Macros in Common Lisp

Christopher Mark Gore

http://www.cgore.com

cgore@cgore.com

@cgore

Thursday, May 1, AD 2014

Getting Started with Common Lisp

1. Install Linux.

   ```
   http://aptosid.com
   ```

2. Install SBCL and some libraries.

   ```
   apt-get install sbcl{,-doc,-source} \
   cl-{asdf,cffi}
   ```

3. Install Emacs and SLIME *(Not strictly required.)*

   ```
   apt-get install emacs{,-goodies-el} cl-swank \
   cl-swank slime common-lisp-controller
   ```

Since my new employer bought me a shiny new MacBook Pro:

```
brew install sbcl
```

Lisp Macros Are Vastly Superior To:

1. Not having macros at all.
2. Vim/Emacs[a] macros.
3. Microsoft Word/Excel macros.
4. C pre-processor macros.
5. Scheme “hygenic” macros.

---

[a] Excluding `defmacro` in Elisp, which is equivalent.

Defun versus Defmacro

We can define functions with `defun`, and we can define macros with `defmacro`. These two definitions achieve the same goal in quite different ways.

```
(defun add-one (x)
  (+ 1 x))
(defmacro add-one (x)
  `(+ 1 ,x))
```

Why Not Just Functions?

You shouls prefer functions instead of macros if they can do the job. There are lots of operations that can only be done with macros: functions can't make it happen.

- Linguistic extensions
- Preventing the execution of the arguments
- Controlling the execution of the arguments
- Rewriting the arguments before they are executed
- Natural DSLs

Backquote

Backquotes let you quote out a list, but selectively evaluate some of the list elements. This shorthand notation makes it really easy to build up complicated s-expressions on the fly.

```
(let ((a 12)
      (b 'x)
      (c '(q r s)))
  `( a  b   c)          ; Becomes '(a  b c)
  `(,a  b   c)          ; Becomes '(12 b c)
  `( a ,b   c)          ; Becomes '(a  x c)
  `( a  b  ,c)          ; Becomes '(a  b (q r s))
  `( a  b ,@c)          ; Becomes '(a  b  q r s)
  `( a  b ,(+ 1 2 3))   ; Becomes '(a  b 6)
```

Nested Backquotes

Backquotes are a bit confusing when they are nested inside each other. It is generally best to avoid it as much as possible, but sometimes it is necessary. The general rule is they apply inside-to-outside.

```
(let ((a 'x))
  (eval `(let ((a 'y)
               (b ',a))
           `(,a ,b)))) ; Returns '(y x)
```

How Macros Work

Macros have two main features:

1. Controlling, preventing, or manipulating the evaluation of their arguments.
2. Local expansion within the calling context.

If you really want a macro, it is because you need one of those abilities.

Controlling Argument Evaluation

```
(defmacro upside-down (first last)
  `(progn ,last
          ,first))
(upside-down
  (format t "The␣first␣shall␣be␣last~%")
  (format t "The␣last␣shall␣be␣first~%"))
```

This prints:

```
The last shall be first
The first shall be last
```

```
(let ((x 42))
  (upside-down (setf x (/ x 2))
               (setf x (+ x 2)))) ; 22, not 23.
```

A Weird If . . .

The Normal `if` in Common Lisp has the following basic form:

```
(if (some-conditional)
  (true-form)
  (optional-false-form))
```

For example:

```
(defun is-it-positive? (x)
  (if (< 0 x)
    (format t "It's␣positive!~%")
    (format t "It's␣not␣positive!~%")))
```

But what if we want a weird version of `if`?

. . . A Weird If . . .

What if we think this is the
ONE TRUE WAY TO WRITE IF STATEMENTS?

```
(weird-if (true-form)
          (some-conditional)
          (optional-false-form))
```

An infix-operator variant of if? Or something like that. For example:

```
(defun is-it-positive? (x)
  (weird-if (format t "It's␣positive!~%")
            (< 0 x)
            (format t "It's␣not␣positive!~%"))
```

. . . A Weird If

We can make it happen with a very simple macro.

```
(defmacro weird-if (true-form
                    conditional
                    &optional (false-form nil))
  `(if ,conditional ,true-form ,false-form))
```

Now we can use or `weird-if` everywhere!

```
(weird-if "true" (< 12 22) "false") ; Returns "true"
(weird-if "true" (< 44 22) "false") ; Returns "false"
```

Wordy If . . .

