# Lecture 3

Recursion - the most important topic in this course.

## Structures

### User-defined data type

Integer, String, Boolean

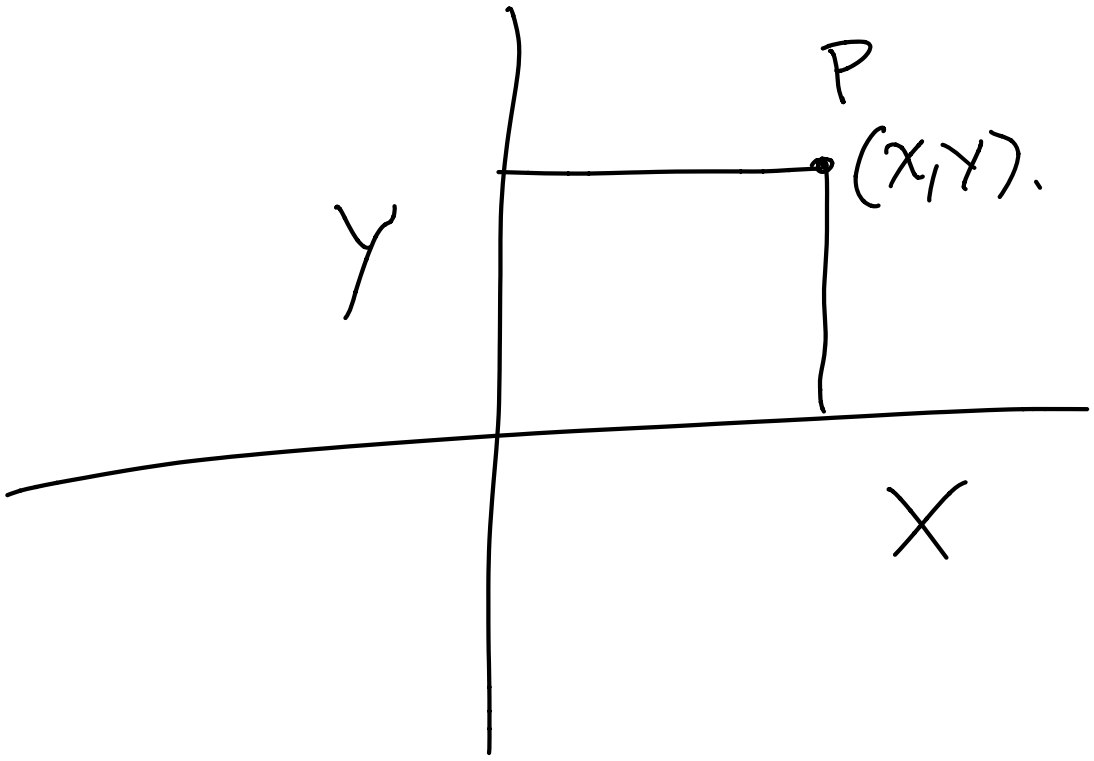
App for maps google maps

positions or points.

Denote a point

x-axis, y-axis

x-coordinate, y-coordinate.




```
(define-struct Point
  ([X : Real]
   [y : Real]))
```

Your name for the structure

Fields

You have to define.

(: my-point : Point) ← Type Annonatation

(define my-point (Point 3 -2))

- Creating a point with x value 3 and y value -2 and storing it to my-point.

my-point ← variable

↓

Storing (Point 3 -2)

(Point 3 -2) → "Constructor"

↳ Creates a point.

Selector functions.

(Point-x my-point) ⇒ 3

(Point-y my-point) ⇒ -2

(Point? ) if data type is of Point.

Creating a structure to denote points in 3-d

```
(define-struct 3D-Point
  ([x : Real]
   [y : Real]
   [z : Real]))
```

Suppose you have two points, you want to compute distance between them. $P1 = (x1, y1)$

$P2 = (x2, y2)$

$$dist(P1, P2) = \sqrt{(x1-x2)^2 + (y1-y2)^2}$$

Writ a racket function which takes two points any inputs and computes the distance between them.

```
(: distance : Point Point -> Real)
(define (distance p1 p2)

(sqrt (+ (* (- (Point-x p1) (Point-x p2))
            (- (Point-x p1) (Point-x p2)))
         (* (- (Point-y p1) (Point-y p2))
            (- (Point-y p1) (Point-y p2))))))
```

Common mistake Point ≠ point

Point-x using point-x ✗

The contructor functions, selector function depend on structure and field names.

```
(define struct 3D-Point
   ( [x : Real]
     [y : Real]
     [z : Real]) )
```

```
(3D-Point 1 0 1)
 3D-Point-x
 3D-Point-y
 3D-Point-z
```

Check expect for the distance function

```
(check-within (distance (Point 1 0)
                        (Point 2 0))
              1
              0.01)
```

Functions you get for free

(Point — Constructor.

