

Lecture Notes: Macros

Macros in Lisp provide a very powerful and flexible method of extending Lisp syntax. They are much, much more powerful than `#define` macros in C: Lisp macros are a full-fledged code-generation system, while C `#define` macros are simple string substitutions. Although extremely powerful and useful, macros are also significantly harder to design and debug, and are normally considered a topic for the advanced Lisp developer.

Lisp functions take Lisp values as input and return Lisp values. They are executed at run-time. Lisp macros take Lisp code as input, and return Lisp code. They are executed at compiler pre-processor time, just like in C. The resultant code gets executed at run-time. Almost all the errors that result from using macros can be traced to a misunderstanding of this fact.

1. Basic Idea: Macros take unevaluated Lisp code and return a Lisp form. This form should be code that calculates the proper value. Example:

```
(defmacro Square (X)
  `(* ,X ,X))
```

This means that wherever the pre-processor sees `(Square XXX)` to replaces it with `(* XXX XXX)`. The resultant code is what the compiler sees.

2. Debugging technique: `macroexpand-1`

When designing a function, you can type a call to the function into the Lisp Listener (prompt), and see if it returns the correct value. However, when you type a macro "call" into the Lisp Listener, two things happen: first, the macro is expanded into its resultant code, and then that code is evaluated. It is more useful during debugging to be able to examine the results of these two steps individually. The function `macroexpand-1` returns the result of stage one of this process:

```
(macroexpand-1 '(Square 9)) ⇒ (* 9 9)
```

"If in doubt, `macroexpand-1` it out."

3. Purpose: To control evaluation of the arguments.

Since macros are so much harder to use than functions, a good rule of thumb is: *don't use `defmacro` if `defun` will work fine*. So, for example, there would be no reason to try to use a macro for `Square`: a function would be much easier to write and test. In Lisp, unlike in C, there is no need to use macros to avoid the very small runtime overhead of a function call: there is a separate method for that (the "inline" proclamation) that lets you do this without switching to a different syntax. What macros can do that functions cannot is to control when the arguments get evaluated. Functions evaluate all of their arguments before entering the body of the function. Macros don't evaluate any of their arguments at preprocessor time unless you tell it to, so it can expand into code that might not evaluate all of the arguments. For example, suppose that `cond` was in the language, but `if` wasn't, and you wanted to write a version of `if` using `cond`.

```
(defun Iff-Wrong (Test Then &optional Else)
  (cond
    (Test Then)
    (t Else)))
```

The problem with this is that it always evaluates all of its arguments, while the semantics of `if` are supposed to be that exactly one of the `Then` and `Else` arguments gets evaluated. For example:

```
(let ((Test 'A))
  (Iff-Wrong (numberp Test)
             (sqrt Test)
             "Sorry, SQRT only defined for numbers"))
```

will crash, since it tries to take `(sqrt 'A)`. A correct version, with behavior identical to the built-in `if` (except that the real `if` only has one required arg, not two), would be:

```
(defmacro Iff (Test Then &optional Else)
  "A replacement for IF, takes 2 or 3 arguments. If the first evaluates to
  non-NIL, evaluate and return the second. Otherwise evaluate and return
  the
  third (which defaults to NIL)"
  `(cond
    (,Test ,Then)
    (t      ,Else)))
```

A similar example would be NAND ("Not AND"), which returns true if at least one of the arguments is false, but, like the built-in `and`, does "short-circuit evaluation" whereby once it has the answer it returns immediately without evaluating later arguments.

```
(defmacro Nand (&rest Args)
  `(not (and ,@Args)))
```

4. Bugs:

- (A) Trying to evaluate arguments at run-time
- (B) Evaluating arguments too many times
- (C) Variable name clashes.

