Logic Programming: Prolog

Yifeng Zeng

yfzeng@cs.aau.dk

Review

- Use FOL to formulate sentences from knowledge base
- Conversion to Normal Form
  - Standardize variables
    - Avoid confusion
  - Implication elimination
    - Remove $\Rightarrow$
  - Negation inwards
    - $\neg\forall x\ \ p$ becomes $\exists x\ \ \neg p$
    - $\neg\exists x\ \ p$ becomes $\forall x\ \ \neg p$
  - Skolemization
    - Remove $\exists$ quantifier
  - Distribute $\wedge$ over $\vee$
    - Flatten out nested conjunction and disjunction
  - Universal implicit
    - Remove $\forall$ quantifier
- Add the negation of the goal to the set of clauses
- Use resolution to deduce a refutation

Outline

- Prolog Overview
- Program Execution
- Example: Monkey World
- List
- Exercising Control
- Mini-project

Overview

- Declarative Languages
  - Specify representations, not control flow

	Logic Programming	**Ordinary Programming**
1.	Identify problem	Identify problem
2.	Assemble information	Assemble information
3.	Coffee break	Figure out solution
4.	Encode information in KB	Program solution
5.	Encode problem instances as facts	Encode problem instance as data
6.	Ask queries	Apply program to data
7.	Find false facts	Debug procedure errors

Program Syntax: Examples

- $parent(tom, bob).$
  - Predicate: Starting with lower-case letters
  - Constant: Starting with lower-case letters
  - Termination by a full stop .
- $offspring(Y, X) :- parent(X, Y).$
  - $:-$ indicates rules
  - A conclusion part(**head**: the left-hand side of the rule): $offspring(Y, X)$
  - A condition part(**body**: the right-hand side of the rule):$parent(X, Y)$
  - Implicit universial quantifers
- $mother(X, Y) :- parent(X, Y), female(X).$
  - Use a comma , to encode $\wedge$ relation
  - Use a semicolon ; to encode $\vee$ relation

How Prolog answers questions: Examples

```
/*PROGRAM*/

likes(thomas,beer).
likes(kate,wine).
likes(thomas,X):-
          likes(kate,X).
```

- $? - likes(thomas, wine).$
- $? - likes(thomas, X).$
- $? - likes(Y, beer).$
- $? - likes(X, Y).$

Prolog and Logics

- Syntax as FOL formulas
  - Clause form: conjunctive normal form
  - Quantifiers are not explicitly written
- Operation: **Matching**
  - Given two terms if they are identical or
  - the variables in both terms can be instantiated to objects in such a way that after the substitution of variables by these objects the terms become identical
- Backward chaining
  - Transform a goal into another goal
  - **Backtrack to the original goal if the goal fails**
- Prolog Implementation
  - **SWI-Prolog**, YAP-Prolog, $\cdots$

Program Execution

```
/*PROGRAM*/
big(bear).              % C1
big(elephant).          % C2
small(cat).             % C3

brown(bear).            % C4
black(cat).             % C5

gray(elephant).         % C6
dark(Z):-
        black(Z).       % C7:Anything black is dark
dark(Z):-
        brown(Z).       % C8: Anything brown is dark
```

- $? - dark(X), big(X).$
  - $X = bear$

Ordering Matter

- Define the **Predecessor** relation
  - **pred1**

```
pred1(X,Z):-  parent(X,Z).
pred1(X,Z):-  parent(X,Y),pred1(Y,Z).
```

  - **pred2**

```
pred2(X,Z):-  parent(X,Y),pred2(Y,Z).
pred2(X,Z):-  parent(X,Z).
```

  - **pred3**

```
pred3(X,Z):-  parent(X,Z).
pred3(X,Z):-  pred3(X,Y),parent(Y,Z).
```

