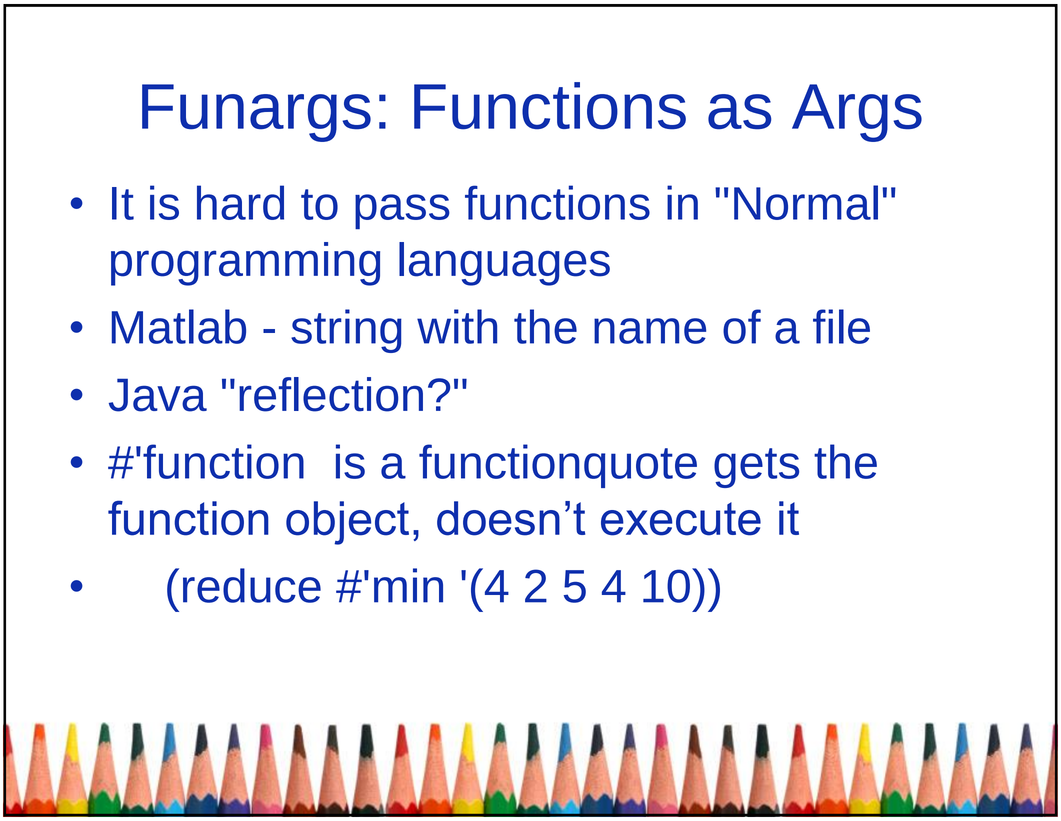
LISP 301

Advanced Lisping

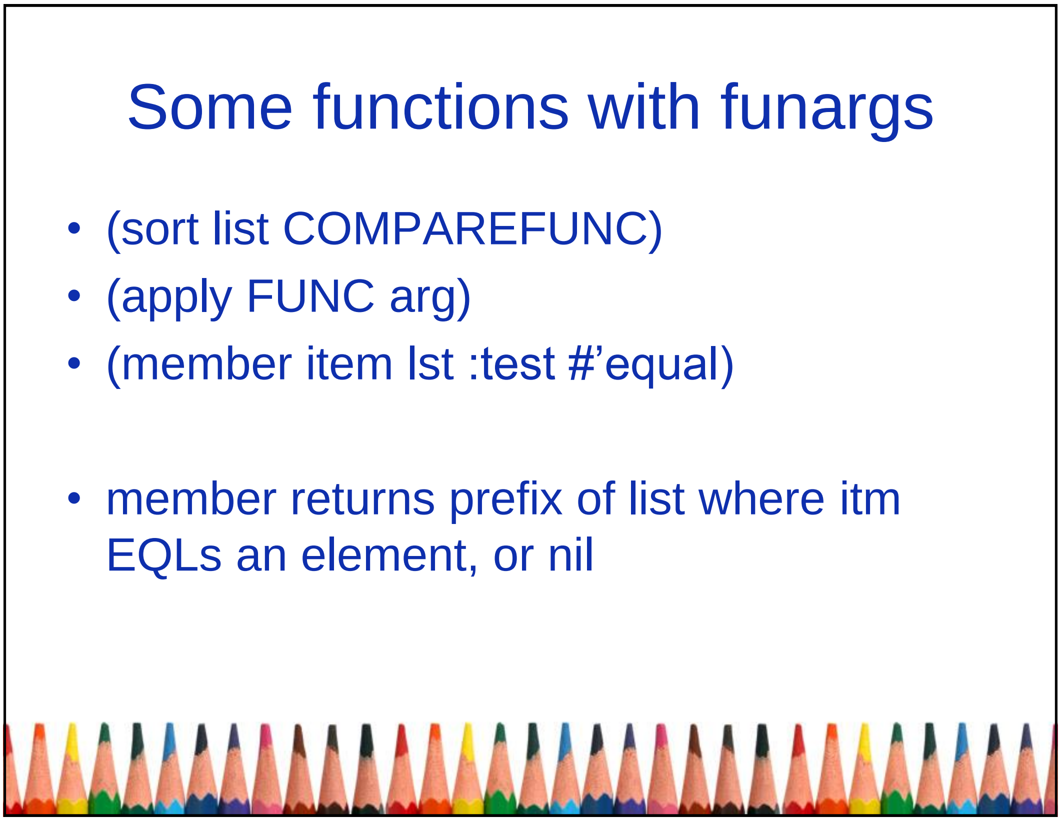
Advanced Lisp

- FUNARGS: Functions as Arguments
- Lambda and Let
- Macros
- Assoc Lists and hash tables
- Memoization: Trading Time for Space



Funargs: Functions as Args

- It is hard to pass functions in "Normal" programming languages
- Matlab - string with the name of a file
- Java "reflection?"
- #'function is a functionquote gets the function object, doesn’t execute it
-     (reduce #'min '(4 2 5 4 10))



Some functions with funargs

- (sort list COMPAREFUNC)
- (apply FUNC arg)
- (member item lst :test #’equal)

- member returns prefix of list where itm EQLs an element, or nil

Lisp has 4 different equals

- = for numbers
- eq for symbols and integers
- eql for strings
- equal for lists (slowest)

Sorting

- (sort ‘(5 4 3 2 1) #’<)
  - ->(1 2 3 4 5)
- What about this:
- (defvar chrismas
-       '((3 French hens) (10 lords a-leaping) (2 turtle doves)
-        (4 calling birds) (12 drummers drumming) (7 swans a-swimming)
-        (9 ladies dancing) (1 partridge in a pear tree) (6 geese a-laying)
-        (8 maids a-milking) (5 gold rings) (11 pipers piping)))

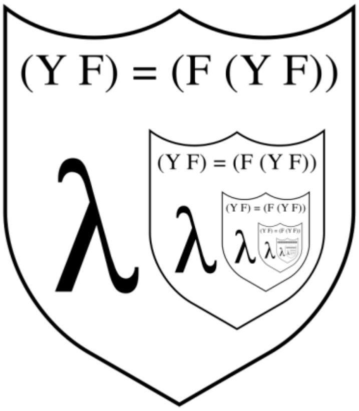
- What is the right function for sorting chrismas?

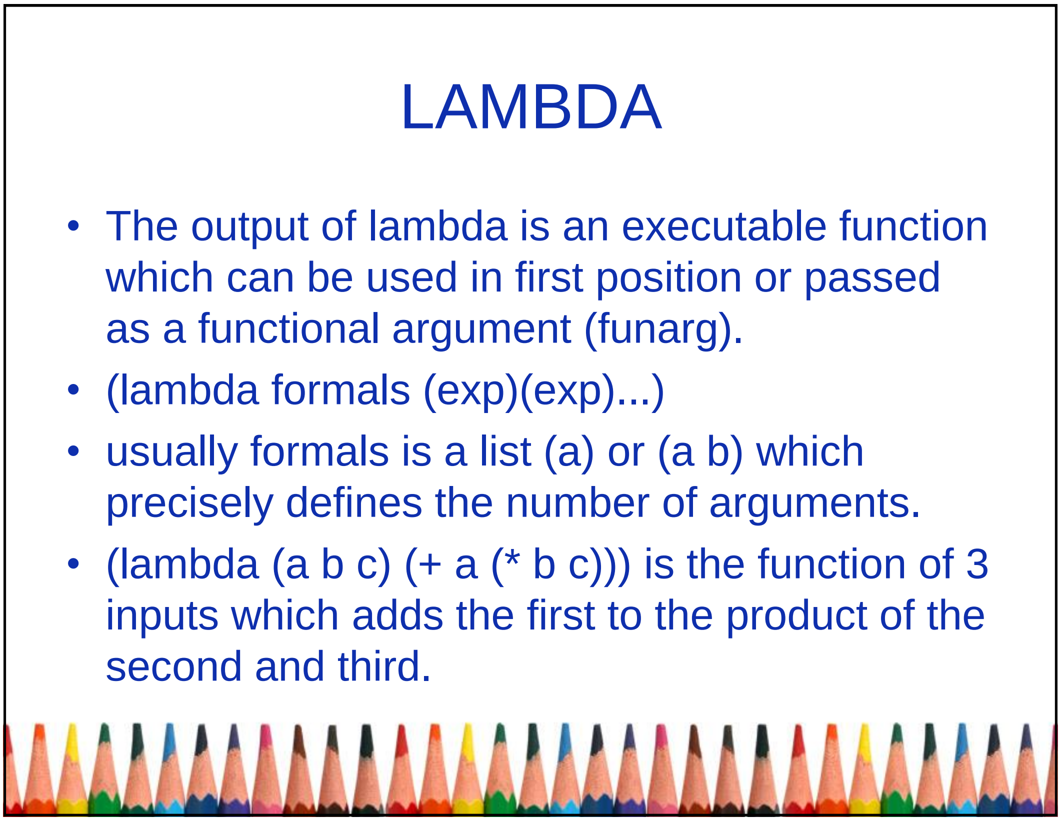
data as code

- In conventional languages source code is text and object code is binary.
- In LISP, source code is made of LISTS.
- (list ‘+ 3 4 5) -> (+ 3 4 5) as data.
- The EVAL function executes data as code:
- (eval ‘(+ 3 4 5)) -> 12
- DEFUN turns data into program, stores it in the symbol.

