?- cousin(mary, jake).

| X_1 = mary
| Y_1 = jake

parent($P1_1$, mary), parent($P2_1$, jake), sibling($P1_1$, $P2_1$)

| $P1_1$ = john

parent($P2_1$, jake), sibling(john, $P2_1$)

| $P2_1$ = susan

sibling(john, susan)

|

mother(M_2, john), mother(M_2, susan)

|

parent(M_2, john), female(M_2), mother(M_2, susan)

| M_2 = jim

female(jim), mother(jim, susan)

|

father(F3, john), father(F3, susan)

|

parent(F3, john), male(F3), father(F3, susan)

| $F_3 = jim$

male(jim), father(jim, susan)

|

father(jim, susan)

|

parent(jim, susan), male(jim)

|

male(jim)

|

yes

?- ancestor(jim, mary)

X_1 = jim
Y_1 = mary

parent(jim, mary)

fail, backtrack

X_2 = jim
Y_2 = mary

parent(jim, Z_2), ancestor(Z_2, mary)

Z_2 = john

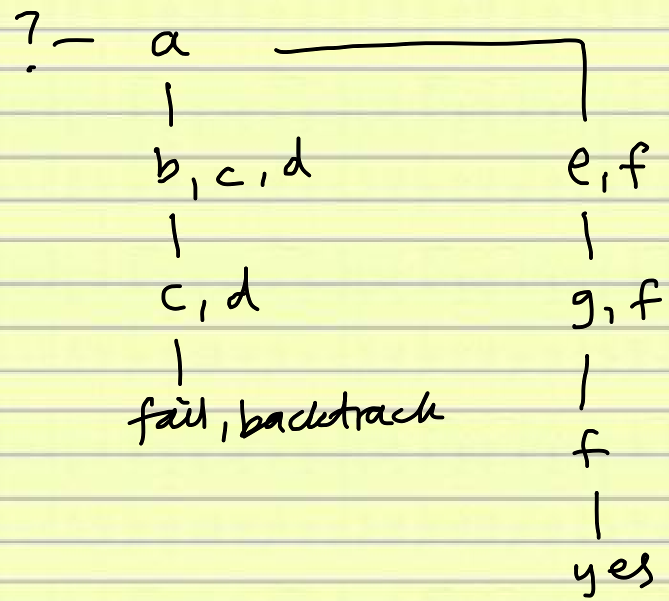
ancestor(john, mary)

X_3 = john
Y_3 = mary

parent(john, mary)

yes

a :- b, c, d.
a :- e, f.
e :- g.
b.
g.
f.
d.



Call / Exit / Redo / Fail model

?- a

CALL a
CALL b
EXIT b
CALL c
FAIL c
REDO a

CALL e
CALL g
EXIT g
EXIT e
CALL f
EXIT f
EXIT a

a :- b, c, d.

a :- e, f.

e :- g.

g.

b.

f.

d.

Goal 'stack' - initially just a

a
↑
a b c d
↑ (under b)
a b c d
↑ (under c)
a
↑
a e f
↑ (under e)
a e g f
↑ (under g)
a e g f
↑ (under f)
a e g
a
yes

Unification needs to very general
 - matching any two terms, with variables (possibly)

terms: a(X)

a

a(b(X))

a(b(X), c(Y))

a([X|Y]) => a(.(X,Y))