  - **pred4**

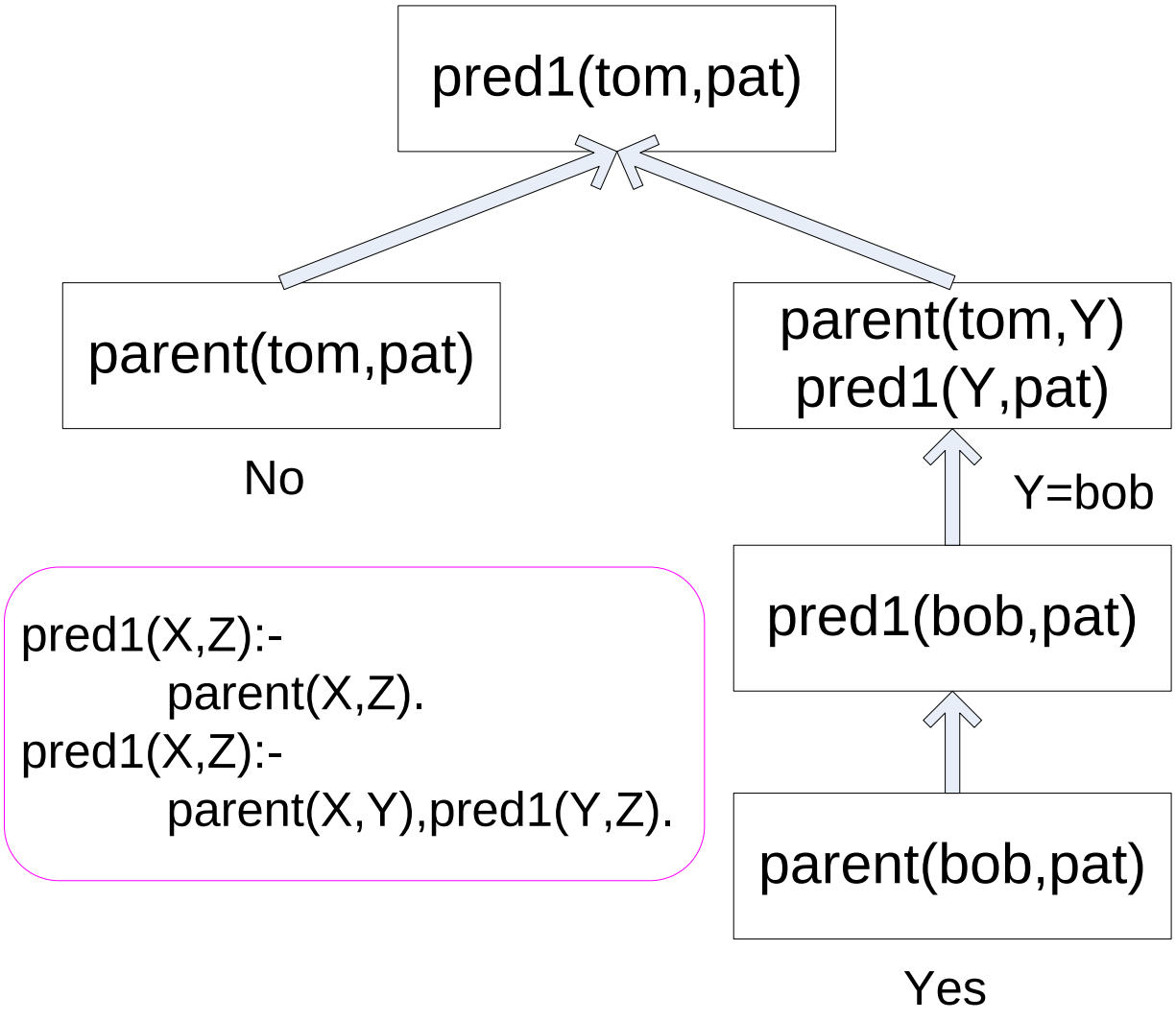
```
pred4(X,Z):-  pred4(X,Y),parent(Y,Z).
pred4(X,Z):-  parent(X,Z).
```

Predecessor 1

```
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
pred1(X,Z):-        parent(X,Z).
pred1(X,Z):-        parent(X,Y),pred1(Y,Z).
```

- ? – *pred*1(*tom*, *pat*).

