

# IT-3105 Lecture: Lisp Macros

The following sections motivate the need for macros and investigate their essential details. Even though macro writing can become quite complex, many common CL programming situations call for macros that are often very straightforward to design.

## 1 The general idea

In the late 1950's, John McCarthy invented Lisp to cover many common weaknesses of languages such as FORTRAN, COBOL and ALGOL. Thus, Lisp was the first language to truly embrace concepts such as recursion, first-class treatment of functions (i.e. they are treated just like any other data), weak typing, and macros. Nearly 50 years later, most of these concepts have been incorporated into a wide array of languages, but Lisp macros remain several steps beyond (and above) those used elsewhere. They literally allows Lisp programmers to *rewrite the lisp-language syntax* to suit their individual preferences. And beyond aesthetics, they also provide many practical advantages, such as allowing call-by-reference behavior in a call-by-value language.

The macro concept is quite simple. Any source code written with macros is converted into conventional lisp primitives prior to compilation. Thus, the source code may display a wide spectrum of syntactic styles, from *nearly lisp* to extremely *anti-lisp*, all depending upon the software writers preferences. The macro itself is identical to a function in the sense that it receives arguments, operates on them, and returns a value. However, for a macro, these input arguments are *code*, the operations are typically symbol-manipulating actions, and the output value is *new code*. The input code is in software-developer syntax, while the output code is standard Lisp syntax. Figure 1 summarizes this basic process.

Unlike arguments to functions, those sent to macros are **not evaluated**. This reflects the basic job of the macro: to convert code into code. The input code should not be analyzed in terms of its value (i.e. evaluated), but viewed as a syntatic construct to be rewritten.

It is important to distinguish the macro code from the macro call. The latter embodies the developer's syntactic preferences and thus is normally quite easy to read and understand. On the other hand, the macro code must specify how this aesthetically-pleasing syntax is converted into standard Lisp. So in bridging this gap between the pretty and the ugly, the macro code is invariably horrendous in appearance, easily driving a Lisp neophyte to heavy drinking and/or vows to never again venture outside the cozy world of JAVA.

But once the macro code is written and debugged (almost inevitably a lengthy process at first), it maintains a stable level of abstraction between developer-chosen syntax and conventional Lisp, allowing the developer to distance herself from nuts-and-bolts Lisp, and, most importantly, from the macro code itself!

Many Lisp programmers write a lot of macros in short periods, when they are in a *macro zone*, for example when setting up useful coding interfaces to new problems, or preparing lectures on macros. After that, they may avoid them completely for many months while happily coding away at a high level of abstraction. Only when a new application (or new programming class) arises do they go back to rewriting the Lisp language to suit their new needs.

