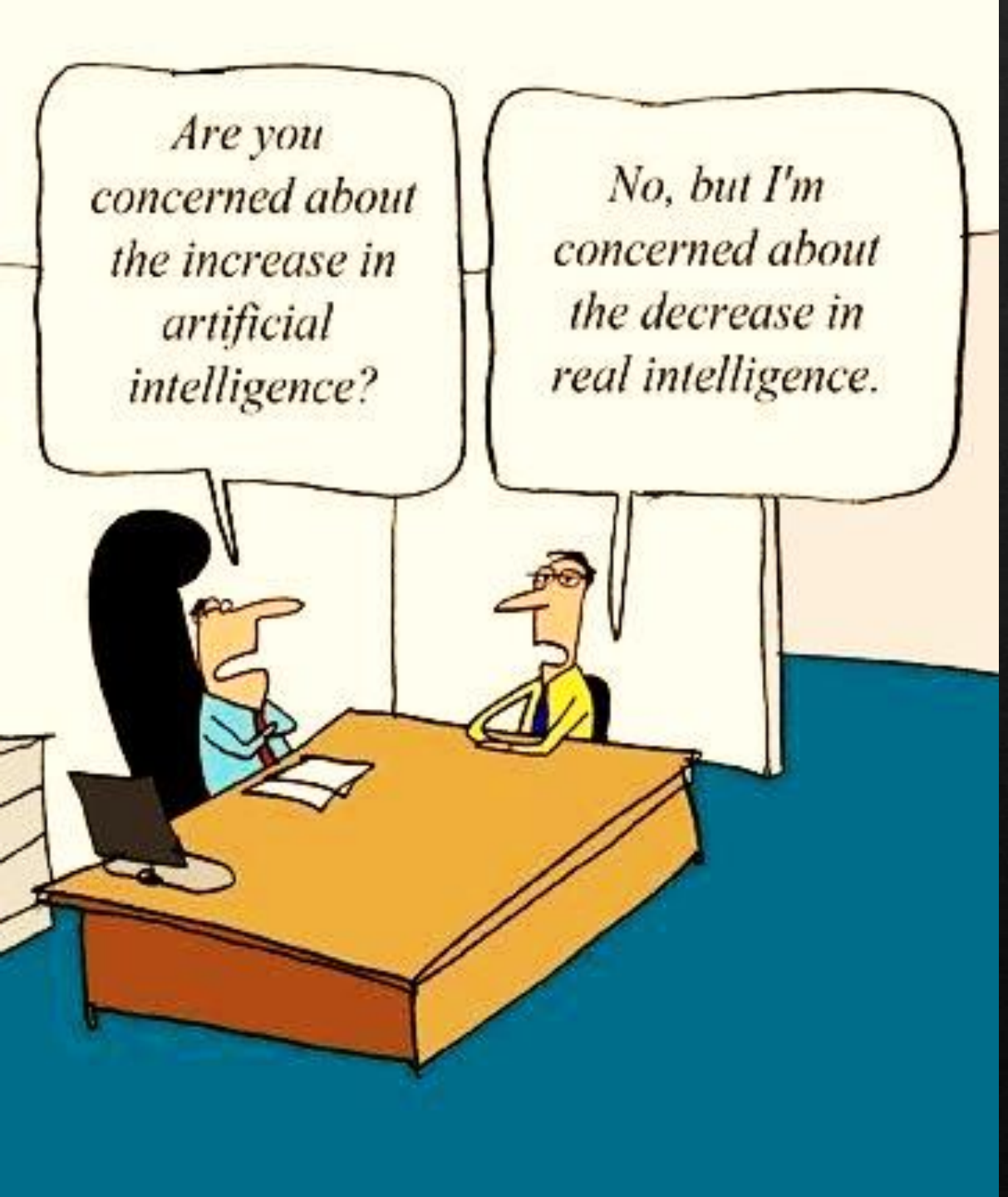


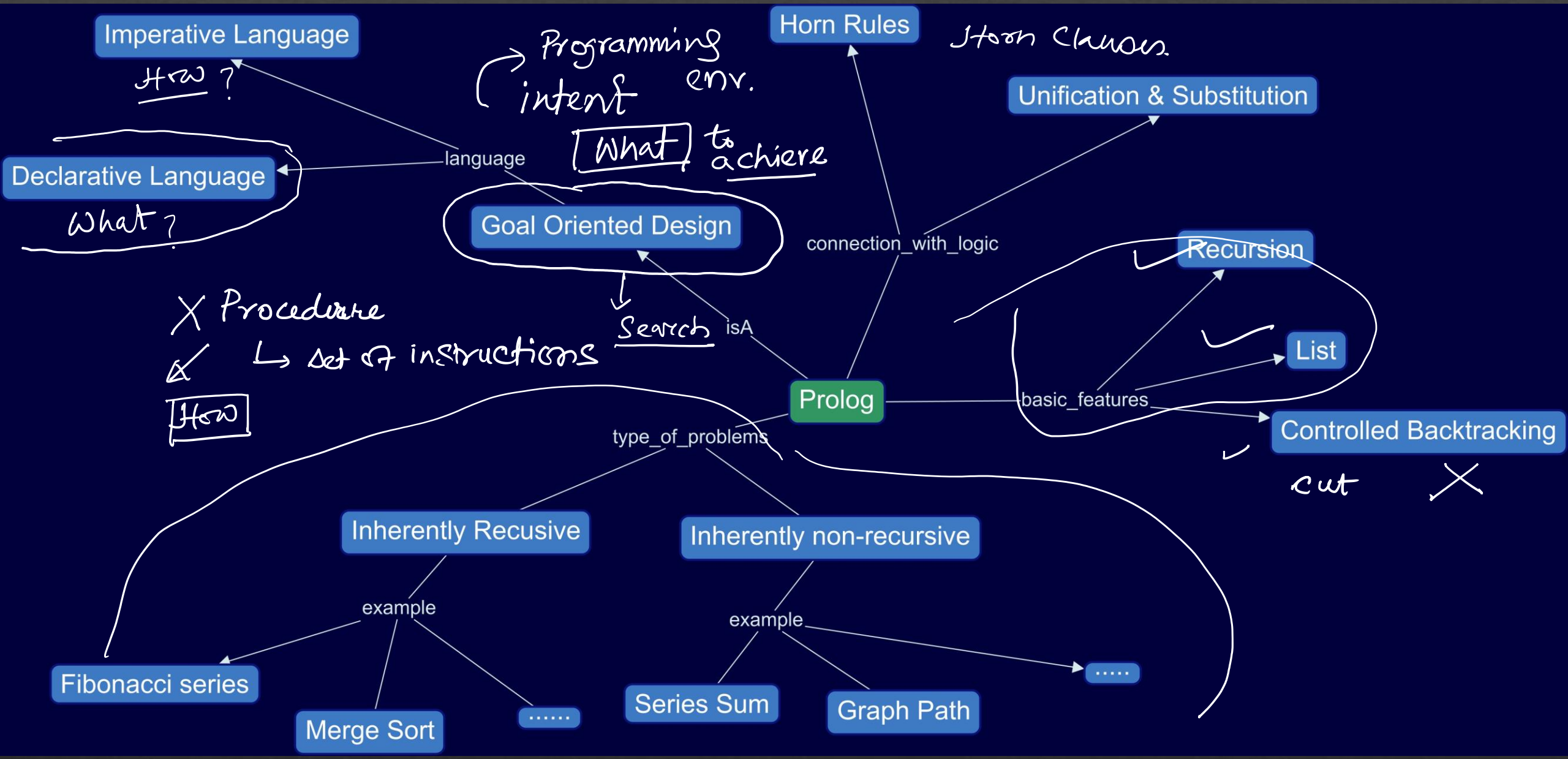
Prolog: Language for AI

Programming

Logic

AIFA, AI61005, 2021 Autumn

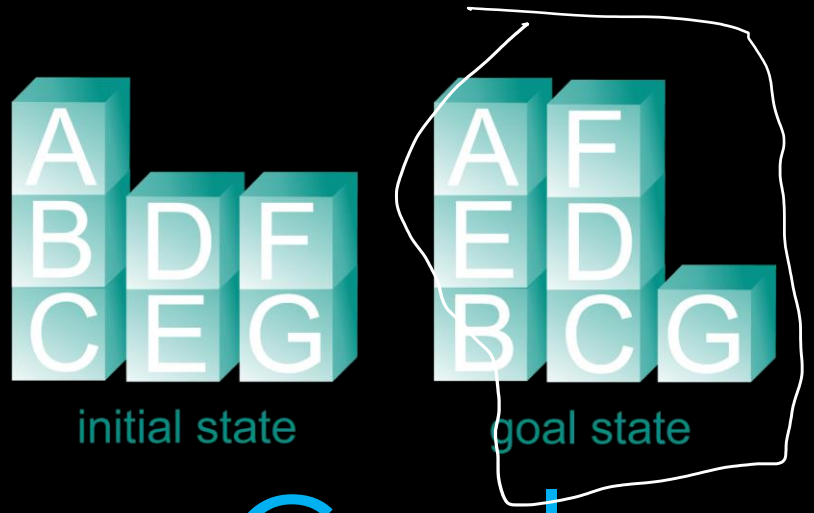
Plaban Kumar Bhowmick


Imperative Language
How ?
Declarative Language
What ?
language
Programming
intent
env.
What to achiere
Goal Oriented Design
Horn Rules
Horn Clauses
Unification & Substitution
connection_with_logic
Recursion
List
Controlled Backtracking
cut
basic_features
X Procedure
L set of instructions
How
Search
isA
Prolog
type_of_problems
Inherently Recusive
Inherently non-recursive
example
example
Fibonacci series
Merge Sort
.....
Series Sum
Graph Path
.....

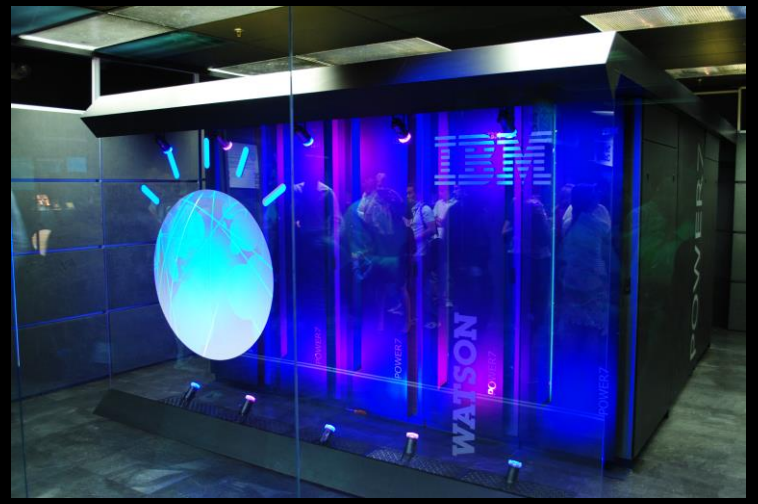
