An Introduction to Macros

jens.jensen@stfc.ac.uk

June 11, 2021

Outline

- Mostly covering both Common and Emacs Lisp
  - Otherwise you should demand your money back (you'll see why)
- Macros receive code unevaluated
- Then macros (pre)process the code
- And the processed code is then evaluated
- In other words, *macros map code to code*

Macros' Processing

- Macros *expand* prior to evaluation of code
- Unlike inferior languages, macros apply both at runtime and compile time
  - (Common Lisp has a compile-time-only macro)
  - Some implementations JIT-compile everything anyway
- Compilers can of course eliminate unreachable code

```
(defmacro brittle-add-5 (x) (+ x 5))
(brittle-add-5 7)
;; 12
```

- How is this different from `defun`?
- It would fail on `(brittle-macro (+ 2 3))` because `(+ 2 3)` is not numeric (it is a list)
  - The macro is called with `x` bound to the list `(+ 2 3)`
  - It is expected to process this list
- In contrast, if it were a defun this is what would happen:
  - Arguments would be evaluated `(+ 2 3)` $\rightarrow$ 5
  - The *function* would be called with `x` bound to 5
  - The function would return 12, as before

Fixing it

```
(defmacro add-5 (x) (list '+ x 5))
```

Consider the evaluation of `(add-5 (+ 2 3))`.

- First, the macro is called with `x` bound to the list `(+ 2 3)`
- The macro then constructs the list `(+ (+ 2 3) 5)`
- This list is returned to Lisp from the macro
- Lisp then evaluates the result, returning `10`.

A function would do it differently (reminder):

- First, Lisp evaluates `(+ 2 3)` to get `5`.
- Lisp then binds `x` to `5` and calls the function body
- If the above were a function, it would return `(+ 5 5)` as a list of three elements

Slightly more interesting Example: if

If a predicate evaluates to True, evaluate the 'then' branch, otherwise evaluate the 'else' branch. The other branch is not evaluated at all.

- (Normal) `if` is a *special form*, not a function
- A *function* would have all its arguments evaluated before it was called

Simple Test code

```
(defun test-my-if ()
  (cons
    (let ((a 0) (b 7))
      ;; False, but needs evaluation to become false
      (and
(eql (my-if (endp '(a b)) (incf a) (incf b)) 8)
(eql a 0) (eql b 8)))
    (let ((a 0) (b 7))
      ;; True
      (and
(eql (my-if t (incf a) (incf b)) 1)
(eql a 1) (eql b 7)))))
```

- Replacing `my-if` with `if` makes the function return `(t . t)`
- This will test any `my-if`, whether a macro or function
  - If you rewrite `my-if`, it will happily test the rewritten one
    - No need to reevaluate `test-my-if` (unless it has been compiled?)

First approach

We need:

```
(defmacro my-if (pred then else) ...)
```

Initial attempt:

- *Evaluate* the pred during macro expansion with `eval`
- Macro then returns `then` or `else` (as yet unevaluated)
- Which will then be evaluated subsequently as normal

Can we do (pseudo code)

```
...  (if (eval pred) then else)
```

except of course it then needs `if` to do `my-if`.

A better approach to `my-if`

Restructure, don't evaluate

- `(and pred then)` would do the trick for the `then` clause alone
  - Problem solved, if there were no `else` clause
- Of course this relies on the `and` special form...

```
(defmacro my-if (pred then else)
  ;; Doesn't work yet, but it illustrates the solution
  (or (and pred then) else))
```

Finally, my-if

- We need to construct the code/list that is returned
- This is done by the macro, using the constituents

```
(defmacro my-if (pred then else)
  "Three part if: evaluate then if pred evaluates to true, c
  (list 'or (list 'and pred then) else))

(test-my-if)
;; (t . t)
```

Shortcomings of the `my-if`

- All larger macros will need some element of code/list construction
- No scope for `progn`
  - Exactly one `then` statement
  - Exactly one `else` statement

Fixing the list construction - backquote

- Like `'(a b)` is a short hand for `(quote (a b))`
  - which *evaluates* to `(a b)`
- *Backquotes* are short hand (in fact, macros!) for constructing lists:
  - Inside the list, comma inserts the value of a variable into the result
  - Inside the list, `,@` splices a variable containing a list into the result

```
`(foo . bar)
;; (foo . bar)
(let ((a 1) (b '(1 2)))
  `(foo ,a bar ,b baz ,@b quux))
;; (foo 1 bar (1 2) baz 1 2 quux)
```

