LISP Programming:

It is important to realize that LISP attempts to interpret the first element of a list.

(cadr (list 1 2 3 4 5))

(caddr (list 1 2 3 4 5))

(caaddr (list (list 1 2) (list 3 4) 5 6))
Recursion 1. The famous programs

(defun factorial (N)
  (cond
    ((zerop N) 1)
    ((< 0 N) (* N (factorial (- N 1)))))
)

(defun memberp (X L)
  (cond
    ((null L) nil)
    ((eq X (car L)) T)
    (T (memberp X (cdr L)))
  ))

Note that LISP’s member function doesn’t work this way.

> (member 3 (list 1 2 3 4 5))
(3 4 5)
>

2
Recursive patterns

- $f$ calls $f$

- 1. $f$ calls $h$
   2. $h$ calls $h$

- There is a sequence $g_1, \ldots g_n$ such that
  1. $f$ calls $g_1$
  2. $g_i$ calls $g_{i+1}$  \( i = 1, \ldots g_{n-1} \)
  3. $g_n$ calls $f$
; substitution

(defun subst (new old main)
  (cond
    ((equal old main) new)
    ((atom main) main)
    (T
      (cons
        (subst new old (car main))
        (subst new old (cdr main)))
    ))
  )
)
Tricks with extra variables

; iterative factorial
(defun facti (n p)
  (cond
    ((zerop n) p)
    (T (facti (- n 1) (* n p))))
))
Tricks with extra variables 2.

; naive reverse

(defun reverse (list)
  (cond
    ((null list) list)
    (t (append (reverse (cdr list))
              (list (car list))))))

; smarter reverse

(defun smartreverse (list accumulator)
  (cond
    ((null list) accumulator)
    (t (smartreverse (cdr list)
                    (cons (car list) accumulator)))
    ))
I/O

I/O is done to and from “streams”.

- (OPEN path :DIRECTION dir) as in
  (read (open "examples1.lsp" :DIRECTION :INPUT))

- (READ Stream)

- (PRINT argument stream)

- (CLOSE Stream)
An example

(setq stream-in (open "examples1.lsp"
  :DIRECTION :INPUT))
#<Input-Stream 4:"/home/jhodgson/courses/ppl/code/lsp"
1> (read stream-in)
(DEFUN REVERSE (LIST) (COND ((NULL LIST) LIST) (T (REVERSE (REST LIST))))
1> (read stream-in)
(DEFUN FACT (N) (COND ((ZEROP N) 1) ((< 0 N) (* N (FACT (+ N -1)))))))
MAPCAR and friends

- (APPLY k-arg-fn list-of-k-arguments)

  (apply ’+ (list 1 2))
  3

- (FUNCALL k-arg-fn arg1 .. argk)

  > (funcall ’+ 1 2)
  3
• (MAPCAR k-arg-fn arg-list1 .. arg-listn)

> (mapcar '+ (list 1 2) (list 3 4))
(4 6)

• (MAPLIST k-arg-fn arg-list1 .. arg-listn)

(maplist 'append (list 1 2 3)
           (list 'a 'b 'c))
((1 2 3 A B C) (2 3 B C) (3 C))
Function closures

- Free

- Bound variables.

(defun adder (Y) (plus X Y))

Then other functions can reset X causing problems. Closures (if you can get them to work) solve this “problem”. Better is to use OO properties in LISP.
Property lists

- (putprop object-arg value-arg attribute-arg)

- (get object-arg value-arg)

(putprop 'chair3 'blue 'color)
(putprop 'chair3 'John 'owner)

(get 'chair3 'owner)
JOHN
Macros 1.

(defmacro M (x1 ...xn) S)

The S-expression (M S1 ...Sn) is evaluated as follows:

1. First S is evaluated in an environment in which S1, ...Sn are bound *unevaluated* to x1, ...xn respectively. (Macro expansion)

2. Second the value resulting from evaluating S during the expansion phase is evaluated.
The Backquote macro

‘S is like a quoted expression ’S except that any S-expression S that preceded by a comma is evaluated.

> (setq y ’(1 2 3))
(1 2 3)
> ’(x y z)
(X Y Z)
> ’(x y z)
(X Y Z)
> ‘(x ,y z)
(X (1 2 3) Z)
> ‘(x ,(car y) z)
(X 1 Z)
> ‘(x (car ,y) z)
(X (CAR (1 2 3)) Z)
LISP Functions
Evaluation

• (eval expr)

• (apply fun arg ... arg)

• (funcall fun arg ... )

• (quote expr)

• (backquote expr)

• (lambda args expr)
Some Imperative stuff.

- \((\text{let } ((x_1 \text{ S}_1) \ldots (x_n \text{ S}_n)) \text{ S})\)

  1. \(\text{S}_1, \ldots \text{S}_n\) are evaluated in that order and their values are bound to \(x_1, \ldots, x_n\) respectively,

  2. \(\text{S}\) is evaluated in the resulting environment

  This is equivalent to \(((\text{lambda } x_1, \ldots x_n) \text{ S}) \text{ S}_1 \ldots \text{S}_n)\)

- \((\text{prog } (x_1 \ldots x_n)\\l_1 \text{ S}_1\\l_2 \text{ S}_2\\\ldots\\l_n \text{ S}_n)\)

  Each \(l_i\) is a label, which can be omitted if not needed.
More LISP functions

- (append L1 L2)

- (assoc S ‘((S11 . S12) ... (Sn1 . Sn2)))

  > (assoc 4 ‘(( 3 . 6) (4 . 5)))
  (4 . 5)

- (not S)

- (gensym)

- (terpri)
LISP in LISP

1. (READ)

2. (EVAL)

3. (PRINT)
Some bigger examples

- Binary trees

- Symbolic differentiation
Binary trees

; A tree is to be represented as a list (Left, Node, Right)
; The empty tree is nil

; (bt-insert Node Tree) returns the new tree

(defun bt-insert (N Tree)
  (cond
    ((null Tree) (list nil N nil)) ; empty tree
    ((< N (second Tree))
     (list (bt-insert N (first Tree))
          (second Tree)
          (third Tree)))
    (T (list (first Tree)
           (second Tree)
           (bt-insert N (third Tree))))
  ))
(defun batch-bt-insert (List Tree)
  (cond
   ((null List) Tree)
   (T (bt-insert (car List)
            (batch-bt-insert (cdr List) Tree)))))
; traverse generate a list of the nodes
; of a binary tree inorder

(defun traverse (Tree)
  (cond
    ((null Tree) nil)
    (T (append (traverse (first Tree))
      (cons (second Tree)
        (traverse (third Tree)))))
   ))
Symbolic differentiation

(defun diff (term var)
  (cond
    ((atom term)
      (if (eq term var) 1 0)
    ((or
      (eq (car term) '+)
      (eq (car term) '-)
    )
     (cons
      (car term)
      (diff-list (cdr term) var)
    ))
    ((eq (car term) '*)
     (cons '+
       (diff* (cdr term) var)
    ))
    ((eq (car term) '/)
     (list '/
       (cons '-'
         (diff*/ (cdr term) var)
       )
       (cons '*
         (append (cddr term) (cddr term))
       ))
    ))
  )
)
; differentiate a list of terms

(defun diff-list (term-list var)
  (mapcar
   '(lambda (trm) (diff trm var))
   term-list))
; differentiate a product

(defun diff* (term-list var)
  (maplist
   '(lambda (tl1)
      (cons '*
        (maplist
         '(lambda (tl2)
            (if
             (equal (length tl1) (length tl2))
             (diff (car tl1) var)
             (car tl2))
             term-list)))
      term-list))