17

Example: Objects

In this chapter we're going to implement our own object-oriented language within Lisp. Such a program is called an *embedded language*. Embedding an object-oriented language in Lisp makes an ideal example. As well as being a characteristic use of Lisp, it shows how naturally the abstractions of object-oriented programming map onto the fundamental abstractions of Lisp.

17.1 Inheritance

Section 11.10 explained how generic functions differ from message-passing. In the message-passing model,

1. objects have properties,
2. and respond to messages,
3. and inherit properties and methods from their parents.

CLOS, of course, uses the generic function model. But in this chapter we are interested in writing a minimal object system, not a rival to CLOS, so we will use the older model.

In Lisp, there are already several ways to store collections of properties. One way would be to represent objects as hash tables, and store their properties as entries within them. We then have access to individual properties through `gethash`:

```
(gethash 'color obj)
```

```
(defun rget (prop obj)
  (multiple-value-bind (val in) (gethash prop obj)
    (if in
        (values val in)
        (let ((par (gethash :parent obj)))
          (and par (rget prop par))))))

(defun tell (obj message &rest args)
  (apply (rget message obj) obj args))
```

Figure 17.1: Inheritance.

Since functions are data objects, we can store them as properties too. This means that we can also have methods; to invoke a given method of an object is to funcall the property of that name:

```
(funcall (gethash 'move obj) obj 10)
```

We can define a Smalltalk style message-passing syntax upon this idea,

```
(defun tell (obj message &rest args)
  (apply (gethash message obj) obj args))
```

so that to tell `obj` to move 10, we can say

```
(tell obj 'move 10)
```

In fact, the only ingredient plain Lisp lacks is inheritance. We can implement a simple version of that by defining a recursive version of `gethash`, as in Figure 17.1. (The name `rget` stands for "recursive get.") Now with a total of eight lines of code we have all three of the minimal elements of object-oriented programming.

Let's try out this code on our original example. We create two objects, one a child of the other:

```
> (setf circle-class                  (make-hash-table)
        our-circle                    (make-hash-table)
        (gethash :parent our-circle) circle-class
        (gethash 'radius our-circle) 2)
2
```

The object `circle-class` will hold the area method for all circles. It will be a function of one argument, the object to which the message is originally sent:

```
> (setf (gethash 'area circle-class)
        #'(lambda (x)
            (* pi (expt (rget 'radius x) 2))))
#<Interpreted-Function BF1EF6>
```

Now we can ask for the area of our-circle, and its value will be calculated according to the method defined for the class. We use rget to read a property, and tell to invoke a method:

```
> (rget 'radius our-circle)
2
T
> (tell our-circle 'area)
12.566370614359173
```

Before going on to improve this program, it's worth pausing to consider what we have done. With eight lines of code we have made plain old pre-CLOS Lisp into an object-oriented language. How did we manage to achieve such a feat? There must be some sort of trick involved, to implement object-oriented programming in eight lines of code.

There is a trick, but it is not a programming trick. The trick is, Lisp already was an object-oriented language, or rather, something more general. All we had to do was put a new facade on the abstractions that were already there.

17.2 Multiple Inheritance

So far we have only single inheritance—an object can only have one parent. But we can have multiple inheritance by making the parent property a list, and defining rget as in Figure 17.2.

With single inheritance, when we wanted to retrieve some property of an object, we just searched recursively up its ancestors. If the object itself had no information about the property we wanted, we looked at its parent, and so on. With multiple inheritance we want to perform the same kind of search, but our job is complicated by the fact that an object's ancestors can form a graph instead of a simple tree. We can't just search this graph depth-first. With multiple parents we can have the hierarchy shown in Figure 17.3: a is descended from b and c, which are both descended from d. A depth-first (or rather, height-first) traversal would go a, b, d, c, d. If the desired property were present in both d and c, we would get the value stored in d, not the one stored in c. This would violate the principle that subclasses override the default values provided by their parents.

