Lisp Macros

What are Macros?
- Macros are an interesting & powerful Lisp feature.
- They let you 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:
  
  `(setq x nil)`
  
  Here's how we do it in CL:
  
  `>(defmacro nil! (var) (list 'setq var nil)) NIL!`

Why this is cool
- It allows one to extend the language.
- It’s a free lunch.
- Using lots of little functions has features (+ modularity, +maintenance, +reusability) and drawbacks (-overhead)
- Compiling macro calls
  - The compiler, on seeing a macro, expands it and compiles in place the code the macro returns.
  - … expanding any embedded calls to macros, of course.
- Interpreting macro calls
  - In the interpreter, you can write you can write a self-replacing macro so that it is expanded only once.
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 the expression in place of 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 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 comma `, and comma-at `, @.

Backquote

- A backquoted list is equivalent to a call to list with elements quoted.
  - `(a b c) is equal to (list 'a 'b 'c)
- A comma before one of the elements of the list cancels 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 where 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.
Why backquote?
- Backquote is usually used for making lists.
- The advantage of a backquote is that it makes expressions easier to read.
- This is very useful in defining macros:
  (defmacro nil! (var) (list 'setf var nil))
  (defmacro nil! (var) `(setf ,var nil))

Comma-at: ,@
- Comma-at is useful in macros having 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)))

Example: Push and Pop
- The representation of lists as conses makes it natural to use them as pushdown stacks.
- This is done so often that CL provides two macros for the purpose, push and pop.
- Both are defined in terms of setf.
  (push obj lst)
  is the same as
  (setf lst (cons obj lst)
Example: Push and Pop

```lisp
> (setf stack nil)
NIL
> (push 'foo stack)
(stack (foo))
> stack
((foo))
(push 'bar stack)
Stack
(bar foo)
(push 'bar stack)
Stack
(bar foo)
```

Implementing push and pop

```lisp
(defun push (s)
  `(setf ,s (cons ,s (pop stack)))
)
```

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

- 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.
Variable Capture

(defmacro ntimes (n &rest body)
  `(do ((x 0 (+ x 1)))
       ((>= x ,n))
       ,@body)))

> (> (ntimes 10 (princ ".")))

NIL

Variable Capture

But...

> (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)

Variable capture solution

Generate a unique name for the variable introduced by the macro.
(gensym) returns a symbol that is guaranteed 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:

```lisp
> (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.

Multiple Evaluation

(\(\text{Solution}\))

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

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

```lisp
(let ((v 10))
  (do ((#:g002 0 (+ #:g002 1)))
      ((>= #:g002  (((>= #:g002 (setf v  (v  (-- v  1))))v  1))))
    (princ ".")
)
```
Don’t try this at home

- Macros provide a hacker’s paradise.
- Not for the novice...
- Lisp macros (and readmacros) have made it a good vehicle for exploring new language features and prototyping new languages.