Prolog Programming for Artificial Intelligence: Ivan Bratko

https://swish.swi-prolog.org/

Goal Oriented Programming



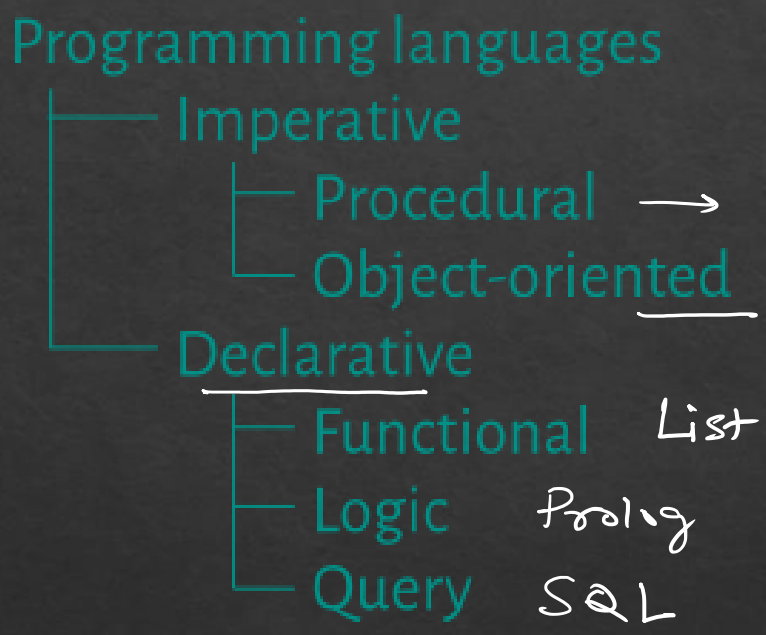
Goal



Language Choice

- "Known" languages like FORTRAN, C/C++, Java, python
  - Imperative: How-type language
- Goal Oriented Languages (Declarative)
  - Declarative: What-type language
  - LISP → functional
    - *"Goal oriented programming is like reading Shakespeare in language other than English"* – Patrick Winston
  - ProLog: Truly what-type language

Imperative vs. Declarative



Imperative: Comprises a sequence of commands

Declarative: Declare what result we want and leave the language to come up with the procedure to produce them