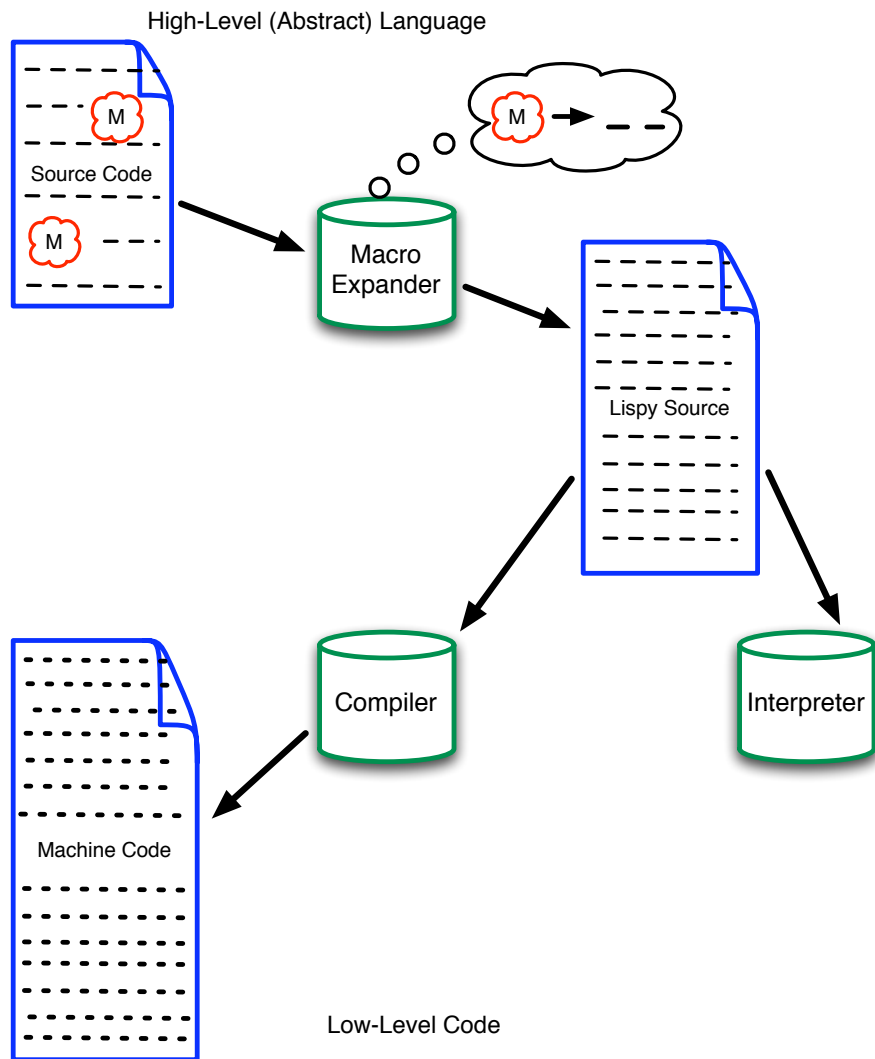


Figure 1: Overview of macros and macro expansion, where long dashed lines denote standard lisp code, dotted lines denote machine code, and clouds labelled  $M$  represent macro calls. Note that the (standard lisp) code produced by macroexpansion can either be compiled or interpreted (i.e. run directly).

## 2 Some Simple Examples

Assume that you are writing a program to generate web pages for a university. To put up a *front of formality*, you want to consistently convert all faculty names, such as *Robert Jones*, into their formal equivalents, i.e., *Professor Robert Jones (Ph.D)* - even though everyone calls him *Bob*.

As shown in Figure 2, a function to generate the formal string from the basic name string is a snap to write. This is a standard example of simplifying your code by modularizing an often-used operation.

We can call **formalize-name** with either a string or a variable bound to a string, as shown, and in either case, the return value is a new string. However, the new string does not become the new binding of the variable, *z*. In short, our function cannot *side-effect* the *z*. That would be a destructive act in a functional language, and a functional language designer would certainly never include behavior of that type as default. That's one reason why Lisp functions are, by default, call-by-value.

However, we occasionally desire the side-effects to our variables. For example, when we write an increment function, we often want to update the variable's value and not merely return its old value plus 1. That is, if  $X = 5$  and we call (increment X), we typically want the function to return 6 **and** change the value of X to 6. Unfortunately, with call-by-value functions, that will not happen, as shown in Figure 3.

```
? (defun formalize-name (name-string)
  (concatenate 'string "Professor " name-string " (Ph.D.)"))
? (formalize-name "Frank_Reed")
"Professor_Frank_Reed_(Ph.D.)"
? (defvar z "Albert_Einstein")
"Albert_Einstein"
? (formalize-name z)
"Professor_Albert_Einstein_(Ph.D.)"
? z
"Albert_Einstein"
```

Figure 2: Definition and use of a simple macro to enclose normal names in a formal title.

However, we occasionally desire the side-effects to our variables. For example, when we write an increment function, we often want to update the variable's value and not merely return its old value plus 1. That is, if  $X = 5$  and we call (increment X), we typically want the function to return 6 **and** change the value of X to 6. Unfortunately, with call-by-value functions, that will not happen, as shown in Figure 3. Within **increment**, *z* gets bound to *x*'s value, but *z* and *x* do not share a memory location. So changes to *z* do not affect *x*. When *x*'s value is a list, array or other more complex list object, the function can side-effect the original variable (e.g. *x*) in the sense that the object pointed to by *x* will change. But when that object is single value, the called function simply receives a copy.

To achieve the desired side-effect during incrementing, swapping and other standard operations, a Lisp macro is an elegant solution. Naturally, we could always write (setf x (increment x)), but that's almost as verbose as (setf x (+ x 1)). Similarly, to achieve the side-effect in Figure 2, (setf x (formalize-name x)) would work. However, we'd like to simplify the code as much as possible.

With macros, you write (increment X) in the source code, but it gets rewritten (a.k.a. *expanded*) to (setf x (+ x 1)) prior to compilation. The increment macro (renamed to a shorter word, **incr**), and a similar macro for formalizing names, appear in Figure 4.

```

(defun increment (z) (setf z (+ z 1)))

? (setf x 5)
5
? (increment x)
6
? x
5

```

Figure 3: The use of an increment function to illustrate the general problem of achieving variable side-effects while using call-by-value functions.

```

(defmacro incr (z)
  (list 'setf z
        (list '+ z 1)))

? (setf x 5)
5
? (incr x)
6
? x
6

(defmacro f-name (name)
  (list 'setf name
        (list 'concatenate '(quote string) "Professor_" name "_ (Ph.D.)")))

(defun formalize-all-names (&rest strings)
  (loop for s in strings
        collect (f-name s)))

;; =====>> Macro Expansion =====>

(defun formalize-all-names (&rest strings)
  (loop for s in strings
        collect
        (setf s (concatenate 'string "Professor_" s "_ (Ph.D.)"))))

? (formalize-all-names "Fred" "Wilma" "Barney")
("Professor_Fred_(Ph.D.)" "Professor_Wilma_(Ph.D.)" "Professor_Barney_(Ph.D.)")

```

Figure 4: Defining and using two simple macros.