Ordering Matter - 1

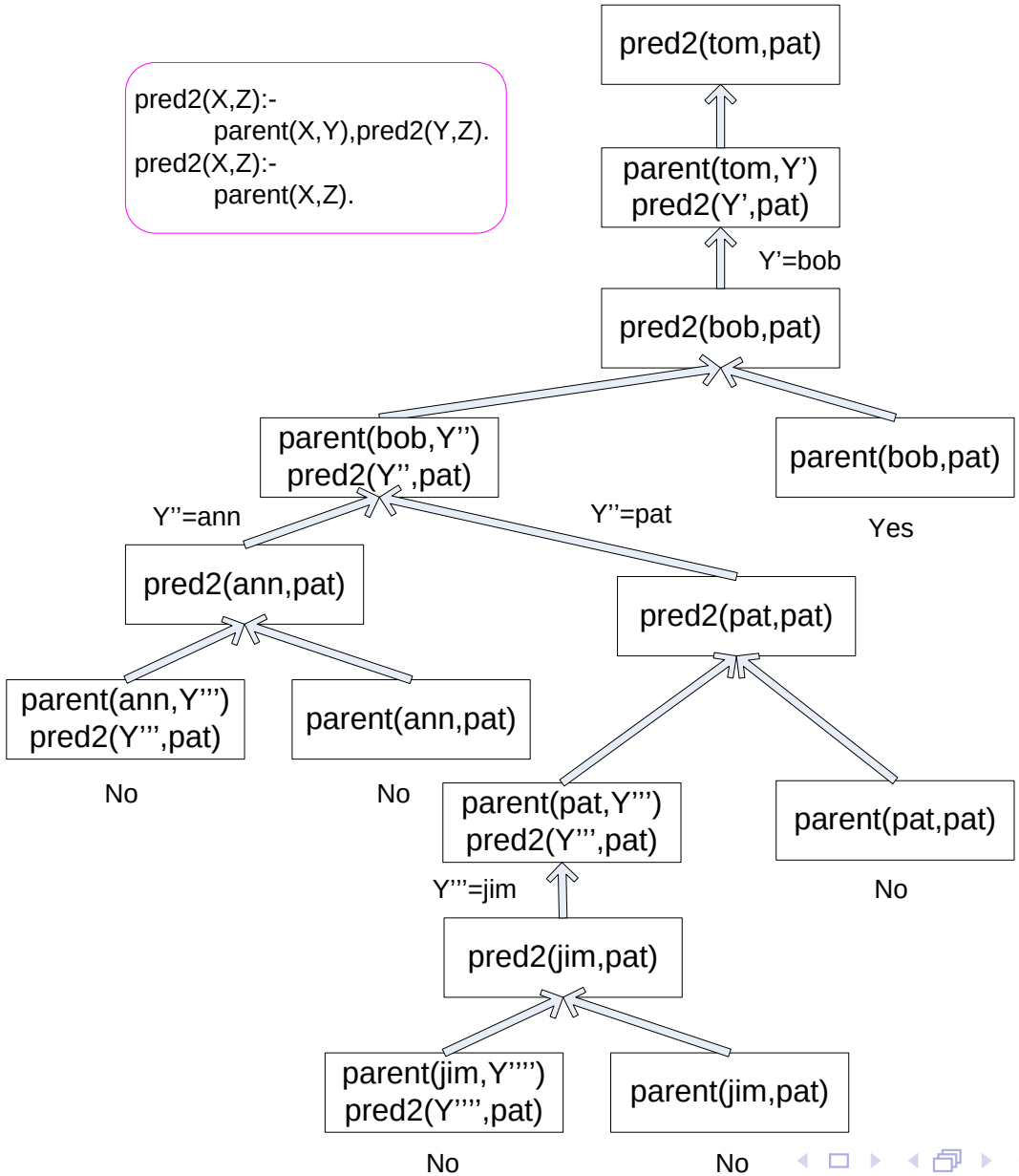


Predecessor 2

```
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
pred2(X,Z):-        parent(X,Y),pred2(Y,Z).
pred2(X,Z):-        parent(X,Z).
```

- ? – *pred2*(*tom*, *pat*).