Prolog and First-Order Logic

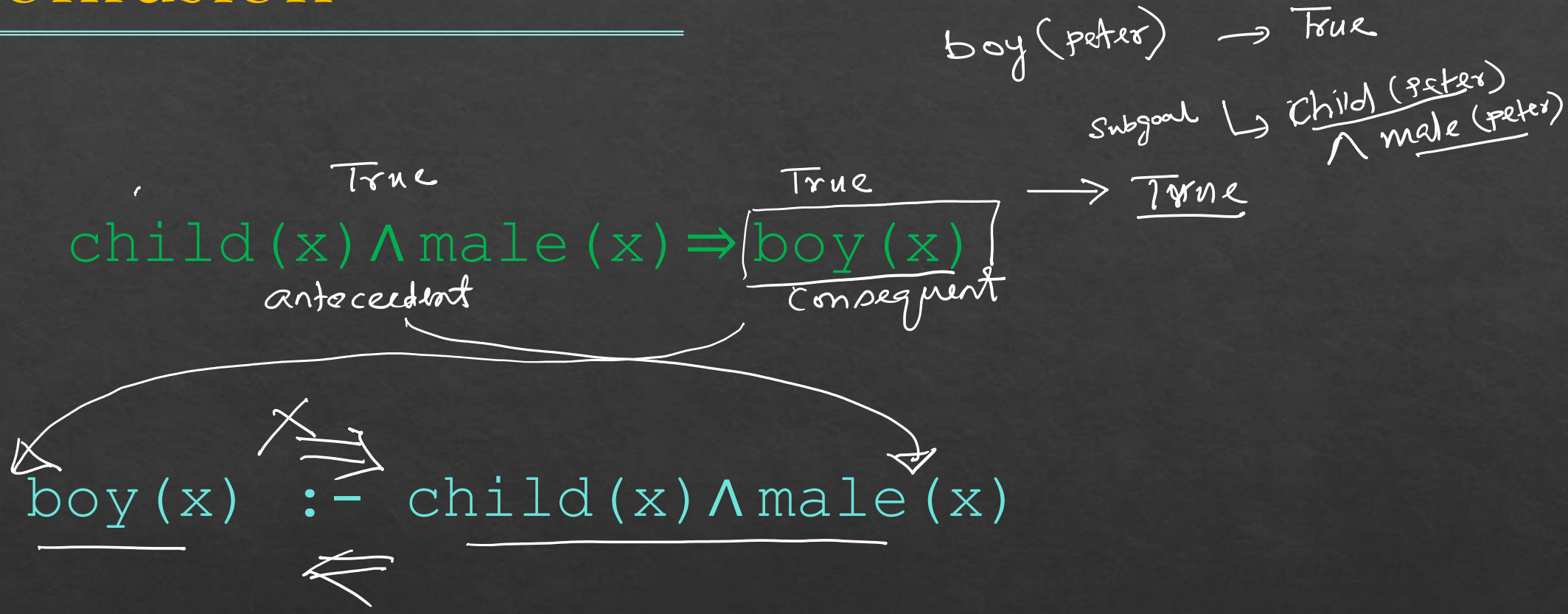
- Prolog language syntax
  - Horn Clause: CNF with implicit quantifiers and with at most one positive literal
    - $child(x) \wedge male(x) \Rightarrow boy(x)$
- Prolog proof procedure
  - Resolution Principle
- Prolog goal matching
  - Unification and substitution

$\neg child(x) \vee \neg male(x) \vee boy(x)$

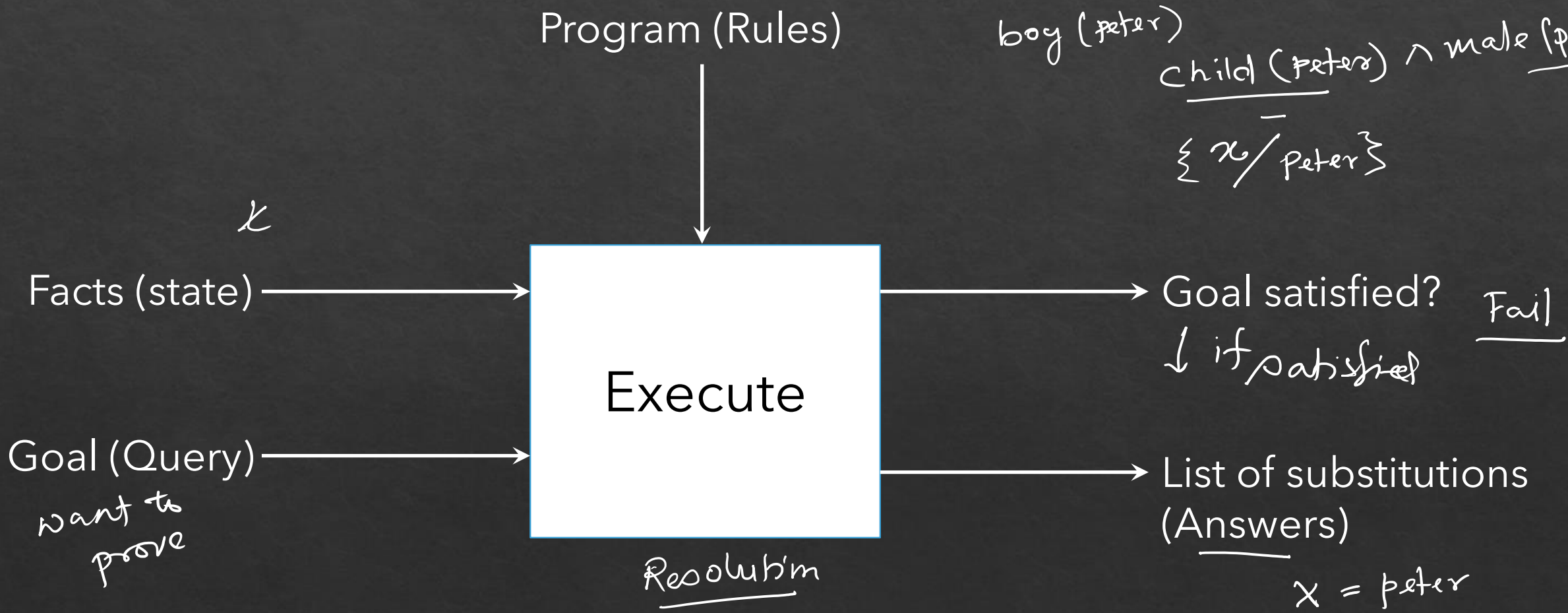
Solving problem in Prolog
$\Rightarrow$ Prove the goal