Looking first at **incr**, notice that the macro body creates a list inside a list as the return value. It is important to note that during a macro call, such as **(incr x)**, the following occur:

1. The argument, *x*, is not evaluated but merely sent to the macro as the symbol *x*.
2. The macro variable *z* is then bound to symbol *x*. So any evaluation of *z* will yield *x*.
3. Within the body of the macro, evaluation occurs just like in a function.
4. Just about any lisp code is valid within the macro body.

So once we get past the key difference of evaluated versus non-evaluated arguments at the beginning of the call, macros and functions behave identically. In practice, the macro is used to create code while the function typically is not, but the tools used in both are essentially the same. Since macros are used to create a developer-friendly syntax, it's safe to say that macros exploit the symbol-processing prowess of Lisp to rewrite Lisp.

Getting back to **(incr x)**, evaluation of the nested **list** commands produces the following return value: **(setf x (+ x 1))**. During that evaluation, the quoted expressions **'setf**, **'+**, along with the **1**, evaluate to the symbols **setf**, **+** and **1**, while the *z* evaluates to *x*.

When you type in a macro call to the Lisp Listener, the REPL actually calls the macro, returns the new code, and evaluates that code. So the user notices no difference from a regular function call. However, when the macro call is embedded inside a function definition, the macro output is pasted into that definition in place of the original macro call. The bottom of Figure 4 illustrates this idea with two versions of **formalize-all-names**; the first is prior to macro expansion (of the macro **f-name**, and the second is afterwards. This is only a simplified rewrite: in reality, **loop** is also a macro, so the whole loop call would also be expanded (into some rather nasty looking code). But regardless, the key point is that our macro call does get rewritten prior to compilation or interpretation.

### 3 Backquote

Before delving into more complicated macros, we must examine Lisp's powerful backquote operator, since it greatly simplifies macro source code.

To create lists inside of lists, one can simply type a single quote followed by all the parentheses and list elements required. For example, **'(a b (c (d e) f) g)** and **'(dotimes (i 20) (print (+ i 50)))**, when typed into the Listener, simply return nested lists of exactly the form shown above (but without the leading quote). Clearly, this is a lot simpler than writing **(list 'a 'b (list 'c (list 'd 'e) 'f) 'g)** and **(list 'dotimes (list 'i 20) (list 'print (list '+ i 50)))**.

We would like to employ similar simplifications when creating the return value of a macro, which is often complex deeply-nested code. Unfortunately, this code normally includes a few variables that need to be evaluated - for example, the variable **z** in **incr** and the variable **name** in **f-name**.

Fortunately, Lisp provides a very simple mechanism, the backquote, for writing complex expressions in which certain symbols need to be evaluated. As shown in the examples of Figure 5, a backquoted expression behaves similarly to a quoted expression except that any sub-expression that is immediately preceded with a comma is evaluated. Also, as shown at the bottom of the figure, any expression preceded by **`,`@`** is evaluated, and the result is *unraveled*, i.e. merged into the surrounding list.

```

? (defvar x 20)
X
? (defvar y 30)
Y
? (defvar z '(100 200 300))
Z
? '(x y z)
(X Y Z)
? '(,x y ,z)
(20 Y (100 200 300))
? '(x+y= ,(+ x y))
(X+Y= 50)
? '(,x ,y ,@z)
(20 30 100 200 300)

```

Figure 5: A few simple examples of backquoted expressions.

Backquote enables us to rewrite the earlier macros in a much less verbose form, as shown in Figure 6. Rarely does a good Lisp hacker write a macro without exploiting the power of backquote.

```

(defmacro incr2 (z) '(setf ,z (+ ,z 1)))
(defmacro f-name2 (name)
  '(setf ,name (concatenate 'string "Professor " ,name " (Ph.D.)")))

```

Figure 6: Simplifications of **incr** and **f-name** from Figure 4 by using backquote.

Note the use of backquote in the macros in Figure 7. It enables the code of the macro body to mirror the code **produced** by the macro. Unfortunately, this is not always the case, since some macros perform sophisticated code transformations in their bodies, and thus, despite the Herculean contribution of the backquote, the end result is decidedly non-WYSIWYG.

### Exercises

- Write the macro (**pushall-new items list**), which returns code that pushes onto *list* all elements of *items* that are not already members of *list*. Hint: CL provides a primitive macro named **pushnew** for pushing single unique items onto lists.
- Write the macro (**n-of n &rest body-code**), which returns code that executes the body *n* times and collects the results from each execution. As a simple example (n-of 4 8) → (8 8 8 8) when typed into the Lisp Listener.
- Write the macro (**nullify &rest symbols**), which returns code that uses **setf** to assign nil to all argument symbols. E.g. (nullify x y z) sets variables x, y and z to nil when typed into the Lisp Listener.

