LISP

John MaCarthy, MIT, 1960

pure Lisp – functional programming language

data types – atom and list

dynamic scoping: most recent association rule for non-local

interactive interpreter: evaluate each expression and return

```
Ex.   → (+  3  5)
         8
```

A.I.

Dialects:

- MacLISP (MIT) – Franz Lisp (Vax)
- INTERLISP (BBN)
- SCHEME – static scope
- Common Lisp – Guy Steele (1984)

<u>Common Lisp</u> (esus)

```
clisp
  :
(exit)
```

error recovery: unwind

To obtain output,

```
script  filename
clisp
  :
(exit)
Ctrl-D
----------------------------------------
more  filename            (to see)
lpr –Ppanda filename  (to print)
```

Data Types and Structures

- o atom

    numeric atom - number (integer, real)
    non-numeric atom – symbol

- o list (atom atom … atom)

    Note.

( )	empty list
(( ))	nonempty

Expression

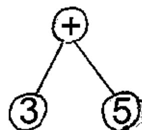
S-expression (cf. M-notation)

1. An atom is an S-expression.
2. If α and β are S-expression, then $(\alpha . \beta)$ is an S-expression.

Note.

(1) function call : (function arg1 arg 2 … arg n)
    data : (A B C …)

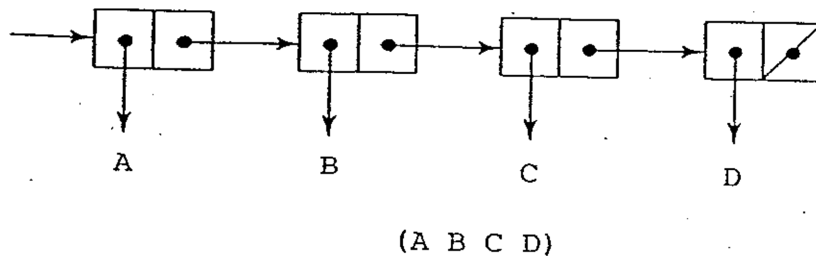
(2) Preorder (Cambridge Polish notation)

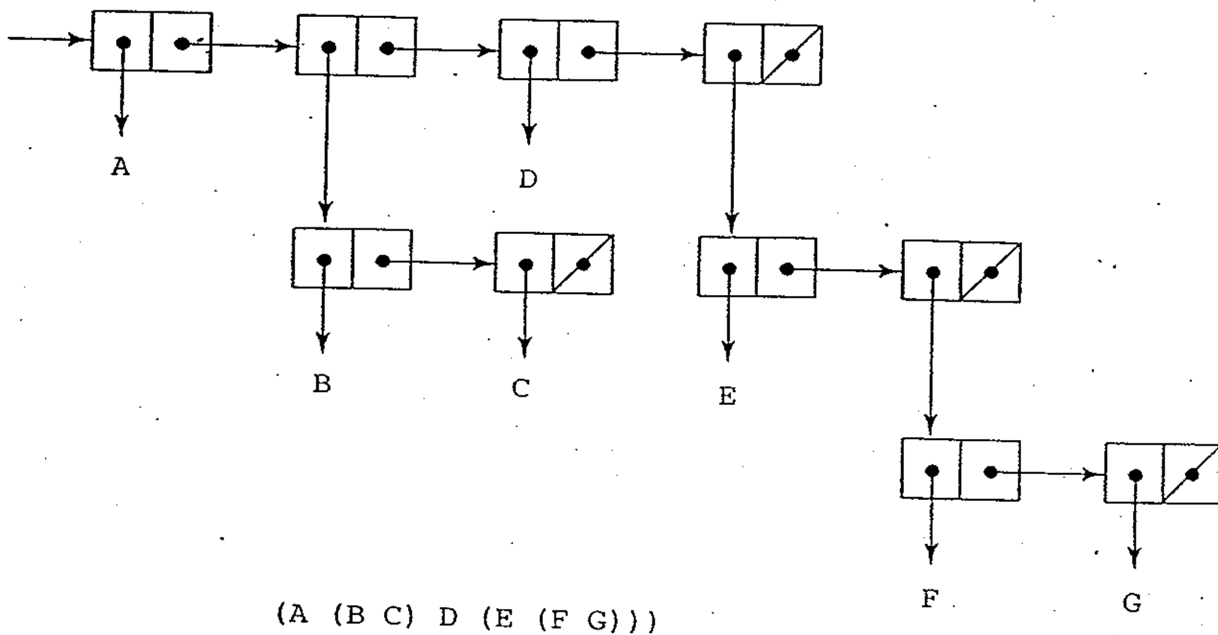


→ (+ 3 5) : S-expression

8 : va;ue

Figure 15.1
Internal representation of two LISP lists





- **Assignment**

  → (setq x 5)   **Note.** setq = set quote, a special function
  5

  → (* x 4)
  20

  → x
  5

  → (setq pi 3.14)
  3.14

- **Define a list**

  → (quote (a b c)) or ‘(a b c)
  (A B C)