$boy(peter)$ $\{x/peter\}$

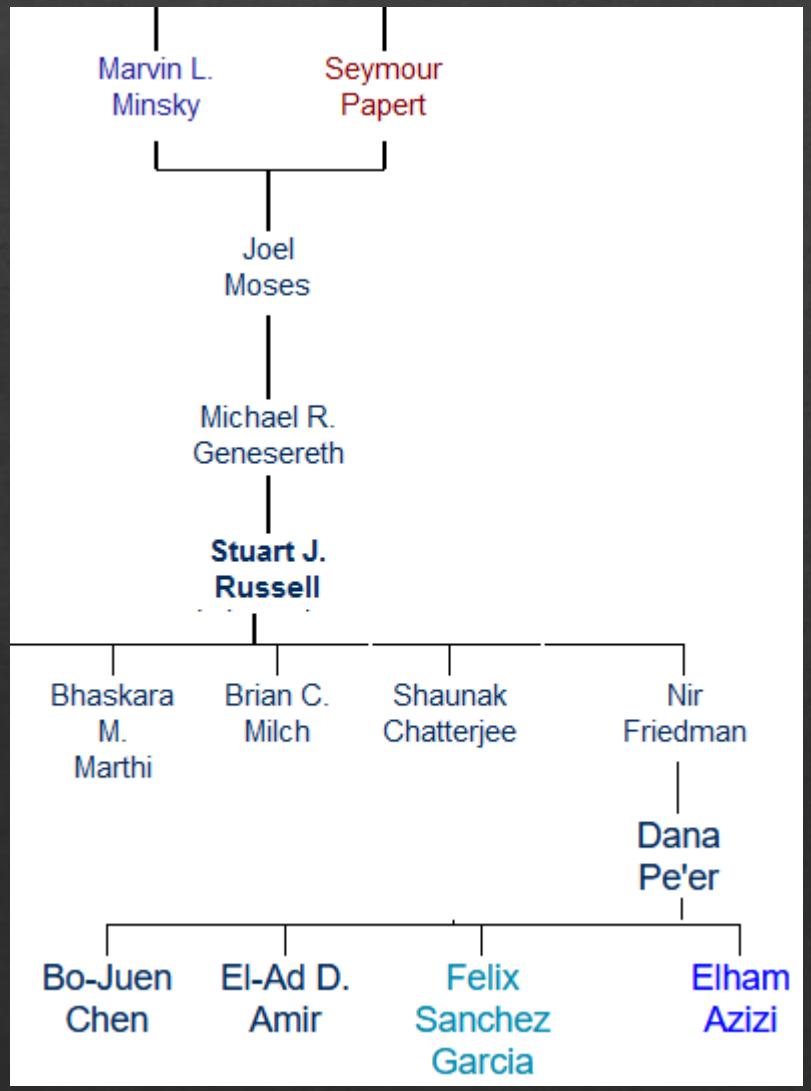
A Confusion


boy(peter) → True
subgoal ↳ child(peter) ∧ male(peter)
→ True
True
True
child(x) ∧ male(x) ⇒ boy(x)
antecedent
consequent
boy(x) :- child(x) ∧ male(x)

Prolog Computation Model



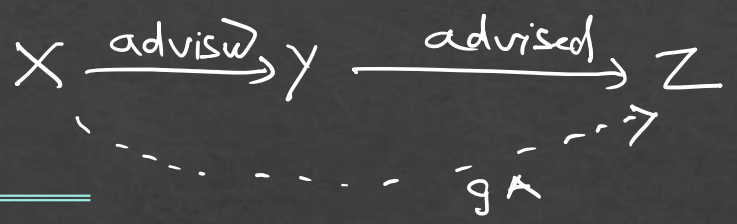
Prolog Facts




```
advisor(minsky, moses).
advisor(papert, moses).
advisor(moses, genesereth).
advisor(genesereth, russell).
advisor(russell, bhaskara).
advisor(russell, milch).
advisor(russell, shaunak).
advisor(russell, friedman).
advisor(friedman, dana).
advisor(dana, felix).
advisor(dana, chen).
advisor(dana, amir).
advisor(dana, azizi).

male(felix).
female(dana).
```

$advisor(X,Y) \wedge advisor(Y,Z) \Rightarrow grand\_advisor(X,Z)$



Prolog Rules

```
grand_advisor(X,Z) :- advisor(X,Y), advisor(Y,Z).
```

head (consequent) — body (antecedent)

$\forall_{X,Z} \exists_Y advisor(X,Y) \wedge advisor(Y,Z) \Rightarrow grand\_advisor(X,Z)$

IF there is a Y such that X is advisor of Y AND Y is advisor of Z
THEN X is a grand advisor of Z

Prolog rules are Horn Clauses:

$(P_{11} \vee P_{12} \vee \ldots P_{1m}) \wedge \cdots \wedge (P_{n1} \vee P_{n2} \vee \ldots P_{nr}) \Rightarrow Q$

Generic

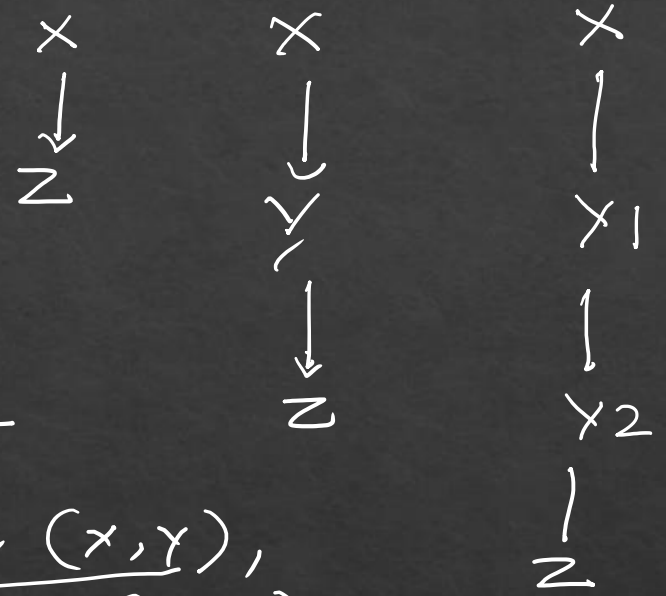
$Q :- P_{11}; P_{12}; \ldots; P_{1m}, \ldots \ldots, P_{n1}; P_{n2}; \ldots; P_{nr}$

Prolog Rules: Recursion

```
ancestor(X,Z) :-
        advisor(X, Z).

ancestor(X,Z) :-
        advisor(X, Y),
        advisor(Y, Z).

ancestor(X,Z) :-
        advisor(X, Y1),
        advisor(Y1, Y2),
        advisor(Y2, Z),
```



ancestor (x,z) :-
ancestor (x,y),
ancestor (y,z).

↓

ancestor (x,z) : —
advisor (x,y),
ancestor (y,z).

Prolog Rules: Recursion

```
ancestor(X,Z) :-
        advisor(X, Z).

ancestor(X,Z) :-
        advisor(X, Y),
        ancestor(Y, Z).
```

X is an ancestor of Z if
    X is an advisor of Y AND
    Y is an ancestor of Z

How Prolog Answers?