## 4 A General Strategy for Macro Design

A common approach to macro design is to begin with two patterns:

```

(defmacro += (x val) `(setf ,x (+ ,x ,val)))

(defmacro pushend (item list)
  `(if (null ,list)
      (setf ,list (list ,item))
      (setf (cdr (last ,list)) (list ,item))))

(defmacro popend (elems)
  `(progl
    (first (last ,elems))
    (setf ,elems (nbutlast ,elems 1))))

```

Figure 7: Macros for a) incrementing a variable, b) pushing an item onto the end of a list, and c) popping an element from the end of a list. The CL special form **progl** delineates a block of code that is executed in sequence, but unlike **progn**, **progl** returns the value of the **first** expression in the sequence.

```

? (setf x 4)
4
? (+= x 10)
14
? x
14
? (macroexpand-1 '(+= x 10))
      (SETF X (+ X 10))

? (setf x '(1 2 3))
(1 2 3)
? (pushend 555 x)
(555)
? x
(1 2 3 555)
? (macroexpand-1 '(pushend 555 x))
      (IF (NULL X)
          (SETF X (LIST 555))
          (SETF (CDR (LAST X)) (LIST 555)))

? (popend x)
555
? x
(1 2 3)
? (macroexpand-1 '(popend x))
      (PROGL
        (FIRST (LAST X))
        (SETF X (NBUTLAST X 1)))

```

Figure 8: Examples using the macros of Figure 7.

```

(defmacro pushall-new (new-items list &optional (test 'equal))
  (let ((item (gensym)))
    '(dolist (,item ,new-items ,list)
      (pushnew ,item ,list :test ,test))))

(defmacro n-of (n &rest body)
  (let ((i (gensym)))
    '(loop for ,i from 1 to ,n collect (progn ,@body))))

(defmacro nullify (&rest vars)
  '(setf ,@(mapcan (lambda (var) (list var nil)) vars)))

```

Figure 9: Answers to the exercises

1. Source - The syntax of the macro call, which will reflect the preferences of the designer.
2. Target - The syntax of the standard lisp code into which the macro will be expanded.

The general form of the first pattern is then reflected in the parameter list of the macro, while the specification for the last pattern appears as an often-backquoted expression at the end of the macro. The macro code between the parameter list and the return pattern is transformation code: that needed to convert the first pattern into the second. Since backquotes allow anything to be evaluated within them, some of these transformations may occur within the backquoted expression as well. We use the term *preamble* to refer to all code between the parameter list and the returned backquoted expression. Figure 10 summarizes this general structure, which applies to most macros. A common exception is the lack of a preamble for many simple macros.

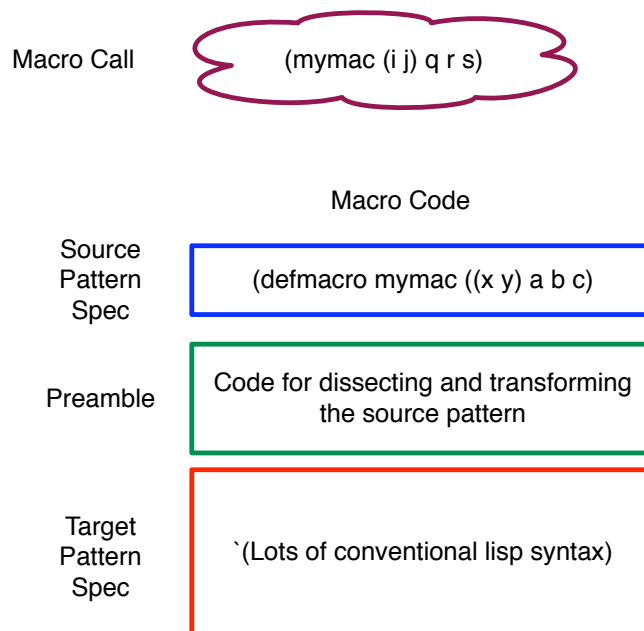


Figure 10: Basic structure of a macro

Figure 11 displays a macro for swapping the values of two variables. This is clearly an operation that a) is tedious to write out in its entirety everytime two variable values need exchanging, and b) cannot be easily handled by call-by-value functions. It almost demands a macro.

The source pattern is simply *(swap x y)*, while the target is more verbose code involving the declaration of a temporary variable, the saving of one value in that variable, etc. The final 3 lines of the **swap** macro express this target.

In the middle, the preamble is one simple line of code, a **let** statement, which binds the local variable **temp** to the result of a call to **gensym**, which is guaranteed to produce a unique symbol. Thus, this new symbol cannot possibly conflict with the name of any other symbol anywhere in current lisp environment. This is essential, since macros create code that may be placed anywhere in the source code, and they may *wrap around* any existing source code. So they cannot simply introduce variables willy-nilly, since this could easily cause name conflicts - especially when using standard names such as *temp*.

So, in the preamble, the macro uses a local variable named *temp* to hold a unique variable name, which turns out to be `#:G8217` in the example of Figure 11. Then, when **temp** is evaluated inside the backquote, it produces `#:G8217`, and thus, the macro returns code with three variables: *x*, *y* and `#:G8217`, with no mention of **temp**.

```
(defmacro swap (x y)
  (let ((temp (gensym)))
    `(let ((,temp ,x))
      (setf ,x ,y)
      (setf ,y ,temp))))

? (macroexpand-1 '(swap x y))
(LET ((#:G8217 X))
  (SETF X Y)
  (SETF Y #:G8217))
```

Figure 11: Definition of a swap macro and one example of its application. **macroexpand-1** returns the code produced by a quoted macro call. Note the use of **gensym** to guarantee that the variable name introduced by the macro does not conflict with any existing variable names.