how? LAMBDA

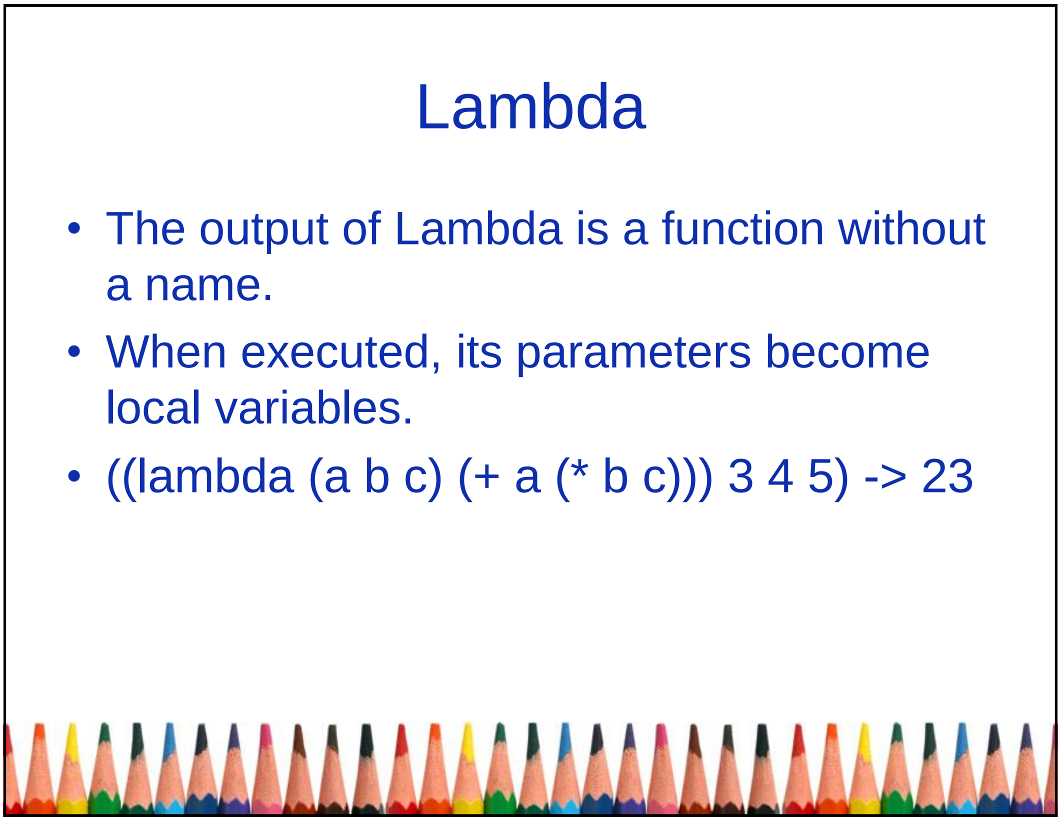
- Lambda is a powerful gizmo.
- It turns data into program
- You too can also become a Knight of the Lambda Calculus!





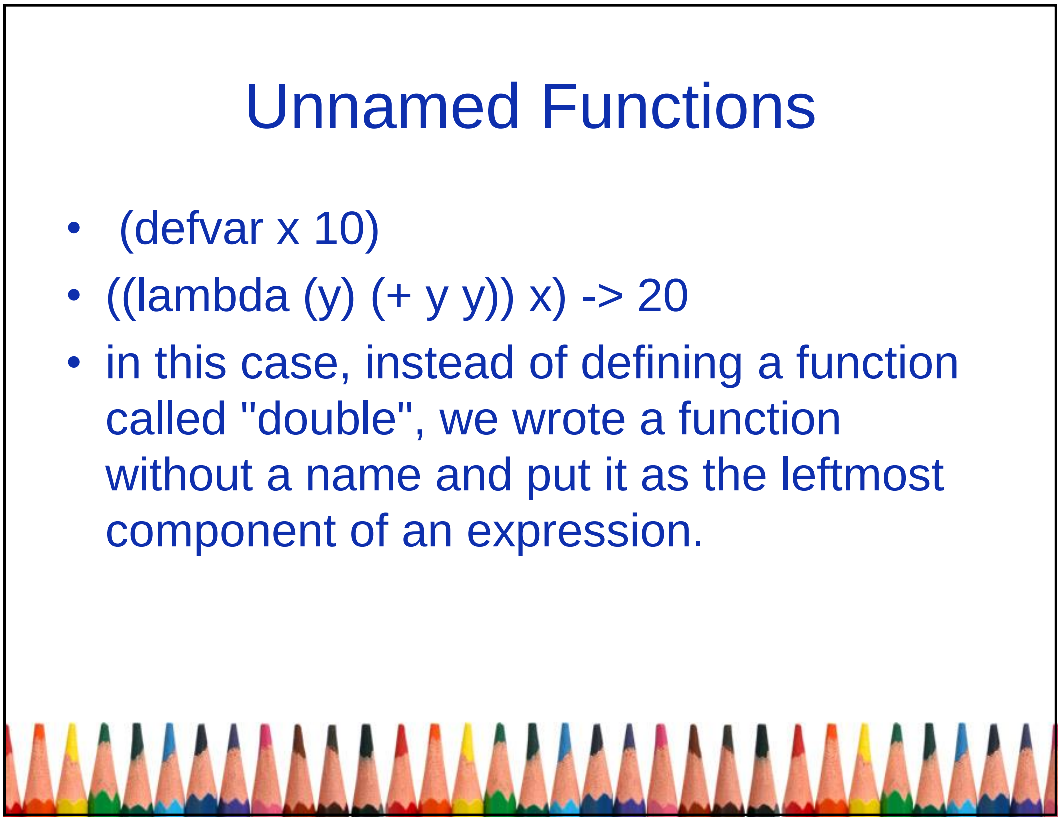
LAMBDA

- The output of lambda is an executable function which can be used in first position or passed as a functional argument (funarg).
- (lambda formals (exp)(exp)...)
- usually formals is a list (a) or (a b) which precisely defines the number of arguments.
- (lambda (a b c) (+ a (* b c))) is the function of 3 inputs which adds the first to the product of the second and third.



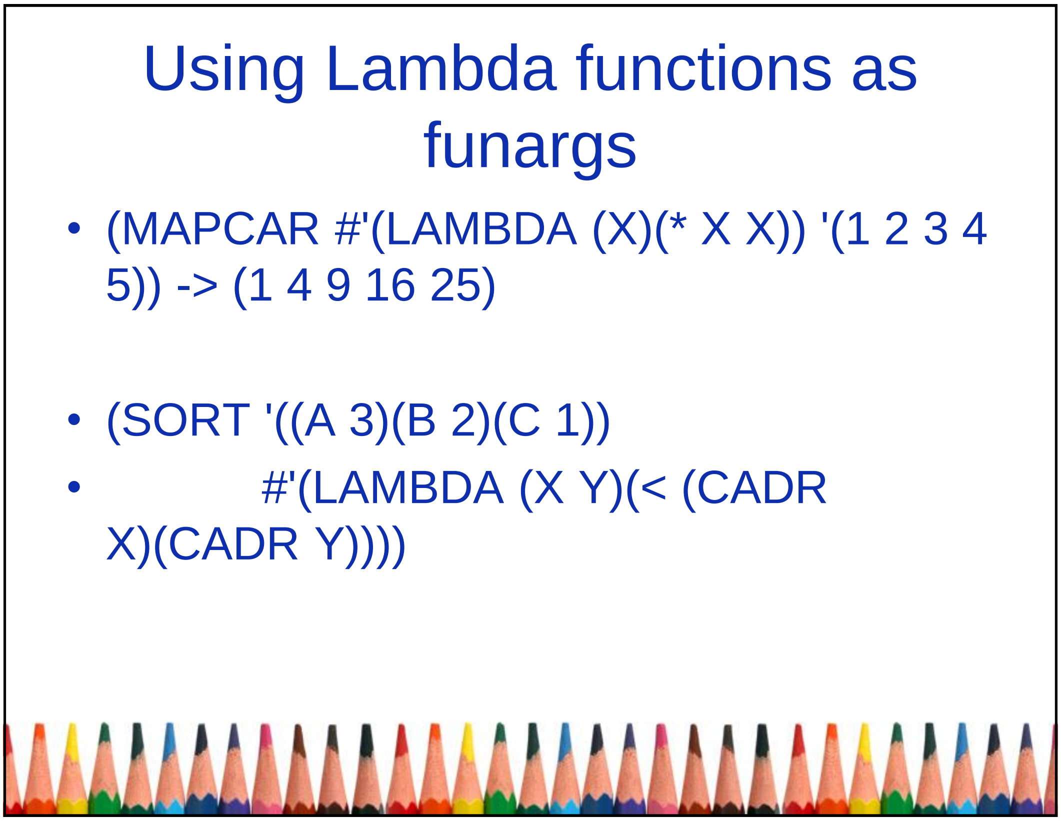
Lambda

- The output of Lambda is a function without a name.
- When executed, its parameters become local variables.
- ((lambda (a b c) (+ a (* b c))) 3 4 5) -> 23



