# Lisp Macros



## What are Macros?

- Lisp macros allow you to define operators that are implemented by transformation.
- The definition of a macro is essentially a function that generates lisp code.
  - A program that writes programs.
- Functions vs. macros:
  - A function produces *results*.
  - A macro produces *expressions* - which, when *evaluated*, produce *results*.



## Example: the macro nil!

- We want to write a macro *nil!*, which sets its arguments to nil.

(nil! x)

should be the same as:

(setf x nil)

Here's how we do it in CL:

> (defmacro nil! (var) (list 'setq var nil))

NIL!



## Macroexpansion

- What happens when we type the macro call (nil! x) into the toplevel?
- Lisp interprets *nil!* has a macro and:
  - builds the expression specified by the definition, (list 'setq var nil), then
  - evaluates that expression in place of the original macro call.
- What happens when the compiler discovers a call to nil!?
  - builds the expression specified by the definition, (list 'setq var nil), then
  - compiles that expression in place of the original macro call.

# Backquote

- *Backquote* is a special version of *quote*.
- It is used to create templates.
- It is used mostly in macro expressions.

  `(a b c) is equal to ‘(a b c)
- Backquote becomes useful only when it appears in combination with common **,** and comma-at **,@**.



- A backquoted list is equivalent to a call to *list* with the elements quoted.

  `(a b c) is equal to (list ‘a ‘b ‘c)
- When a comma appears before one of the elements of the list, it has the effect of canceling out the quote that would have been put in there.

  `(a ,b c ,d) is equal to (list ‘a b ‘c d)
- Commas work no matter how deeply they appear within a nested list:

```
> (setf  a 1  b 2   c 3)
3
> `(a  ,b  c)
(A  2  C)
> `(a  (,b  c))
(A   (2  C))
```



- And, they may even appear within quotes, or within quoted sublists:

```
> `(a  b  ,c  (‘,(+  a b c)) (+  a  b) ‘c ‘((,a  ,b)))
(A   B   3   (‘6)  (+  A B)   ‘C   ‘((1  2)))
```

- One comma counteracts the effect of one backquote, so commas must match backquotes.



# What is a backquote for?

- Backquote is usually used for making lists.
- The advantage of a backquote is that it makes expressions easier to read:

```
(defmacro nil! (var) (list ‘setf  var nil))
(defmacro nil! (var)  `(setf  ,var  nil))
```

# Comma-at: ,@

- Comma-at is useful in macros that have rest parameters representing, for example, a body of code.
- Suppose, we want a while macro that will evaluate its body so long as an initial test expression remains true:

```
> (let ((x 0))
    (while (< x 10)
      (princ x)
      (incf x)))
0123456789
NIL
```



# Example: while macro

- We define the macro while by using the rest parameter to collect a list of the expression in the body, then using comma-at to splice this list into the expansion:

```
(defmacro while (test &rest body)
    `(do () ((not ,test)) ,@body))
```

So (while (< x 10) (print x)(setq x (+ x 1))) becomes

```
(do (( not (< x 10)) (print x) (setq x (+ x 1)))
```



# Macro Design and Problems

- Writing macros is a distinct kind of programming, with its own unique aims and problems.
- When you start writing macros, you have to start thinking like a language designer.
- Two problems
  - *Variable capture*
  - *Multiple evaluations*



# Variable Capture

```
(defmacro ntimes (n &rest body)
  `(do ((x 0 (+ x 1)))
       ((>= x ,n))
     ,@body))
> (ntimes 10
    (princ ","))
.....
NIL
```

- Variable capture happens when a variable used in a macro expansion happens to have the same name as a variable existing in the context where the expansion is inserted:

```
(let ((x 10))
    (ntimes 5 (setf  x  (+  x  1)))
    x)
10
```



# Macro expansion

Macro expansion highlights the problem:

```
(macroexpand
  '(let ((x 10)) (ntimes 5 (setf  x  (+  x  1)))  x))
(let ((x  10))
 (do  ((x  0  (+  x  1)))
       ((>=  x  5))
    (setf x  (+  x  1)))
 x)
```



# Solution

Generate a unique name for the variable introduced by the macro.

(gensym) returns a symbol that is guarenteed not to be in use.

```
 > (gensym)
 #:G0001
> (defmacro ntimes (n &rest body)
    (let ((g  (gensym)))
      `(do ((,g 0 (+  ,g  1)))
           ((>=  ,g  ,n))
         ,@body)))
```



# Multiple Evaluation

- Because the first argument is inserted directly into the do, it will be evaluated on each iteration.
- This mistake shows most clearly when the first argument is an expression with side-effects:

```
> (let ((v  10))
      (ntimes  (setf  v (-  v 1))
      (princ ".")))
.....
NIL
```

- Since v is initially 10 and setf returns the value of its second argument, this should print nine periods. In fact it prints only five.
- We need to look at the macroexpansion.



```
(let ((v  10))
  (do ((#:g002 0 (+ #:g1 1)))
      ((>= #:g002  (setf  v  (-  v  1))))
    (princ ".")
```

- On each iteration we compare the iteration variable not against 9, but against an expression that decreases each time it is evaluated.



# Solution

- Set a variable to the value of the expression in question before any iteration. This involves another gensym:

```
(defmacro ntimes (n &rest body)
  (let ((g (gensym))  (h (gensym)))
       `(let ((,h ,n))
            (do ((,g 0  (+  ,g  1)))
                ((>=   ,g ,h))
              ,@body))))
```