Macroexpand-All: An Example of a Simple Lisp Code Walker

Richard C. Waters

Mitsubishi Electric Research Laboratories
201 Broadway; Cambridge MA 02139
Dick@MERL.COM

If you like to write Lisp macros, or even just use the macros other people write, you have no doubt felt the desire to see what particular macro calls expand into. The standard Common Lisp function **macroexpand** is very useful in this regard; however, since it only expands the topmost form in an expression, it does not necessarily show you the full result of a macro expansion.

For example, suppose that you wrote (or have available to you) the following implementation of the standard Lisp macro **cond**.[1]

```
(defmacro cond (&rest clauses)
  (when clauses
    `(if ,(caar clauses)
         (progn ,@(cdar clauses))
         (cond ,@(cdr clauses)))))
```

If you evaluate

```
(macroexpand '(cond (a b) (c d)))
```

you obtain the result

```
(if a (progn b) (cond (c d)))
```

which is not as informative as you might wish, because it does not show you the complete result that will be obtained when the nested instance of **cond** that is created by the macro is eventually expanded.

As shown below, it is trivial to write a program that applies **macroexpand** to every sublist in a Lisp expression.

[1]This is an ugly implementation of **cond**, because it produces a lot of excess code. Worse, it is an erroneous implementation of **cond**, because it assumes that every clause contains at least two elements. However, it is a convenient example for the purposes of this discussion.

```
(defun macroexpand-tree (tree)
  (setq tree (macroexpand tree))
  (if (atom tree)
      tree
      (mapcar #'macroexpand-tree tree)))
```

This can be used to show you the complete macroexpansion of a form in many situations. For instance,

```
(macroexpand-tree '(cond (a b) (c d)))
```

yields

```
(if a (progn b) (if c (progn d) nil))
```

Unfortunately, **macroexpand-tree** is severely flawed, because it does not operate in the same manner as the Common Lisp compiler and evaluator. In particular, while **macroexpand-tree** macroexpands every sublist in a Lisp expression, the evaluator and compiler only macroexpand sublists that are in positions where they can be evaluated.

For example, when it encounters the form

```
(mapcar #'(lambda (cond) (car cond)) list)
```

the compiler does not do any macroexpansion. However, applying **macroexpand-tree** produces

```
(mapcar #'(lambda nil (car cond)) list)
```

since `(macroexpand '(cond))` is **nil**.

To macroexpand everything that should be expanded in a Lisp expression, and nothing else, you have to write a function that understands which parts of which Lisp forms are evaluated and which parts are not. A function called **macroexpand-all** that does this is presented in the next section.

Taken by itself, `macroexpand-all` is useful, but not all that interesting. However, the way that `macroexpand-all` is written is quite interesting, because it is an example of an important class of programs known as code walkers.

One of the beauties of Lisp is that everything that any programming tool has to know about the syntax of Lisp can be stated in a couple of short paragraphs. Further, this information is built into the Lisp reader so that nobody has to expend much effort dealing with Lisp syntax.

In comparison to many other programming languages, the semantics of Lisp is also very simple, because almost everything is a mere function call or a macro that expands into function calls. However, there is a residue of some 25 special forms each of which has its own special semantics.[2]

Unfortunately, 25 is a pretty large number when you consider that each tool that manipulates programs in non-trivial ways has to have embedded knowledge of all 25 special forms. The way this typically comes about is that the tool has to traverse a Lisp expression and either change parts of it (e.g., macroexpanding subforms or renaming variables) or collecting information (e.g., about free variables or bound variables). This kind of process is generally referred to as *code walking*.

People have attempted to implement general code walkers that encode everything any tool has to know about Lisp semantics (see for example, the PCL code walker described in [1]). However, none of the resulting code walkers have been generally accepted as really containing everything anyone would need. As a result, the writers of Lisp programming tools and complex macros are typically required to write there own code walkers. The implementation of `macroexpand-all` is a valuable tutorial introduction to how this can be done.

