LISP (a Schema dialect)

Primitive Operations

```
(+ 3 (* 5 8))
```
40
43

```
(+ 1 3 4)
```
8

LISP LAMBDAS

```
(lambda (x) (* x x))
((lambda (x) (* x x)) 5)
```
param

```
>> 25
```

PYTHON LAMBDAS

```
square = lambda x : x * x
square(5)
```
param

```
>>25
```

LISP CONSTANTS

```
(define PI 3.14)
(* 2 PI)
>> 6.28
```

LISP CONDITIONALS

```
(= 3 5) >> #F
(< 3 5) >> #T
(EVEN ? 4) >> #T
```

LISP FUNCTIONS

params

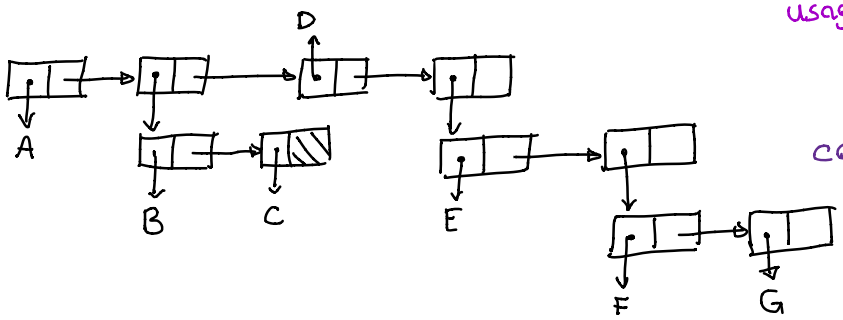
```
(define (hypotenuse a b)
    (sqrt (+ (* a a) (* b b)))
)

(define (factorial n)
    (if (= a 1)
        1
        (* n (factorial (- n 1)))
    )
)
```

```
(define (leap ? y)
    (cond
        ((zero ? (modulo y 400)) #T)
        ((zero ? (modulo y 100)) #F)
        (else (zero ? (modulo y 4)))
    )
)
```

LISP LISTS

'(A (BC) D (E (F G)))



CAR → returns first element of a list.

CDR → returns first Element.next

usage: (car (car (cdr x)))
↓
list

CONS → constructs a list

usage: (cons 'A 'BC) >> 'ABC

LISP CAR / CDR / CONS EXAMPLES

(car ('AB)) >> A

(car ('(AB)CD)) >> (AB)

(car 'A) >> error

(car '(A)) >> A

(car '()) >> error

(cdr '(ABC)) >> (BC)

(cdr '((AB)C D)) >> (C D)

(cdr 'A) >> error

(cdr '(A)) >> ( )

(cdr '()) >> error

(cons 'A '(BC)) >> '(ABC)

(cons 'A '((BC)) >> '(A (BC))

(cons '(A) '(BC)) >> '((A) BC)

(cons '(AB) '(CD)) >> '((AB) CD)

(cons '() '(BC)) >> '(() BC)

(cons 'A 'B) >> '(AB)

(list 'A 'B 'C) >> '(ABC)

LISP NULL & EQUIVALENCE CHECK

(list? '()) >> #T

(list? '(A)) >> #T

(list? 'A) >> ~~#T~~

(null? '()) >> #T

(null? '(A)) >> ~~#F~~

(null? 'A) >> #T

(eq? 'A 'A) >> #T

(eq? 'A 'B) >> ~~#F~~

(eq? '(A) '(AB)) >> #F

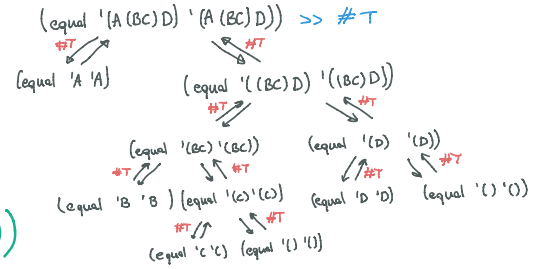
(eq? '(A B) '(A B))
>> #T or #F

(eq? 3.4 (+ 3.0 0.4))
>> #T or ~~#F~~

eq? cannot be used to compare lists

LISP EQUALS METHOD

```
(define (equal x y)
        (cond
          ((not (list ? x)) (eq?   x y))
          ((not (list ? y)) #f)
          ((null ? x) (null? y))
          ((null ? y) #f)
          ((equal (car x) (car y)) (equal (cdr x) (cdr y)))
          (else #f)
        )
)
```



LISP LENGTH METHOD

```
(define (length x)
    (if (null? x)
        0
        (+ 1 (length (cdr x)))
    )
)
```

LISP APPEND METHOD

```
(define (append x y)
    (if (null ? x)
        y
        (cons (car x) (append (cdr x) y))
    )
)
```

usage: (append '(AB) '(CD))
>> '(ABCD)

(append '(AB) '(CD))
↓
(cons 'A (append '(B) '(CD)))
↓
(cons 'A (cons 'B (append '() '(CD))))

'(CD)
'(BCD)
'(ABCD)

LISP MEMBER METHOD

```
(define (member (x y)
    (if (null? y)
        #F
        (if (equal x (car y)
            #T
            (member x (cdr y))
        )
    )
)
```

```
(define (member x y)
    (cond
        ((null? y) #F)
        ((equal x (car y) #T)
        (else (member x (cdr y)))
    )
)
```

usage: (member '(A B) '(X Y (A B) D))
>> #T

LISP DISCRIMINANT FUNCTION

```
(def (roots a b c)
    (let* (
        (minus_b (- 0 b))
        (two_a (+ a a))
        (discr (- (* b b) (* 4 a c)))
        (delta (sqrt discr))
    )
    (list (/ (+ minus_b delta) two_a) (/ (- minus_b delta) two_a))
    )
)
```

↳ returns a list.

usage: (roots 1 0 -1) → $(x^2 - 1)$
>> (1 -1)

LISP ADD FUNCTION

```
(define (add x)
    (if (null ? x)
        0
        (+ (car x)(add (cdr x)))
    )
)
```

usage: (add '(1 2 3))
>> 6

```
(define (add x)
    (if (null ? x)
        0
        eval (cons '+ x))
    )
)
```

adding a plus at the beginning and evaluating.