Let's consider a real classic, the **if...then...else** construct. Common Lisp provides an **if** statement, but keywords such as *then* and *else* are not permitted. So, if we feel a longing for a more FORTRAN-like **if**, macros provide a relatively straightforward (albeit more difficult than the previous examples) solution.

First, let's assume that we want free reign to include as many expressions as needed for the *then* and *else* clauses. Common Lisp's **if** only permits one of each. Thus, our source pattern will have the form: *(if C then A<sub>1</sub>A<sub>2</sub>...A<sub>m</sub> else B<sub>1</sub>B<sub>2</sub>...B<sub>n</sub>)*

The target pattern involves Common Lisp's **if**, but with the actions wrapped inside blocks. In Lisp, **progn** is a useful block construct: every expression within a **progn** is evaluated, with the last one producing the return value of the **progn**. Thus, the target pattern would be:

```
(if C (progn A1A2...Am) (progn B1B2...Bn))
```

Our macro needs to find and gather up the *A<sub>i</sub>* and the *B<sub>j</sub>* from the source pattern, bundle them into **progn**'s, and wrap the **progn**'s inside a standard **if** statement. This finding and gathering of symbols is a simple job for a Lisp function.

The approach taken by **split-at** in Figure 12 is to find the key (such as *then*) and to then split the list into two sublists: all elements prior to the key, and all elements after it. **split-at** then returns a list that contains the two sublists. For example:

(split-at '((> X 5) then (setf X 4) (print X) else (incr X)) 'then) returns two lists:

- ((> X 5))
- ((setf X 4) (print X) else (incr X)))

So, to collect the 3 key sections of an if-then-else, simply call **split-at** twice: the first time with the entire **if** statement (minus the *if* keyword) and a key of *then*, and the second time with the second sublist returned by the first call and a key of *else*. The preamble of **myif** handles this work, leaving very little to be done in the backquoted portion.

Note that the parameter list for **myif** employs the **&rest** descriptor. This causes all parameters to be wrapped into a single list and bound to the argument variable, which is *body* in this case. Also note that the arguments to a macro call are everything except the macro name itself. So for a call such as (*myif* (> Y 10) then 55 else 66), *body* is bound to ((> Y 10) then 55 else 66).

In that final section, the three code chunks: **condition**, **act1** and **act2** are unpacked via the **”,@”** prefix. Note that although the macro uses many local variables, it introduces no new variables into the return expression. Hence, no calls to **gensym** are necessary.

#### Exercises

- Write a **repeat** macro to handle the source pattern (repeat action(s) until condition), where *until* is a necessary keyword.
- Write a **while** macro to handle the source pattern (while condition do action(s)), where *do* is a necessary keyword.

## 5 Handling Complex Source Patterns

In many cases, the developer-friendly syntax of a macro source pattern involves a combination of nested lists, as exemplified by Common Lisp’s **do cond**, **case** and **with-open-file** macros, among others.

One approach to transforming these patterns into lisp primitives is to a) accept them as a single macro argument (using the *rest* option in the parameter list, and b) use a rather complex preamble to pick out the different pieces, based on either keyword recognition or sublist structure. When the source pattern is a relatively flat list with several keywords and an indefinite number of expressions between keywords (such as the *if-then-else* above), then this approach is unavoidable.

However, if the main complexity of the source pattern involves nested lists, CL’s **destructuring-bind** provides a nice labor-saving tool. And best of all, a similar tool is automatically invoked during macro expansion. You never really need to think about it.

Figure 14 provides a few simple examples. In each, notice that the parameter list (the first list) can include nested sublists, and each of these can include the standard parameter-list keywords such as **&key**, **&op-**

```

(defun split-at (l key &optional (part1 nil))
  (cond ((null l) (list (reverse part1) nil))
        ((equal (car l) key) (list (reverse part1) (cdr l)))
        (t (split-at (cdr l) key (cons (car l) part1)))))

(defmacro myif (&rest body)
  (let* ((pair (split-at body 'then))
         (condition (first pair))
         (pair2 (split-at (second pair) 'else))
         (act1 (first pair2))
         (act2 (second pair2)))
    `(if ,@condition
        (progn ,@act1)
        (progn ,@act2))))

? (macroexpand-1 '(myif C then A1 A2 A3 else B1 B2 ))

(IF C
    (PROGN A1 A2 A3)
    (PROGN B1 B2))

? (macroexpand-1 '(myif (> X Y) then (setf Y X) (print Y)
                       else (setf X (+ X Y)) (print X)))

(IF (> X Y)
    (PROGN
     (SETF Y X)
     (PRINT Y))
    (PROGN
     (SETF X (+ X Y))
     (PRINT X)))