(Point-x — Selector functions.

(Point-y

(Point? — Checks if a data type is of type Point.

# Union types

```
(define-struct Circle
  ([radius : Real]
   [center : Point]
   [color : Symbol]))

(define-struct Square
  ([side-length : Real]
   [center : Point]
   [color : Symbol]))

(U Circle Square)
```

→ Data type which is union of Circle or Square.

Write a function which takes either a square or circle as input and outputs its color.

```
(: shape-color: (U Circle Square) → Symbol)
(define (shape-color shape)
  (cond
  [(Circle? shape) (Circle-color shape)]
  [(Square? shape) (Square-color shape)]))

(check-expect (shape-color
               (Circle 3.0 (Point 0 0)
                       'R))  'R)
```

Input

Write a function that takes a Circle or Square as input and outputs its area.

```
(: area : (U Circle Square) -> Real)
(define (area shape)
  (cond
  [(Circle? shape) (* pi (expt
                           (Circle-radius shape)
                           2))]
[else (expt (Square-side-length shape)
            2) ]))
```

# Type - Definitions

An alias for types.

(U Circle Square)

(define-type Shape (U Circle Square))

If you use Shape → (U Circle Square)

Phonebook is now alias for Name → String. (a function which takes a Name as input

and outputs a string.

```
(define-type Name String)
(define-type Phonebook (Name -> String))
```

```
(define-type Email-Address-Book
              (Name -> String))
```

Idea is the make your code more redable.

```
(define-type INT Integer)
```

# Functions on Types

Basically functions which takes types as inputs

Example : Listof

(Listof String)
(Listof Integer)

It takes a data type as input

and a new data type which is list of that datatype.

# Symbols

Symbols are both data types and values.

```
( define-type Color (U 'red 'blue
                       'green) )
```

```
(U 2 S 3)
```

```
define-type piece-color (U 'red
                           'black))
```

```
( define-type Dir (U 'north 'east
                     'south 'west) )
```

Suppose you are making a game. of Cards

Data definitions or Interface.

Card - Suit (diamond, heart, club or spade)

Card - rank (A, J, Q, K 2-10))

A=1, J=11, Q=12, K=13

```
(define-type Suit (U 'diamond
                     'heart 'club
                     'spade))
```

```
(define-struct Card
  ([suit : Suit]
   [rank : Integer]))

(define-type Deck (Listof Card))
```

Write a function to check if a card is red.

```
(: red? : Card → Boolean)
  (define (red? card)
    (or
(symbol=? (Card-suit card) 'heart)
(symbol=? (Card-suit card) 'diamond)))
```

```
(check-expect (red? (Card 'heart 6))
              #t)
(check-expect (red? (Card 'club 5))
              #f).
```

Write a function which checks if a card is black.

```
(: black? : Card → Boolean)
(define (black? card)
  (not (red? card)))
```

```
(: black? : Card → Boolean)
(define (black? card)
```

```
(or
 (symbol=? (Card-suit card) 'club)
 (symbol=? (Card-suit card) 'spade))
```

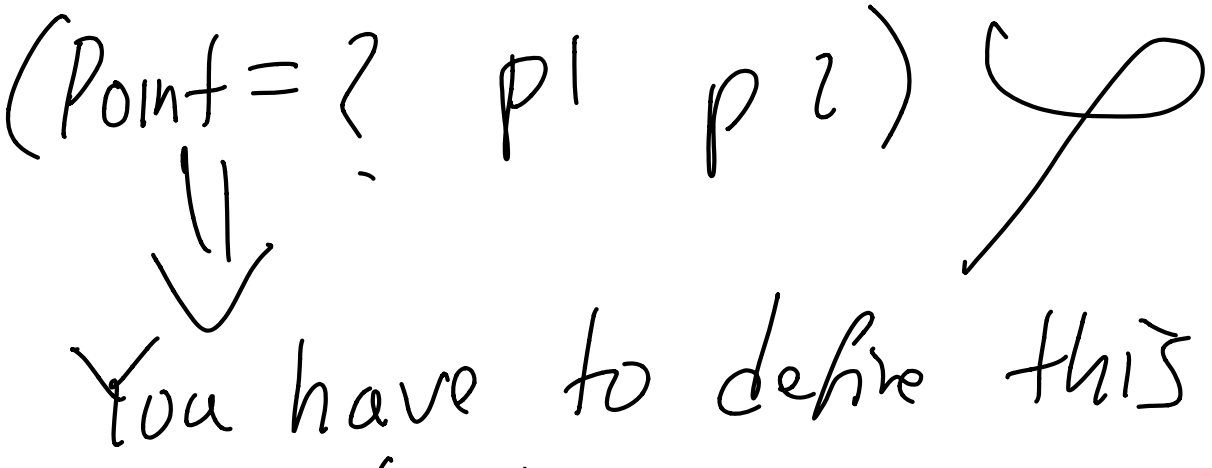
Write a function to check if two cards are the same.

```
(: card=? : Card Card → Boolean)

(define (card=? card-1 card-2)
 (and
  (symbol=? (Card-suit card1) (Card-suit
                               card2))
  (= (Card-rank card-1) (Card-rank card-2))))
```