Backquote

- Backquote is not just for macros, it works any time you want lists!
- Interpolated lists must be proper lists (not dotted)
  - (Unless it's at the end where CDR can be not NIL)

```
(let ((a 1) (b '(1 2)))
  `(foo ,a bar ,b baz ,@b quux))
;; (foo 1 bar (1 2) baz 1 2 quux)
(let ((a 1) (b '(1 2)))
  ;; Backquote above could expand to (in CL)...
  (concatenate 'list (list 'foo a 'bar b baz) b (list 'quux
```

Backquote magic

- Interpolation works on any sexp, not just symbols that name variables:

```
(let ((a 1) (b 2))
  `(fred ,a wilma ,b barney ,(+ a b) betty ,@(list b a)))
;; (fred 1 wilma 2 barney 3 betty 2 1)
```

Fixing `my-if`

We can now tidy up `my-if`:

```
(defmacro my-if (pred then else)
  "If pred evaluates to true, evaluate then, otherwise evalu
  `(or (and ,pred ,then) ,else))
```

- Using short-circuit features of `and` and `or`
- This is still a bit like using `if` to implement `my-if`

Another example - timing code

- ELisp doesn't have a `time` function
- (adapted from last year's Advent of Code)

```
(defmacro time (&rest code)
  "Return the wallclock runtime of code.  Returns a cons of
  `(let* ((now (current-time))
  (retval (progn ,@code))
  (then (time-subtract (current-time) now)))
     (cons (+ (ash (nth 0 then) 16) (nth 1 then)
      (/ (nth 2 then) 1000000.0)
      (/ (nth 3 then) (expt 10. 12)))
   retval)))
```

`time` macro

- Notice the lambda list use of `&rest` to give us `progn`
- These are ELisp specific, CL macros have a `&body` to capture the body
  - `&body` and `&rest` are functionally equivalent but `&body` indicates intent
- These *do* have implicit `progn`
- Using named variables inside the backquote is not usually a good idea
  - Because they run at execution time, after macro expansion has finished
  - Sometimes it's required, sometimes it's OK, sometimes it's forbidden

Named variables (anaphora examples)

```
(defmacro awhile (test &rest body)
  "Anaphoric while; the value of the test is available as ':
  `(let ((it ,test))
     (while (setq it ,test) ,@body)))

(defmacro awhen (test &rest body)
  "Anaphoric when"
  `(let ((it ,test)) (when it ,@body)))
```

Anaphoric when/if example

```
(awhile (next-case *state*) (print it))
```

- Instead of

```
(let (m)
  (while (setq m (next-case *state*))
    (print m)))
```

Temporary variables 1

- In *functions* (lexical) temp vars will just shadow anything else with the same name
- In macros, naming t.v.s is potentially bad:

```
(defmacro run-twice (expr)
  "(bad impl) Run an expression twice and return a cons of t
  `(let ((a ,expr) (b ,expr))
    (cons a (equal a b))))
```

Temporary variables 2

- Solution: generate the variable names in the macro

```
(defmacro run-twice (expr)
  "Run an expression twice and return a cons of the result a
  (let ((var1 (gensym)) (var2 (gensym)))
    `(let ((,var1 ,expr) (,var2 ,expr))
       (cons ,var1 (equal ,var1 ,var2)))))
```

- Note how the *names* of the temporary variables are held in `var1` and `var2`
- These will never conflict with any variable named in `expr`

Consider this implementation of a `pushnew` macro:

```
(defmacro pushnew (item place &optional test)
  "Bad pushnew"
  `(unless (funcall (or test #'member) ,item ,place)
     ;; call returns updated place, from push
     (push ,item ,place)))
```

Code passed in as `item` and `place` get duplicated so could be called twice.

We will return to `pushnew` later. . .

Destructuring

- *Destructuring* in CL makes it easy to extend the language:
- The lambda list can be a *structure*:

```
(defmacro my-dolist
  ((var-name list &optional final-form) . body)
  ...)
```

In a call `(dolist (i '(a b c) (frob)) (let ((x (blarf i))) ...))` the macro will expand with `var-name` bound to `i`, `list` is bound to `(quote (a b c))`, `final-form` is bound to `(frob)` (as yet unevaluated).
Note the `.` before the body variable, ensuring that body binds to everything else in the `dolist` statement - as opposed to just capturing the next single statement. Thanks to the dot, `dolist` has implicit progn (in its body):
`(dolist (a b) c d e) == (dolist (a b) . (c d e))`