[2]Due primarily to the urgings of Kent Pitman [2], a key advance of Common Lisp over its predecessors (e.g., MacLisp) was reducing the number of special forms to only 25 and preventing users from defining new ones. A significant attraction of Scheme is that it goes even farther in the direction of semantic simplification.

Macroexpand-All

Except as noted below, `macroexpand-all` is addressed solely to Common Lisp as defined in *Common Lisp the Language, first edition* (CLtL1) [3], rather than the proposed standard version described in *Common Lisp the Language, second edition* (CLtL2) [4]. The decision to stick to CLtL1 was motivated by two issues. First, except where noted below, moving to CLtL2 would make `macroexpand-all` a bit more complex, because there are a few more special forms, but it would not make it any more interesting. In addition, like many people, I still have to work in CLtL1 and so this is the version of `macroexpand-all` I am using.

The main body of the code for `macroexpand-all` is shown in Figure 1. `Macroexpand-all` (at the top left of the figure) takes two arguments: a form and an optional macro environment (i.e., the same kind of environment that `macroexpand` takes). `Macroexpand-all` copies the form to be expanded to protect the code that contains the form from being destructively modified during macro expansion, and then calls `mexp` to do the real work. (Some implementations of Common Lisp implement `copy-tree` recursively. If this is the case in the Lisp you use, you will have to write an iterative implementation of `copy-tree` to use in `macroexpand-all` or risk stack overflow occurring.)

`Mexp` is the central control point of the code walking process. It calls `macroexpand-1` repeatedly until the form has been converted into a use of a special form whose semantics is understood by `mexp` or reduced to an ordinary function call or other vanilla object. `Mexp` then recurses by calling an appropriate handler as discussed shortly. (`Mexp` checks for special forms each time before calling `macroexpand-1`, because some implementations of Common Lisp implement some special forms as macros.)

It should be noted that while `mexp` is a very simple code walker, every code walker has to have essentially the same structure. A code walker has to expand every macro call, because the only way for it to understand the semantics of a macro call is to determine what it expands

```
(in-package :mexp)
(export '(macroexpand-all))

(defun macroexpand-all (f &optional env)
  (mexp (copy-tree f) env))

(defun mexp (f env &aux (flag t) m)
  (loop
    (cond ((atom f)
           (return f))
          ((not (symbolp (car f)))
           (return (all-mexp f env)))
          ((setq m (get (car f) 'mexp))
           (return (funcall m f env)))
          ((not flag)
           (return (funcall-mexp f env))))
    (multiple-value-setq (f flag)
      (macroexpand-1 f env))))

(defun all-mexp (list env)
  (do ((f list (cdr f))
       (r () (cons (mexp (car f) env) r)))
      ((atom f) (nreconc r f))))

(defun funcall-mexp (f env)
  `(,(car f) ,@(all-mexp (cdr f) env)))

(defun quote-mexp (f env)
  (declare (ignore env))
  f)

(defun block-mexp (f env)
  `(,(car f)
    ,(cadr f)
    ,@(all-mexp (cddr f) env)))

(defun let-mexp (f env)
  `(,(car f)
    ,(mapcar #'(lambda (p)
                 (bind-mexp p env))
      (cadr f))
    ,@(all-mexp (cddr f) env)))

(defun bind-mexp (p env)
  (if (and (consp p) (consp (cdr p)))
      (list (car p) (mexp (cadr p) env))
      p))

(defun lambda-mexp (f env)
  `(,(car f)
    ,(mapcar #'(lambda (p)
                 (arg-mexp p env))
             (cadr f))
    ,@(all-mexp (cddr f) env)))

(defun arg-mexp (arg env)
  (if (and (consp arg) (consp (cdr arg)))
      `(,(car arg)
        ,(mexp (cadr arg) env)
        ,@(cddr arg))
      arg))

(defun get-var (b)
  (if (consp b) (car b) b))

(defun get-val (b)
  (eval (if (consp b) (cadr b) nil)))
```

```
(defun compiler-let-mexp (f env)
  (progv (mapcar #'get-var (cadr f))
         (mapcar #'get-val (cadr f))
    (mexp
      (if (null (cdddr f))
          (caddr f)
          `(let nil ,@(cddr f)))
      env)))

(defun macrolet-mexp (f env)
  (with-env env `(macrolet ,(cadr f))
            #'mexp
            (if (null (cdddr f))
                (caddr f)
                `(let nil ,@(cddr f)))))

(defun flet-mexp (f env)
  `(flet
    ,(all-lambda-mexp (cadr f) env)
    ,@(with-env env `(flet ,(cadr f))
                #'all-mexp
                (cddr f))))

(defun labels-mexp (f env)
  (with-env env `(labels ,(cadr f))
            #'labels-mexp-2 f))

(defun labels-mexp-2 (f env)
  `(labels
    ,(all-lambda-mexp (cadr f) env)
    ,@(all-mexp (cddr f) env)))

(defun all-lambda-mexp (list env)
  (mapcar #'(lambda (f)
              (lambda-mexp f env))
          list))

(mapc #'(lambda (x)
          (setf (get (car x) 'mexp)
                (eval (cadr x))))
  '((block          #'block-mexp)
    (catch          #'funcall-mexp)
    (compiler-let   #'compiler-let-mexp)
    (declare        #'quote-mexp)
    (eval-when      #'block-mexp)
    (flet           #'flet-mexp)
    (function       #'funcall-mexp)
    (go             #'quote-mexp)
    (if             #'funcall-mexp)
    (labels         #'labels-mexp)
    (lambda         #'lambda-mexp)
    (let            #'let-mexp)
    (let*           #'let-mexp)
    (macrolet       #'macrolet-mexp)
    (multiple-value-call #'funcall-mexp)
    (multiple-value-prog1 #'funcall-mexp)
    (progn          #'funcall-mexp)
    (progv          #'funcall-mexp)
    (quote          #'quote-mexp)
    (return-from    #'block-mexp)
    (setq           #'funcall-mexp)
    (tagbody        #'funcall-mexp)
    (the            #'block-mexp)
    (throw          #'funcall-mexp)
    (unwind-protect #'funcall-mexp)))
```

