LISP

↙
expression oriented

prefix notation - easy to compute for repeated operation

(+ 3 5 4)

(define r 10) → defining variables

(define (SQUARE x) (* x x))

- Tree acculumation

(*(+ 3 5)(- 2 7))

Substitution < Applicative / normal order

Applicative - whenever an unknown is computed and then passed (i.e till primitive operation is obtained)

Normal

cond < cond / else / if

cond (<P1> <e1>)
(<P2> <e2>)
(<P3> <e3>)

If all fails then value of cond is undef cond ((> x 0) x)

else <e>

if condition → only two

(define (>= x y) (or (> x y)(= x y)))

(define (sumsquare x y) (+ (* x x) (* y y)))

(define (somefun x y z)

(and
~~(cond~~ (and (~~>x~~ y) (> ~~x~~ z)

~~(<y~~ z)

(cond (and (< x y) (< x z)

(sumsquare y z))

(~~con~~ and (< y x) (< y z)

(sumsquare x y)

)

else (sumsquare x z)

(define squareroot(x)

free variable → global scope

local variable → local scope

Bound/ Bind

(i) normal order evaluation
→ substitute the procedure and compute values only when needed

(ii) applicative order evaluation
↳ evaluate the parameters first and then use

Lisp uses this

cuberoot $\rightarrow \dfrac{(x/y^2 + 2y)}{3}$

```
(define (fib n) (
    (cond =n 0    0)
    ( =n 1    1)
    ~~else~~
    (+(fib (n-1)) (fib (n-2)))
```

fib(5)

fib(4)   fib(3)

```
(define (fib n)
    (fib_iter 1 0 n)

(define (fib_iter a b count)
    (if ( = count 0)
        b
        (fib_iter(+a b) a (- count 1))
```

```
(define (power b n)
    ( power_iter b n res)
(define (pow_iter b n res)
    (if (= n 0)
        res
        iter b (- n 1) (* res
```

```
(define power b n)
    ( cond (= n 0) 1)
    ~~(conde (remainder n 2) 1)~~

    ((even ? n) (square (fast_expt b(/n 2))

    (else (* b (fast_except b (-n 1))))


  ( define (even ? n)

        ( = remainder n 2) 0)


  (define  gcd (a b)
       if ( = b 0)

         a

       gcd( b (remainder a b))
```

Primality :-

```
   (define (smaller_divisor n)
         ( find divisor n 2)

    define ( first_divisor n test_driver)
       ( cond (( > square test-divisor) n ) n)
      ((divides ? test-divior n)test_divisor)
       (else (find_divisor n) test-driver)

  (define (divides ? a b)
      (= remainder b a) 0))

  (define prime? n)
      (= (smaller_divisor 0))
```

Fermet's theorem (for checking prime

$a^n \bmod n = a$

Higher order → provides abstraction for another procedure

define (square x) (* x x) ← Higher order

~~define~~ (othersquare x (* x x 1))

lambda

lambda (x) (* x x)

let

↳ local variables

(let (var1) (expr1)
(var2) (exp2)

) (body))

let (a . (+ 1 2)
b . (+ a 5)
(+ a b))

let always bind the value only inside the body of it.

(cons arg1 arg2)

construction

car → arg1 → content of address register

cdr → arg2 → content of decrement register.

define x (cons 1 2)

define y (cons 3 4)

define z (cons x y)

$x^2 - x = 0$

$x(x-1)$

$x^2 = x$

$x^2 - x = 0$

$x(x-1) = 0$

$x = 0$ or 1

(i) car z

1 2

(ii) car (car z)

↳ 1

General methods

1. Finding roots → Interval halving $f(x) \le 0 \le f(a)$

2. fixed point

$f(x) = x$

Procedure can be reused → first class. (variable

passed as arg

have return value

can be used as data structure)

Returning procedure and passing procedure

(define (arg-dump f)

(lambda (x) (arg' (f x)))

((arg_dump square) 5)