Two cards are same if they have the same rank and suit.

Remark: If you want to check if two structures of a specific type are equal you have to write your own functions.

(Point=? p1 p2)



You have to define this function.

Write a function

input1: Suit

input2: Rank

output: Deck (List of cards)

Suppose my input is 'diamond and 3

Output: (list (Card 'diamond 1)
(Card 'diamond 2)
(Card 'diamond 3))

↓

Here 3 is Rank of input.

['D 1) ('D 2) ('D 3)]

Bar case:

(define (cards-in-suit suit rank)

If rank is 1
then there will be only one
element in ony output.
Input is 'diamond and 1
& ouput.
[(list (Card 'diamond 1))]

When rank=1 ouput ↓
(list (Card suit 1))

# How build list recursively

## Template

1 → Create the first element of the list.

2 → Recursively create a list of n-1 elements.

There are two ways to combine lists - list-appends

→ append list any two size.

Cons

→ add a single element to the list.

```
(: cards-in-suit-helper: Suit Integer
   -> Deck)
(define (cards-in-suit-helper suit rank)
  (cond
    [(= rank 1) (list (Card suit 1))]
    [else (cons (Card suit rank)
                (cards-in-suit-helper suit (- rank 1)))]))
```

There are two functions.

- list-appends - it combines two or more than two lists of any size.
- cons - which add an element to the front of the list.

[('diamonds 4) ('diamonds 3)

('diamonds 2) ('diamonds 1)].

('diamond 4) ← we create.

[('diamond 3) ('diamonds 2)
('diamonds 1)] →

Take one
by recursion

cons

[('diamond 4) ('diamond 3)
('diamond 2) ('diamond 1)]