These are stored as nested structures

a(b(c(X))) is represented as

Term a
 Term b
  Term c
   Var X

e.g. a(b(X)) matches a(b(c)) with X bound to c

We need to record bindings so they can be undone on backtrack

We have looked at two semantics for Prolog

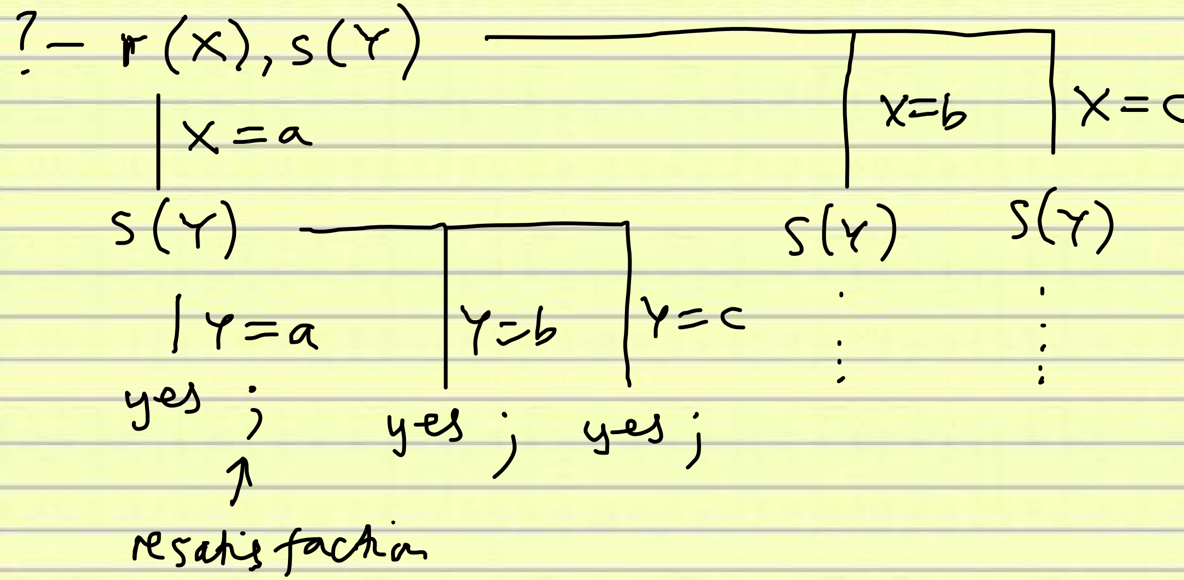
1. procedural (match, backtrack etc.)
2. declarative (logical statements)

There are 2 extra-logical features of Prolog

1. cut
2. negation

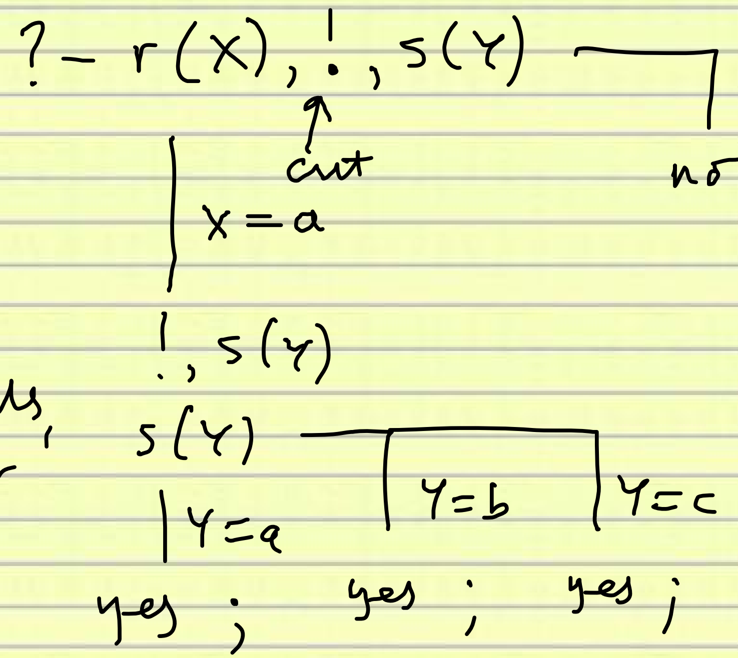
Cut - procedural mechanism for cutting off backtracking

r(a).
r(b).
r(c).
s(a).
s(b).
s(c).



going forwards,
cut is transparent
(does nothing)

going backwards,
cut immediately fails,
cutting off any other
possible matches



Example using cut : member

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

- assume that a goal using member succeeds
- assume a later goal fails. Having succeeded, we matched the fact. However anything that matches the fact also matches the recursive rule. Backtracking to member causes (useless) search through the rest of the list.

```
?- member(a, [b,a,c]), fail.
      | X1 = a
      | T1 = [a,c]
   member(a, [a,c]), fail ──────────┐ X3 = a
      | X2 = a                     │ T3 = [c]
   fail, backtrack            member(a, [c])
                                   |
```

| $X_4 = a$
| $T_4 = []$

member(a, []), fail

|

no

Rewrite the predicate :

```
member(X, [X|_]) :- !.
member(X, [_|T]) :- member(X, T).
```

?- member(a, [b, a, c]), fail

| $X_1 = a$
| $T_1 = [a, c]$

member(a, [a, c]), fail — the branch is cut off — no

| $X_2 = a$

!, fail

|

fail, backtrack