Destructuring 2

Destructuring must have the exact structure, or it won't match.
(We will see a version later not using destructuring)

```
(let ((stuff '(var-name (1 2 3 4))))
  ;; Doesn't work, stuff is not expanded to match
  (dolist stuff (print var-name)))
```

This is correct:

```
(dolist (var-name '(1 2 3 4)) (print var-name)))
```

Final notes on destructuring

- Destructuring macros are cleaner than the `(let ...)`
  - Bindings in Lambda list are valid only during macro expansion
- Destructuring can introduce *declarations* on their parameters

```
(defmacro my-dolist ((var my-list) . body)
  (declare (type symbol var) (type list my-list))
  ...)
```

- Common Lisp has `destructuring-bind` available for general code

dolist 1

- Let's use this to write a `dolist` macro (for CL initially):

```
(defmacro my-dolist ((var lst &optional final) . body)
  ;; preamble
  ;; build-expr
  )
```

as in
`(my-dolist (i '(1 2 3)) (print i))` where during expansion,

- `var` is bound to `i`
- `lst` is bound to `(quote (1 2 3))`
- `final` is bound to `NIL`
- `body` is bound to `((print i))`

dolist 2

- We need a variable to hold the list
  - Because the list is bound to lst during macro expansion
  - But the binding is not active when the code is to be run

```
(defmacro my-dolist ((var lst &optional final) . body)
  (let ((list-var (gensym)))
    ...))
```

- The `lst` must be evaluated only once
  - In case it has side effects
  - E.g. `(my-dolist (i (expensive-read-returning-list)) (print i))`

dolist 3

Full `dolist` (CL version), using the `tagbody` primitive (very low level construct) to demonstrate code without using other macros

```
(defmacro my-dolist ((var lst &optional final) . body)
  (let ((list-var (gensym)))
    `(let (,var (,list-var ,lst))
       (tagbody
loop
  (when (endp ,list-var) (go done))
  (setq ,var (car ,list-var) ,list-var (cdr ,list-var))
  (progn ,@body)
  (go loop)
done)
       ,final)))
```

dolist 4

Testing `my-dolist`:

```
(my-dolist (i '(1 2 3) 'foo) (print i))
1
2
3
FOO

(my-dolist (i nil 'safe) (destroy-universe))
SAFE
```

dolist 5

Back to Emacs; temporary version using named temporaries - first attempt:

```
(defmacro my-dolist (&rest everything)
  (let ((var (caar everything))
(lst (cadar everything))
(final (caddar everything))
(body (cdr everything)))
    `(let (,var (list-var ,lst))
       (while list-var
 (setq ,var (car list-var) list-var (cdr list-var))
 ,@body)
      ,final)))
```

- Emacs macros do not have destructuring, so we have to pick the structure apart by hand

dolist 6

Final version for Emacs Lisp:

```
(defmacro my-dolist (&rest everything)
  (let ((var (caar everything))   ; iterator var
(list-var (gensym)))      ; name for list
    `(let (,var (,list-var ,(cadar everything)))
       (while ,list-var
 ;; pop list into var
 (setq ,var (car ,list-var) ,list-var (cdr ,list-var))
 ,@(cdr everything))      ; body
       ,(caddar everything))))    ; finally eval final
```

- This is likely different from the one in the CL package

Debugging macros

What is the macro doing:

```
(macroexpand-1 '(incf (aref b 3)))
;; (setf (aref b 1) (1+ (aref b 1)))
```

- It does not matter that `b` is not bound, as the expression is not evaluated
- `macroexpand-1` needs the car to be a macro or it does no magic
- `macroexpand` will do a complete expansion until the expression doesn't change

Back to pushnew

```
(macroexpand-1 '(pushnew 'x a))
(cl-pushnew 'x a)
(macroexpand-1 '(cl-pushnew 'x a))
(if (memql 'x a) (with-no-warnings a) (setq a (cons 'x a)))
(macroexpand-1 '(cl-pushnew (incf x) (aref b 2)))
(cl-callf2 cl-adjoin (incf x) (aref b 2))
```

Compilation

- Macros must be defined and available at compile time

(for another talk, or two)

- `macroexpand-hook` (CL)
- Local and recursive macros (`macrolet` instead of `flet`)
- Nested backquotes `` `(...foo `(,fred ,,ernie ,@,blezp)) ``
- Compilation and macros
  - Compiler macros (CL)
- Reader macros (CL?)
  - Like C++'s `operator""` only, of course, more powerful
- Environments (CL) (when the macro is *defined* vs *expanded*)
  - Environments provide (non-CLOS) *introspection*