Unnamed Functions

- (defvar x 10)
- ((lambda (y) (+ y y)) x) -> 20
- in this case, instead of defining a function called "double", we wrote a function without a name and put it as the leftmost component of an expression.



Using Lambda functions as funargs

- (MAPCAR #'(LAMBDA (X)(* X X)) '(1 2 3 4 5)) -> (1 4 9 16 25)
- (SORT '((A 3)(B 2)(C 1))
- #'(LAMBDA (X Y)(< (CADR X)(CADR Y))))

What about Chrismas?

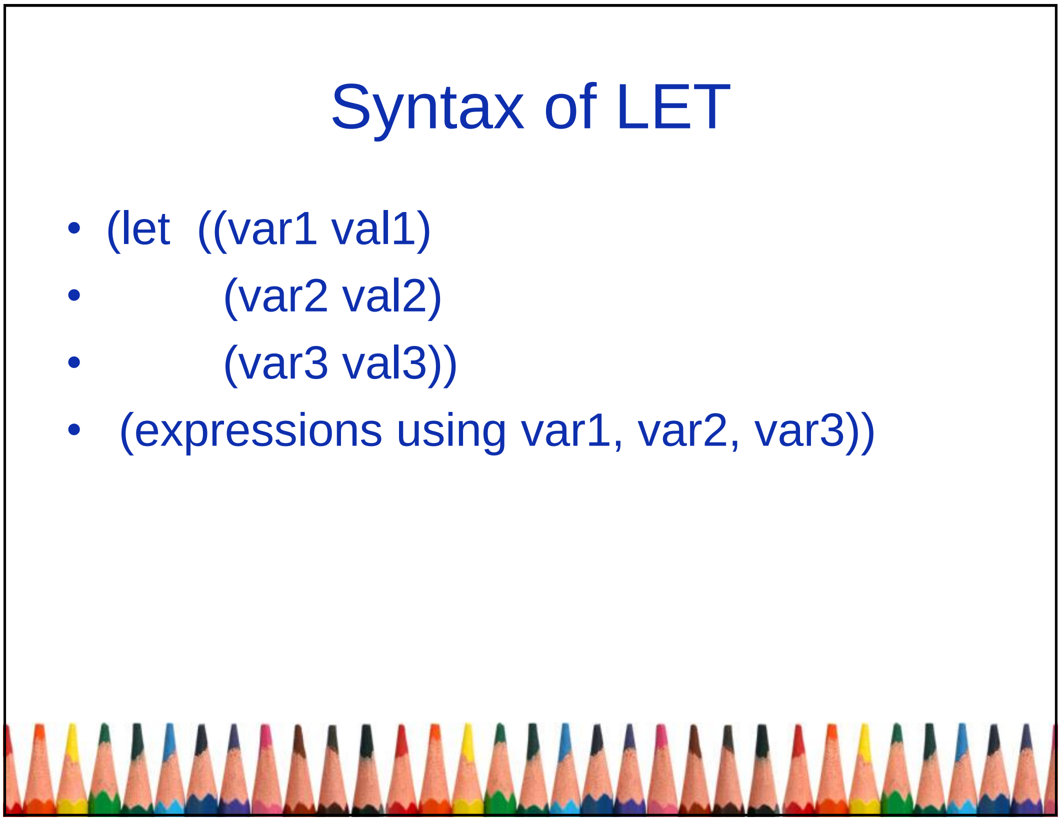
- (sort chrismas
  - #'(lambda (a b)(> (car a)(car b)))) →
  - ((12 DRUMMERS DRUMMING) (11 PIPERS PIPING) (10 LORDS A-LEAPING)
  - (9 LADIES DANCING) (8 MAIDS A-MILKING) (7 SWANS A-SWIMMING) (6 GEESE A-LAYING)
  - (5 GOLD RINGS) (4 CALLING BIRDS) (3 FRENCH HENS) (2 TURTLE DOVES)
  - (1 PARTRIDGE IN A PEAR TREE))

Lambda Binds new local parameters,

- This ghosts” symbols with same name.
- This allows a way to make temp variables

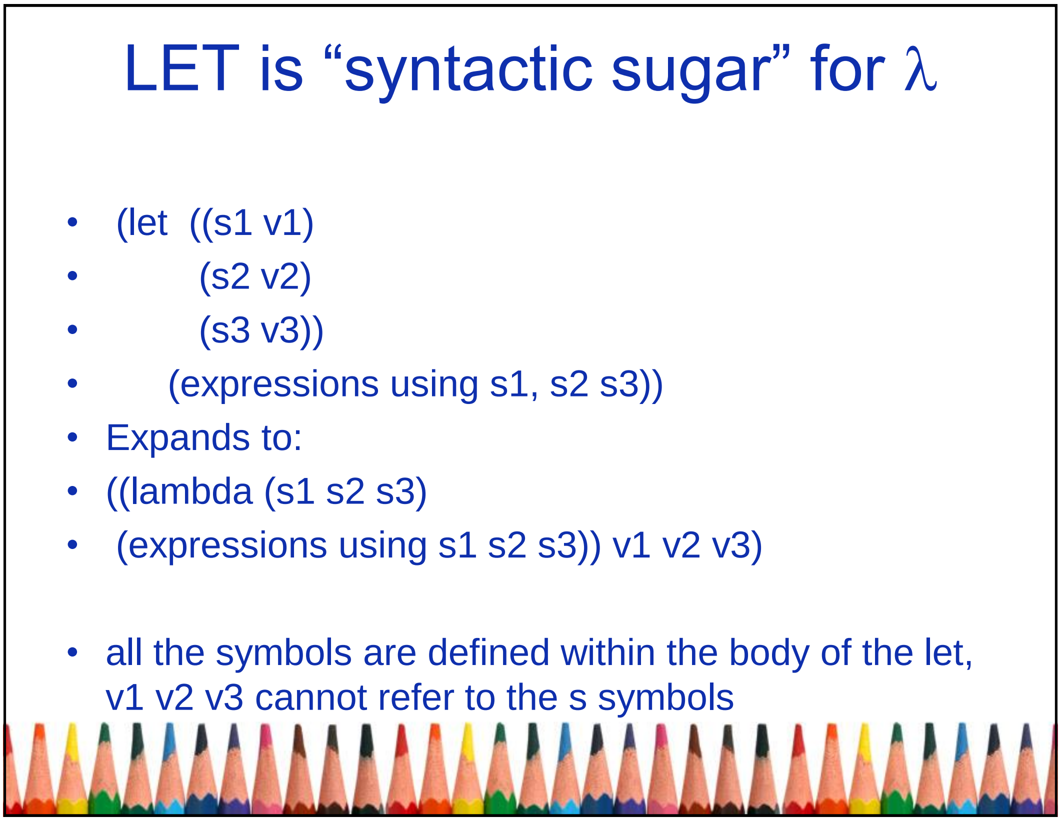
- (defvar j 10)
- ((lambda (j)(setf j (* j j))) 100) -> 10000
- j still -> 10