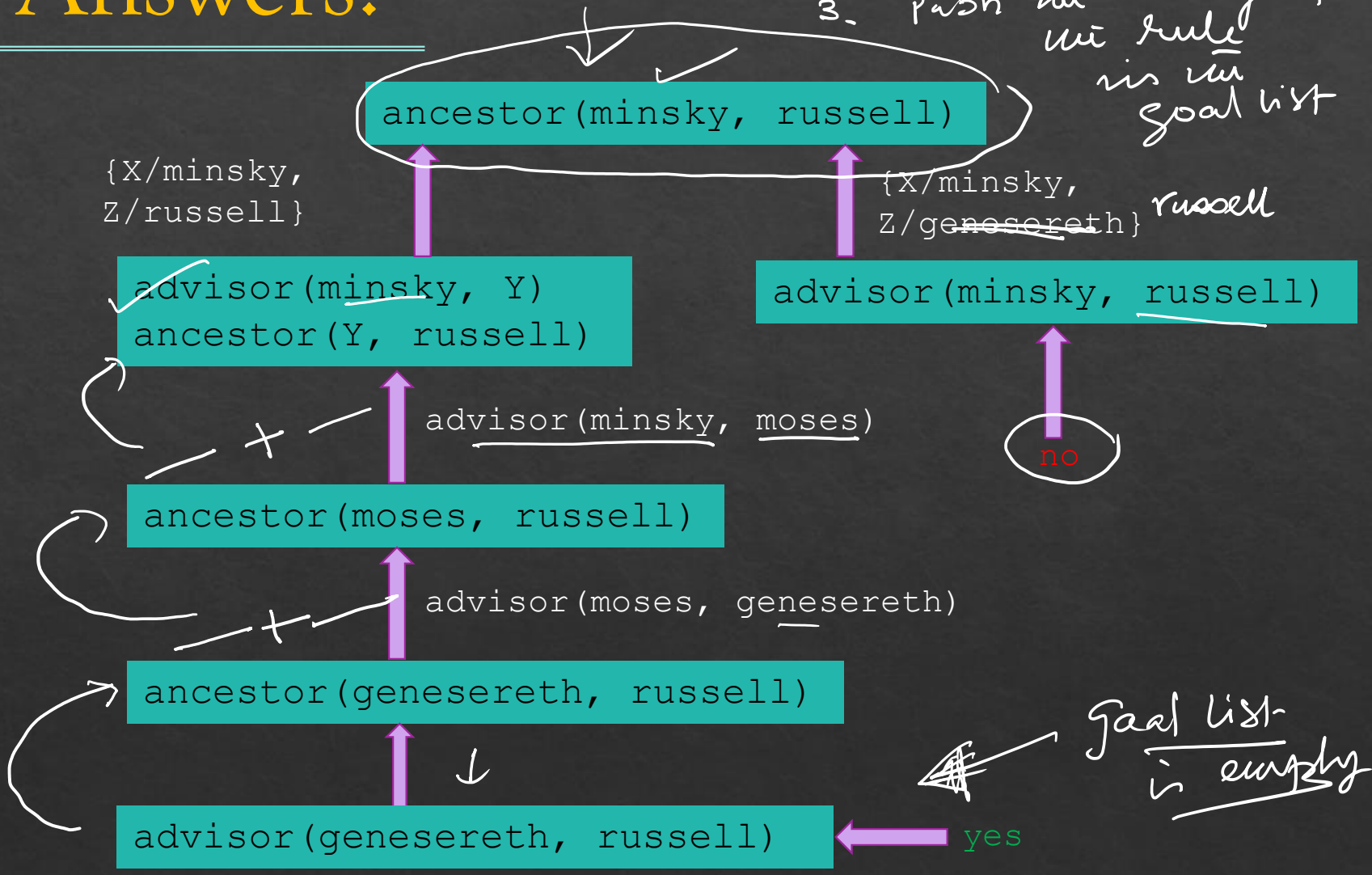
1. Start with a goal list
2. match the goal with rule head
3. Push the body of the rule in the goal list

```
ancestor(X,Z) :-
        advisor(X, Z).

ancestor(X,Z) :-
        advisor(X, Y),
        ancestor(Y, Z).

?- ancestor(minsky, russell)
```

R1



How Prolog Answers?

```
function EXECUTE (program, GoalList) returns [success, instance]/failure
    if empty(GoalList) then return [true, instance]
    else
        goal = head(GoalList)
        other_goals = tail(GoalList)
        satisfied = false
        while not satisfied and more clauses in program then
            H :- B1, ..., Bn //Next clause C of the program
            H' :- B1', ..., Bn' //Variant C' of clause C
            [match_OK, instance] = match(goal, H')
            if match_OK then
                new_goals = append([ B1', ..., Bn' ], other_goals)
                new_goals = substitute(instance, other_goals)
                [satisfied, instance] = EXECUTE(program, new_goals)
        return [satisfied, instance]
```

Repeated use of resolution

?anc(m,r), anc(d,c)

substitutions

Single goal

goal → anc(m,r)

other_goals → anc(d,c)

new_goals ← [B1', ... Bn'] + other goals

adv(x,z)

new goals ← adv(m,r)

{x/m, z/r}

anc(x,z) :- adv(x,z)

Q: whether reordering of clauses or goals have any effect in execution?

Reordering of Clauses

V1

```
ancestor1(X,Z) :-
        advisor(X, Z).

ancestor1(X,Z) :-
        advisor(X, Y),
        ancestor1(Y, Z).
```

Original

V2

```
ancestor2(X,Z) :-
        advisor(X, Y),
        ancestor2(Y, Z).

ancestor2(X,Z) :-
        advisor(X, Z).
```

Clause swap

V3

```
ancestor3(X,Z) :-
        advisor(X, Z).

ancestor3(X,Z) :-
        ancestor3(Y, Z),
        advisor(X, Y).
```

Goal swap

V4

```
ancestor4(X,Z) :-
        ancestor4(Y, Z),
        advisor(X, Y).

ancestor4(X,Z) :-
        advisor(X, Z).
```

Clause & Goal swap

Reordering of Clauses

```
ancestor1(X,Z) :-
        advisor(X, Z).

ancestor1(X,Z) :-
        advisor(X, Y),
        ancestor1(Y, Z).
```

Original

?- ancestor (dana, azizi)

```
Call: ancestor1(dana, azizi)
Call: advisor(dana, azizi)
Exit: advisor(dana, azizi)
Exit: ancestor1(dana, azizi)
```

True

True

goal satisfied

Reordering of Clauses

```
ancestor2(X,Z) :-
        advisor(X, Y),
        ancestor2(Y, Z).

ancestor2(X,Z) :-
        advisor(X, Z).
```

Clause swap

Success

```
Call: ancestor2(dana, azizi)
Call: advisor(dana, _4612)
Exit: advisor(dana, felix)
Call: ancestor2(felix, azizi)
Call: advisor(felix, _4614)
Fail: advisor(felix, _4614)
Redo: ancestor2(felix, azizi)
Call: advisor(felix, azizi)
Fail: advisor(felix, azizi)
Fail: ancestor2(felix, azizi)
Redo: advisor(dana, _4612)
Exit: advisor(dana, chen)
Call: ancestor2(chen, azizi)
Call: advisor(chen, _4614)
Fail: advisor(chen, _4614)
Redo: ancestor2(chen, azizi)
Call: advisor(chen, azizi)
Fail: advisor(chen, azizi)
Fail: ancestor2(chen, azizi)
Redo: advisor(dana, _4612)
Exit: advisor(dana, amir)
Call: ancestor2(amir, azizi)
Call: advisor(amir, _4614)
Fail: advisor(amir, _4614)
Redo: ancestor2(amir, azizi)
Call: advisor(amir, azizi)
Fail: advisor(amir, azizi)
Fail: ancestor2(amir, azizi)
Redo: advisor(dana, _4612)
Exit: advisor(dana, azizi)
Call: ancestor2(azizi, azizi)
Call: advisor(azizi, _4614)
Fail: advisor(azizi, _4614)
```