  → (setq x ‘lisp)
  LISP

  → x
  LISP

- **Primitive Functions**

  Arithmetic functions (integer/real)

  +
  -
  *
  /

Predecessor/Successor functions (x)

sub1
add1

Max/min functions

→ (max 7.5 1.3 9.6)
9.6

Relational tests

equal or = (x)
<, <=, >, >=

- **Predicates**

  zerop
  onep (x)
  minusp
  plusp
  numberp

  atom
  listp
  member

Non-destructive functions (pure functions)

- **car (content of address register) : first element**
- **cdr (content of decrement register) : list without 1st element**

  car ≡ first
  cdr ≡ rest

- **cons (construct)**

  (cons expr list)

  Note: cons requires new storage.

- **list – list building function**

Examples

Examples:

```
> (setq  x  '(A  B  C) )                 (A  B  C)
> (car  x)                                A
> (cdr  x)                               (B  C)
> (car (cdr  x))  =  (cadr  x)           B
> (car '((A B) C D))                      (A  B)
> (car  '(A))                             A
> (car  'A)                               error: A is not a list
> (car '( ))                              NIL
> (cdr  'A)                               error: A is not a list
> (cdr  '(A))                             NIL
> (cons (car  x)  (cdr  x))               (A  B  C)   -  x itself
> (cons  'A  '(B  C)))                    (A  B  C)
> (cons  'A  '( ) )                       (A)
> (cons  '(A  B)  '(C  D))                ( (A  B)  C  D)
> (cons  '( )  '(A  B))                  ( NIL  A  B)
> (list  'A  'B  'C)                      (A  B  C)
```

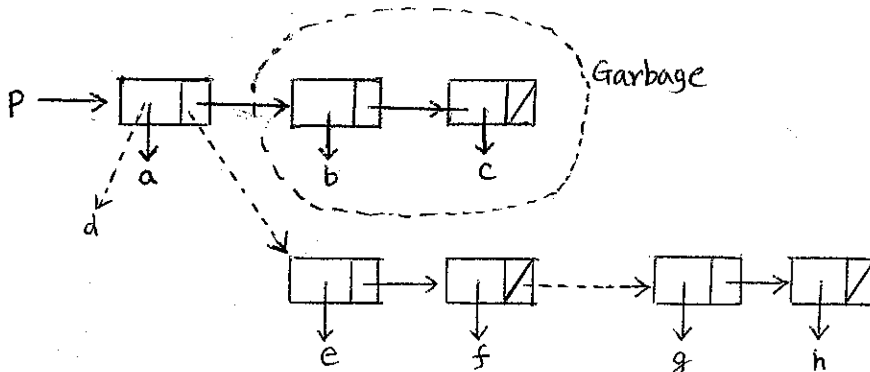
To build ((A B) C),

```
> (list (list  'A  'B)  'C)     or
> (cons (cons  'A  (cons  'B  nil))  (cons  'C  nil))       ((A B) C)
```

Destructive functions (side-effect)

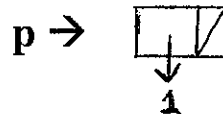
- rplaca (replace car)
- rplacd (replace cdr)
- nconc

> (setq p '(a b c))	(A B C)
> (rplaca p 'd)	(D B C)
> (rplacd p '(e f))	(D E F)
> (nconc p '(g h))	(D E F G H)



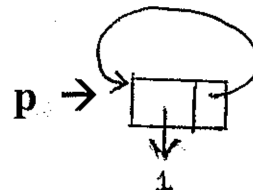
Note Garbage collection
- referencing count

Note: Be careful.



> (setq p '(1))
(1)

> (rplacd p p)
(1 1 1 1 ...)

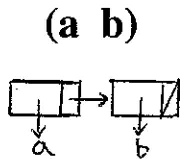


List representation

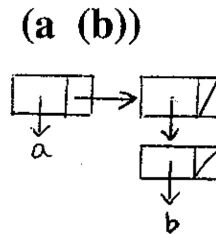
- **Atoms in Lisp are unique – global**
  Ob-list (harsh table)

- **Lists are stored in linked list.**

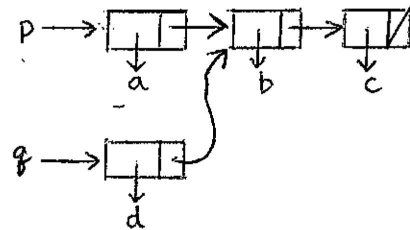
(a b)



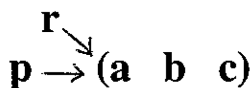
(a (b))

