

Macros in LISP

- ✓ Suppose `foo` is a function... Then

`(foo a b c d)`

evaluates its arguments, and calls `foo` with the values bound to `foo`'s parameters. Then

- ✗ the expressions in the body of `foo`'s function definition are evaluated, and...
- ✗ the value of the last one is returned as the value of the call to `foo`

- ✓ Suppose `foo` is a macro... Then

`(foo a b c d)`

calls the macro `foo` with the *unevaluated* arguments bound to `foo`'s parameters. Then

- ✗ “macroexpansion” occurs:
 - the expressions in the body of `foo`'s macro definition are evaluated, and...
 - the value of the last one is returned as *a LISP expression to be evaluated*
- ✗ the result of macroexpansion is evaluated, and its value is returned as the value of the call to `foo`

Macros in LISP

✓ Why use macros instead of functions?

- ✗ functions in LISP always evaluate all of their arguments; you may want a “function” that controls the evaluation of its arguments
 - example: creating a construct which introduces unevaluated symbols as local variables, like `let` or `dolist`
 - example: creating a construct which controls the course of evaluation of expressions, like `if` or `cond`
- ✗ functions always involve a function call; you may want to avoid this for reasons of efficiency and elegance
 - the result of macroexpansion is inserted directly into the code stream at compile time
 - this permits having compact, understandable source code that does not involve unnecessary function calls at run time

Macros in LISP

- ✓ when a macro is called:
 - ✗ macroexpansion occurs: a LISP expression is built as specified by the macro definition
 - ✗ then that LISP expression is evaluated “in place of” the original macro call
 - ✗ think of the macro as actually writing LISP code which is executed
- ✓ Example: a macro which sets its argument to 0

```
;; a macro which sets its argument to 0
(defmacro zero (x)
  (list 'setq x 0))
```

- ✓ Now when a call to `zero` is evaluated ...

```
USER: (zero var)
```

- ✗ the symbol `var` (not its value!) is bound to the parameter `x` in the definition of the macro `zero`
- ✗ macroexpansion: the body of the macro definition is evaluated, resulting in the list

```
(setq var 0)
```
- ✗ this expression is evaluated, setting `var` to 0 as desired

```
0
USER: var
0
```

Writing macros in LISP

- ✓ When writing a macro definition
 - ✗ think of what LISP expression would perform the desired operation
 - ✗ write the macro definition to construct this expression
- ✓ Example: write a macro `nif` which performs a 3-way conditional branch, depending on whether its first argument is negative, 0, or positive:

```
(nif test expr1 expr2 expr3)
```

will evaluate `test`, and then will evaluate and return the value of `expr1`, `expr2`, or `expr3` depending on whether the value of `test` was `<0`, `0`, or `>0` respectively

What LISP expression would do that?...

```
(cond
  ((< test 0) expr1)
  ((> test 0) expr3)
  ((= test 0) expr2))
```

```
(if (< test 0) expr1
    (if (= test 0) expr2
        expr3))
```

Writing macros in LISP

- ✓ write a macro definition for `nif` that will expand the call

```
(nif test expr1 expr2 expr3)
```

as

```
(if (< test 0) expr1  
    (if (= test 0) expr2  
        expr3))
```

- ✓ what LISP expression will construct this nested `if`?

```
USER: (list 'if (list '< 'test 0) 'expr1  
              (list 'if (list '= 'test 0) 'expr2 'expr3))
```

```
(if (< test 0) expr1 (if (= test 0) expr2 expr3))
```

- ✓ substitute parameters for `test`, `expr1`, `expr2`, `expr3` and make this the body of the definition of the macro `nif`...

```
(defmacro nif (test-expr neg zero pos)  
  (list 'if (list '< test-expr 0) neg  
        (list 'if (list '= test-expr 0) zero pos)))
```

- ✗ keep in mind that the arguments to `nif` will not be evaluated in the macroexpansion...
- ✗ but they may be evaluated when the result of macroexpansion is evaluated!