```

Figure 12: Definition of a classic if-then-else (named **myif**). Notice that most of the code involves picking out the *condition* clause, plus the *then* and *else* actions, via calls to **split-at**.

```

(defmacro while (&rest body)
  (let ((pair (split-at body 'do)))
    '(loop
      (if (not ,(first pair)) (return))
      ,(second pair))))

? (macroexpand-1 '(while (< x 20) do (setf x (+ x 2)) (print x)))

(LOOP
  (IF (NOT (< X 20)) (RETURN))
  (SETF X (+ X 2))
  (PRINT X))

(defmacro repeat (&rest body)
  (let ((pair (split-at body 'until)))
    '(loop
      ,(first pair)
      (if ,(second pair) (return)))))

? (macroexpand-1 '(repeat (setf x (- x 2)) (print x) until (< x 10)))

(LOOP
  (SETF X (- X 2))
  (PRINT X)
  (IF (< X 10) (RETURN)))

```

Figure 13: Definition of **while** and **repeat** macros with syntax (while condition do action(s)) and (repeat action(s) until condition). Legal source patterns must include the keywords *do* and *until*, respectively.

**tional**, and **&rest**. The arguments (the second list) are then picked out of their nested lists and assigned to the corresponding variables in the parameter list. This allows considerable flexibility in parsing rather intricate argument lists. Since macro parameter lists can exploit similar destructuring, we can design complicated (but aesthetically pleasing) source patterns for our macros and have them automatically dissected during macro expansion.

```

? (destructuring-bind (x y z) '(10 20 30)
  (list x (* x y) (* x y z)))
(10 200 6000)
? (destructuring-bind ((x y) z &rest extras) '((10 20) 30 40 50 60)
  (apply #'+ x y z extras))
210
? (destructuring-bind
  ((x &rest extras-1) y &rest extras-2)
  '((10 20 30 40) 50 60 70 80 90)
  (apply #'+ x y (append extras-1 extras-2)))
450

```

Figure 14: Several examples of **destructuring-bind**, which binds the variables in the parameter list to corresponding elements of the argument list before executing the body code.

Consider the problem of working through 2 lists in parallel. CL provides **dolist** for walking down a single list, but nothing simple for 2 or more lists. The **do** macro does the job, but its syntax is rather complicated. Since the 2-list case comes up very frequently, we will specialize our macro for it and ignore the n-list case for the moment.

Our source pattern is shown in the call to **do2lists** near the top of Figure 15. Notice that the pattern has two main parts:

1. A 3-element list, where the first two elements are pairs of the form (list-element list), and the 3rd element is an expression to be returned when we have walked through both lists.
2. The code body, which will be performed on each pair of corresponding elements in the two lists.

The two lists need not have the same length; **do2lists** will stop when it reaches the end of either list.

Figure 15 shows the code for **do2lists**. Notice that it fully exploits destructuring by including all important variables in the parameter list, nested at various levels. This renders a preamble unnecessary, since these variables are exactly the pieces needed to construct the target pattern. In this case, a call to **mapc** - with the **body** code unraveled into the body of the lambda and the list-item variables as the two parameters to the lambda - works fine.

Without destructuring, we would be forced to write something more complicated, such as **do2lists-hard** at the bottom of Figure 15. Note that this has a large preamble, and one that still does not work all the way down to the level of the necessary variables. This is done by calls to **first** and **second** in the backquoted target pattern.

Clearly, destructuring is a labor-saving process of which all Lisp programmers should take advantage when writing macros (and performing other operations in which returned lists require dissection).

```

? (defvar j '(1 2 3 4 5))
J
? (defvar k '(a b c d e f g))
K
? (do2lists ((j-elem j) (k-elem k) "Finished")
            (print (list j-elem k-elem)))

(1 A)
(2 B)
(3 C)
(4 D)
(5 E)
"Finished"

(defmacro do2lists (((x1 l1)(x2 l2) &optional (return-val t)) &rest body)
  `(progn
    (mapc #'(lambda (,x1 ,x2) ,@body) ,l1 ,l2)
    ,return-val))

(defmacro do2lists-hard (params &rest body)
  (let* ((pair1 (first params))
         (pair2 (second params))
         (return-val (third params)))
    `(progn
      (mapc #'(lambda (,(first pair1) ,(first pair2)) ,@body)
            ,(second pair1) ,(second pair2))
      ,return-val)))

```

Figure 15: Use of the built-in destructuring-bind of macro parameter lists to create **do2list**, which works through two lists in parallel.

## 6 Macro Wrappers

In many programming contexts, a primary code chunk is *wrapped* inside auxiliary code that may perform bookkeeping, monitoring, gating or pre-processing functions for the primary code. If this auxiliary functionality reappears often in source code, then, as with any reoccurring operation, it can (and should) be abstracted into a module.

Since functions cannot accept code objects as arguments without first evaluating them, it is up to macros to cleanly handle the abstractions associated with wrapper code.