If we want to implement the usual idea of inheritance, we should never examine an object before one of its descendants. In this case, the proper

```
(defun rget (prop obj)
  (dolist (c (precedence obj))
    (multiple-value-bind (val in) (gethash prop c)
      (if in (return (values val in))))))

(defun precedence (obj)
  (labels ((traverse (x)
             (cons x
                   (mapcan #'traverse
                           (gethash :parents x)))))
    (delete-duplicates (traverse obj))))
```

Figure 17.2: Multiple inheritance.

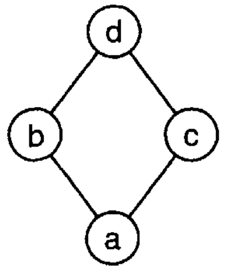

Figure 17.3: Multiple paths to a superclass.

search order would be `a`, `b`, `c`, `d`. How can we ensure that the search always tries descendants first? The simplest way is to assemble a list of an object and all its ancestors in the proper precedence order, then look at each one in turn.

The function `precedence` returns a list of an object and all its ancestors in the correct order. It begins by calling `traverse` to build a list representing the objects encountered in a depth-first traversal. If any of the objects share parents, there will be duplicates in this list. If we preserve only the last of each set of duplicates, we will get a precedence list in the natural order defined by CLOS. (Deleting all but the last duplicate corresponds to rule 3 in the algorithm described on page 183.) The Common Lisp function `delete-duplicates` is defined to behave this way, so if we just call it on the result of the depth-first traversal, we will get the correct precedence list. Once the precedence list is created, `rget` searches for the first object with the desired property.

By taking advantage of precedence we can say, for example, that a patriotic scoundrel is a scoundrel first and a patriot second:

```
> (setf scoundrel            (make-hash-table)
        patriot              (make-hash-table)
        patriotic-scoundrel (make-hash-table)
        (gethash 'serves scoundrel) 'self
        (gethash 'serves patriot)   'country
        (gethash :parents patriotic-scoundrel)
                 (list scoundrel patriot))
(#<Hash-Table C41C7E> #<Hash-Table C41F0E>)
> (rget 'serves patriotic-scoundrel)
SELF
T
```

At this point we have a program that's powerful, but ugly and inefficient. In the second stage of the life of a Lisp program, we refine this sketch into something usable.

17.3 Defining Objects

Among the first improvements we need is a function to create objects. The way our program represents objects and their parents need not be visible to the user. If we define a function to build objects, users will be able to make an object and specify its parents in one step. And we can build an object's precedence list at the time it is created, instead of expensively reconstructing it every time we need to find a property or a method.

If we are going to maintain precedence lists instead of constructing them as we need them, we have to deal with the possibility of the lists becoming outdated. Our strategy will be to keep a list of all existing objects, and whenever something's parents are modified, to remake the precedence list of every object affected. This is expensive, but since queries are likely to be much more common than the redefinition of objects' parents, we will get a net saving. Our program will not become any less flexible by this change; we just shift costs from a frequent operation to an infrequent one.

Figure 17.4 contains the new code.° The global `*objs*` will be a list of all current objects. The function `parents` retrieves an object's parents; its converse `(setf parents)` not only sets an object's parents, but calls `make-precedence` to rebuild any precedence list that might thereby have changed. The lists are built by `precedence`, as before.

Now instead of calling `make-hash-table` to make objects, users will call `obj`, which creates a new object and defines its parents in one step. We also redefine `rget` to take advantage of stored precedence lists.

```
(defvar *objs* nil)

(defun parents (obj) (gethash :parents obj))

(defun (setf parents) (val obj)
  (prog1 (setf (gethash :parents obj) val)
         (make-precedence obj)))

(defun make-precedence (obj)
  (setf (gethash :preclist obj) (precedence obj))
  (dolist (x *objs*)
    (if (member obj (gethash :preclist x))
        (setf (gethash :preclist x) (precedence x)))))

(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (push obj *objs*)
    (setf (parents obj) parents)
    obj))

(defun rget (prop obj)
  (dolist (c (gethash :preclist obj))
    (multiple-value-bind (val in) (gethash prop c)
      (if in (return (values val in))))))
```

Figure 17.4: Creating objects.