- But its kinda ugly



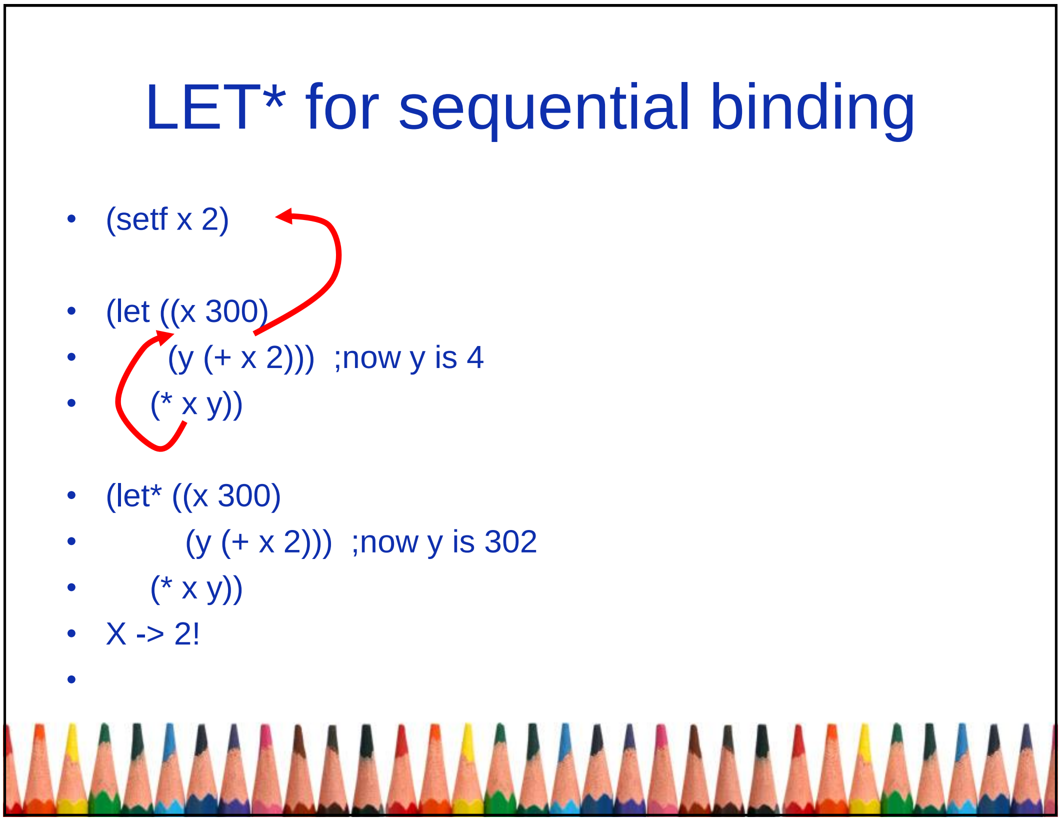
Syntax of LET

- (let ((var1 val1)
- (var2 val2)
- (var3 val3))
- (expressions using var1, var2, var3))



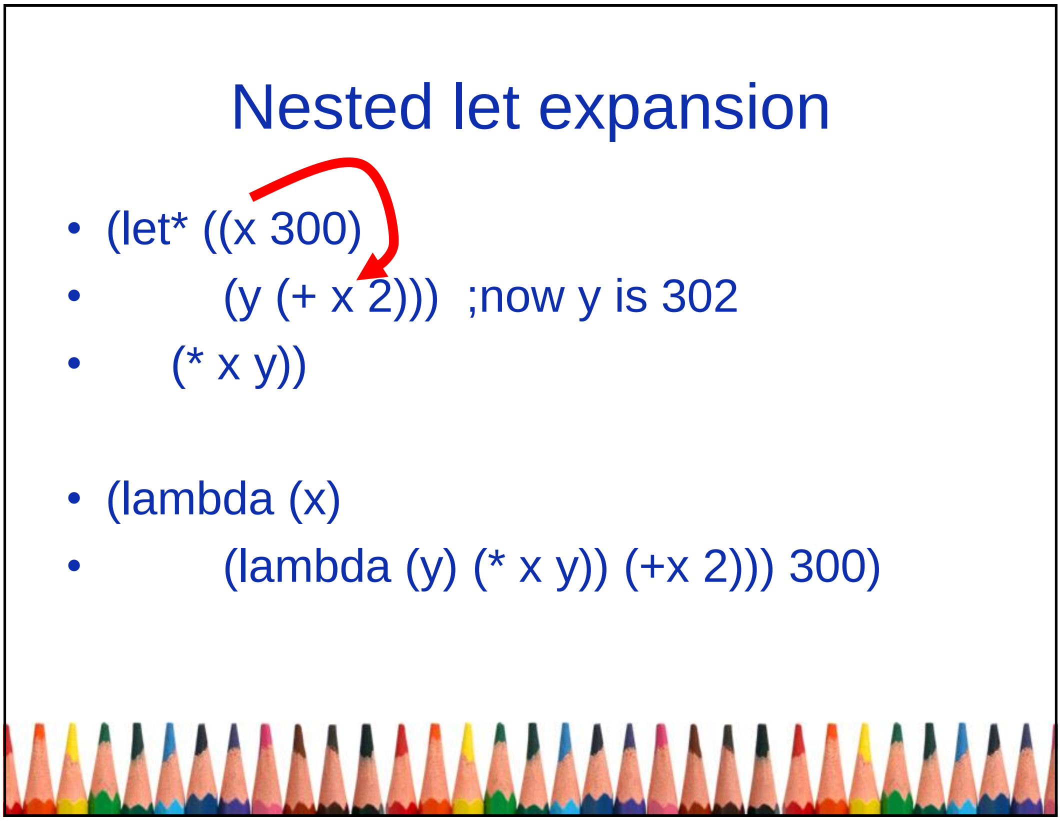
LET is “syntactic sugar” for λ

- (let ((s1 v1)
- (s2 v2)
- (s3 v3))
- (expressions using s1, s2 s3))
- Expands to:
- ((lambda (s1 s2 s3)
- (expressions using s1 s2 s3)) v1 v2 v3)
- all the symbols are defined within the body of the let, v1 v2 v3 cannot refer to the s symbols



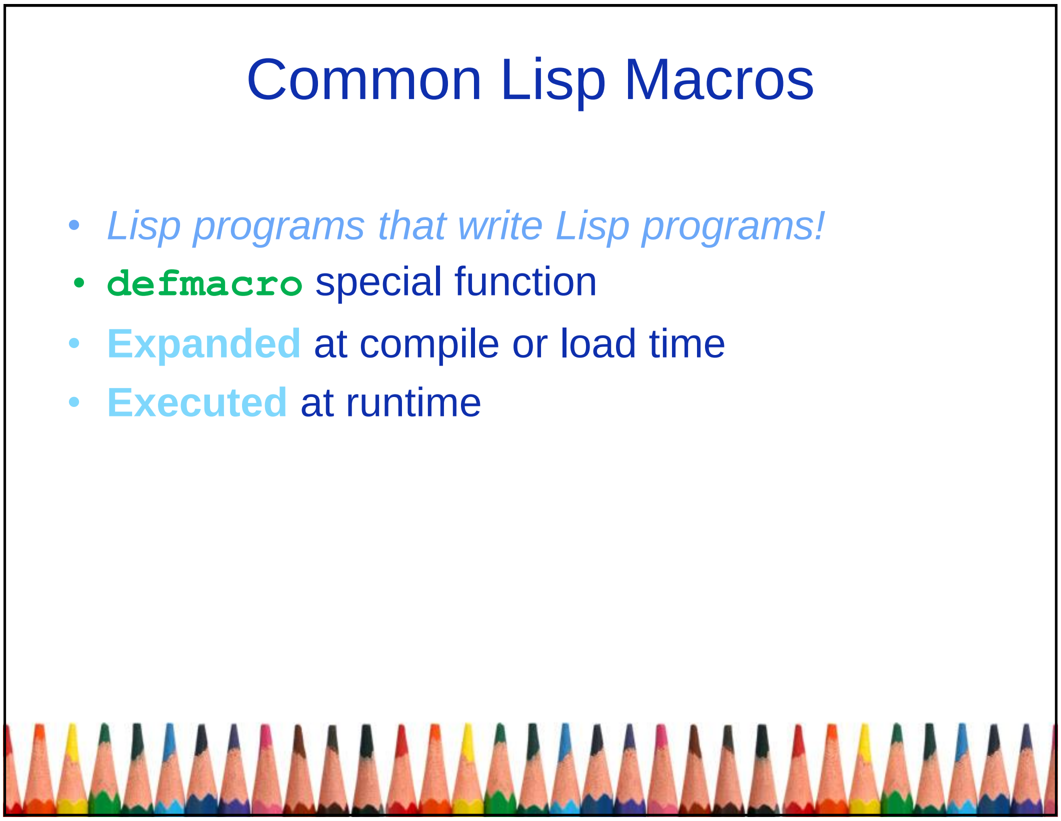
LET* for sequential binding

- (setf x 2)
- (let ((x 300)
-     (y (+ x 2))) ;now y is 4
-   (* x y))
- (let* ((x 300)
-       (y (+ x 2))) ;now y is 302
-    (* x y))
- X -> 2!
-