Homework 2- question 6

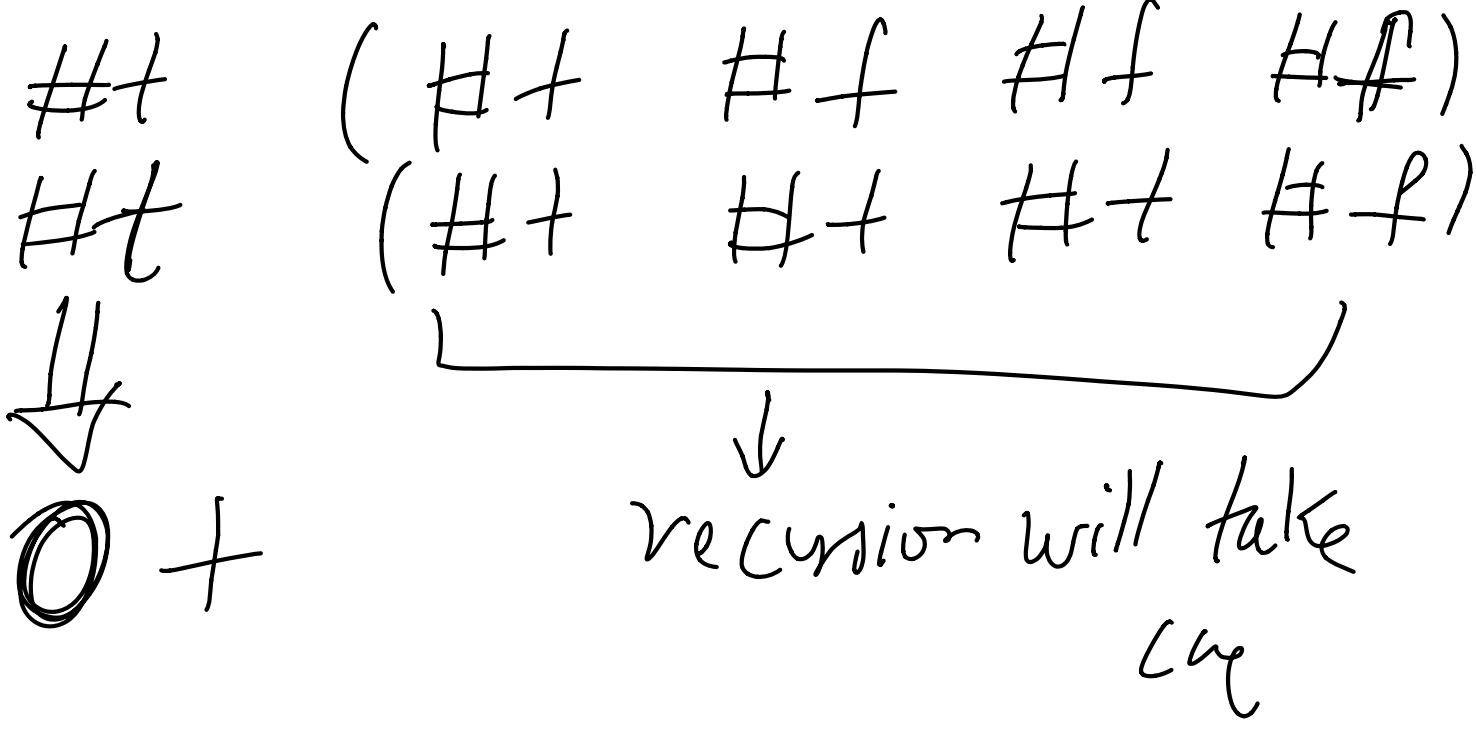
(#t #t #f #f #f) ✓ — my list

(#t #t #t #t #f) ✓ — friend's

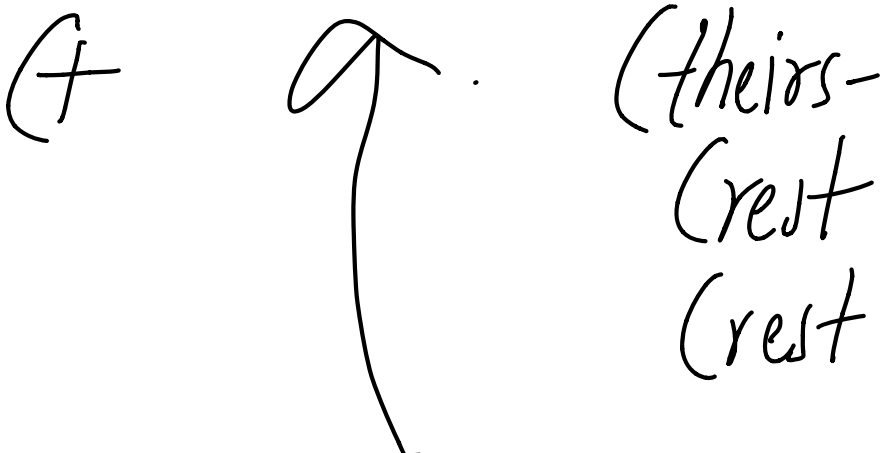
↑ I don't have this pokemone but my friend has it.

(#t #t #f #f #f)

(#t #t #t #t #f)

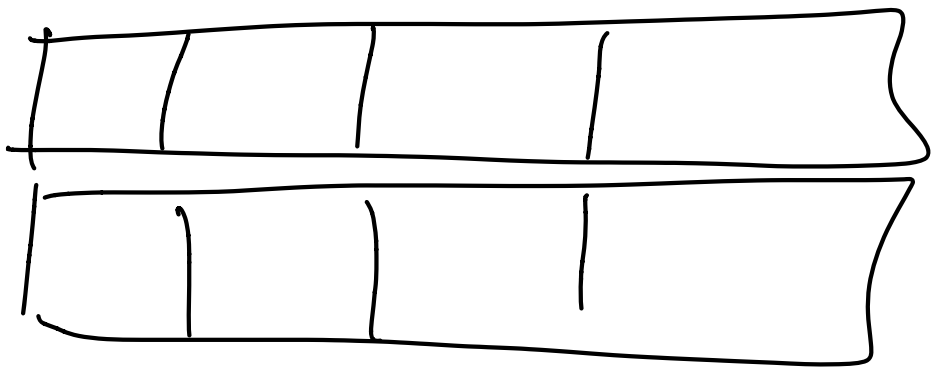


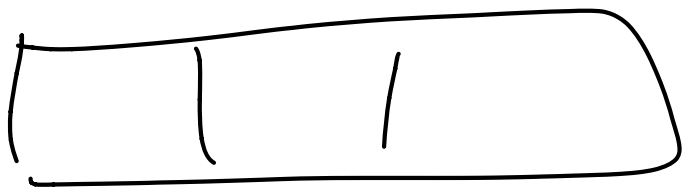

Extract the first element
from both the lists.
If it is false in my list
and true in my friend's list
then 1 else it is 0.