Reordering of Clauses

```
ancestor3(X,Z) :-
        advisor(X, Z).

ancestor3(X,Z) :-
        ancestor3(Y, Z),
        advisor(X, Y).
```

Goal swap

```
?- ancestor3(bhaskara, felix)
```

Quick Solution

Infinite Loop

```
ancestor4(X,Z) :-
        ancestor4(Y, Z),
        advisor(X, Y).

ancestor4(X,Z) :-
        advisor(X, Z).
```

Clause & Goal swap

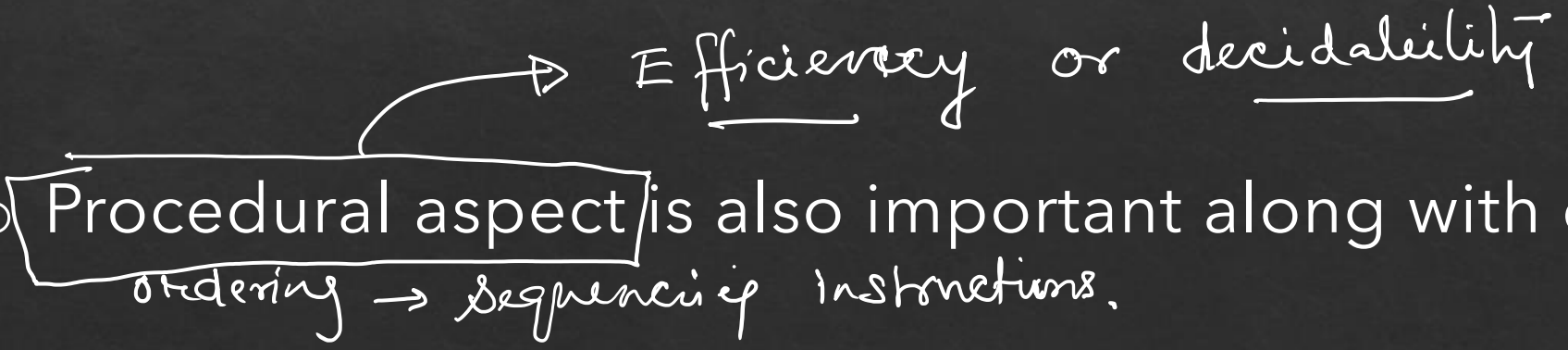
Infinite Loop

Takeaways from Ordering

- Try simplest idea first (practical heuristics in problem solving)
  - `ancestor1` being the simplest, `ancestor4` being the most complex

- Check your clause ordering to avoid infinite recursion

- Procedural aspect is also important along with declarative



Some Example Programs

Numbers between two numbers:

```
between_number(N1,N2) :-
    N1<N2-1, N is N1+1,
    print(N),nl, NN1 is N1+1,
    between_number(NN1,N2).
```

Sum of the numbers in a range:

```
series_sum(N,N,N).
series_sum(N1, N2, Sum) :- N1<N2, N is N1+1,
                           series_sum(N,N2,SumInter),
                           Sum is SumInter + N1.
```

Prolog List Data Structure

List Data Structure:

empty list

[1|[2,3,4,5]] = [1,2,3,4,5]

`[]` OR `[Head|Tail]` OR `[Item1, Item2,…|Others]`

[1,2,3|others] = [1,2,3,4,5]

others = [4,5]

Examples:

```
Color1 = [maroon, green].
Color2 = [red, yellow].
Clubs = [mohanB, Color1, eastB, Color2].
L = [X1,X2|[X3,X4,X5]].
```

Nested List

Prolog List Data Structure

- Concatenation of two lists

```
conc([], L, L).
conc([X|L1], L2, [X|L3]) :- conc(L1, L2, L3).
```

divide [X|L1]

conc([X], L1, L2, L3)

L1 + L2 = L1 | L2 → L3

conc(L1, L2, L3)

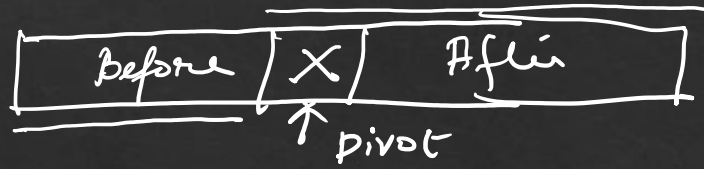
[X|L1]...

- Membership in list

base step: X [X|—] → satisfied

recursive: X mem(X, [—|Tail]) :- mem(X, Tail).

- Partition a list wrt a pivot



⇒ list

1, 2, 3, 4, 5, 6, 7, 8, 9

Before: 1, 2, 3, 4 — after: 6, 7, 8, 9

conc(Before, [X|After], List)

⇒ list

conc(—, [B, X, A|—], List)

Prolog List Data Structure

- Delete from list

base step: X [X | L1] ⇒ [L1]

recursive step: X [Y | L1] ⇒ [Y | L2] >> del X L1 L2

- A list is ordered or not

base step: [ ] [X]

recursive step: [H | Tail] [H | H1 | Tail1]

Tail = Con (H1, Tail1)

ordered ([H1 | Tail1])