Abstraction of data

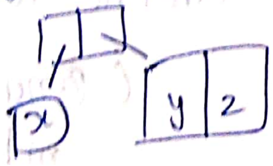
* Message Passing

Abstraction Barrier

Box & Pointers

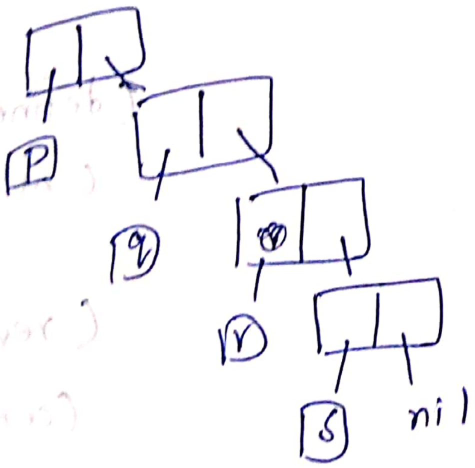
(cons x y)

(cons x (cons y z))



(cons p ((cons q (cons r (cons s nil))))

List



(define newlist (list 1 2 3 4 5))

(car (cdr newlist))

define (scale-list items factor)
    (if (null? items)
reveal
        nil
        (cons (* (car items factor))
            (scale-list (cdr items) factor)

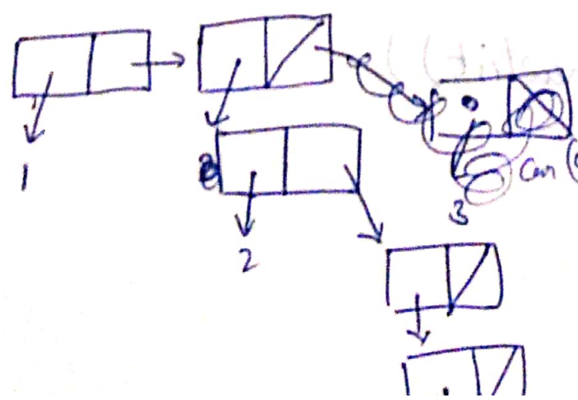
null? cdr(cdr (x))

(define reverse list)
    (null? list)
        nil
    (cons(reverse cdr(list)
        car(list))

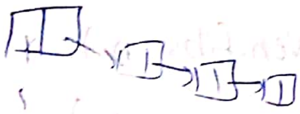
(define reverse list) list2
    (null ? list)
        nil
    (reverse cdr(list)list2
    (cons car(list) list2)

(list 1 (list 2 (list 3))

Con 2



pair?



define x list(1 2 3)

define y list(4 5 6)

append x y

'(1 2 3 4 5 6)

cons x y

('(1 2 3) '(4 5 6))

(list (x y))

sequence

'((1 2 3) 4 5 6)

sequences as conventional interface

```
(define (sum-odd square tree)
  ( cond((null? tree) 0)
   ((not (pair? tree))
    (if (odd? tree) (square tree) 0))
    (else (+ sum-odd-square (car tree)
          (sum-oddsquare (cdr tree))))
```

```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
      nil
      (let ((f (fib n))
        (if (even? f)
          (cons f (next (+ x 1))
          (next (+ k 1))))
   (next 0)))
```

1. enumerate 2. map 3. filter ~~odd~~ 4. accumulate.

```
(define (filter predicate sequence)
(cond ((null? square) nil)
  ((predicate (car sequence)
   (cons (car sequence)
     (filter predicate (cdr sequence)))
   ( else (filter predicate (cdr sequence))

(define (accumulate op initial sequence)
  (if (null? square)
    ~~nil~~ initial
    (op (car sequence)
  accumulate op initial (sequence)))
```

```
(define (enumerate-tree tree)
 (cond ((null? tree) nil))
  ((not (pair? tree)) (list tree))
  (else (append (enumerate-tree (car tree)
         (enumerate-tree (cdr tree) )))
```

```
(define (sum-odd square tree) )
                        (map square
                                  odd
 ( ~~enumerate~~ accumulate + 0 ( filter (enumerate-tree tree))
```

nested mapping

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
i+j	3	5	5	7	7	7	11

```
(accumulate append
        nil
        (map (lambda(x)
            (map( lambda (j) (list i j))
            (enumerate interval i -i 1))
            (enumerate interval 1 n))
```

```
(define (Prime-sum? Pair)
    ( Prime? (+ Car pair) (cdr pair) )
```

```
(define (make-pair-sum pair)
    (list (car pair) (cadr pair)
        (+ (car pair) (cdr pair)))
```

```
(define (prime-sum-pairs n)
    (map make-pair-sum (filter prime-sum?)
        (filtermap (lambda (i) (map (lambda (j)
                                      (list i j))
            (enumerate_interval 1 (-1 i))
        (enumerate-interval 1 n)))
```