Figure 1: The main body of the code for `macroexpand-all`.

into. When confronted by a form that is not a macro call, any code walker has to have special-purpose handlers for each kind of form since the various special forms are totally idiosyncratic.

As summarized in Table 5-1 on page 57 of [3], CLtL1 has 24 special forms. However, from the perspective of tools that operate on programs `lambda` should be added to this list, since it can appear in code and certainly has special semantics. `Mexp` maintains an index between special forms and their handlers by storing the handler functions as properties of the special form symbols. This is set up by the expression in the lower right of Figure 1. The remainder of the figure shows the handlers themselves.

Since `mexp` primarily only cares about what parts of a special form macroexpansion should be applied to and what parts it should not be applied to, most of the handlers are very simple, and many of the special forms are treated in the same way. For example, the handler `funcall-mexp` specifies that everything except the first element in a form should be macroexpanded. For the purposes of `mexp` this is appropriate for handling ordinary function calls and 11 of the 25 special forms.

A code walker that is more complex than `mexp` will require more complex handlers that keep track of additional information such as what variables are bound. However, the handlers will be basically upward compatible from the ones shown here.

There are only three classes of `mexp`'s handlers that are at all complex. The handlers for forms that bind variables (i.e., `let{*}` and `lambda`) are a bit complex due to the somewhat complex syntax that is used to specify variables and values for them.

The handler for `compiler-let` is complex because it must cause a change in the variable bindings that are in effect while macro expansion proceeds. Conveniently, only special variables are involved, so the change can be straightforwardly made by using `progv` to change the evaluation environment before recursing into the body of the `compiler-let`.

The handlers for `flet`, `labels`, and `macrolet` are by far the most interesting. They are complicated because they potentially change the environment that controls the way macros expand. `Flet` and `labels` can shadow a macro definition with a function definition. `Macrolet` can introduce a new macro definition.

For example, consider the form

```
(flet ((cond (x) (cond (x (1+ x)))))
  (cond (car y)))
```

Assuming the definition of `cond` used above, this should macroexpand into

```
(flet ((cond (x)
         (if x (progn (1+ x)) nil)))
  (cond (car y)))
```

The use of `cond` in the body of the local function definition is an instance of the macro `cond` defined above, but the instance of `cond` in the body of the `flet` is an instance of the locally defined function instead.

