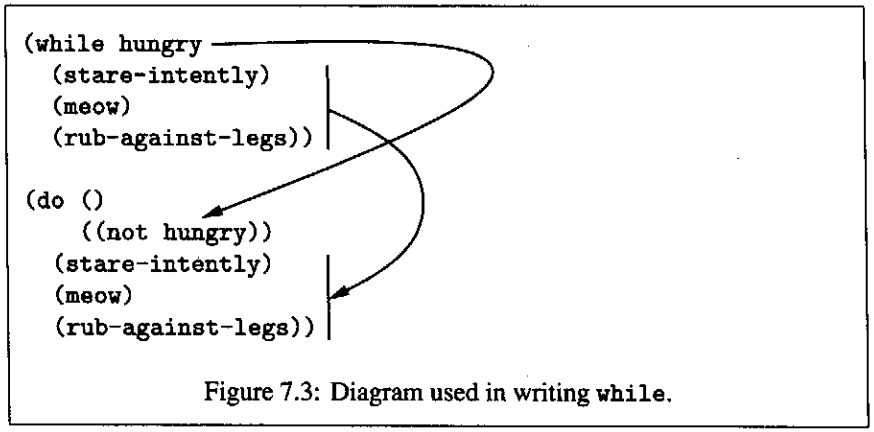

Figure 7.3: Diagram used in writing `while`.

```
(defmacro memq (obj lst)
  `(member ,obj ,lst :test #'eq))
```

So far, we can only write macros which take a fixed number of arguments. Now suppose we want to write a macro `while`, which will take a test expression and some body of code, and loop through the code as long as the test expression returns true. Figure 7.3 contains an example of a `while` loop describing the behavior of a cat.

To write such a macro, we have to modify our technique slightly. As before, begin by writing down a sample macro call. From that, build the parameter list of the macro, but where you want to take an indefinite number of arguments, conclude with an `&rest` or `&body` parameter:

```
(defmacro while (test &body body)
```

Now write the desired expansion below the macro call, and as before draw lines connecting the arguments in the macro call to their position in the expansion. However, when you have a sequence of arguments which are going to be sucked into a single `&rest` or `&body` parameter, treat them as a group, drawing a single line for the whole sequence. Figure 7.3 shows the resulting diagram.

To write the body of the macro definition, proceed as before along the expansion. As well as the two previous rules, we need one more:

3. If there is a connection from a series of expressions in the expansion to a series of the arguments in the macro call, write down the corresponding `&rest` or `&body` parameter, preceded by a comma-at.


```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))
```

To build a macro which can have a *body* of expressions, some parameter has to act as a funnel. Here multiple arguments in the macro call are joined together into `body`, and then broken up again when `body` is spliced into the expansion.

The approach described in this section enables us to write the simplest macros—those which merely shuffle their parameters. Macros can do a lot more than that. Section 7.7 will present examples where expansions can't be represented as simple backquoted lists, and to generate them, macros become programs in their own right.

7.4 Testing Macroexpansion

Having written a macro, how do we test it? A macro like `memq` is simple enough that one can tell just by looking at it what it will do. When writing more complicated macros, we have to be able to check that they are being expanded correctly.

Figure 7.4 shows a macro definition and two ways of looking at its expansion. The built-in function `macroexpand` takes an expression and returns its macroexpansion. Sending a macro call to `macroexpand` shows how the macro call will finally be expanded before being evaluated, but a complete expansion is not always what you want in order to test a macro. When the macro in question relies on other macros, they too will be expanded, so a complete macroexpansion can sometimes be difficult to read.

From the first expression shown in Figure 7.4, it's hard to tell whether or not `while` is expanding as intended, because the built-in do macro gets expanded, as well as the `prog` macro into which *it* expands. What we need is a way of seeing the result after only one step of expansion. This is the purpose of the built-in function `macroexpand-1`, shown in the second example; `macroexpand-1` stops after just one step, even if the expansion is still a macro call.

When we want to look at the expansion of a macro call, it will be a nuisance always to have to type

```
(pprint (macroexpand-1 '(or x y)))
```

Figure 7.5 defines a new macro which allows us to say instead:

```
(mac (or x y))
```

Typically you debug functions by calling them, and macros by expanding