Ordering Matter - 2

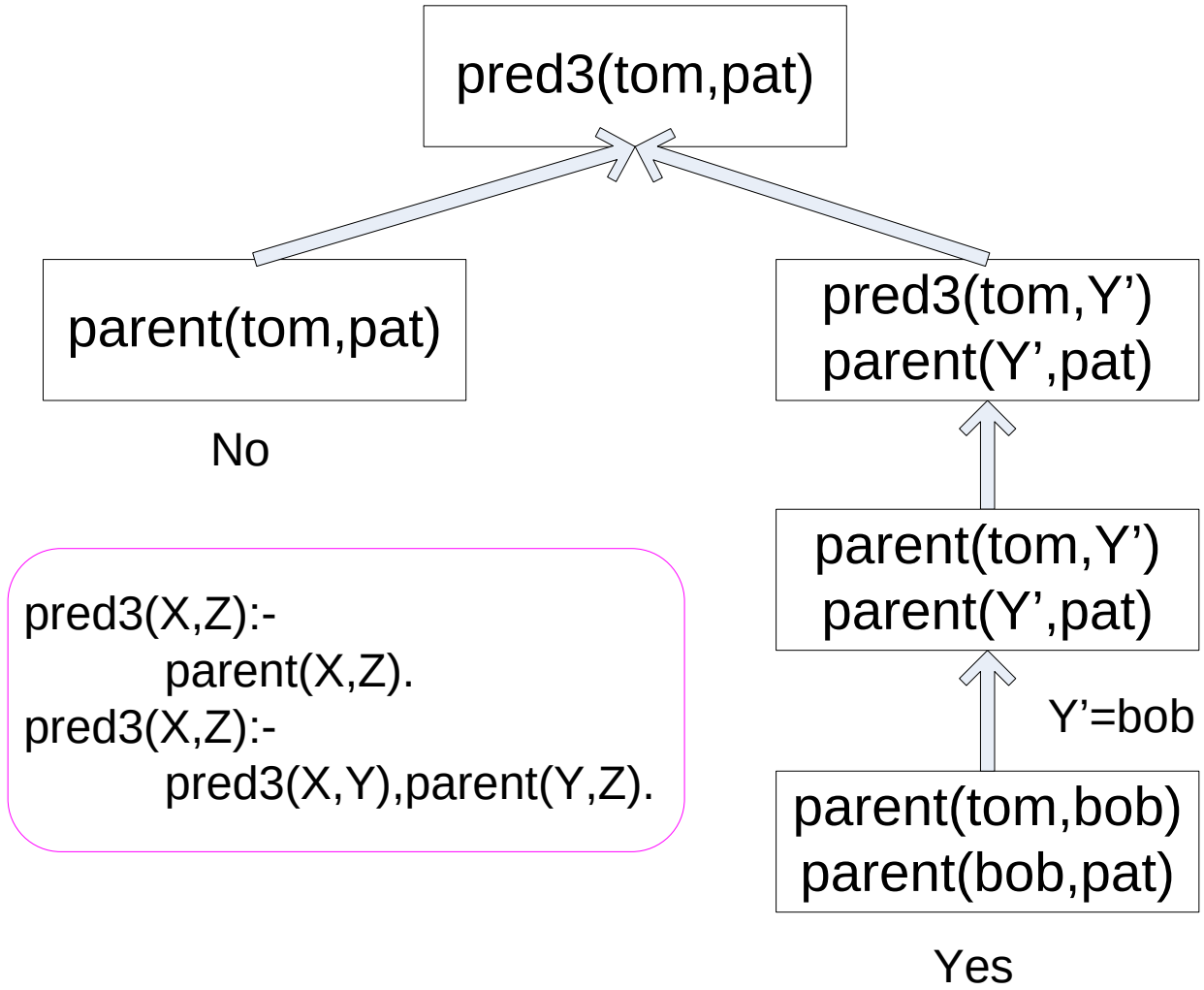


Predecessor 3

```
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
pred3(X,Z):-        parent(X,Z).
pred3(X,Z):-        pred3(X,Y),parent(Y,Z).
```

- $?-$ *pred*3(*tom*, *pat*).

