



Lisp 201: Recursion and Iteration



Recursion

- A native style to lisp
- Very useful
- a good way to think of programming

Writing in a Recursive Style

- Solve Easy (Base) Case
- Make Predicate for Base Case
- [Magic] Assume it works for some arbitrary size (call this N-1)
- Show how to solve size N problem, given function works on size N-1 or smaller
 - Lisp compiler is good at converting recursion to iteration

Factorial

- `(defun fact (n)`
- `(if (< n 2) 1 (* n (fact (- n 1)))))`



Recursive Fibonacci

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



Length

- (defun leng (lst)
- (if (null lst)
- 0
- (+ 1 (leng (cdr lst)))))

- $O(n)$ time
- Remember a list is a right-branching tree ending in NIL, not a vector or 1d array!

beyond caddr:

Accessing Nth item in a list

- if $n=0$, nth should return the car
- if $n=1$, return the cadr
- if $n=2$, return the caddr, etc
- (defun enth (n lst)
- (if (= n 0) (car lst)
- (enth (- n 1)(cdr lst)))

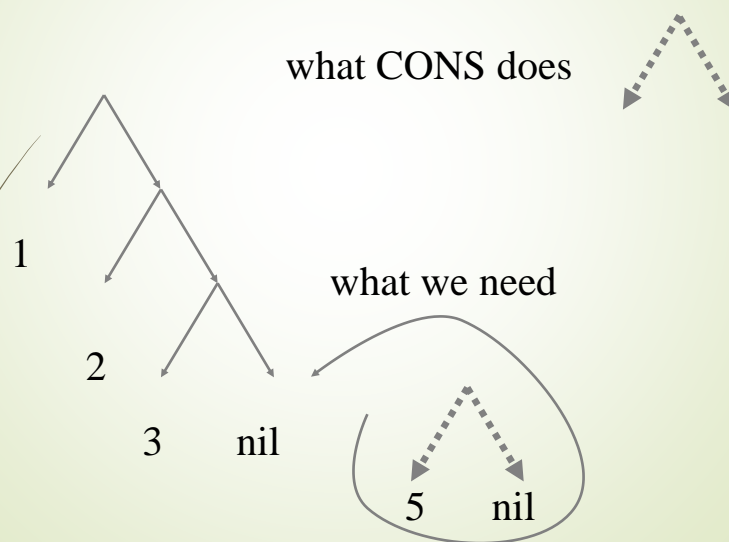
extending a list?

- $x \rightarrow (1\ 2\ 3\ 4)$
- how do we make $(0\ 1\ 2\ 3\ 4)$?
- How do we make $(1\ 2\ 3\ 4\ 5)$?
- How about $(1\ 2\ 3\ 4\ 1\ 2\ 3\ 4)$?

extending a list?

- (setf x '(1 2 3 4))
- how do we make (0 1 2 3 4)?
 - very easily: (cons 0 x)
 - CONS and CAR work as PUSH and POP!
- How do we make (1 2 3 4 5)?
 - (cons x 5)? (1 2 3 4 . 5)
 - (cons x (list 5))? ((1 2 3 4) 5)
- How about (1 2 3 4 1 2 3 4)
 - (cons x x)? (1 2 3 4 (1 2 3 4))
 - (list x x)? ((1 2 3 4)(1 2 3 4))

Append to the end



Recursive definition for append

- base case:
If list1 is the empty list, then result is just list2
- recursive step:
 - cons the first element of list1 onto the front of (append (cdr list1) with list2)

Append

- (defun append (list1 list2)
- (if (null list1) list2
- (cons (car list1)
- (append (cdr list1) list2))))

reverse a list

- `(last x) -> (4)`
- `(butlast x) (1 2 3)`
- `(defun revrse (lst)`
- `(if (null lst) nil (cons (car (last lst))(revrse (butlast lst))))`

square root, recursively

- start with a guess of 1
- if $|n - g^2| < \text{epsilon}$, halt
- else average g and n/g
- `(defun sqrt1 (n &optional (guess 1))`
- `(if (< (abs (- n (* guess guess))) .0001)`
- `guess`
- `(sqrt1 n (/ (+ guess (/ n guess)) 2.0))))`

Factoring

- Iteration in a recursive style, with bug
- (defun factor (n &optional (i 2))
 - (cond ((> i (sqrt n)) nil)
 - ((zerop (rem n i))
 - (cons (/ n i)
 - (cons i (factor n (+ 1 i))))))
 - (t (factor n (+ i 1))))))