17.4 Functional Syntax

Another place for improvement is the syntax of message calls. The `tell` itself is unnecessary clutter, and because it makes verbs come third, it means that our programs can no longer be read like normal Lisp prefix expressions:

```
(tell (tell obj 'find-owner) 'find-owner)
```

We can get rid of the `tell` syntax by defining property names as functions, using the macro `defprop` in Figure 17.5. The optional argument `meth?`, if true, signals that this property should be treated as a method. Otherwise it will be treated as a slot, and the value retrieved by `rget` will simply be returned. Once we have defined the name of either kind of property,

```
(defprop find-owner t)
```

```
(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget ',name obj)))
     (defun (setf ,name) (val obj)
       (setf (gethash ',name obj) val))))

(defun run-methods (obj name args)
  (let ((meth (rget name obj)))
    (if meth
        (apply meth obj args)
        (error "No ~A method for ~A." name obj))))
```

Figure 17.5: Functional syntax.

we can refer to it with a function call, and our code will read like Lisp again:

```
(find-owner (find-owner obj))
```

Our previous example now becomes somewhat more readable:

```
> (progn
    (setf scoundrel           (obj)
          patriot             (obj)
          patriotic-scoundrel (obj scoundrel patriot))
    (defprop serves)
    (setf (serves scoundrel) 'self
          (serves patriot)   'country)
    (serves patriotic-scoundrel))
SELF
T
```

17.5 Defining Methods

So far we define a method by saying something like:

```
(defprop area t)

(setf circle-class (obj))
```

```
(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (setf (gethash ',name ,gobj)
             (labels ((next () (get-next ,gobj ',name)))
               #'(lambda ,parms ,@body))))))

(defun get-next (obj name)
  (some #'(lambda (x) (gethash name x))
        (cdr (gethash :preclist obj))))
```

Figure 17.6: Defining methods.

```
(setf (area circle-class)
      #'(lambda (c) (* pi (expt (radius c) 2))))
```

Within a method we can get the effect of the built-in `call-next-method` by calling the first thing we can find under the same name in the cdr of the object's `:preclist`. So, for example, if we want to define a special circle that prints something in the process of returning its area, we say:

```
(setf grumpy-circle (obj circle-class))

(setf (area grumpy-circle)
      #'(lambda (c)
          (format t "How dare you stereotype me!~%")
          (funcall (some #'(lambda (x) (gethash 'area x))
                         (cdr (gethash :preclist c)))
                   c)))
```

The `funcall` here is equivalent to a `call-next-method`, but it shows more internals than we want to look at.

The macro `defmeth` in Figure 17.6 provides a convenient way to define methods, and makes it easy to call the next method within them. A call to `defmeth` expands into a `setf`, but the `setf` occurs within a `labels` expression that defines `next` as a function to retrieve the next method. This function is like `next-method-p` (page 188), but returns something we can call, and so serves the purpose of `call-next-method` as well.° Now the preceding two methods could be defined:

```
(defmeth area circle-class (c)
  (* pi (expt (radius c) 2)))
```

```
(defmeth area grumpy-circle (c)
  (format t "How dare you stereotype me!~%")
  (funcall (next) c))
```

Incidentally, notice that the definition of `defmeth` takes advantage of symbol capture. The body of the method is inserted into a context where the function `next` is locally defined.

17.6 Instances

So far we have not distinguished between classes and instances. We have used a single term, *object*, to cover both. It is elegant and flexible to treat all objects the same, but grossly inefficient. In most object-oriented applications the inheritance graph will be bottom-heavy. In a simulation of traffic, for example, we might have less than ten objects representing classes of vehicles, and hundreds of objects representing particular vehicles. Since the latter will all share a few precedence lists, it is a waste of time to create them, and a waste of space to store them.

Figure 17.7 defines a macro `inst`, for making instances. Instances are like other objects (which now may as well be called classes), but have only one parent and do not maintain precedence lists. They are also not included in the list `*objs*`. In the preceding examples, we could have said:

```
(setf grumpy-circle (inst circle-class))
```

Since some objects will no longer have precedence lists, the functions `rget` and `get-next` are now redefined to look at the parents of such objects instead. This gain in efficiency has cost us nothing in flexibility. We can do everything with an instance that we can do with any other kind of object, including make instances of it and redefine its parents. In the latter case, `(setf parents)` will effectively convert the object to a "class."

