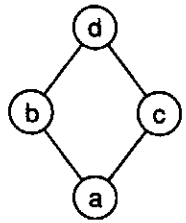

Figure 25.2: Multiple paths to a superclass.

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 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 all the ancestors of the original object, sort the list so that no object appears before one of its descendants, and then look at each element in turn.

This strategy is used by `get-ancestors`, which returns a properly ordered list of an object and its ancestors. To sort the list, `get-ancestors` calls `stable-sort` instead of `sort`, to avoid the possibility of reordering parallel ancestors. Once the list is sorted, `rget` merely searches for the first object with the desired property. (The utility `some2` is a version of `some` for use with functions like `gethash` that indicate success or failure in the second return value.)

The list of an object's ancestors goes from most specific to least specific: if `orange` is a child of `citrus`, which is a child of `fruit`, then the list will go `(orange citrus fruit)`.

When an object has multiple parents, their precedence goes left-to-right. That is, if we say

```
(setf (gethash 'parents x) (list y z))
```

then y will be considered before z when we look for an inherited property. For example, we can say that a patriotic scoundrel is a scoundrel first and a patriot second:

```
> (setq scoundrel (make-hash-table)
        patriot (make-hash-table)
        patriotic-scoundrel (make-hash-table))
```



```
(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (setf (gethash 'parents obj) parents)
    (ancestors obj)
    obj))

(defun ancestors (obj)
  (or (gethash 'ancestors obj)
      (setf (gethash 'ancestors obj) (get-ancestors obj))))

(defun rget (obj prop)
  (some2 #'(lambda (a) (gethash prop a))
         (ancestors obj)))
```

Figure 25.3: A function to create objects.

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

Let's make some improvements to this skeletal system. We could begin with a function to create objects. This function should build a list of an object's ancestors at the time the object is created. The current code builds these lists when queries are made, but there is no reason not to do it earlier. Figure 25.3 defines a function called `obj` which creates a new object, storing within it a list of its ancestors. To take advantage of stored ancestors, we also redefine `rget`.

Another place for improvement is the syntax of message calls. The `tell` itself is unnecessary clutter, and because it makes verbs come second, 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 each property name as a function, as in Figure 25.4. 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