Ordering Matter - 3



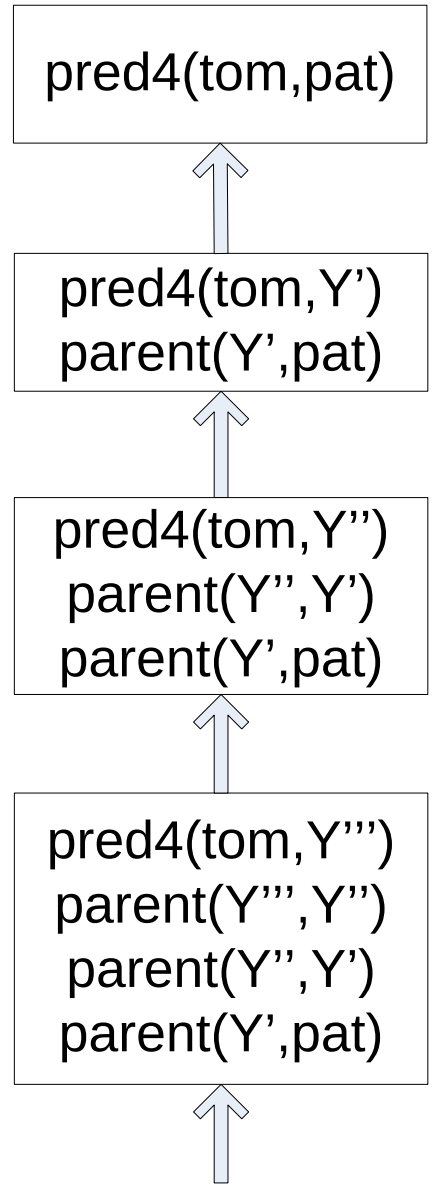
Predecessor 4

```
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
pred4(X,Z):-        pred4(X,Y),parent(Y,Z).
pred4(X,Z):-        parent(X,Z).
```

- ? – *pred*4(*tom*, *pat*).

Ordering Matter - 4

```
pred4(X,Z):-
        pred4(X,Y),parent(Y,Z).
pred4(X,Z):-
        parent(X,Z).
```



Monkey World

- There is a monkey at the door into a room
- In the middle of the room a banana is hanging from the ceiling. The monkey wants it, but cannot jump high enough from the floor
- At the window of the room there is a box that the monkey may use
- The monkey can perform the following actions:
  - Walk on the floor
  - Climb the box
  - Push the box around
  - Grasp the banana if it is standing on the box directly under the banana
- **QUERY**: Can the monkey get the banana?

Formulation - 1

- World state
  - Describe the positions of the objects: *horizontal position of monkey*,*vertical position of monkey*, *position of box*, and *monkey has or has not banana*
  - E.g.: Initial state: $state(atdoor, onfloor, atwindow, hasnot)$
- State changes after a specific action
  - E.g.: $move(State1, Move, State2)$
    - $State1$ is the state before the move
    - $Move$ is the move executed
    - $State2$ is the state after the move

Formulation - 2

- $move(state(middle, onbox, middle, hasnot),$
  $grasp,$
  $state(middle, onbox, middle, has))$
  - After the move the monkey has the banana and it has remained on the box in the middle of the room
- $move(state(Pos1, onfloor, Box, Has),$
  $walk(Pos1, Pos2),$
  $state(Pos2, onfloor, Box, Has))$
  - The move executed was 'walk from some position $Pos1$ to some position $Pos2$'
  - The monkey is on the floor before and after the move
  - The box is at some point Box which remains the same after the move
  - After the move the monkey still remains Has status

Formulation - 3

- Our query is $canget(State)$
- For any state if the monkey already has the banana, the predicate $canget(\cdot)$ must certainly be true
  - $canget(state(\_, \_, \_, has))$.
- In other states one or more moves are necessary
  - $canget(State1) :- move(State1, Move, State2),$
    $canget(State2)$.