17.7 New Implementation

None of the improvements we've made so far have been made at the expense of flexibility. In the latter stages of its development, a Lisp program can usually benefit from some sacrifice of flexibility, and this case is no exception. So far we have been representing all objects as hash tables. This gives us more flexibility than we need, at greater cost than we want. In this section we will rewrite our program to represent objects as simple vectors.

```
(defun inst (parent)
  (let ((obj (make-hash-table)))
    (setf (gethash :parents obj) parent)
    obj))

(defun rget (prop obj)
  (let ((prec (gethash :preclist obj)))
    (if prec
        (dolist (c prec)
          (multiple-value-bind (val in) (gethash prop c)
            (if in (return (values val in)))))
        (multiple-value-bind (val in) (gethash prop obj)
          (if in
              (values val in)
              (rget prop (gethash :parents obj)))))))

(defun get-next (obj name)
  (let ((prec (gethash :preclist obj)))
    (if prec
        (some #'(lambda (x) (gethash name x))
              (cdr prec))
        (get-next (gethash obj :parents) name))))
```

Figure 17.7: Defining instances.

This change will mean giving up the possibility of defining new properties on the fly. So far we can define a property of any object simply by referring to it. Now when a class is created, we will have to give a list of the new properties it has, and when instances are created, they will have exactly the properties they inherit.

In the previous implementation there was no real division between classes and instances. An instance was just a class that happened to have one parent. If we modified the parents of an instance, it would become a class. In the new implementation there will be a real division between classes and instances; it will no longer be possible to convert instances to classes.

The code in Figures 17.8–17.10 is a complete new implementation. Figure 17.8 defines the new operators for creating classes and instances. Classes and instances are represented as vectors. The first three elements of each will contain information used by the program itself, and the first three macros in Figure 17.8 are for referring to these elements:

```
(defmacro parents (v) `(svref ,v 0))
(defmacro layout (v) `(the simple-vector (svref ,v 1)))
(defmacro preclist (v) `(svref ,v 2))

(defmacro class (&optional parents &rest props)
  `(class-fn (list ,@parents) ',props))

(defun class-fn (parents props)
  (let* ((all (union (inherit-props parents) props))
         (obj (make-array (+ (length all) 3)
                          :initial-element :nil)))
    (setf (parents obj)  parents
          (layout obj)   (coerce all 'simple-vector)
          (preclist obj) (precedence obj))
    obj))

(defun inherit-props (classes)
  (delete-duplicates
    (mapcan #'(lambda (c)
                (nconc (coerce (layout c) 'list)
                       (inherit-props (parents c))))
            classes)))

(defun precedence (obj)
  (labels ((traverse (x)
             (cons x
                   (mapcan #'traverse (parents x)))))
    (delete-duplicates (traverse obj))))

(defun inst (parent)
  (let ((obj (copy-seq parent)))
    (setf (parents obj)  parent
          (preclist obj) nil)
    (fill obj :nil :start 3)
    obj))
```

Figure 17.8: Vector implementation: Creation.

1. The `parents` field takes the place of the `:parents` hash table entry in the old implementation. In a class it will contain a list of parent classes. In an instance it will contain a single parent class.

2. The `layout` field will contain a vector of property names, indicating the layout of the class or instance from the fourth element on.

3. The `preclist` field takes the place of the `:preclist` hash table entry in the old implementation. It will contain the precedence list of a class, or `nil` in an instance.

Because these operators are macros, they can all be used in the first argument to `setf` (Section 10.6).