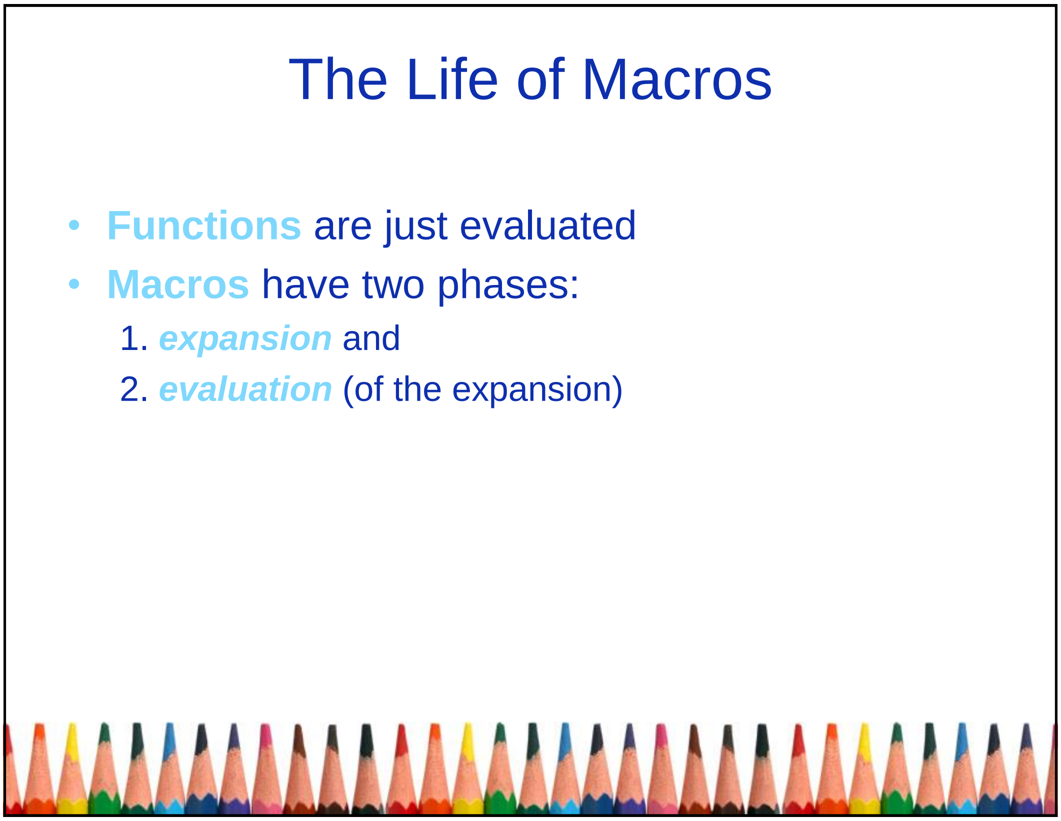
Nested let expansion

- (let* ((x 300)
-       (y (+ x 2))) ;now y is 302
-    (* x y))
- (lambda (x)
-       (lambda (y) (* x y)) (+x 2))) 300)



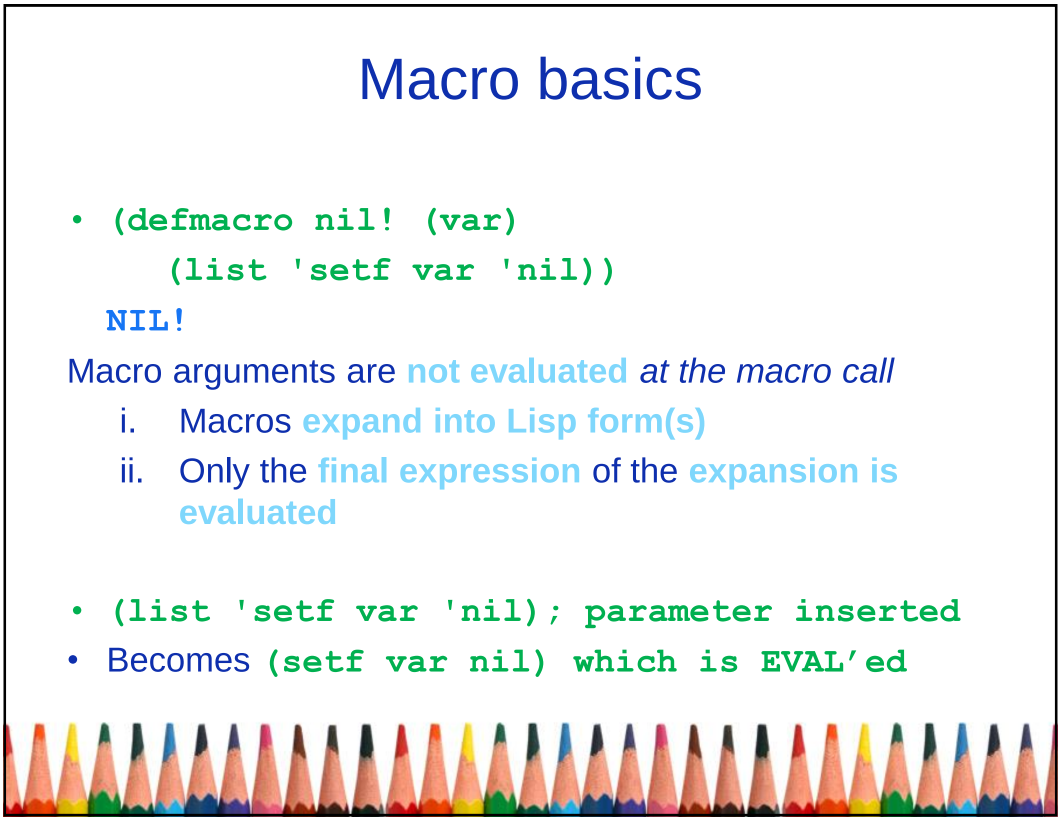
Common Lisp Macros

- *Lisp programs that write Lisp programs!*
- **`defmacro`** special function
- **Expanded** at compile or load time
- **Executed** at runtime



The Life of Macros

- **Functions** are just evaluated
- **Macros** have two phases:
  1. ***expansion*** and
  2. ***evaluation*** (of the expansion)



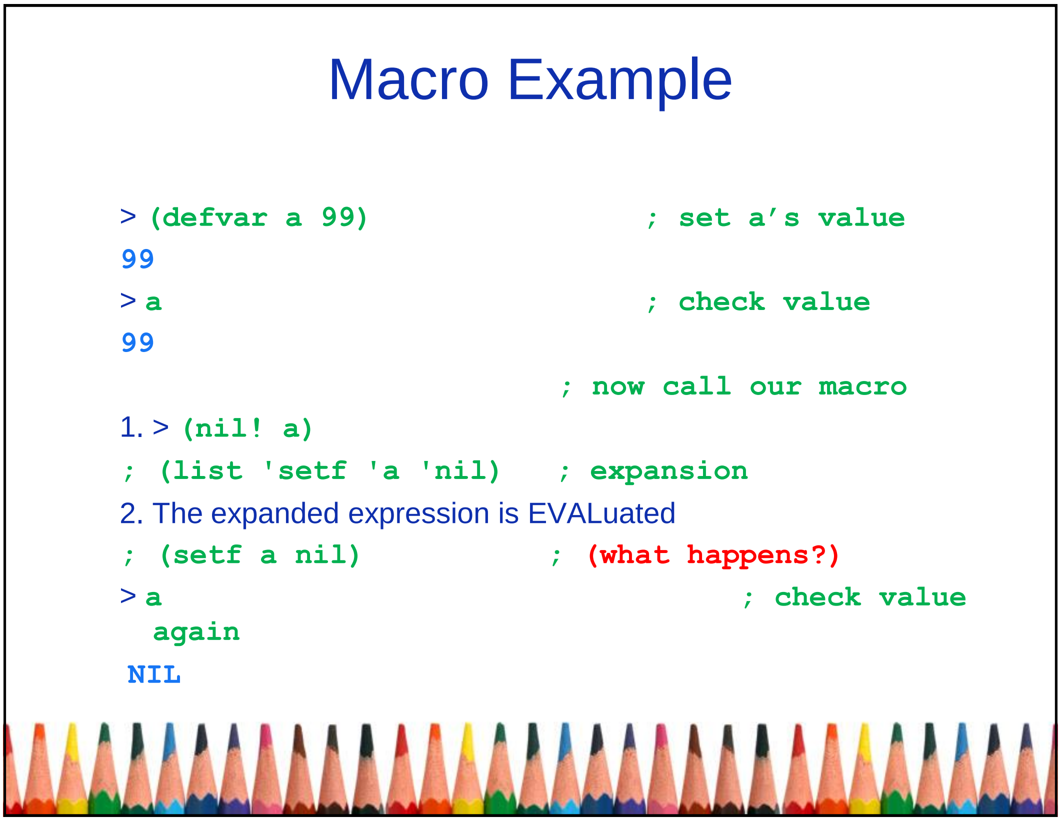
Macro basics

-
  ```
  (defmacro nil! (var)
       (list 'setf var 'nil))
  NIL!
  ```

Macro arguments are not evaluated *at the macro call*

1. Macros expand into Lisp form(s)
2. Only the final expression of the expansion is evaluated

- `(list 'setf var 'nil); parameter inserted`
- Becomes `(setf var nil) which is EVAL'ed`



Macro Example

```
> (defvar a 99)                    ; set a's value
99
> a                                ; check value
99
                               ; now call our macro
1. > (nil! a)
; (list 'setf 'a 'nil)     ; expansion
```