Program

```
/*PROGRAM*/

move(state(middle,onbox,middle,hasnot),
     grasp,
     state(middle,onbox,middle,has)).

move(state(P,onfloor,P,H),
     climb,
     state(P,onbox,P,H)).

move(state(P1,onfloor,P1,H),
     push(P1,P2),
     state(P2,onfloor,P2,H)).

move(state(P1,onfloor,B,H),
     walk(P1,P2),
     state(P2,onfloor,B,H)).

canget(state(_,_,_,has)).

canget(State1):-
        move(State1,Move,State2),
        canget(State2).
```

- $? - canget(state(atdoor, onfloor, atwindow, hasnot))$.

Lists

- Either empty or consists of two parts: a *head* and a *tail*. The *tail* itself has to be a list
- Forms
  - $[Item1, Item2, \cdots]$
  - $[Head|Tail]$
  - $[Item1, Item2, \cdots|Others]$
  - E.g.: $[a, b, c] = [a|[b, c]] = [a, b|[c]] = [a, b, c|[]]$
- Items can be lists as well
  - $[[a, b], c, [d, [e, f]]]$
  - *head*=?

List::Membership

- Description: X is a member of L if either
  - X is the head of L, or
  - X is a member of the tail of L

$$member(X, [X|Tail]).$$
$$member(X, [Head|Tail]) \quad :- \quad member(X, Tail).$$

- E.g.
  - ?-$member(a, [b, c, a])$.
  - ?-$member(a, [b, [c, a]])$.
  - ?-$member(X, [b, [c, a]])$.

List::Concatenation

- $conc(L1, L2, L3)$
  - $conc([a], [b, c], [a, b, c])$ - **TRUE**
  - $conc([a, b], [c, d], [a, c, d, b])$ - **FALSE**
- If $L1$ is an empty list then $L2 = L3$
- If $L1$ is a non-empty list($[X|L1]$) then $L3$ is the concatenation of $L1$ and $L2$

$$conc([], L, L).$$
$$conc([X|L1], L2, [X|L3]) \quad :-$$
$$conc(L1, L2, L3).$$

Examples

- $? - conc([a, b], [c, d], [a, b, c, d])$.
- $? - conc([a], L, [a, b])$.

List::Add

- Put the new item in front of the list so that it becomes the new head

$$add(X, L, [X|L]).$$

List::Deletion

- $del(X, L, L1)$ where $L1$ is equal to the list L with the item X removed
- If X is the head of the list then the result after the deletion is the tail of the list
- If X is in the tail then it is deleted from there

$$del(X, [X|Tail], Tail).$$
$$del(X, [Y|Tail], [Y|Tail1]) \text{ :- } del(X, Tail, Tail1).$$

Examples

- how to define $Insert(X, List, BiggerList)$
  - $Insert(X, List, BiggerList) :- del(X, BiggerList, List).$

List::Permutations

- *permutation*(L, P).
- If L is empty then P must also be empty
- If L is non-empty and has the form $[X|L]$ then permute L and insert X to get P.

```
permutation([], []).
permutation([X|L], P)   :-
                        permutation(L, L1),
                        insert(X, L1, P).
```

Arithmetic and Logical Operators

- $+, -, \times, /, mod$
- *is*
  - ?-*X* *is* 7/2 will return $X = 3.5$ (force evaluation)
- $X > Y, X < Y, X =< Y, X >= Y$
- Equality: $X =:= Y$
- Un-Equality: $X = \backslash = Y$

List::Length

- *length*(*List*, *N*): count the elements in a list *List* and instantiate *N* to their number
- Description
  - If the list is empty then its length is 0
  - If the list is not empty then $List = [Head|Tail]$ then its length is equal to 1 plus the length of the tail *Tail*

```
length([], 0).
length([_|Tail], N)   :-
                      length(Tail, N1),
                      N is 1 + N1.
```

Cut - Controlling Backtracking

- Given KB define $f(X, Y)$

$$f(X, 0) :- \ X < 3. \qquad \text{C1}$$
$$f(X, 2) :- \ 3 =< X, \ X < 6. \qquad \text{C2}$$
$$f(X, 4) :- \ 6 =< X. \qquad \text{C3}$$