(A) Trying to evaluate arguments at run-time

Macros are expanded at compiler pre-processor time. Thus, the values of the arguments are generally not available, and code that tries to make use of them will not work. I.e. consider the following definition of `Square`, which tries to replace `(Square 4)` with `16` instead of with `(* 4 4)`.

```
(defmacro Square (X)
  (* X X))
```

This would indeed work for `(Square 4)`, but would crash for `(Square X)`, since `X` is probably a variable whose value is not known until run-time. Since macros do sometimes make use of variables and functions at expansion time, and to simplify debugging in general, *it is strongly recommended that all macro definitions and any variables and functions that they use at expansion time (as opposed to code they actually expand into) be placed in a separate file that is loaded before any files containing code that makes use of the macros.*

(B) Evaluating arguments too many times

Let's take another look at our first definition of the `Square` macro.

```
(defmacro Square (X) `(* ,X ,X))
```

This looks OK on first blush. However, try `macroexpand-1`'ing a form, and you notice that it evaluates its arguments twice:

```
(macroexpand-1 '(Square (Foo 2))) ⇒ (* (Foo 2) (Foo 2))
```

Foo gets called twice, but it should only be called once. Not only is this inefficient, but could return the wrong value if **Foo** does not always return the same value. I.e. consider **Next-Integer**, which returns 1 the first time called, then 2, then 3. (**Square (Next-Integer)**) would return $N*(N+1)$, not N^2 , plus would advance N by 2. Similarly, (**Square (random 10)**) would not necessarily generate a perfect square! With Lisp you have the full power of the language available at preprocessor time (unlike in C), so you can use ordinary Lisp constructs to solve this problem. In this case, **let** can be used to store the result in a local variable to prevent multiple evaluation. There is no general solution to this type of problem in C.

```
(defmacro Square2 (X)
```

```
  `(let ((Temp ,X))
```

```
    (* Temp Temp)))
```

```
(macroexpand-1 '(Square2 (Foo 2)))
```

```
⇒ (let ((Temp (Foo 2)))
```

```
    (* Temp Temp))
```

This is what we want.

(C) Variable name clashes.

When using **let** to suppress multiple evaluation, one needs to be sure that there is no conflict between the local variable chosen and any existing variable names. The above version of **Square2** is perfectly safe, but consider instead the following macro, which takes two numbers and squares the sum of them:

```
(defmacro Square-Sum (X Y)
```

```
  `(let* ((First ,X)
```

```
         (Second ,Y)
```

```
         (Sum (+ First Second)))
```

```
    (* Sum Sum)) )
```

This looks pretty good, even after macroexpansion:

```
(macroexpand-1 '(Square-Sum 3 4))
```

```
⇒ (LET* ((FIRST 3)
```

```
        (SECOND 4)
```

```
        (SUM (+ FIRST SECOND)))
```

```
    (* SUM SUM))
```

which gives the proper result. However, this version has a subtle problem. The local variables we chose would conflict with existing local variable names if a variable named **First** already existed. E.g.

```
(macroexpand-1 '(Square-Sum 1 First))
```

```
⇒ (LET* ((FIRST 1)
```

```
        (SECOND FIRST)
```

```
        (SUM (+ FIRST SECOND)))
```

```
    (* SUM SUM))
```

The problem here is that (**SECOND FIRST**) gets the value of the new *local* variable **FIRST**, not the one you passed in. Thus

```
(let ((First 9)) (Square-Sum 1 First))
```

returns 4, not 100! Solutions to this type of problem are quite complicated, and involve using **gensym** to generate a local variable name that is guaranteed to be unique.

Moral: even seemingly simple macros are hard to get right, so don't use macros unless they really add something. Both **Square** and **Square-Sum** are inappropriate uses of macros.

```
(defmacro Square-Sum2 (X Y)
  (let ((First (gensym "FIRST-"))
        (Second (gensym "SECOND-"))
        (Sum (gensym "SUM-")))
    `(let* ((,First ,X)
            (,Second ,Y)
            (,Sum (+ ,First ,Second)))
      (* ,Sum ,Sum))
  ))
```

Now

```
(macroexpand-1 '(Square-Sum2 1 First))
⇒ (LET* ((#:FIRST-590 1)
         (#:SECOND-591 FIRST)
         (#:SUM-592 (+ #:FIRST-590 #:SECOND-591)))
    (* #:SUM-592 #:SUM-592))
```

This expansion has no dependence on any local variable names in the macro definition itself, and since the generated ones are guaranteed to be unique, is safe from name collisions.