2. The expanded expression is EVALuated

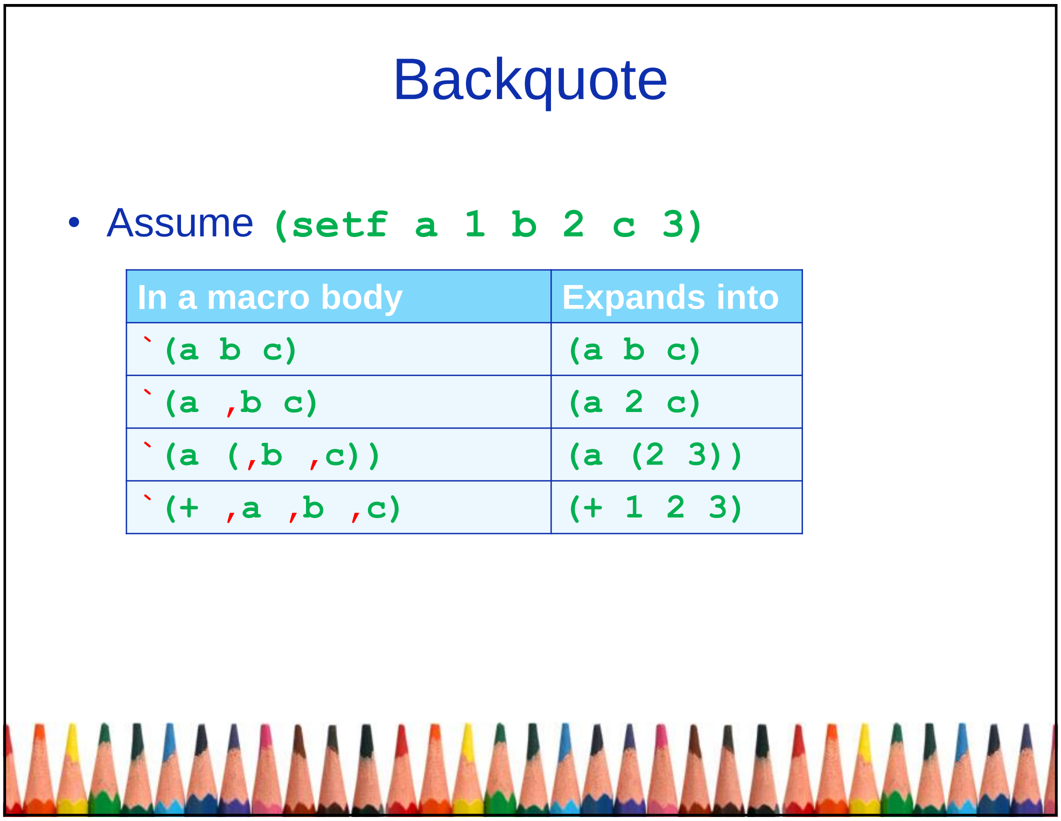
```
; (setf a nil)            ; (what happens?)
> a                                    ; check value
  again
 NIL
```

Backquote is used to make macros more readable

- (defmacro nil! (var) ; without backquote
-     (list 'setf var 'nil))
- (defmacro nil! (var) ; shorter with backquote
-         `(setq ,var nil))

Inside Backquote Context…

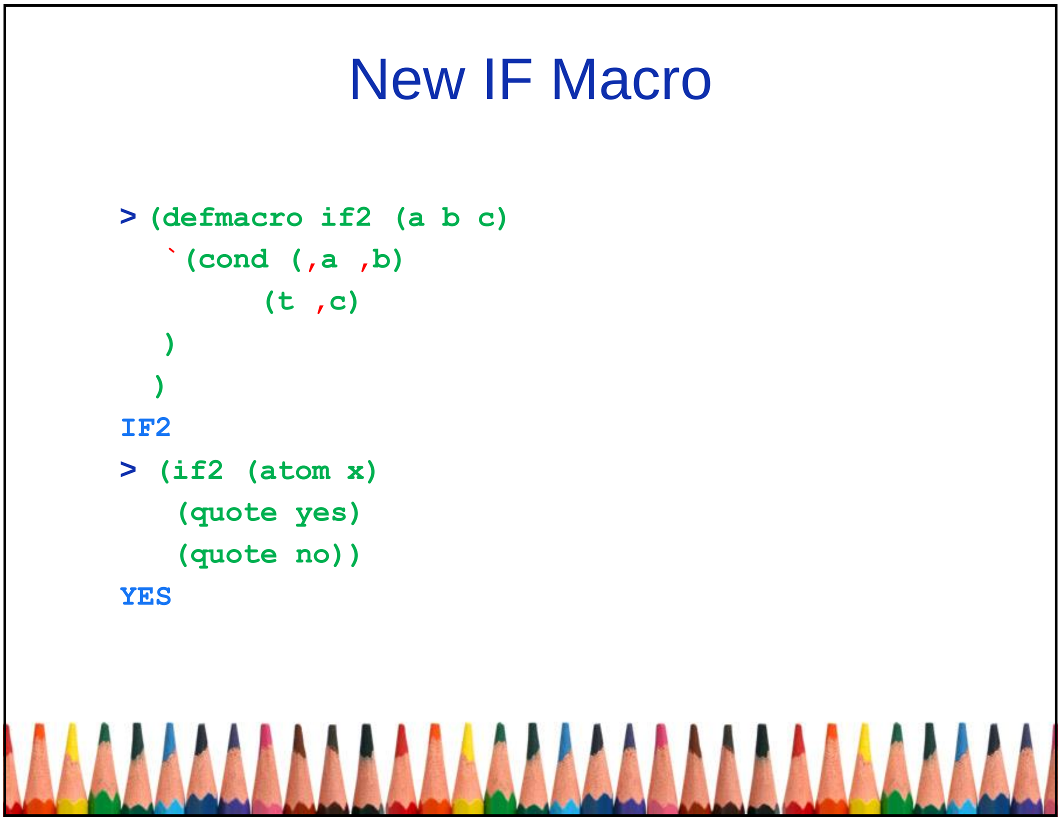
- `   backquote specifies template
  - prevents evaluation similar to quote in functions
- ,   COMMA evaluates the item just after the comma
- ,@ SPLICE evaluates an item and “splices it into” the expression using append



Backquote

- Assume **(setf a 1 b 2 c 3)**

In a macro body	Expands into
`` `(a b c) ``	(a b c)
`` `(a ,b c) ``	(a 2 c)
`` `(a (,b ,c)) ``	(a (2 3))
`` `(+ ,a ,b ,c) ``	(+ 1 2 3)

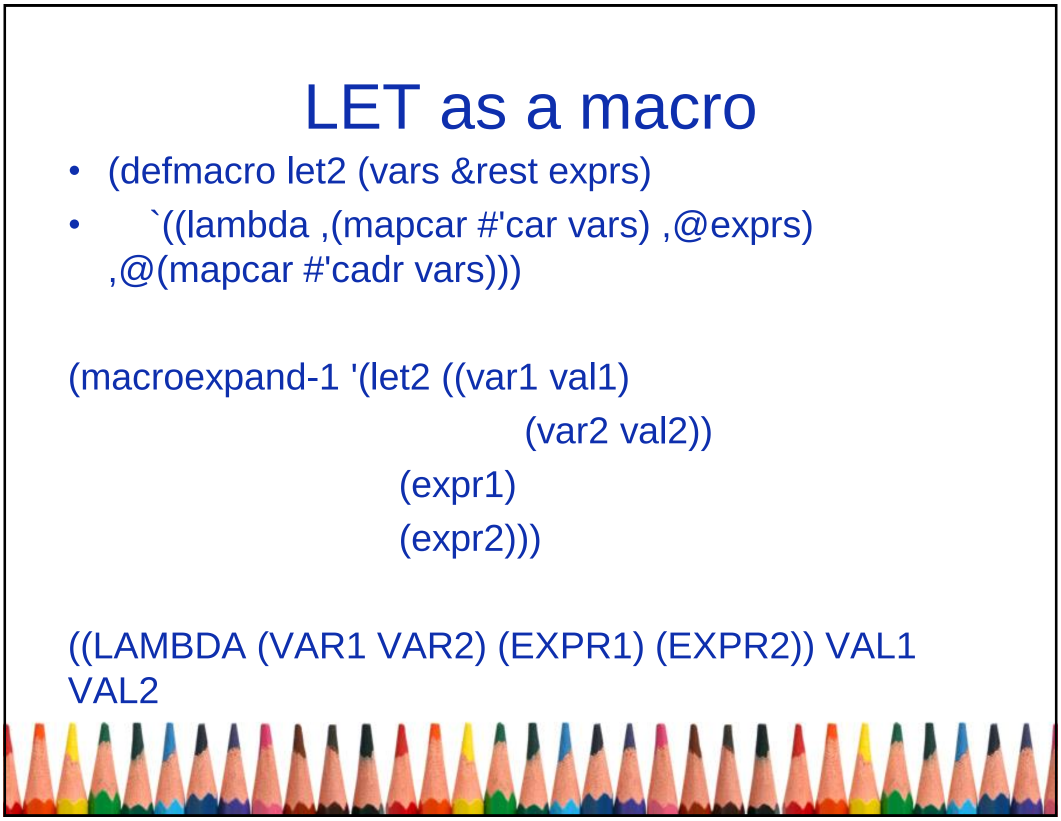


New IF Macro

```
> (defmacro if2 (a b c)
    `(cond (,a ,b)
           (t ,c)
    )
   )
IF2
> (if2 (atom x)
     (quote yes)
     (quote no))
YES
```

Backquote with “splice” ,@

- (defvar d (quote (some list)))
- `(a ,d b) -> (a (some list) b)
- `(a ,@d)   -> (a some list)
- `(a ,@d b) -> (a some list b)