Consider a simple gating condition that all arguments to a function must be non-negative. If many functions require such a test, then a simple macro wrapper can save a lot of typing. Figure 16 shows the **neg-check** macro.

```
(defmacro neg-check (func &rest args)
  '(if (not (member-if (lambda (x) (< x 0)) ',args))
      (apply ',func ',args)))

? (macroexpand-1 '(neg-check #'* x y z))
(IF (NOT (MEMBER-IF (LAMBDA (X) (< X 0)) '(X Y Z)))
    (APPLY #'* '(X Y Z)))
```

Figure 16: A simple wrapper macro for enforcing the gating condition that all arguments to a function must be non-negative.

Even within the code body of macros, certain patterns reoccur, such as the creation of gensym'ed names in the preamble. A wrapper macro for simplifying source macro code, **with-gensyms** of Figure 17, provides an elegant abstraction.

```
(defmacro with-gensyms (syms &rest body)
  '(let ,(mapcar #'(lambda (s) '(,s (gensym)))
                syms)
      ,@body))

(macroexpand-1 '(with-gensyms (x y z) (do-something-with-x-y-and-z)))

(LET ((X (GENSYM))
      (Y (GENSYM))
      (Z (GENSYM)))
    (DO-SOMETHING-WITH-X-Y-AND-Z))
```

Figure 17: Definition and use of the ubiquitous **with-gensyms** macro.

The **with-gensyms** macro comes in handy for designing the **timed-while** macro, shown in Figure 18. This performs a standard while-do loop, but in addition, it specifies the maximum time that the loop can run. Thus, the loop will exit when either the condition becomes false or it times out. Notice that the brunt of **timed-while** involves the use of internal timing functions and constants, details that a programmer probably does not want to deal with everytime she writes a time-bounded function. The macro abstracts away that detail so that she only needs to think about it once: when she writes the macro.

```

(defmacro timed-while (seconds &rest body)
  (let ((pair (split-at body 'do)))
    (with-gensyms (target lastval)
      '(let* ((,target (+ (round (* ,seconds internal-time-units-per-second))
                        (get-internal-real-time)))
              ,lastval)
          (loop
            (if (or (>= (get-internal-real-time) ,target)
                    (not ,@(first pair)))
                (return ,lastval))
              (setf ,lastval (progn ,@(second pair))))))))

? (macroexpand-1
   '(timed-while 60 (moving-forward? mars-robot) do
     (take-pictures mars-robot)
     (send-images-to-earth mars-robot)))

(LET* ((#:G433 (+ (ROUND (* 60 INTERNAL-TIME-UNITS-PER-SECOND))
                 (GET-INTERNAL-REAL-TIME))) #:G434)
  (LOOP (IF (OR (>= (GET-INTERNAL-REAL-TIME) #:G433)
                (NOT (MOVING-FORWARD? MARS-ROBOT)))
           (RETURN #:G434))
        (SETF #:G434
              (PROGN
                (TAKE-PICTURES MARS-ROBOT)
                (SEND-IMAGES-TO-EARTH MARS-ROBOT)))))

```

Figure 18: The **timed-while** macro, useful in real-time applications.

Consider situations in which a set of variables will be involved in code that may modify their values, but afterwards, they should be reset to their original values. If these situations arise often, then it would be tedious to repeatedly write code to save the original variable bindings in the beginning and then restore them at the end of a code chunk. The macro **reset-protect** solves the problem, as shown in Figure 19.

```
(defmacro reset-protect (vars &rest body)
  (with-gensyms (oldvals var val)
    '(let ((,oldvals (list ,@vars))
          ,var ,val)
      (progn
        (progn ,@body)
        (do2lists ((,var ',vars) (,val ,oldvals))
                  (eval '(setf ',var ',val)))))))

? (setf x 1 y 2 z 3)
3
? (reset-protect (x y z) (setf x 10 y 20 z 30) (print (+ x y z)))
60
? (+ x y z)
6
```

Figure 19: The **reset-protect** macro for saving the bindings of variables, performing calculations on them, and then restoring their original bindings.

## 7 Commas, Quotes, Backquotes and Eval

In the backquoted expression of **neg-check** in Figure 16, notice the quote prior to the comma that precedes *args*. This does not block the operation of the comma, which forces the evaluation of the variable *args*. Rather, it insures that the result of that evaluation is quoted. The quote-comma combination is prevalent in macros. Figure 20 provides a few simple examples of the quote-comma combination.

```
? (setf x 1 y 2 z 3)
3
? '(5 4 ,(list x y z))
(5 4 (1 2 3))
? '(5 4 ',(list x y z)) ;; quote-comma combination
(5 4 '(1 2 3)) ;; quoted result of evaluating (list x y z)
? '(5 4 ,@(list x y z))
(5 4 1 2 3)
? '(apply #'+ ,(list x y z))
(APPLY #'+ (1 2 3))
? (eval '(apply #'+ ,(list x y z)))
> Error: Car of (1 2 3) is not a function name or lambda-expression.
1 > (eval '(apply #'+ ',(list x y z))) ;; quote-comma avoids the error
6
```

Figure 20: The combination of quote and comma inside a backquoted expression. The error message near the bottom arises because **apply** is a function; hence, its arguments are evaluated. Evaluating the list (1 2 3) yields an error, since 1 is not a valid function. By putting a quote before the comma, the argument to **apply** becomes '(1 2 3).