The macro `class` is for creating classes. It takes an optional list of superclasses, followed by zero or more property names. It returns an object representing a class. The new class will have the union of its local properties (that is, property names) and those inherited from all its superclasses.

```
> (setf *print-array* nil
        geom-class    (class nil area)
        circle-class  (class (geom-class) radius))
#<Simple-Vector T 5 C6205E>
```

Here we create two classes: `geom-class` has no superclasses, and only one property, `area`; `circle-class` is a subclass of `geom-class`, and adds the property `radius`.[1] The layout of `circle-class`

```
> (coerce (layout circle-class) 'list)
(AREA RADIUS)
```

shows the names of the last two of its five fields.[2]

The `class` macro is just an interface to `class-fn`, which does the real work. It calls `inherit-props` to assemble a list of the properties of all the new object's parents, builds a vector of the right length, and sets the first three fields appropriately. (The `preclist` is built by `precedence`, which is essentially unchanged.) The remaining fields of the class are set to `:nil` to indicate that they are uninitialized. To examine the `area` property of `circle-class` we could say:

```
> (svref circle-class
         (+ (position 'area (layout circle-class)) 3))
:NIL
```

[1]When classes are displayed, `*print-array*` should be `nil`. The first element in the `preclist` of any class is the class itself, so trying to display the internal structure of a class would cause an infinite loop.

[2]The vector is coerced to a list simply to see what's in it. With `*print-array*` set to `nil`, the contents of a vector would not be shown.

```
(declaim (inline lookup (setf lookup)))

(defun rget (prop obj next?)
  (let ((prec (preclist obj)))
    (if prec
        (dolist (c (if next? (cdr prec) prec) :nil)
          (let ((val (lookup prop c)))
            (unless (eq val :nil) (return val))))
        (let ((val (lookup prop obj)))
          (if (eq val :nil)
              (rget prop (parents obj) nil)
              val)))))

(defun lookup (prop obj)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off (svref obj (+ off 3)) :nil)))

(defun (setf lookup) (val prop obj)
  (let ((off (position prop (layout obj) :test #'eq)))
    (if off
        (setf (svref obj (+ off 3)) val)
        (error "Can't set ~A of ~A." val obj))))
```

Figure 17.9: Vector implementation: Access.

Later we will define access functions that do this automatically.

Finally, the function `inst` is used for making instances. It does not have to be a macro, because it takes just one argument:

```
> (setf our-circle (inst circle-class))
#<Simple-Vector T 5 C6464E>
```

It's instructive to compare `inst` to `class-fn`, which does something similar. Because instances have only one parent, there is no need to determine what properties are inherited. The instance can just copy the layout of its parent class. Nor is there any need to build a precedence list, because instances don't have them. Building instances will thus be much faster than building classes—which is as it should be, because creating instances is more common than creating classes in most applications.

Now that we can build a hierarchy of classes and instances we need functions to read and write their properties. The first function in Figure 17.9 is the new definition of rget. It is similar in shape to the rget in Figure 17.7. The two branches of the conditional deal with classes and instances respectively.

1. If the object is a class, we traverse its precedence list until we find an object in which the value of the desired property is not :nil. If we don't find one we return :nil.
2. If the object is an instance, we look for the property locally, and make a recursive call to rget if it isn't there.

The new third argument to rget, next?, will be explained later. For now suffice it to say that if it is nil, rget will behave as usual.

The function lookup and its inverse play the role that gethash did in the old rget. They use an object's layout to retrieve or set a property with a given name. This query duplicates the one we made earlier:

```
> (lookup 'area circle-class)
:NIL
```

Since the setf of lookup is also defined, we could define an area method for circle-class by saying:

```
(setf (lookup 'area circle-class)
      #'(lambda (c)
          (* pi (expt (rget 'radius c nil) 2))))
```

In this program, as in the earlier version, there is no hard distinction between slots and methods. A "method" is just a field with a function in it. This will soon be hidden by a more convenient front-end.

Figure 17.10 contains the last of the new implementation. This code does not add any power to the program, but makes it easier to use. The macro defprop is essentially unchanged; now it just calls lookup instead of gethash. As before, it allows us to refer to properties in a functional syntax:

```
> (defprop radius)
(SETF RADIUS)
> (radius our-circle)
:NIL
> (setf (radius our-circle) 2)
2
```

If the optional second argument to defprop is true, it expands into a call to run-methods, which is also almost unchanged.

```
(declaim (inline run-methods))

(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget ',name obj nil)))
     (defun (setf ,name) (val obj)
       (setf (lookup ',name obj) val))))

(defun run-methods (obj name args)
  (let ((meth (rget name obj nil)))
    (if (not (eq meth :nil))
        (apply meth obj args)
        (error "No ~A method for ~A." name obj))))

(defmacro defmeth (name obj parms &rest body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (defprop ,name t)
       (setf (lookup ',name ,gobj)
             (labels ((next () (rget ,gobj ',name t)))
               #'(lambda ,parms ,@body))))))
```

Figure 17.10: Vector implementation: Interface macros.