H ≤ H1

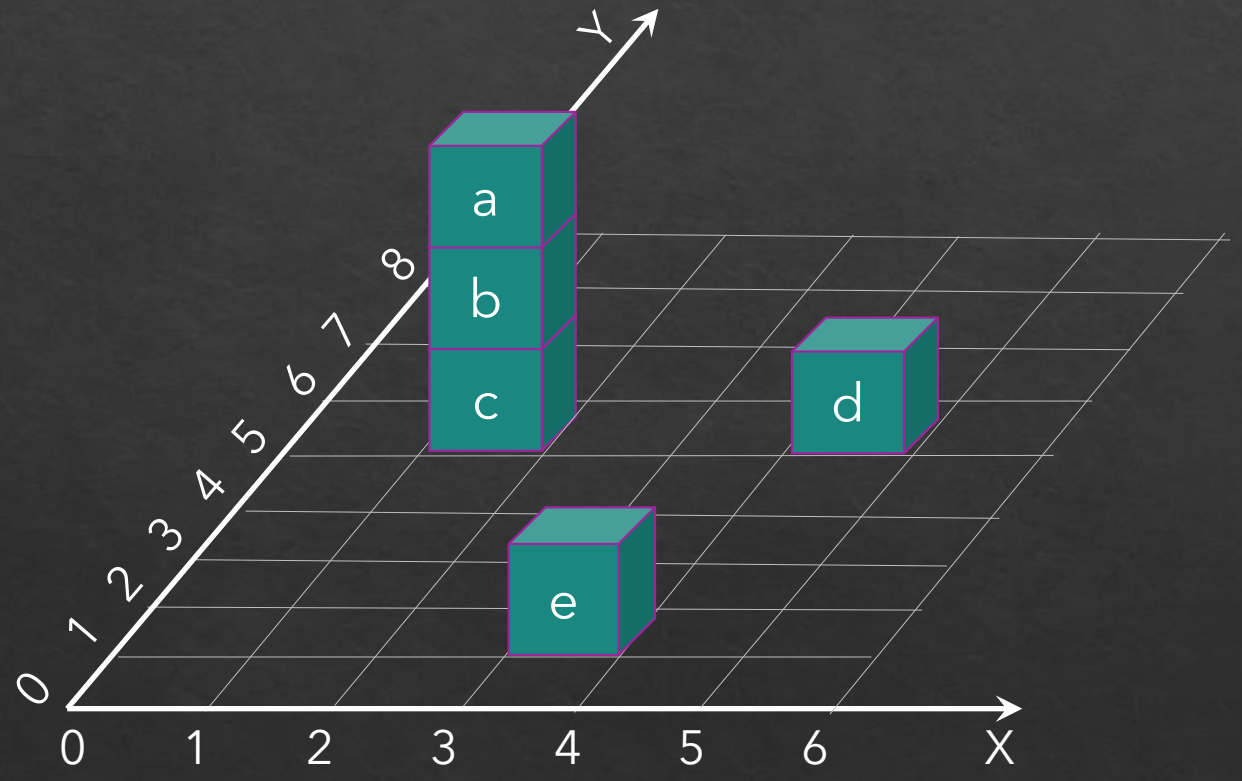
- Max in a list

base step: [X] ⇒ X max

recursive step: [H | Tail]

max (N1, Tail)
if H > N1, M is H
if H ≤ N1, M is N1

A Robot Playing with Blocks



- Robot sees from the top
- Robot can name the visible blocks and x-y coordinates.

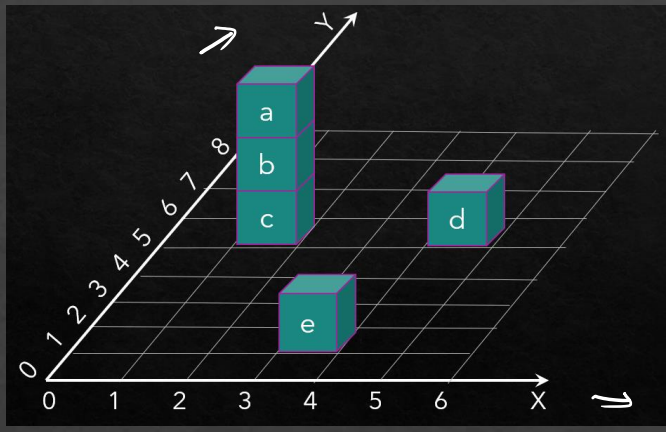
```
see(a, 1, 5).
see(d, 4, 5).
see(e, 3, 1).
```

- Positional relations

```
on(a, b).
on(b, c).
on(c, table).
on(d, table).
on(e, table).
```

on(A, _), \+see(A, _, _)

A Robot Playing with Blocks



- What are blocks in this world?

```
?- on(Block, _).
```

Robots

- Pairs of blocks having the same y-coordinate

```
?- see(B1,_,Y),
   see(B2,_,Y),
   B1 \= B2.
```

- Boxes that are not visible

```
?- on(B, _),
   \+see(B, _, _).
```

- Leftmost visible block

```
?- see(B,X,_),
   \+ (see(B2,X2,_), (X2<X)).
```

- Find the Z-coordinate of a block

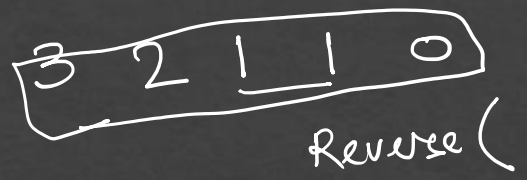
```
z(B,0) :- on(B,table).
z(B,Z):- on(B,B0), z(B0,Z0),
         Z is Z0+1.
```

- Find blocks b/w two blocks

```
recursive_on(B1,B2) :- on(B1,B2).
recursive_on(B1,B2) :-  on(B1,BX),
    print(BX), recursive_on(BX,B2).
```

Inherently Recursive Problems

Fibonacci Series



fib series

```
fib_seq(S, N) :-
    N > 1,
    fib_seq_(N, SR, 1, [1,0]),
    reverse(SR,S).

fib_seq_(N,Seq,N,Seq).
fib_seq_(N,Seq,N0,[B,A|Fs]) :-
    N > N0,
    N1 is N0+1,
    C is A+B,
fib_seq_(N,Seq,N1,[C,B,A|Fs]).
```

reversed

Current

inbuilt

[C,B|A+Fs]

L = [H|Tail]

=>[H1, H2|Tail]

Reverse

i I=1 [0,1] [1,0]

[1,0|[]]

I=2 1 [1,1,0|[]]

[1,1|[0]]

I=3 2 [2,1,1|[0]]

[2,1|[1,0]]

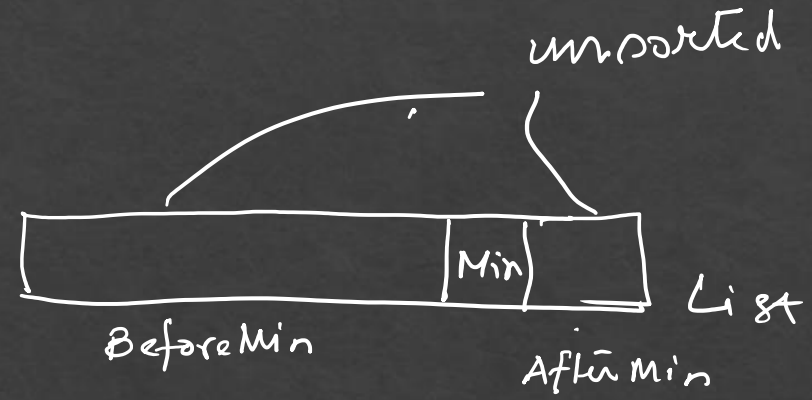
Simple Sort

```
min(A, A, B) :- A =< B.
min(B, A, B) :- B =< A.

smallest(A, [A|[]]).
smallest(Min, [A|B]) :- smallest(SB, B),
                        min(Min, A, SB).


sorted([], []).

sorted([Min|RestSorted], List) :-
  smallest(Min, List),
  append(BeforeMin, [Min|AfterMin], List),
  append(BeforeMin, AfterMin, RestUnsorted),
  sorted(RestSorted, RestUnsorted).
```



