LISP EXPRESSIONS

- atom / number
  \ symbol

- list

  (op arg1 ··· argn)

  ↓

  function: built-in / user specified

  special operations - setq
  quote.

- selectors:

  car, cdr (first, rest)

- constructors:

  cons, list.

  ↳ only 2 arguments

* Create list (1, 2)

```
> (cons 1 (cons 2 NIL))
```

› Create list (1 (2 3) 4)

```
> (setq x (cons 2 (cons 3 NIL)))
> (cons 1 (cons x (cons 4 NIL))
```

LIST:

```
> (list 1 2 3)
     > (1 2 3)

> (list 1 (list 2 3) 4)
     > (1 (2 3) 4)
```

CONS vs LIST:

```
(cons H T)
      ↳ α
      (car α) → H

      (cdr α) → T

(list a1 a2 ... an)
      (a1 a2 ... an)
```

BOOLEAN EXPRESSIONS:

false : NIL (empty list)

true : t (anything other than NIL)

PREDICATES:

```
( >  .  . )

( <  .  . )

( =  .  . )

( <=  .  . )

( >=  .  . )
```

```
> (> 3 1)
> t
> (< 3 1)
> NIL
```

→ atom : is the expression an atom

→ listp : is it a list

→ null : is it null

→ equal : are they equal.

```
> (atom 3)
      > t

> (atom 'x)
      > t

> (atom '(a b))
      > NIL

> (listp '(a b))
      > t

> (setq x '(a b))

> (setq y '(a b))

> (equal x y)
      > t
```

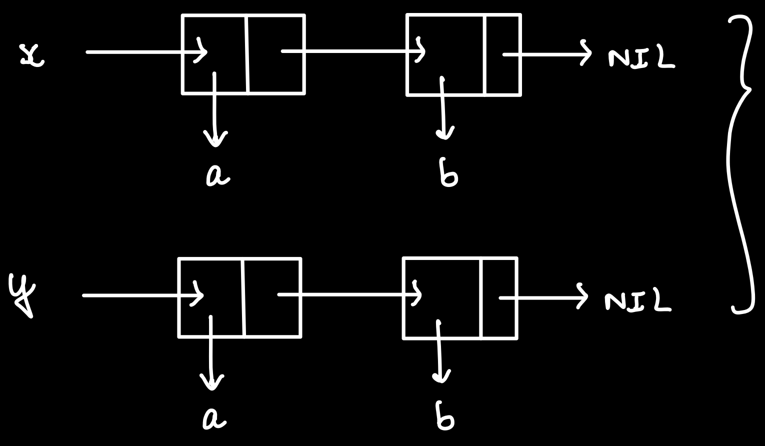
```
> (atom NIL)
      > t

> (listp NIL)
      > t
```

equal vs eqL

```
> (setq x '(a b))
> (setq y '(a b))
> (eqL x y)
      > NIL
```

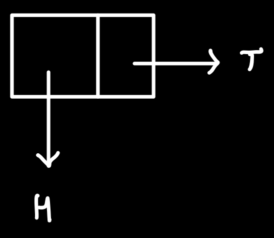
In memory



different in memory.

(cons H T)

Creates a unit and points to atom H. Next set to tail list, T.



BOOLEAN CONNECTORS:

NOT, AND, OR

```
> (NOT t)
    > NIL
> (NOT NIL)
    > t

> (NOT 3)
    > NIL
> (NOT (> 3 1))
    > NIL

> (AND arg1 ... argn)
```

→

evaluates left to right

- if any NIL, stop, return NIL
- else return argn.

```
> (AND (+ 2 3) (+ 1 3))
        > 4
> (AND (+ 2 3) (cdr '(a)) (+ 1 3))
        > NIL
> (OR arg1 ... argn)
```

⟶

evalutes left to right

- if all NIL, return NIL

- else return first NON-NIL

```
> (OR (+ 2 3) (+ 1 3))
        > 5

> (OR NIL (+ 2 3) t)
        > 5
```

BRANCHING :

```
(cond (bexp exp1 ... expn)
      (bexp exp1 ... expm)
       ⋮
      (bexp exp1 ... expk))
```