Symbolic Data :-

```
(list 'a b)
    (a 5)
```

Symbolic Data

```
(define (derive exp var)
    (cond ((number? exp) 0)
        ((variable? exp)
        (if (same-variable? exp var) 1 0))
(make_sum (deriv (addend exp) var)
        (deriv (augend exp) var))
```

```
((product? exp)
(make-sum (make-product (multiplier exp)
 (deriv (multiplicand exp) var)
    else
 (make -product (deriv (multiplier exp) var)


(define (variable? x) (symbol? x))
(define (some-variable? v1 v2)
      (and (variable? v1) (variable? v2) (eq? v1 v2)

(define (make-sum a1 a2) (list '+ a1 a2))
  (define (make-product m1 m2) (list '* m1 m2))

(define (sum? x)
     (and (pair? x) (eq? (car x)' +))
(define (addend s) (cadr s))
(define augend s) (cadr 3))

(define (product? x)
   (and (pair? x) (eq? (car x) '*))
(define (multiplier p) (cadr p))

(define (multiplier p) (cadr p))

(define (multiplicand p) (cadr p))
```

Sets :- Union, Intersection, Adjoint, element

```
(define (element-of-set? x set)
    ((null? set) false)
     ((equal? x (car set)) true)
      (else (element-if-set? x (cdr set))))

(define (intersection-set set1 set2)
 (cond ((or (null? set1) (null? set2)) '())
 ((element-of set? (car set1) set2)
     (cons (cars set1)
                 (intersection-set (else set1 set2))
    (else (intersection-set (cdr set1) set2))
```

multiple data representation:-

1) Complex
2) Polar

$x = r\cos\theta$
$y = r\sin\theta$

$\frac{x}{y} = ta$

$\frac{y}{x} = \tan\theta$

$\theta = \tan^{-1}(y/x)$

Scanned by CamScanner

chapter 3 objects

```
set!    &lt;var&gt; &lt;exp&gt;

begin (exp, exp ...   exp)


(define balance 100)
(define (withdraw amount)
    (if (>= balance amout)
        (begin (set! balance (-balance amout))
               balance)
               "insufficient funds"))


Decrementy balance
    ↳ (set!   balance (-balance amout)) )


(define  (make-accout  balance)
   (define  (withdraw amout )
     (if  (>= balance  amout)
          (begin(set! balance (-balance amout)
                        balance)
                   "insufficient balance")


   (define  ( deposit  amout )
     (set!  balance   (+ balance  amout) balance)

     (define (dispatch m) (cond(eq? m 'withdraw)
                                     withdraw)
                               ( eq? m  'deposit)
                                        deposit)
                               ( else (error "
```