- ?-$f(1, Y), 2 < Y$.
  - All clause have been tried

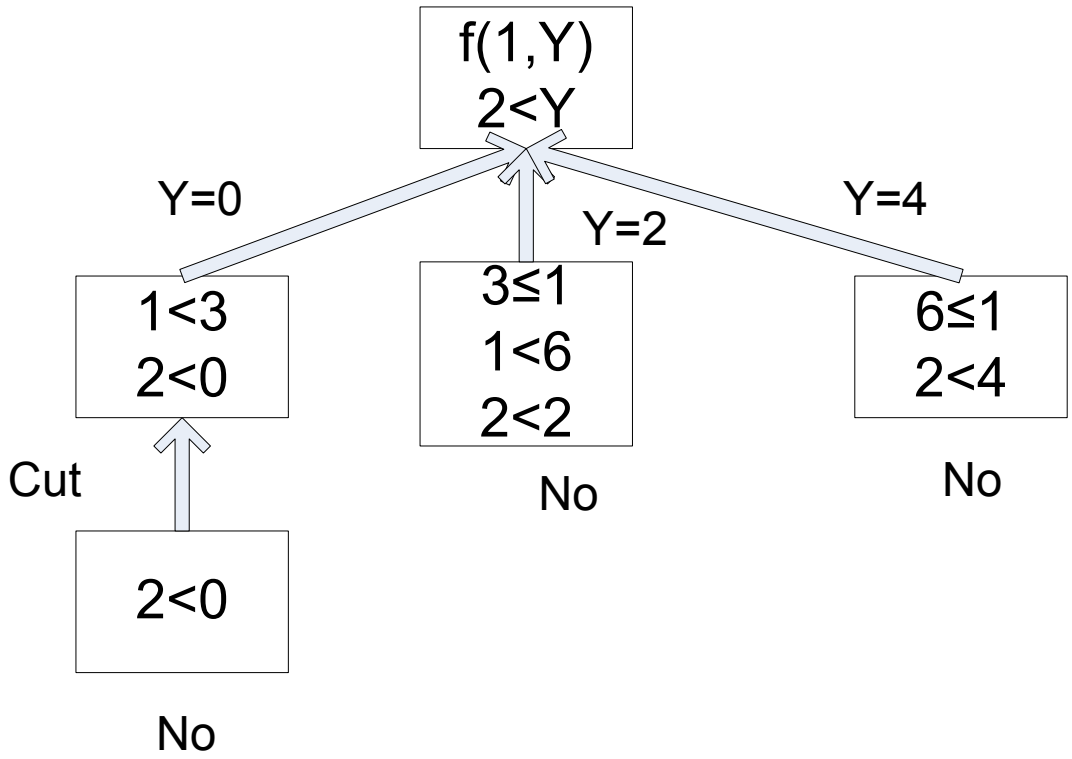
Cut - Controlling Backtracking

- Given KB define $f(X, Y)$

$$f(X, 0) :- \ X < 3, !. \qquad \text{C1}$$
$$f(X, 2) :- \ 3 =< X, \ X < 6, !. \qquad \text{C2}$$
$$f(X, 4) :- \ 6 =< X. \qquad \text{C3}$$

- ?-$f(1, Y), 2 < Y$.
  - Only $C1$ is tried

Cut - Controlling Backtracking



Cut - Controlling Backtracking

$C :- P, Q, R, !, S, T, U.$
$C :- V.$
$A :- B, C, D.$
$?- A.$

- Backtracking within the goal list P, Q, R
- As soon as the cut is reached:
  - All alternatives of P, Q, R are suppressed
  - The clause $C :- V$ will also be discarded
  - Backtracking possible within S, T, U
  - No effect within $A :- B, C, D$, that is, backtracking within B, C, D remains active

