```
(if (eligible obj)
    (progn (do-this)
           (do-that)
           obj))
```

Most macros for iteration splice their arguments in a similar way.

The effect of comma-at can be achieved without using backquote. The expression `'(a ,@b c)` is equal to `(cons 'a (append b (list 'c)))`, for example. Comma-at exists only to make such expression-generating expressions more readable.

Macro definitions (usually) generate lists. Although macro expansions could be built with the function `list`, backquote list-templates make the task much easier. A macro defined with `defmacro` and backquote will superficially resemble a function defined with `defun`. So long as you are not misled by the similarity, backquote makes macro definitions both easier to write and easier to read.

Backquote is so often used in macro definitions that people sometimes think of backquote as part of `defmacro`. The last thing to remember about backquote is that it has a life of its own, separate from its role in macros. You can use backquote anywhere sequences need to be built:

```
(defun greet (name)
  `(hello ,name))
```

7.3 Defining Simple Macros

In programming, the best way to learn is often to begin experimenting as soon as possible. A full theoretical understanding can come later. Accordingly, this section presents a way to start writing macros immediately. It works only for a narrow range of cases, but where applicable it can be applied quite mechanically. (If you've written macros before, you may want to skip this section.)

As an example, we consider how to write a variant of the the built-in Common Lisp function `member`. By default `member` uses `eql` to test for equality. If you want to test for membership using `eq`, you have to say so explicitly:

```
(member x choices :test #'eq)
```

If we did this a lot, we might want to write a variant of `member` which always used `eq`. Some earlier dialects of Lisp had such a function, called `memq`:

```
(memq x choices)
```

Ordinarily one would define `memq` as an inline function, but for the sake of example

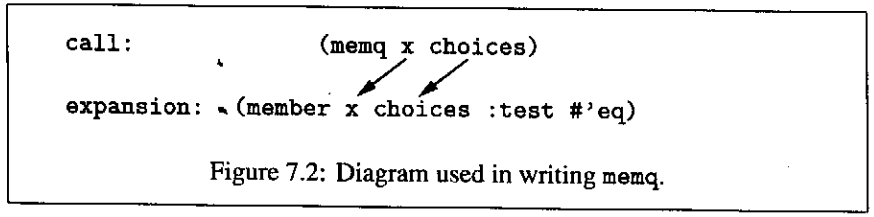

Figure 7.2: Diagram used in writing `memq`.

The method: Begin with a typical call to the macro you want to define. Write it down on a piece of paper, and below it write down the expression into which it ought to expand. Figure 7.2 shows two such expressions. From the macro call, construct the parameter list for your macro, making up some parameter name for each of the arguments. In this case there are two arguments, so we'll have two parameters, and call them `obj` and `lst`:

```
(defmacro memq (obj lst)
```

Now go back to the two expressions you wrote down. For each argument in the macro call, draw a line connecting it with the place it appears in the expansion below. In Figure 7.2 there are two parallel lines. To write the body of the macro, turn your attention to the expansion. Start the body with a backquote. Now, begin reading the expansion expression by expression. Wherever you find a parenthesis that isn't part of an argument in the macro call, put one in the macro definition. So following the backquote will be a left parenthesis. For each expression in the expansion

1. If there is no line connecting it with the macro call, then write down the expression itself.
2. If there is a connection to one of the arguments in the macro call, write down the symbol which occurs in the corresponding position in the macro parameter list, preceded by a comma.

There is no connection to the first element, `member`, so we use `member` itself:

```
(defmacro memq (obj lst)
  `(member
```

However, `x` has a line leading to the first argument in the source expression, so we use in the macro body the first parameter, with a comma:

```
(defmacro memq (obj lst)
  `(member ,obj
```