A similar situation arises with `macrolet`.

```
(macrolet ((cond (x) (cond (x '(1+ ,x)))))
  (cond (car y)))
```

macroexpands into

```
(1+ (car y))
```

The `macrolet` form itself does not need to be retained once macro expansion has occurred. The information it specifies is only relevant to the expansion of macro calls syntactically nested within it and these calls have all been eliminated by expanding them.

The handlers for `flet`, `labels`, and `macrolet` are each implemented using a function called `with-env`, which takes four arguments, a macro environment, a form that potentially modifies this environment, a function `fn`, and an argument `x` to apply the function to. `With-env` updates the macro environment as specified by the form and then applies `fn` to `x` and the modified environment. `With-env` returns whatever `fn` returns.

For example, `flet-mexp` uses `all-lambda-mexp`, which calls `lambda-mexp`, to macroexpand the local function definitions. It then uses `with-env` to create the altered macro environment

that corresponds to the `flet` and uses `all-mexp` to macroexpand the body of the `flet` in this new environment. `Labels-mexp` operates the same way as `flet-mexp` except that it uses the altered environment when macroexpanding the local function definitions.

Before discussing how `with-env` works, it is useful to note that all the code in Figure 1 is portable Common lisp. Unfortunately, this is not true for `with-env`.

Evaluation and Macro Environments

Lisp evaluation is controlled by an evaluation environment that specifies the values of variables and what functions and macros symbols refer to. In order not to over constrain implementors, Common Lisp documentation says almost nothing about this environment. CLtL1 merely describes a couple of situations where evaluation environments appear. In particular, if an `*evalhook*` function is specified, then whenever an attempt is made to evaluate something, the `*evalhook*` function will be called and passed the form to evaluate and an appropriate evaluation environment. Just about the only thing that this environment can be used for is as an argument to the function `evalhook`, which can be used to resume evaluation of the form passed to the `*evalhook*` function. (Stepping and tracing tools can be implemented using `*Evalhook*` functions and `evalhook`.)

The expansion of macros is controlled by a macro environment that specifies which symbols refer to macros and which do not. As above, Common Lisp documentation says almost nothing about this environment. CLtL1 merely describes two situations where macro environments appear. When a macro function (a function that implements a macro) is called, it is passed the macro environment that is appropriate for the place in the source program where the macro call appeared. The function `macroexpand` can be passed a macro environment that specifies the context that should be used when expanding the specified form. This is needed so that a macro (e.g., `setf`) can call `macroexpand` on part of its argument and get the results that are appropriate for the place where the original macro call appeared.

`With-env` has to modify the macro environment given to it to reflect the changes implied by the specified form. This is difficult to do, because CLtL1 does not provide any functions for creating or inspecting either evaluation or macro environments. All that it provides is a few obscure functions that are not intended to be at all relevant to our task.

Solving the Puzzle

For those that delight in getting Lisp to do things that the builders of the language never dreamed that you would want to do, successfully extending a macro environment is a puzzle much too interesting to pass up. The key to solving the puzzle is realizing that whatever a macro environment is, the Lisp evaluator succeeds in extending it appropriately whenever it encounters an `flet`, `labels`, or `macrolet`. We can get the evaluator to make the modification we want, by simply passing it the form we have.

Unfortunately, there is a problem with this simple idea. The evaluator descends into an expression creating appropriate evaluation and macro environments, but while you can specify an initial execution environment with `evalhook`, there is no way to specify an initial macro environment. In contrast, you can specify an initial macro environment to `macroexpand`, but `macroexpand` does not descend into an expression and therefore does not lead to the construction of an extended macro environment. As a result, while it is easy to get the evaluator to extend an evaluation environment, it is not clear how to get it to extend a macro environment for us.