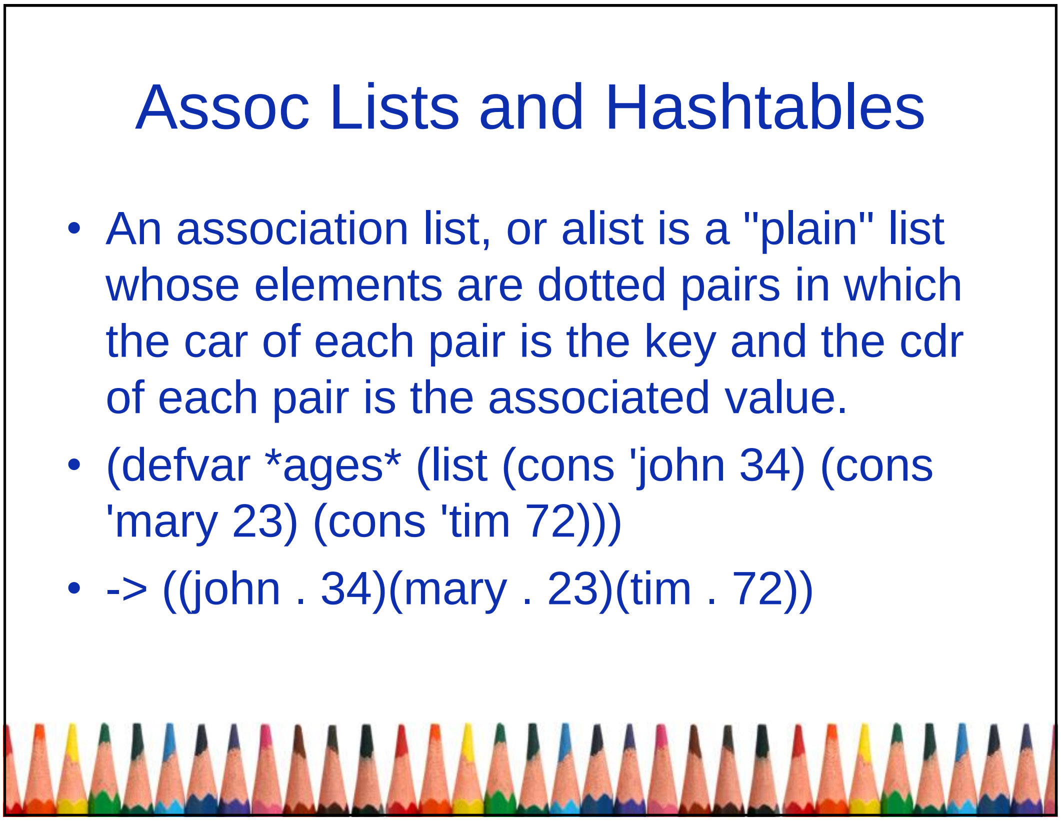


LET as a macro

- (defmacro let2 (vars &rest exprs)
-     `((lambda ,(mapcar #'car vars) ,@exprs) ,@(mapcar #'cadr vars)))

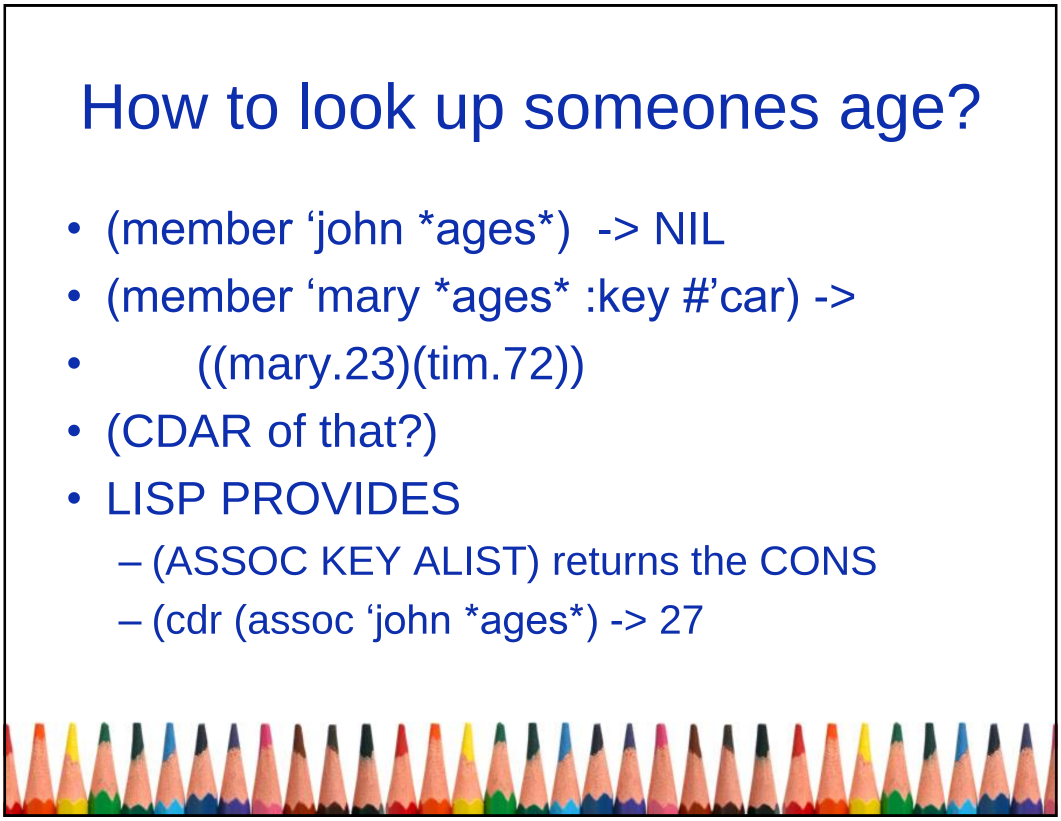
```
(macroexpand-1 '(let2 ((var1 val1)
                                   (var2 val2))
                         (expr1)
                         (expr2)))
```

```
((LAMBDA (VAR1 VAR2) (EXPR1) (EXPR2)) VAL1
VAL2
```



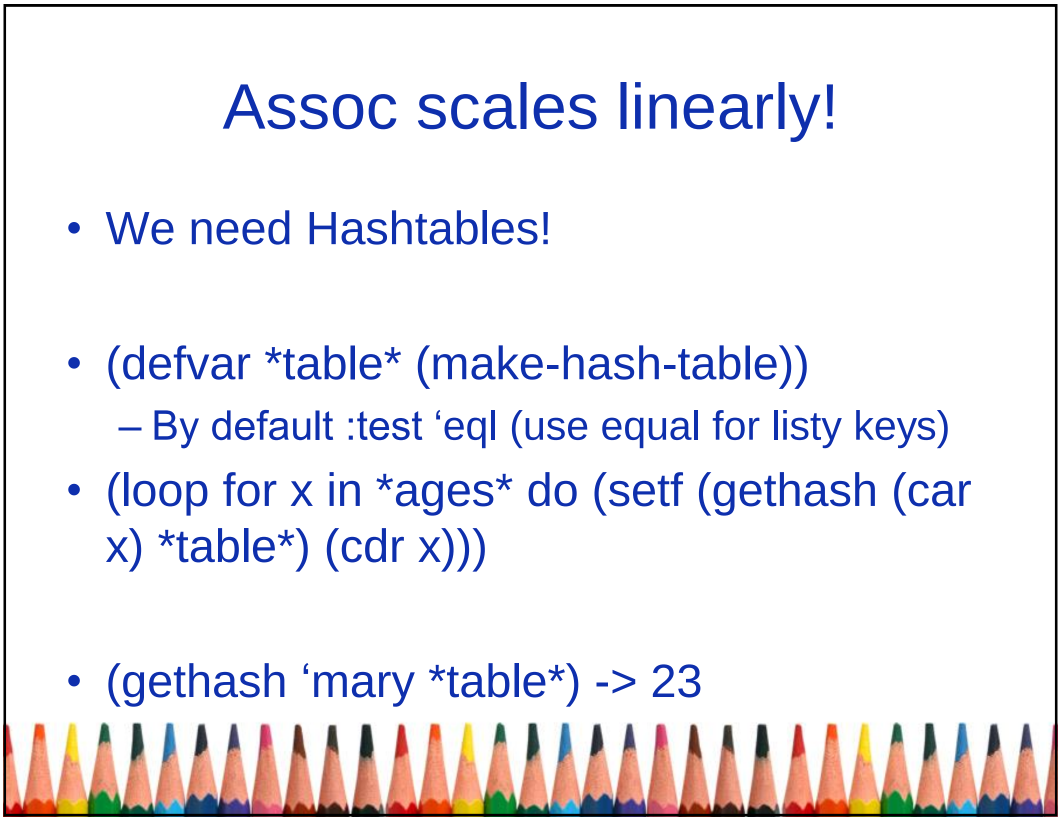
Assoc Lists and Hashtables

- An association list, or alist is a "plain" list whose elements are dotted pairs in which the car of each pair is the key and the cdr of each pair is the associated value.
- (defvar *ages* (list (cons 'john 34) (cons 'mary 23) (cons 'tim 72)))
- -> ((john . 34)(mary . 23)(tim . 72))