Also notice the use of `eval` in Figure 20. This function performs the simple operation of evaluating its argument. However, since it is a function, the argument is evaluated prior to the call. Hence, the net effect of a call to `eval` is TWO evaluations. Figure 21 shows a few examples of `eval` in action.

```
? (setf x 333 y 'x z 'y)
Y
? y
X
? z
Y
? (eval y)
333
? (eval z)
X
? (eval (eval z))
333
? '(+ ,x ,(eval y) ,(eval (eval z)))
(+ 333 333 333)
? (eval '(+ ,x ,(eval y) ,(eval (eval z))))
999
```

Figure 21: Examples using `eval` alone and in combination with backquote and comma.

To further complicate matters, two or more commas may appear together within a backquoted expression, but this does NOT imply that the argument following the commas should be evaluated twice, at least not immediately. Each comma *belongs* to a particular backquote and is only activated when that backquote is being evaluated.

Consider the double commas in the expression `(eval '(setf ,,var ,,val))` of Figure 19. Commas match to their appropriate backquotes in much the same way that right parentheses match to left parentheses: innermost to innermost and outermost to outermost. So in this example, the leftmost commas match to the innermost backquote (i.e. that which precedes `(setf..)`), while the rightmost commas match to the outermost backquote (i.e. that which precedes `(eval..)`).

When the macro is called, only the outermost backquote is evaluated, so only its commas are active in the sense of forcing internal evaluations. The inner backquote and its commas are preserved in their syntactic form until later, when the code returned by the macro is actually invoked.

To determine whether or not a given comma is active within a particular backquoted expression, use the guidelines given by Graham (pp. 399-400),<sup>1</sup> which state that a comma, C, matches a backquote, B, if and only if the counts of commas and backquotes *between* B and C are the same, where the definition of *between* is as follows:

A symbol Y, (i.e. a comma or a backquote) is between symbols X and Z if and only iff Y appears within the scope of X and Z appears within the scope of Y. Everything in the expression immediately following a backquote is within its scope, and, similarly, everything in the expression immediately following a comma is within its scope. For a chain of k adjacent commas, each is within the scope of all those that precede it.

A few simple examples (see Figure 22) should make this clearer. Although backquote is similar to quote in being a special-form call, the expressions returned by a Lisp Listener tend to replace unevaluated backquotes and commas with more complex syntax. In the examples of Figure 22, we leave the unevaluated backquotes

<sup>1</sup>ANSI Common Lisp, Paul Graham (1996)

and commas unchanged to more clearly illustrate the evaluation process.

```
? (setf x 10 y 20 z 30)
30
? '(,x '(,y ,z))
(10 '(,y ,z))
? '(,x ,(,y ,z)) ;; The 2nd comma forces the 2nd backquote to be eval'ed
(10 (20 30))
? '(,x '(, ,y , ,z))
(10 '(,10 ,20)) ;; innermost commas get activated
? '(,x '(cons , ,y , ,z))
(10 '(cons ,20 ,30)) ;; again, innermost commas activated.
```

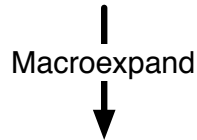
Figure 22: Examples of nested backquotes and commas.

Looking back at the **reset-protect** macro of Figure 19, we can now classify each comma as belonging to the first or second backquote. This helps clarify which commas get evaluated during the macro call, and which are evaluated later, when the macro's return code is executed.

**Exercises** Determine the return values of the backquoted calls in Figure 24.

```
(defmacro reset-protect (vars &rest body)
  (with-gensyms (oldvals var val)
    `(let ((,oldvals (list ,@vars))
          ,var ,val)
      (prog1
        (progn ,@body)
        (do2lists ((,var ',vars) (,val ,oldvals))
          (eval `(setf ,var ,val))))))))
```

```
(reset-protect (a b c) (setf a (setf b (setf c 100))))
```



```
(LET ((#:G227 (LIST A B C))
      #:G228
      #:G229)
  (PROG1
    (PROGN
      (SETF A (SETF B (SETF C 100))))
    (DO2LISTS ((#:G228 '(A B C)) (:G229 #:G227))
      (EVAL `(SETF ,#:G228 ,#:G229))))))
```

Figure 23: Code for the **reset-protect** macro, with color coding for backquotes and their associated commas. Note that most of the commas belong to the outermost backquote (in red) and are thus evaluated during the execution of the macro itself. Conversely, the innermost backquote (in black) and its two commas are preserved in the macro's return code, only to be evaluated when that code is run.

```
? (setf mom 'granny granny 'susan susan 75)
75
? '(Mom has mom ,(eval mom) who is ,(eval ',mom) )
? '(Mom has mom ,(eval mom) who is ,(eval (eval mom)))
? '(Mom has mom ,(eval mom) '(who is ,(eval (eval mom)) years old))
? '(Mom has mom ,(eval mom) ,(eval (eval mom)) years old))
```

Figure 24: Determine the return value of each of these backquoted expressions.