Checking macroexpansion

- ✓ in a macro call, the result of macroexpansion is an expression which is evaluated, and the result of that evaluation is the value of the macro call
- ✓ when debugging a macro definition, it is helpful to be able to inspect the result of macroexpansion, before final evaluation
- ✓ macroexpand is a built-in LISP function that does this

```
USER: (macroexpand '(zero var))  
(SETQ VAR 0)
```

```
USER: (macroexpand '(nif z (print z) (setq z 3) NIL))  
(IF (< Z 0) (PRINT Z) (IF (= Z 0) (SETQ Z 3) NIL))
```

```
USER: (macroexpand '(dotimes (i 4) (setq x (+ x i))))  
  
(LET ((I 0))  
  (DECLARE (TYPE (INTEGER 0 4) I))  
  (BLOCK NIL  
    (TAGBODY  
      #:G407 (COND (# #))  
              (TAGBODY (SETQ X #))  
              (PSETQ I (1+ I))  
              (GO #:G407))))
```

Backquote

- ✓ some macros expand to produce complex LISP expressions
- ✓ constructing a complex LISP expression using `list` can be tedious
- ✓ the “backquote” facility permits constructing a complex LISP expression by providing a template for the expression
- ✓ backquote `'` is like quote `'` in that both of them are used to quote lists
- ✓ however, in a backquoted list, any expression preceded by a comma `,` is considered to be *unquoted*, and is evaluated

```
USER: (setf name 'sandy)
SANDY
```

```
USER: '(this is name from san diego)
(THIS IS NAME FROM SAN DIEGO)
```

```
USER: '(this is name from san diego)
(THIS IS NAME FROM SAN DIEGO)
```

```
USER: '(this is ,name from san diego)
(THIS IS SANDY FROM SAN DIEGO)
```

```
USER: '(i gave ,name about (* 25 4) dollars)
(I GAVE SANDY ABOUT (* 25 4) DOLLARS)
```

```
USER: '(i gave ,name about ,(* 25 4) dollars)
(I GAVE SANDY ABOUT 100 DOLLARS)
```

Writing macros using backquote

```
;; a macro which sets its argument to 0
(defmacro zero (x)
  (list 'setq x 0))
```

```
;; defining the zero macro using backquote
(defmacro zero (x)
  `(setq ,x 0))
```

```
;; a macro which does a 3-way numerical branch
(defmacro nif (test-expr neg zero pos)
  (list 'if (list '< test-expr 0) neg
        (list 'if (list '= test-expr 0) zero pos)))
```

```
;; defining the nif macro using backquote
(defmacro nif (test-expr neg zero pos)
  `(if (< ,test-expr 0) ,neg
      (if (= ,test-expr 0) ,zero
          ,pos)))
```

Using backquote in macro definitions

- ✓ a common use of macros is to define “functions” that do not require quoting arguments
- ✓ quote can be combined with backquote to define such macros
- ✓ Example:
 - ✗ a function **FETCH** takes a 3-element list as pattern and returns facts in a database that match the pattern

```
USER: (fetch '(block-1 color ?))  
(BLOCK-1 COLOR RED)
```

- ✗ since **FETCH** is a function, its argument must be quoted to prevent evaluation
- ✗ for convenience, we can define a macro **FETCHQ** which quotes its argument and passes it to **FETCH**:

```
(defmacro fetchq (patt)  
  '(fetch ',patt))
```

```
USER: (fetchq (block-1 color ?))  
(BLOCK-1 COLOR RED)
```

```
USER: (macroexpand '(fetchq (block-1 color ?)))  
(FETCH '(BLOCK-1 COLOR ?))
```

Macro parameter lists

- ✓ as in parameter lists for functions, &optional, &key, &rest arguments can also be used in parameter lists for macros

- ✓ Example:

- ✗ a function **FETCH** takes a required 3-element list as pattern, and a database of facts as an optional second argument

- ✗ **FETCH** returns facts in the database that match the pattern

```
(defun fetch (patt &optional (database *default-db*)) ...
```

- ✗ for convenience, we can define a macro **FETCHQ** which quotes its first argument and passes it to **FETCH**, and also accepts an optional second argument, a symbol which has a database as value:

```
(defmacro fetchq (patt &optional (db '*default-db*))  
  '(fetch ',patt ,db))
```