Check bexp

→ if true, execute that

→ all false return NIL.

```
> (setq x 3)
> (cond ((= x 0) 'zero)
        ((> x 0) 'positive)
        ((< x 0) 'negative))
    > positive .
```

```
> (cond ((= x 0) 'zero)
        ((> x 0) 'positive)
        ( t      'negative))
    > positive .
```

→ default

FUNCTIONS:

```
> (defun   square (x)
         (*   x   x))

> (square   3)
       > 9

> ( square   (square   3))
       > 81

> (square   (+   1  2))
       > 9

> (defun   sum   (a   b)
        (+   a   b))

> (sum    3   7)
      > 10

> (defun   abs   (x)
       (cond    ((=   x    0)    0)
                ((>   x    0)    x)
                (t              (-x))))
```

```
> (abs  -3)
      > 3.
```

LET:

```
> (let  ((x  3)
          (y  4))
         (+  x  y))
```

local binding in parallel

```
> 7

> x
  ↳ error

> (setq  x  2)
```

→ x = 2

```
> (let  ((x  3)
          (y  (+  x  2)))
      (*  x  y))
```

parallel

x = 3

y = x+2
= 2+2 = 4

```
      > 12
```

```
>(setq  x  5)                              x = 5

> (+  (let  ((x  3))                        x = 3              ]  local
                                                               ]
          (+  x  (*  x  10)))               3 + 30 = 33        ]

      x)                                    x = 5

    >  38
```

LET *

Sequential evalution, still local

```
>(setq  x  5)                               x = 5

>(let*  ((x  3)                             x = 3

          (y  (+  x  2)))                   y = x + 2 = 3 + 2 = 5

     (*  x  y))

    > 15

> x

    > 5
```

RECURSIVE FUNCTIONS:

```
(defun odd? (x)
    (cond ((= x 0) NIL)
          ((= x 1) t)
          (t (odd? (- x 2)))))
```

```
> (odd? 5)      ↖ t
    ↘ (odd? 3)    ↖ t
        ↘ (odd? 1)

    > t
```

⟶ FACTORIAL

$$n! = \begin{cases} 1 & \text{if} \quad n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

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

SUM OF LIST:

```
(1 2 3) → 6

(1) → 1

( ) → 0
```

```
(defun sum_list (L)
       (cond ((null L) 0)
             (t (+ (car L) (sum_list (cdr L))))))
```

IS THE NUMBER, x IN LIST, L:

```
> (member? 'a '(a b c))
        > t

> (member? '(x y) '(1 (x y) 7))
        > t

> (member? 'x '(1 (x y) 3))
        > NIL
```

```
(defun member? (x L)
    (cond ((null L) NIL)
          ((equal x (car L)) t)
          (t (member? x (cdr L)))))
```

LAST ELEMENT OF LIST:

```
> (last '(1 (a b) 7))
      > 7
> (last '(a))
      > a
> (last NIL)
      > NIL
 (defun last (L)
      (cond ((null (cdr L)) (car L))
            (t (last (cdr L)))))
```

n^{TH} ELEMENT OF A LIST:

```
> (nth '(a b c) 0)
        > a
 > (nth '(a b c) 1)
        > b
```

Assume $n \in [0, len(L) - 1]$

```
(defun nth (L n)
       (cond ((= n 0) (car L))
             (t (nth (cdr L) (- n 1)))))
```

REMOVE ALL OCCURRENCES OF AN ELEMENT:

```
> (remove 'x '(1 x 7))
      > (1 7)

 > (remove 3 '(1 3 4 3))
      > (1 4)

 > (remove '(a b) '(7 (a b) 2))
      > (7 2)
```

```
(defun remove (x L)
    (cond ((null L) NIL)
          ((equal x (car L)) (remove x (cdr L)))
          (t (cons (car L) (remove x (cdr L))))))
```

APPEND A LIST TO ANOTHER LIST:

```
> (append '(a b) '(1 2))
      > (a b 1 2)

> (append NIL '(x y))
      > (x y)

> (append '(a) '(3 x))
      > (a 3 x)
```

```
(defun append (L1 L2)
    (cond ((null L1) L2)
          (t (cons (car L1) (append (cdr L1) L2)))))
```