Cost of Assignments

i) Functional programming

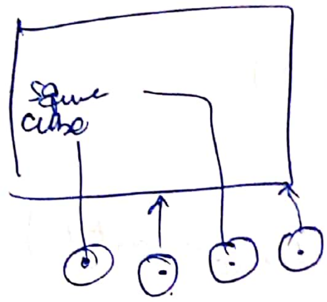
ii) Imperative

Pitfalls of imperative programming

Order of evaluation

Environment variable

Environment model



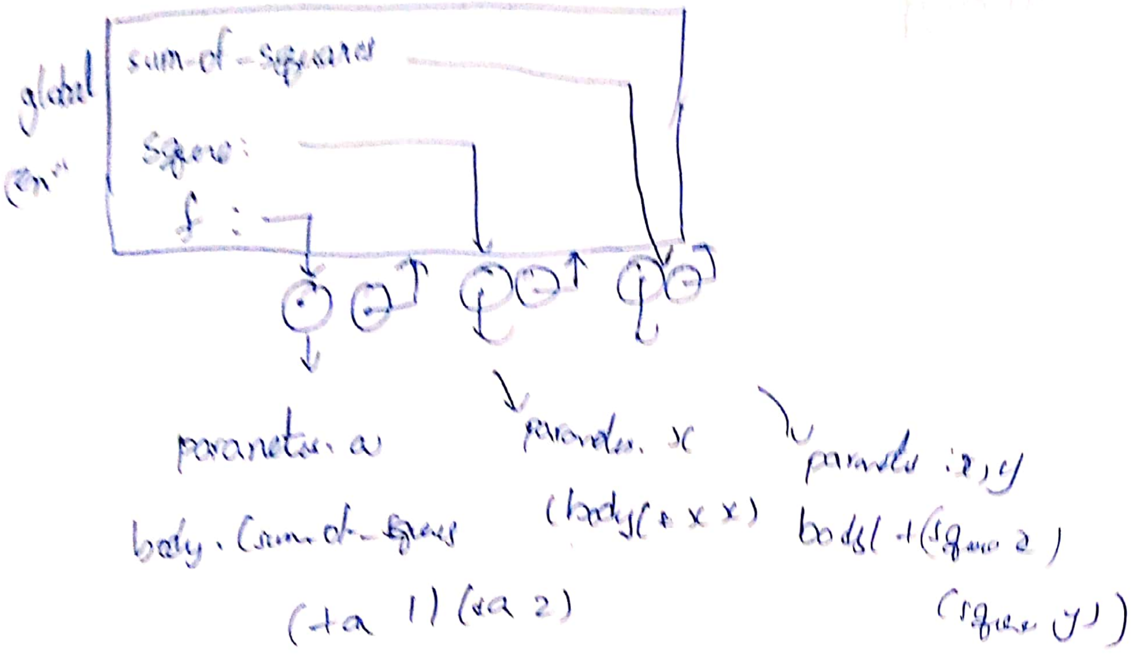
Parameter: x

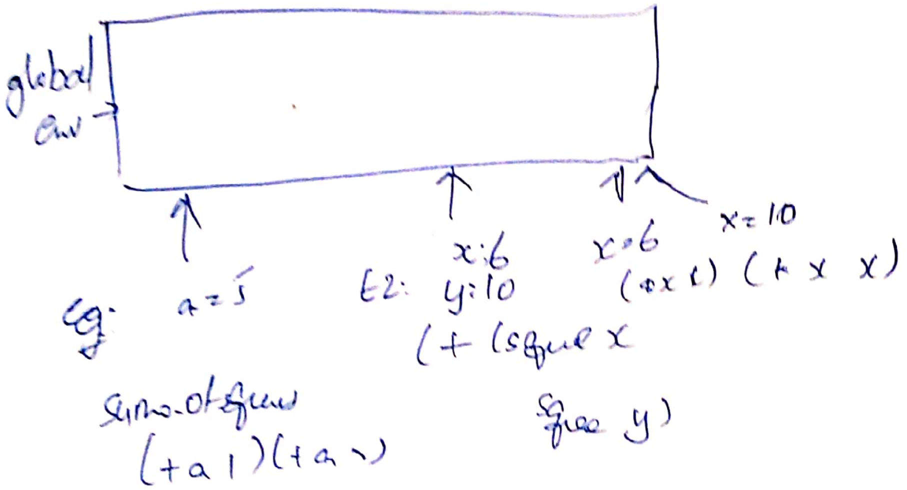
body:

```
(define (square x)
        (* x x))

(define (sum_of_squares x y)
        (+ (square x) (square y))

(define (f a)
        (sumofsquares (+ a 1) (+ a 2))
```





Whenever a function call → new environment is created.

Ex 3.10

mutable structures

lists are settled ~~passed~~ by reference in lisp

set!

set-car!
set-cdr!

```
(define (cons x y)
  (let ((new (get-new-pair)))
    (set-car! new x)
    (set-cdr! new y)
```

Concurrency