Writing in an Iterative Style

- map Forms: Map, Mapc, Mapcar, Mapl, Maplist, mapcan, mapcon, reduce
- uses function as argument, and applies it to each element (or each tail) of a list, collecting (or not) the results

Mapcar and #' (bangquote)

- mapcar maps a function "across" a list or multiple lists
- #' or (function quote) is a special form which gets the f-value of the symbol
- (mapcar #'square '(1 2 3)) -> (1 4 9)
- (mapcar #'+ '(1 2 3) '(4 5 6))-> (5 7 9)

Example Arrays as nested Lists

- Representing arrays as lists of lists
 - (setf A '((1 2 3)(4 5 6)(7 8 9)))
- Rotating and transposing...
- So how do you get
 - ((3 2 1)(6 5 4)(9 8 7))
 - (MAPCAR #'REVERSE A)
- ((1 4 7)(2 5 8)(3 6 9)) ;transpose the array

Thinking Transpose, Recursively

- MAPCAR #'CAR A will get the (1 4 7)
- MAPCAR #'CDR A will get ((2 3)(5 6)(8 9))
 - MAPCAR #'CAR will get (2 5 8)
 - MAPCAR #'CDR THAT gets ((3)(6)(9))
 - MAPCAR #'CAR THAT gets (3 6 9)
 - MAPCAR #'CDR THAT gets (? ? ?)
- What is the base case? How to test for it?

(NIL NIL NIL)

Transpose, Recursively

- (defun transpose (x)
- (if (null (car x)) nil
- (cons (mapcar #'car x)
- (transpose (mapcar #'cdr x))))

Loop Macros to the rescue!

- The LOOP macro Package (powerful and elegant)
 - Added in CLTL2
 - Loops with lists, integers, arrays
 - Powerful collection facilities
 - powerful stop logic
 - use of auxiliary variables


Compare to Mapcar

- `(mapcar #'square '(1 2 3))`
- `(loop for i from 1 to 3 collect (square i))`
- or
- `(loop for i in '(1 2 3) collect (square i))`



works like C forloop

- (loop for i from 1 to 5 by 2 collect i)
- (loop for j from 20 downto 10 sum j)



In and On (tails) lists

- (loop for x in '(a b c) do (print x))
- (loop for x on '(a b c) do (print x))

Conditional Logic for do

- When, if, unless, while, until
 - (loop for x in '(a b c d e 1 2 3 4)
 - until (numberp x)
 - collect (list x 'foo))

Collection Facilities

- Do, Collect, Append, Nconc, Count, Sum, Minimize, Maximize
- (loop for x in (factor 36) maximize x)
- (loop for x in '(a b c) as y from 1
 - collect (list x y))
- (loop for x in '(a b c) as y from 1
 - append (list x y))
- (loop for i from 1 to 10 when (evenp i) sum i)
- (loop for x on '(5 4 3 2 1) collect
 - (reduce #' + x)))

Termination Logic

- (loop for x in (factor 36) thereis (evenp x))
- T
- VERY IMPORTANT FEATURE:
 - Thereis, Always, Never, while, until
 - Thereis stops at first non-NIL
 - Always stops at first NIL
 - Never means ALL NILS
 - Always means ALL non-NILS
 - While and Until are like other languages

WITH Auxiliary Variables

- (loop for x from 1 to n with y = 10 and z = 20
-

Redo Fibonacci

- (defun fib (n)
- (loop for i from 1 to n with f1 = 0 and f2 = 1 and f3 do
- (setf f3 f1)
- (setf f1 f2)
- (setf f2 (+ f3 f2))
- finally (return f2))

You can also iterate using recursion, often prettier!

- (defun fib (n &optional (f1 1)(f2 1))
- (if (= n 1)
- f1
- (fib (- n 1) f2 (+ f1 f2))))

Lets Redo Factoring

- (defun factor (n)
- (loop for i from 2 to (floor (sqrt n)) *append*
- (if (zerop (rem n i))
- (list i (/ n i))
- nil)))
- Instead of collect, append NILS and (numbers). Append “throws away” Parens.

simplified factoring no bug

- (defun factor (n)
- (loop for i from 1 to n
- when (zerop (rem n i))
- collect i))