How about a `wordy-if`? Some people might actually think this one is a good idea, unlike the last one.

```
(defmacro wordy-if (conditional then &body body)
  (let* ((else-position (position 'else body))
         (true-body (subseq body 0 else-position))
         (false-body (when else-position
                       (subseq body
                               (1+ else-position)))))
    `(if ,conditional
       (progn ,@true-body)
       (progn ,@false-body))))
```

. . . Wordy If

What does that give us? Now we can write our conditionals as such:

```
(wordy-if (< 12 22)
  then
    (format t "This␣is␣obviously␣true.~%")
    (format t "But␣a␣good␣demonstration␣anyway.~%")
    "it␣is␣true" ; returns this string
  else
    (format t "This␣won't␣happen.~%")
    (format t "Neither␣will␣this.~%")
    "and␣this␣won't␣be␣returned")
```

Even Stranger and Wordier If's

We can make this `wordy-if` a lot more complex.

1. We could add `else-if` support.
2. We could add `otherwise`, to support a default case.
3. We could add `break`, to break out of the if block.
4. Or anything else you can think of.

Common Lisp macros give us the ability to write not just easy-to-use domain-specific languages (DSL's, their most typical real-world use), they give us the ability to easily write **whole new programming languages inside of Common Lisp itself, with no real challenge of writing a compiler or any other nonsense like Lex and Yacc.**

Local Expansion

Macros work in the local calling namespace, and can therefore capture variables from that namespace or pollute variables into that namespace.

- If that is what you want, then it is awesome.
- If that isn't what you want, then it can be a big headache and endless source of bugs, if you don't work with it correctly and pay attention to what you are doing.

The whole reason for "hygenic macros" and the like are because lots of people don't know how to correctly reason about variable pollution and capture.

Bad Variable Capture

Let's write a `swap` macro!

```
(defmacro bad-swap (x y)
  `(let ((temp))
     (setf temp ,x)
     (setf ,x ,y)
     (setf ,y temp)))
```

We are intentionally altering the `x` and `y` passed in, that's good. But what about temp? What if there already is a temp variable in the local scope? What if they try to do `(swap temp something)`? What if they try to do nested calls to `swap`? These are all broken.

Gensym

How do we avoid capturing a variable? By *generating brand new symbols.*[a] Each call to `gensym` makes a brand new uninterned symbol.

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

[a] In real life, just use `(psetf x y y x)`.

~~Polluting~~ Gifting the Namespace

We can add to the namespace intentionally. It's not pollution, it's a free gift! Just like all of those cigarette butts!

```
(defmacro aif (conditional t-action &optional nil-action)
  `(let ((it ,conditional))
     (if it ,t-action ,nil-action)))
(defmacro awhen (test-form &body body)
  `(aif ,test-form (progn ,@body)))
```

Now we could change (from our `wordy-if` macro):

```
(when else-position (subseq body (1+ else-position)))
```

Into:

```
(awhen else-position (subseq body (1+ it)))
```

WITH-* Macros

A very common use of "pollution" is special-purpose `with-*` macros. The best example of what these do is the canonical `with-open-file`.

```
(with-open-file (input "/some/file/path.txt")
(do ((line (read-line input) (read-line input nil 'eof)))
    ((eq line 'eof) "Reached␣end␣of␣file.")
  (format t "Read␣a␣line:␣~A~%" line))
```

You can define your own `with-db`, `with-kb`, `with-csv-file`, `with-json-file`, or whatever makes sense in your specific example.

- The macro does setup.
- The macro provides one or more resources to use.
- The macro does cleanup.

Macroexpand-1

Debugging macros can be a little bit hairy. One of the first set of tools to reach for is `macroexpand` and `macroexpand-1`. These two macros both expand out the macros but don't evaluate them, so you can view their expansions.

You typically want to use `macroexpand-1`, it expands one level deep.

```
(defmacro swap (x y)
  `(psetf ,x ,y
          ,y ,x))
(macroexpand-1 '(swap a b)) ; Expands to (psetf a b b a)
```

Macroexpand

Usually you don't want `macroexpand`, it expands all the way down the stack of macro calls, including macros you didn't make. This call `(macroexpand '(swap a b))` produced for me:

```
(LET* ()
  (MULTIPLE-VALUE-BIND (#:NEW644)
      B
    (LET* ()
      (MULTIPLE-VALUE-BIND (#:NEW645)
          A
        (PROGN (SETQ A #:NEW644)
               (SETQ B #:NEW645)
               NIL)))))
```

Destructuring Parameter Lists

TO DO

Macro-Related Things

These are all really cool topics, but each would really need a lot more time on their own to do them justice.

- Symbol macros.
- Reader macros.
- M-expressions.
- F-expressions.

Perhaps another day.

Questions?