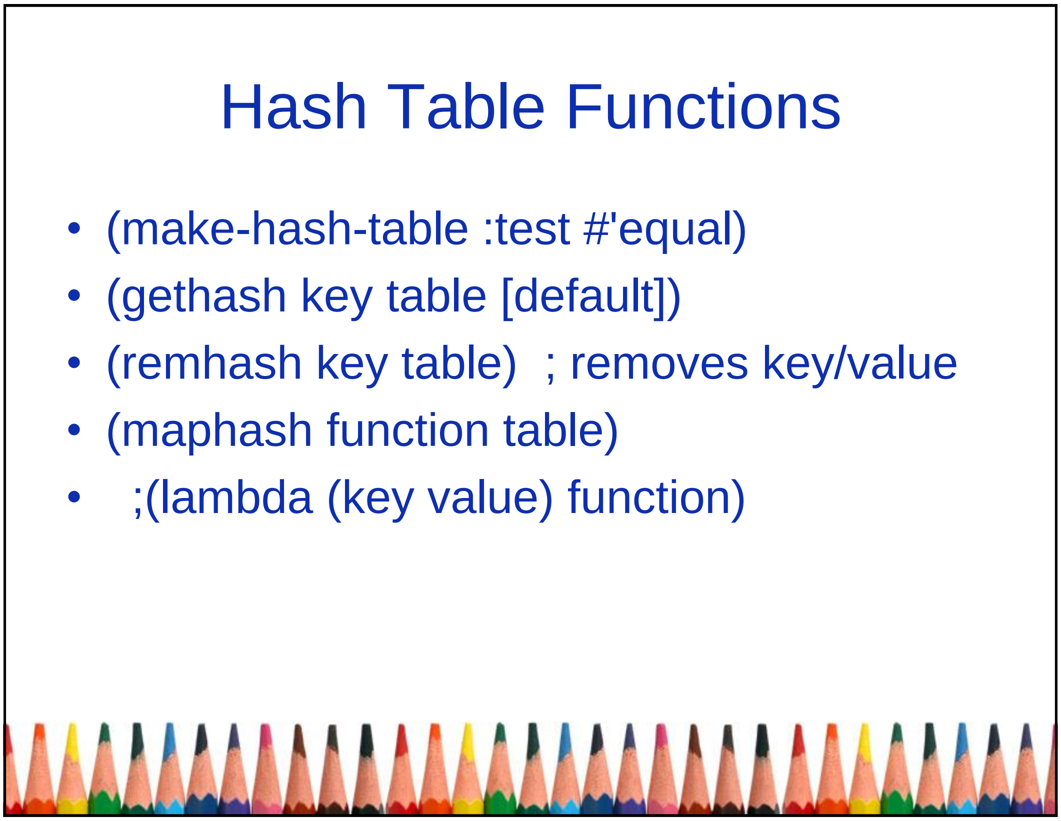
How to look up someones age?

- (member ‘john *ages*) -> NIL
- (member ‘mary *ages* :key #’car) ->
-         ((mary.23)(tim.72))
- (CDAR of that?)
- LISP PROVIDES
  – (ASSOC KEY ALIST) returns the CONS
  – (cdr (assoc ‘john *ages*) -> 27



Assoc scales linearly!

- We need Hashtables!
- (defvar *table* (make-hash-table))
  - By default :test ‘eql (use equal for listy keys)
- (loop for x in *ages* do (setf (gethash (car x) *table*) (cdr x)))
- (gethash ‘mary *table*) -> 23



Hash Table Functions

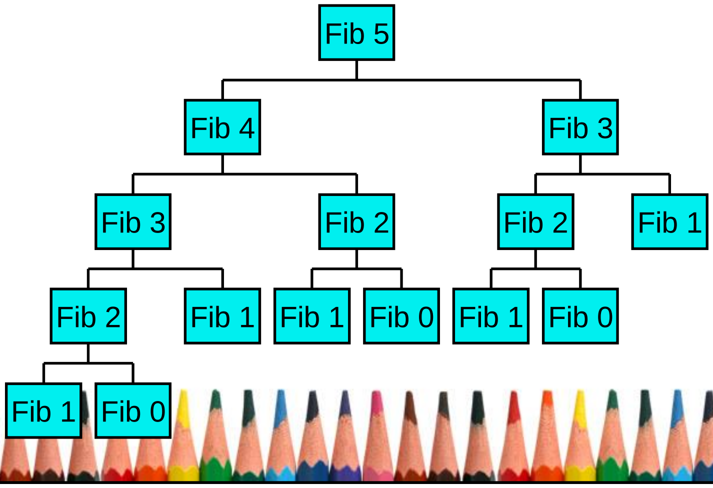
- (make-hash-table :test #'equal)
- (gethash key table [default])
- (remhash key table) ; removes key/value
- (maphash function table)
-   ;(lambda (key value) function)

Fibonacci sequence 0 1 1 2 3 5 8 13 ...

- (defun fib (n)
-     (if (< n 2)
-         n
-         (+ (fib (- n 2))(fib (- n 1)))))

- Inefficient because of repeated recursive operations.
- Can we fix it?

What does it look like?



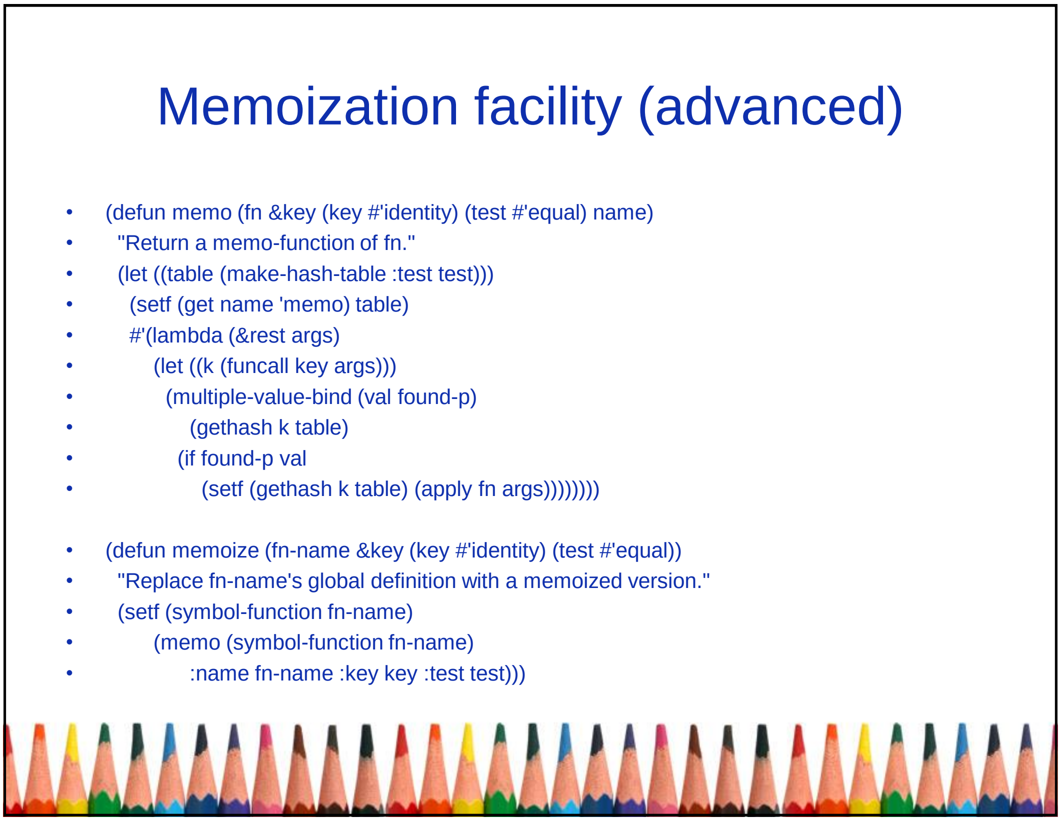
Memoization
a search/knowledge tradeoff

- Imagine if every time a function was called, its answer was memorized in a hashtable.
- The second time you call it with a same argument, the answer is returned instantly…

hashed fibonacci

```
• (defvar *fibhash* (make-hash-table))

• (defun fib (n)
•   (or (gethash n *fibhash*)
•       (setf (gethash n *fibhash*)
•         (if (< n 2) 1 (+ (fib (- n 1))
•                          (fib (- n 2)))))))
```



Memoization facility (advanced)

```
•  (defun memo (fn &key (key #'identity) (test #'equal) name)
•    "Return a memo-function of fn."
•    (let ((table (make-hash-table :test test)))
•      (setf (get name 'memo) table)
•      #'(lambda (&rest args)
•          (let ((k (funcall key args)))
•            (multiple-value-bind (val found-p)
•                (gethash k table)
•              (if found-p val
•                  (setf (gethash k table) (apply fn args))))))))

•  (defun memoize (fn-name &key (key #'identity) (test #'equal))
•    "Replace fn-name's global definition with a memoized version."
•    (setf (symbol-function fn-name)
•          (memo (symbol-function fn-name)
•                :name fn-name :key key :test test)))
```