The function `evalhook` can be used to both prime the evaluator with an initial evaluation environment and to access an extended evaluation environment. For example, suppose you have an evaluation environment *E*.

```
(evalhook '(macrolet ((h (a) '(1+ ,a))) t)
          #'(lambda (x env)
              (print env)
              (eval x))
          nil
          E)
```

```
(defmacro grab-env (fn x
                    &environment env)
  `',(funcall fn x env))
```

```
(defun aug-env (env form fn x)
  (evalhook `(,@ form (grab-env ,fn ,x))
            nil nil env))
```

Figure 2: Manipulating execution and macro environments.

```
#+(or :SYMBOLICS :AKCL :CORAL :FRANZ-INC)
(defun with-env (env form fn x)
  (aug-env (convert-env env) form fn x))

#+(or :SYMBOLICS :AKCL)
(defun convert-env (env)
  env)
```

```
#+:CORAL
(defun convert-env (env)
  (list nil env nil nil nil nil))

#+:FRANZ-INC
(defun convert-env (env)
  (list nil env nil nil))
```

Figure 3: Extending macro environments that are similar to execution environments.

shows you the result of extending E with the information in the `macrolet`. Exactly what this environment is like differs radically from one Common Lisp implementation to another. (If you want to type the expression above at top level, you can use the value `nil` for E, which stands for the top-level environment.)

What we need is some way to convert evaluation environments into macro environments and vice versa. The first of these conversions can be done straightforwardly with a macro, by utilizing an `&environment` argument. In particular, the macro `grab-env` in Figure 2 applies a function to an argument `x` and the macro environment corresponding to the evaluation environment in effect at the place where the macro call appears. It then returns whatever the function returns. For example, the expression

```
(grab-env #,#'(lambda (x env) (print env))
          nil)
```

will show you the macro environment corresponding to the place where the expression appears. If you type this at top level you will see the top-level macro environment. If you type it nested in a form you will see a more complex macro environment.

You can use `evalhook` and `grab-env` together to access the macro environment that corresponds to extending an evaluation environment.

```
(evalhook '(macrolet ((h (a) '(1+ , a)))
             (grab-env #,#'(lambda (x env)
                             (print env))
                       nil))
          nil nil E)
```

shows you the macro environment that results from extending E with the information in the `macrolet`. Exactly what this is like differs radically from one Common Lisp implementation to another. Further, while it is possible that this macro environment will be the same as the extended evaluation environment, there is no guarantee that they will be anything like each other.

The function `aug-env` in Figure 2 embodies the trick shown above. It applies a function to an argument and the macro environment that results from extending an initial evaluation environment as specified by the given form. `Aug-env` then returns whatever the function returns. For example,

```
(aug-env E
         '(macrolet ((h (a) '(1+ ,a))))
         #'(lambda (x env) (print env))
         nil)
```

is identical to the last example, in that it constructs exactly the same form and evaluates it in the same environment.

It does not appear that there is any implementation independent way in CLtL1 to convert a macro environment into an evaluation environment. However, in a given implementation it is usually easy to do. In particular, I used the expressions shown above to inspect macro and execution environments in various implementations of Common Lisp, and determined that in most of them, execution and macro environments are very similar. When this is the case, the function `with-env` needed in Figure 1

```
#+:LUCID
(defun with-env (env form fn x)
  (aug-env nil
           form
           #'with-appended-env
           (list env fn x)))
```

```
#+:LUCID
(defun with-appended-env (z delta)
  (let ((env (car z))
        (fn (cadr z))
        (x (caddr z)))
    (funcall fn x (append delta env))))
```

Figure 4: Extending macro environments that are stacks implemented as lists.

```
(defun with-env (env form bind fn body)
  (funcall fn body
    (if (eq form 'macrolet)
        (augment-env env :macro
                     (mapcar #'parse bind))
        (augment-env env :function
                     (mapcar #'car bind)))))
```

```
(defun parse (b)
  (list (car b)
        (parse-macro (car b)
                     (cadr b)
                     (cddr b)
                     env)))
```

Figure 5: Extending macro environments in CLtL2.

can be directly implemented using **aug-env** as shown in Figure 3.