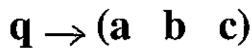


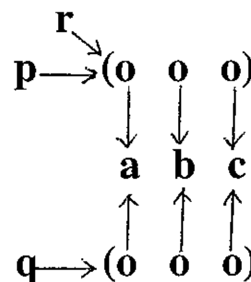
➢ (setq p '(a b c))
(a b c)
> (setq q (cons 'd (cdr p)))
(d b c)
> q
(d b c)



➢ (setq p '(a b c))
(a b c)
> (setq q '(a b c))
(a b c)
> (setq r p)
(a b c)





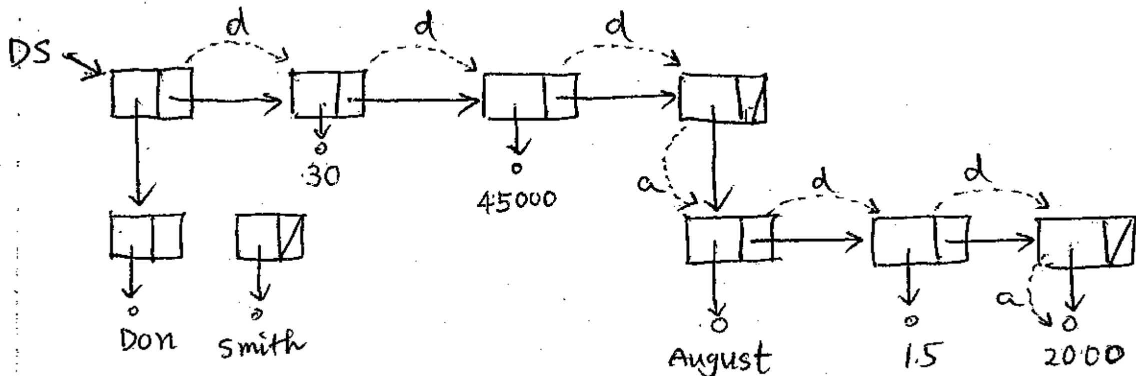


EX Personnel record for Don Smith.

```
                name          age   salary        hire-date
(set 'DS '( (Don Smith)       30    45000    (August 15 2000) ))
                 ①             ②      ③              ④
```

Note: n^{th} element of a list can be accessed by
(n-1) cdr's followed by a car.

- Last name : (car (cdr (car 'DS))) ≡ (cadar DS)
- salary : (caddr DS)
- Hire-date : (cadddr DS)



FUNCTION

- **Define functions – defun**

definition: (defun fname (para1 para2 … para n))

use: (fname arg1 arg2 … arg n)

Pass by value

```
➢ (defun square (num) (* num num))
  square
➢ (square 5)
  25

➢ (defun avg2 (x y)
       (/ (+ x y) 2))
  avg2
> (avg2 5 11)
  8
```

- **free and bound symbols**

```
➢ (defun sum-avg (x y)
     (setq sum (+ x y))
     (/ sum 2)))
  sum-avg
➢ (sum-avg 19 87)
  53
➢ sum
  106
➢ (sum-avg 5 29)
  17
➢ sum
  34
```

sum: free symbol

x, y: bound symbol

parameters

```
Script started on Sun 06 Apr 2008 05:16:48 PM MDT
bash-3.00$ clisp
  i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
  I I I I I I I      8     8   8           8     8     o  8    8
  I  \ `+' /  I      8         8           8     8        8    8
   \  `-+-'  /       8         8           8      ooooo   8oooo
    `-__|__-'        8         8           8           8  8
        |            8     o   8           8     o     8  8
  ------+------       ooooo    8oooooo  ooo8ooo   ooooo   8

Welcome to GNU CLISP 2.41.1 (2007-10-12) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2007

Type :h and hit Enter for context help.

[1]> (set 'ds '( (Don Smith) 30 45000 (August 15 2000) ) )
((DON SMITH) 30 45000 (AUGUST 15 2000))
[2]> (cadddr ds)
(AUGUST 15 2000)
[3]> (/ 8 2)
4
[4]> (/ 7 2))
7/2
[5]> (exit)
Bye.
bash-3.00$
Script done on Sun 06 Apr 2008 05:18:47 PM MDT
```

Scope

Dynamic scope rule (Scheme – static scope rule)

Name binding

- **property list: setq, defun — global**
- **parameter-argument correspondence**
- **local: (let ((x) ( ) ...)) → same as block**

  ↑ **strictly local in the environment**

Example

$$a \cdot x^2 + b \cdot x + c = 0 \qquad x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
(defun  roots (a b c)
    (list ( / ( + (- b) (sqrt (- (expt b 2) (* 4 a c)) )) (* 2 a))
          ( / ( - (- b) (sqrt (- (expt b 2) (* 4 a c)) )) (* 2 a)) ))

