed before the `initialize` clause in the class itself. If the class has two or more parents, their `initialize` code is executed in the order that they appear in the `parent` clause.