Finally, the function `defmeth` provides a convenient way to define methods. There are three things new about this version: it does an implicit `defprop`, it calls `lookup` instead of `gethash`, and it calls `rget` instead of `get-next` (page 278) to get the next method. Now we see the reason for the additional argument to `rget`. It is so similar to `get-next` that we can implement both in one function by adding an extra argument. If this extra argument is true, `rget` takes the place of `get-next`.

Now we can achieve the same effect as the preceding method definition with something a lot cleaner:

```
(defmeth area circle-class (c)
  (* pi (expt (radius c) 2)))
```

Notice that instead of calling `rget` we can simply call `radius`, because we defined it as a function with `defprop`. Because of the implicit `defprop` done by `defmeth`, we can likewise call `area` to get the area of `our-circle`:

```
> (area our-circle)
12.566370614359173
```

17.8 Analysis

We now have an embedded language suitable for writing real object-oriented programs. It is simple, but for its size quite powerful. And in typical applications it will also be fast. In a typical application, operations on instances should be more common than operations on classes. The central point of our redesign was to make operations on instances cheap.

In our program, building classes is slow and generates a lot of garbage. But this will be acceptable if classes are not built at times when speed is critical. The things that have to be fast are access and instance creation. Access in this program will be about as fast as we can expect without compile-time optimizations.° So will instance creation. And neither operation causes consing. Except, that is, for the vector that represents the instance itself. It seems natural enough that this should be dynamically allocated. But we could avoid dynamically allocating even instances, if we used a strategy like the one presented in Section 13.4.

Our embedded language is a characteristic example of Lisp programming. The mere fact of being an embedded language makes it one. But also characteristic of Lisp is the way in which it evolved from a small, limited version, through a powerful but inefficient version, to a fast but slightly restrictive version.

Lisp's reputation for slowness comes not so much from its own nature (Lisp compilers have been able to generate code as fast as compiled C since the 1980s) as from the fact that so many programmers stop at the second stage. As Richard Gabriel wrote,

> In Lisp, writing programs that perform very poorly is quite easy;
> in C it is almost impossible.°

This is simply a true statement, but it can be read as either a point for Lisp or a point against it:

1. By trading speed for flexibility, you can write programs very easily in Lisp ; in C you don't have this option.

2. Unless you optimize your Lisp code, it is all too easy to end up with slow software.

Which interpretation applies to your programs depends entirely on you. But at least in Lisp you have the option of trading execution speed for your time, in the early stages.

One thing our example program is *not* good for is as a model of CLOS (except possibly for elucidating the mystery of how `call-next-method` works). How much similarity could there be between the elephantine CLOS and this 70-line mosquito? Indeed, the contrasts between the two programs are more instructive than the similarities. First of all, we see what a wide latitude the term "object-oriented" has. Our program is more powerful than a lot of things that have been called object-oriented, and yet it has only a fraction of the power of CLOS.

Our program differs from CLOS in that methods are methods *of* some object. This concept of methods makes them equivalent to functions that dispatch on their first argument. And when we use the functional syntax to invoke them, that's just what our methods look like. A CLOS generic function, in contrast, can dispatch on any of its arguments. The components of a generic function are called methods, and if you define them so that they specialize only their first argument, you can maintain the illusion that they are methods *of* some class or instance. But thinking of CLOS in terms of the message-passing model of object-oriented programming will only confuse you in the end, because CLOS transcends it.

One of the disadvantages of CLOS is that it is so large and elaborate that it conceals the extent to which object-oriented programming is a paraphrase of Lisp. The example in this chapter does at least make that clear. If we were content to implement the old message-passing model, we could do it in a little over a page of code. Object-oriented programming is one thing Lisp can do. A more interesting question is, what else can it do?