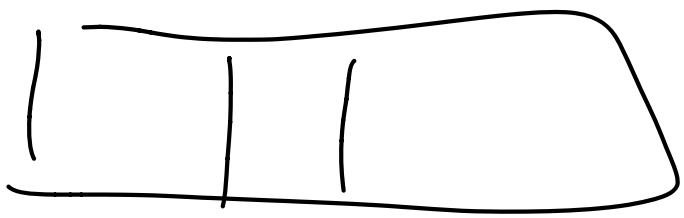


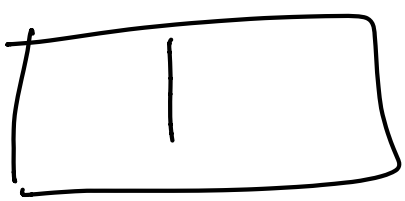
```
(if (and (boolean=? (first my-list)
                    #f)
         (boolean=? (first friend-list)
                    #t))
    1
    0)

(+ (if (and (boolean=? (first my-list) #f)
            (boolean=? (first friend-list)
                       #t))
       1
       0)
   (their-not-mine
     (rest my-list)
     (rest friend-list)))
```

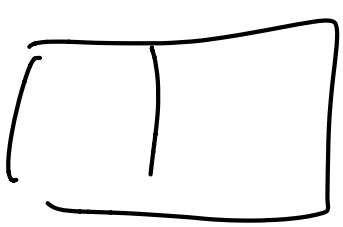
Base case.

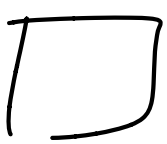


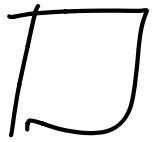








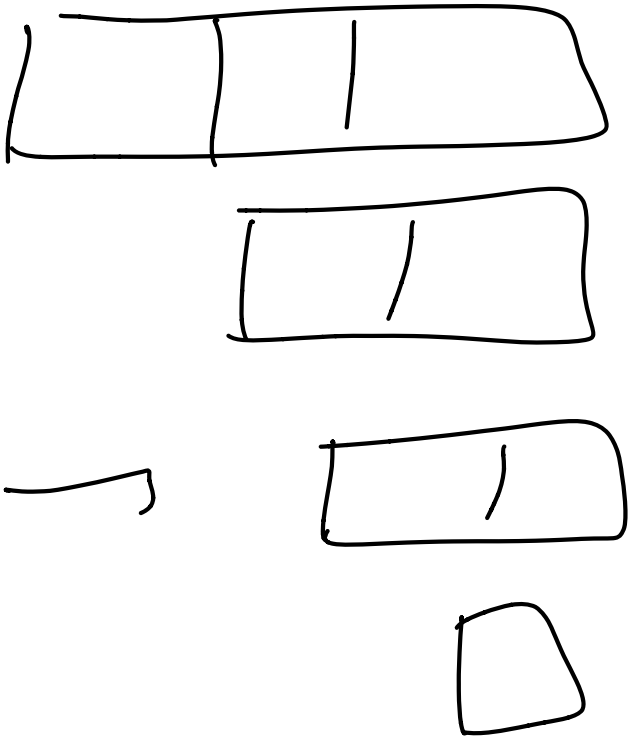




→ empty list.

⇒

If the lists are of the same length ultimate we will get two empty lists.



→ Not empty

→ empty

If the lists are not of the same length one list will be empty other will not be empty.

```
(: theirs-not-mine : (List of Boolean)
                     (List of Boolean)
                  -> Integer)
```

Input 1

Input 2

```
(define (theirs-not-mine my-list
                         friend-list)
```

```
(cond
  [(or (and (empty? my-list)
            (not (empty? friend-list)))
       (and (empty? friend-list)
```

```
      (not (empty? my-list))))
 (error "Length not same")]

[(and (empty? my-list) (empty?
       friend-list)) 0]

[(+ (if (and (boolean=? (first my-list)
                        #f)
             (boolean=? (first friendlist)
                        #t))
        1
        0)

   (theirs-not-mine (rest my-list)
                    (rest friend-list)))]))
```