Examples

```
a(X) :- b(X), !, c(X).
b(1).
b(2).
b(3).
c(2).
```

- $? - \quad a(X).$
  - **Fail**

Examples

- Define $max(X, Y, Max)$ where $Max = X$ if X is greater than or equal to Y, and $Max = Y$ if X is less than Y

$$max(X, Y, X) :- \quad X >= Y, !.$$
$$max(X, Y, Y).$$

- Define $add(X, L, L1)$ where $L1$ has no redundant item by adding X into L

$$add(X, L, L) :- \quad member(X, L), !.$$
$$add(X, L, [X|L]).$$

Negation as failure

- Prolog says ”something is not true” by using a special goal **fail**
- *Thomas* likes all types of beer except Tiger

$$\begin{aligned} likes(thomas, X) :- & \quad tiger(X), !, fail. \\ likes(thomas, X) :- & \quad beer(X). \end{aligned}$$

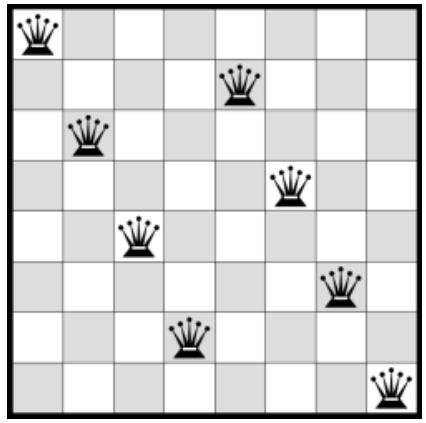
Examples

- Define *different*(X, Y) where it fails if X and Y match; otherwise, it succeeds

$$\begin{array}{l} \mathit{different}(X, X) :- \quad !, \mathit{fail}. \\ \mathit{different}(X, Y). \end{array}$$

Eight Queens Puzzle

- *Solution*$(\cdot)$ formulation
  - *template*$([1/Y1, 2/Y2, 3/Y3, 4/Y4, 5/Y5, 6/Y6, 7/Y7, 8/Y8])$
- Two Cases
  - An empty list of queens: *solution*$([])$
  - Non-empty list of queens: *solution*$([X/Y|Others])$
- No attack occurs between any two queens



Solution - 1

- *Others* must be a solution
  - $solution(Others)$
- X and Y must be integers $\in [1, 8]$
  - $member(Y, [1, 2, 3, 4, 5, 6, 7, 8])$
- A queen at square X/Y must not attack any of the queens in the list *Others*
  - $noattack(X/Y, Others)$

$$solution([X/Y|Others]) :- \quad solution(Others),\ member(Y, [1, 2, 3, 4, 5, 6, 7, 8]),\ noattack(X/Y, Others).$$

Solution - 2

- Predicate: $noattack(Q, Qlist)$
- Case 1: $Qlist$ is empty
  - $noattack(\_, [])$
- Case 2: $Qlist = [Q1, Qlist1]$
  - The queen at Q must not attack the queen at $Q1$
  - The queen at Q must not attack any of the queens in $Qlist1$

Solution - 3

- $noattack(X/Y,[X1/Y1|Others])$:The two squares must not be in the same row, the same column or the same diagonal
  - The Y-coordinates of the queens are different
    - $Y = \backslash = Y1$
  - The distance between the squares in the X-direction must not be equal to that in the Y-direction
    - $Y1 - Y = \backslash = X1 - X$
    - $Y1 - Y = \backslash = X - X1$

$$noattack(X/Y,[X1/Y1|Others]) :- \quad Y = \backslash = Y1, \; Y1 - Y = \backslash = X1 - X, \; Y1 - Y = \backslash = X - X1, \; noattack(X/Y, Others).$$

Program

```
/**PROGRAM**/

solution([]).
solution([X/Y|Others]):-
        solution(Others),
        member(Y,[1,2,3,4,5,6,7,8]),
        noattack(X/Y,Others).

noattack(_,[]).
noattack(X/Y,[X1/Y1|Others]):-
        Y=\=Y1,
        Y1-Y=\=X1-X,Y1-Y=\=X-X1,
        noattack(X/Y,Others).

member(Item,[Item|Rest]).
member(Item,[First|Rest]):-
        member(Item,Rest).

template([1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]).
```