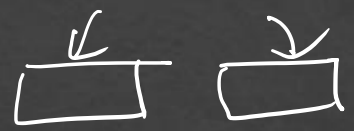
Con(BeforeMin, [Min|AfterMin], List)

Con(BeforeMin, AfterMin, unsorted)

Sort(RestSorted, unsorted)

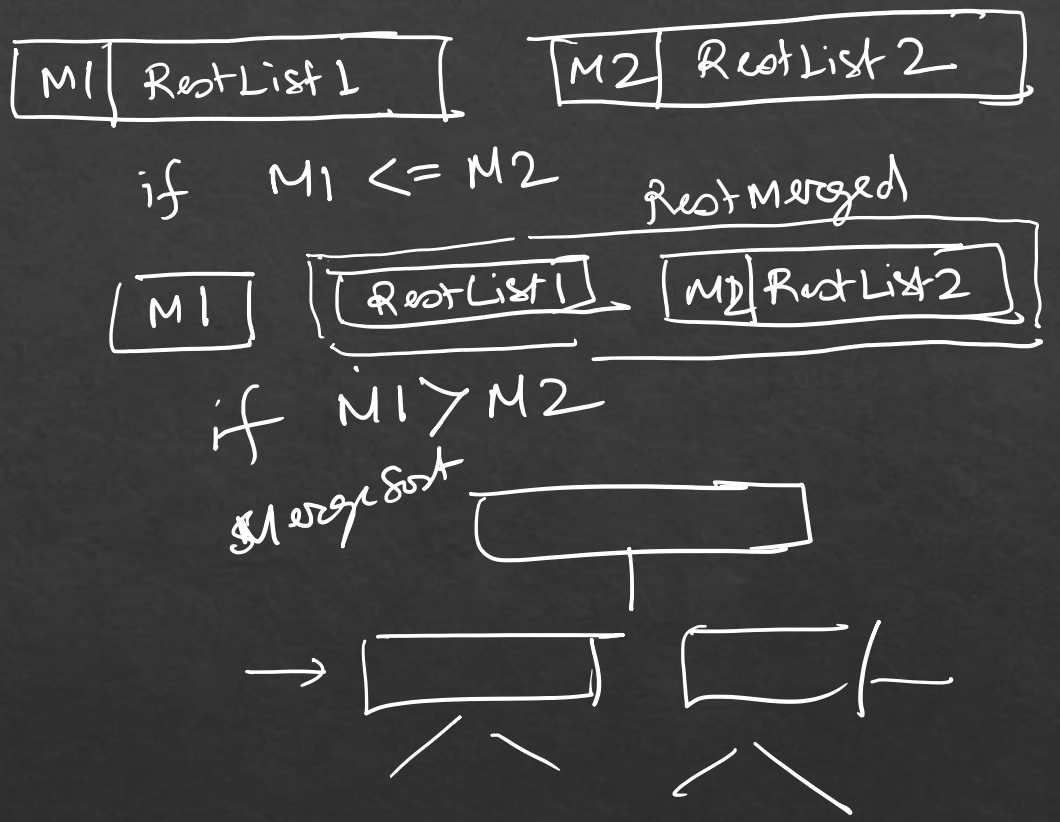
[Min | RestSorted]

Merge Sort



[ ] + List ⇒ List

List + [ ] ⇒ List



Merge Sort

```
merge(List, List, []).
merge(List, [], List).

merge([MinList1|RestMerged], [MinList1|RestList1],
[MinList2|RestList2]) :-
MinList1 =< MinList2,
merge(RestMerged,RestList1,[MinList2|RestList2]).

merge([MinList2|RestMerged], [MinList1|RestList1],
[MinList2|RestList2]) :-
MinList2 =< MinList1,
merge(RestMerged,[MinList1|RestList1],RestList2).
```

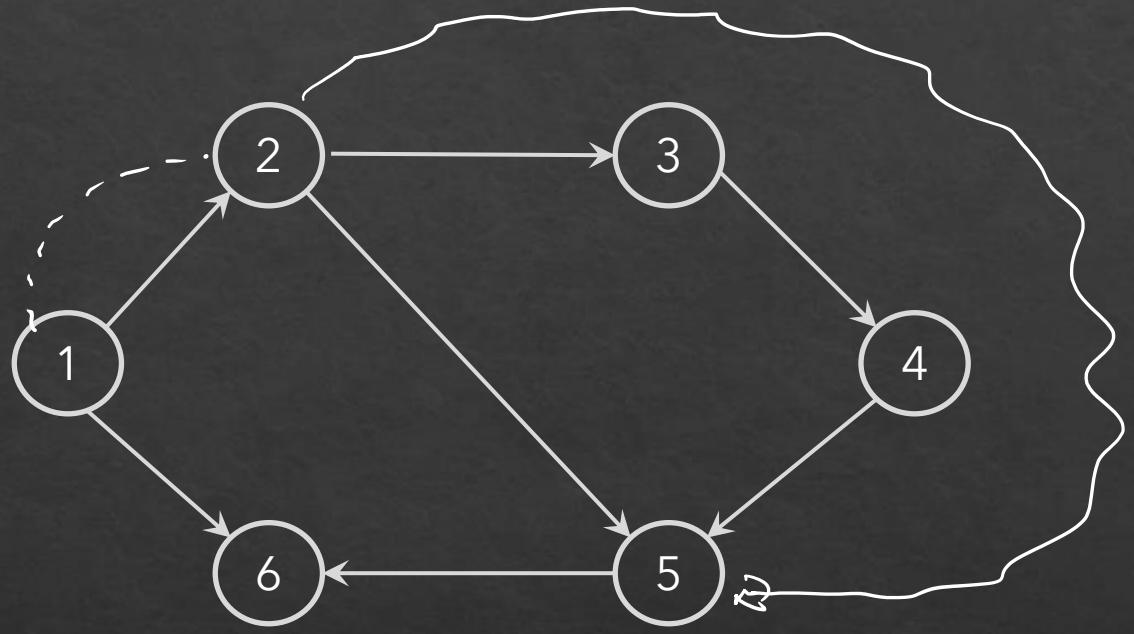
Merge Sort

```
mergeSort([], []).
mergeSort([A], [A]).

mergeSort(Sorted, List) :-
    length(List, N),  FirstLength is //(N, 2),
    SecondLength is N - FirstLength,
    length(FirstUnsorted, FirstLength),
    length(SecondUnsorted, SecondLength),
    append(FirstUnsorted, SecondUnsorted, List),
    mergeSort(FirstSorted, FirstUnsorted),
    mergeSort(SecondSorted, SecondUnsorted),
    merge(Sorted, FirstSorted, SecondSorted).
```

Inherently Iterative Problems

Path Finding in Graph




```
path1(N,N).
path1(S,D)  :- edge(S,N),
               path1(N,D).

path2(N,N,  [N]).
path2(S,D,  L):- edge(S,N),
                 path2(N,D,T),
                 conc([S],T,L).
```