In particular, in two of the Common Lisps I looked at, execution and macro environments are identical. In the other two, an execution environment is a list, one of whose components is a macro environment. Therefore in all four cases, converting a macro environment into an equivalent execution environment is trivial.

In the fifth Common Lisp I looked at, the relationship between execution environments and macro environments is obscured by the use of implementation-specific data structures. However, I noticed that in this implementation a macro environment is a stack implemented as a list. This opens up an alternate approach to modifying a macro environment.

Rather than converting a macro environment to an execution environment and then letting the evaluator extend it, one can determine what extension should be applied and do the extension yourself. This depends on knowing how extension can be done.

If a macro environment is a stack implemented as a list, then a macro environment can be extended using **append**. Further, if the top-level macro environment is the empty stack **nil**, than the change introduced by a form can be determined by determining what macro environment is created by evaluating the form at top level. These observations lead to the implementation of **with-env** shown in Figure 4.

In the figure, **aug-env** is used to determine the change in the macro environment that results from evaluating the specified form in isolation. The function **with-append-env** then combines this change with the original macro environment, creating an extended macro environment, which is passed to the specified function.

Improvements In CLtL2

The problem posed above can be solved in a portable way in CLtL2, because CLtL2 specifies a suite of functions that can extract information from and add information to environments. As a result, a macro environment can be directly extended as shown in Figure 5.

The CLtL2 function **augment-env** is used to add information into an environment. In the figure, it is used to add specifications for the macro definitions in a **macrolet** or the function definitions in an **flet** or **labels**. The function **parse** uses the CLtL2 function **parse-macro** to convert the local macro definitions in a **macrolet** into the form expected by **augment-env**.

Conclusion

The function **macroexpand-all** is a tool that can be useful for anyone who writes or uses complex macros. The code for **macroexpand-all** is primarily implementation independent. However, in order to use it, you have to supply a definition of the critical function **with-env**. If you

are using one of the implementations of Common Lisp where `macroexpand-all` has already been tested, this has been done for you. If not, you have three choices.

First, by inspecting evaluation and macro environments in the Common Lisp you use, you should be able discover enough about the structure of these environments, in order to implement `with-env` in a way that is analogous to Figure 3 or 4.

Second, you can include a vestigial definition of `with-env`, such as

```
(defun with-env (env form fn x)
  (declare (ignore form))
  (funcall fn x env))
```

and live with the fact that `macroexpand-all` will occasionally produce incorrect results. This might be a reasonable thing to do if you are going to use `macroexpand-all` merely as a debugging aid. However, it is not tolerable if you intended to use `macroexpand-all` as part of a macro definition.

Third, you can wait until you have an implementation of CLtL2 available and use the implementation of `with-env` shown in Figure 5.

In addition to being a useful tool in its own right, Figure 1 can be viewed as the minimal skeletal structure on which more complex code walkers can be written. Everything shown in the figure is necessary, because a code walker must expand all the macro calls in an expression in order to work. The code has to be extended in order for the walker to keep track of information about an expression such as what variables are bound. In CLtL2, this is much easier than in CLtL1, because the code walker can retrieve information from the environments used by the implementation, rather than keeping track of it redundantly.

Obtaining Macroexpand-All

The example above is written in Common Lisp and has been tested in several different Common Lisp implementations. The full source is shown in Figures 1–5. In addition, the source can be obtained over the INTERNET by using FTP. Connect to `MERL.COM` (INTERNET number 140.237.1.1). Login as "anonymous" and copy the files shown below.

In the directory `/pub/lptrs/`

`mexp-code.lisp`	source code
`mexp-test.lisp`	test suite
`mexp-doc.txt`	brief documentation

References

[1] Curtis, P., "Algorithms", *ACM Lisp Pointers*, 3(1):48–61, March 1990.

[2] Pitman, K.M., "Special Forms in Lisp", in *Proc. 1980 Lisp Conference*, 179–187, August 1980.

[3] Steele G.L.Jr., *Common Lisp: the Language*, Digital Press, Maynard MA, 1984.

[4] Steele G.L.Jr., *Common Lisp: the Language*, second edition, Digital Press, Maynard MA, 1990.