(defun roots (a b c)
    (let ( (d (sqrt ( - (expr b 2) (* 4 a c)) )) )   ← local
         (list (/ (+ (- b) d) (* 2 a))
               (/  (- (- b) d) (* 2 a)) )) )

  ➢ (roots 1  -3  2)
    (2  1)
```

Control Structure

- **Conditional Expression**

(cond (c_1 e_1) (c_2 e_2) … (c_n e_n))

(1)

```
(cond
   ( (null  x)  0 )
   ( (eq x y) (f  x))
   (  t (g  y)) )
```

←

```
if null(x) then  0
 else if (x = y) then f(x)
 else g(y)
end if
```

(2)

$$sg(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -1, & \text{if } x < 0 \end{cases}$$

```
(defun  sg (x)
   (cond  ( (plusp  x)      1)
          ( (zerop  x)      0)
          ( (minusp x)  -1)
   )
)
```

Note. Conditional interpretation (short-circuit)

Ex. **(or (null L) (eq (car L) 'key))**

If L is empty, second expression is not evaluated.

- **Iteration**

Ex. Find cumul $(n) = \sum_{i=1}^{n} i$

```
> (defun  cumul (n)
     (prog  ( (sum  0))              // if no local var, prog ()//
        loop (cond ( (= n  0) (return sum)))
              (setq sum (+ sum  n))
              (setq  n  (-  n  1))
              (go  loop)))
```

```
➢ (cumul  5)
  15
```

- **Recursion**

```
➢ (defun cumul (n)
     (cond ( (zerop   n)  0)
           (t (+ n  (cumul (-  n  1) )) )))
```

```
➢ (defun  factorial (n)
     (cond  ( (= n  0)  1)
            (t (*  n  (factorial (-  n  1)))) ))
```

- **My-sum**
  - **Find the sum of the numbers in a list**

```
(defun my-sum (a)
    (cond ( (null a) 0)
          ( t (+ (car a) (my-sum (cdr a)) )) ))

➤ (my-sum '(1 2 3 4 5 6 7 8 9 10))
  55
```

- **Mapcar**
  - **applies a function with each element in a list**
  - **Common Lisp has 'mapcar' as a built-in function.**

```
(defun mapcar (f x)
    (cond ( (null x) nil)
    (t (cons (f (car x)) (mapcar f (cdr x)) )) ))
```

Ex.

```
➤ (defun add1 (x)
      (+ x 1))

➤ (mapcar 'add1 '(2 0 0 8))
  (3 1 1 9)
```

- **my-append**
  - **(my-append ‘(a b) ‘(c d)) → (a b c d)**

  Note: (cons ‘(a b) ‘(c d)) → ((a b) c d)

```
(defun my-append (L M)
  (cond
    ( (null L) M)
    ( t (cons (car L) (my-append (cdr L) M)))))
```

- **my-reverse**
  - **reverse a list**

```
(defun my-reverse (L)
   (cond
     ( (null L) nil)
     (t (append (my-reverse (cdr L))
                (list (car L)) ))
   )
 )
```

- **palindrome**

```
(defun palindrome (L)
    (append L (reverse L)) )
```

- **count-atom**
  - **count the number of atoms that appear at all levels in a list**

```
(defun  count-atom (L)
   (cond
      ((null   L)   0)
      ((atom  (car  L))
           (cond
              ((eq (car  L) nil)  (count-atom (cdr L)) )
              (t (+ 1  (count-atom (cdr  L))))
           ))
      (t  (+  (count-atom  (car  L))
               (count-atom  (cdr  L)) ))
   )
 )
```

- **my-union (L1 L2)**

  (a c) (a b d) → (c a b d)

```
(defun my-union (L1  L2)
  (cond
     ((null  L1)  L2)
     ((member  (car  L1)  L2)  (my-union  (cdr  L1) L2))
     (t (my-union  (cdr  L1)  (cons  (car  L1)  L2))) ))
```

- **delete-atom**

  'a (a b a c) → (b c)

```
(defun delete-atom (atm L)
   (cond
     ( (null L) nil)
     ( (atom (car L))
        (cond
          ((eq atm (car L)) (delete-atom atm (cdr L)))
          (t (cons (car L) (delete-atom atm (cdr L))))
        ))
     (t (cons (delete-atom atm (car L))
              (delete-atom atm (cdr L))) )
   )
 )
```

- **transpose**

  - $\begin{pmatrix} a & b \\ c & d \end{pmatrix} \longrightarrow \begin{pmatrix} a & c \\ b & d \end{pmatrix}$

  ((a b) (c d)) ((a c) (b d))

```
(defun transpose (L)
   (cond
      ( (null (car L)) nil)
      (t (cons (mapcar #'car L)
               (transpose (mapcar #'cdr L)) ))
   )
 )
```