Prolog Intro

Language Structure

- Prolog is a declarative language
- A program consists of:
  - A set of *facts*, predicates and relations that are known to hold, and
  - A set of *rules*, predicates and relations that are known to hold if other predicates or relations hold
- To run a Prolog program, you pose a query. The program reports all answers to your query that are true using the rules and facts declared in the program.

Prolog Example

Example (Program)

```
% Facts <-- This is a comment
person_height(norbert, (6,0)).
person_height(nelly,   (5,1)).
person_height(luca,    (5,3)).
% Rules
taller_shorter(X, Y) :- person_height(X, (FX, _)),
   person_height(Y, (FY, _)), FX > FY.
taller_shorter(X, Y) :- person_height(X, (FX, IX)),
  person_height(Y, (FY, IY)),
  FX =:= FY, IX > IY.
```

Example (Queries)

```
?- person_height(norbert, (F, I)). % F = 6, I = 0.
?- taller_shorter(luca, nelly). % true.
?- taller_shorter(X, nelly). % X = norbert; X = luca.
```

Prolog Objects

- Atoms:
  - Composed of letters, digits, and underscores
  - Start with a lowercase letter
  - Examples: `nelly person0 other_Item`
- Numbers:
  - Integers: `1 -3451913`
  - Floating point: `1.0 -12.318 4.89e-3`
- Variables:
  - Composed of letters, digits, and underscores
  - Start with an uppercase letter or underscore
  - Examples: `Person _has_underscore`
  - Special variable(wildcard): `_`

Terms

- Simple term:
  - Atom, number or variable
- Complex term:
  - Predicate:
    - ⟨*atom*⟩(⟨*term*⟩[,...])
    - Examples: `taller_shorter(X,Y)` `person_height(norbert,(6,0))`
  - Infix relation:
    - ⟨*term*⟩⟨*rel*⟩⟨*term*⟩
    - Examples: `X = pred(Y, Z) Number > 4`
  - Tuple:
    - (⟨*term*⟩[,...])
    - Examples: `(6,0) (Tail, Head)`
  - List:
    - [⟨*term*⟩[,...][|⟨*list*⟩]]
    - Examples: `[] [X] [_|_] [A,B|Rest]`

Facts and Rules

- Fact:
  - States what holds.
  - $\langle term \rangle$.
  - Examples: `loves_teaching(norbert).` `siblings(norbert,nelly).`
  - Can be read as a rule: $\langle term \rangle$: -*true*.
- Rule:
  - States how to deduce new facts from known facts.
  - $\langle head \rangle$: -$\langle term_1 \rangle, \ldots$.
  - $\langle head \rangle$ holds if $\langle term_1 \rangle, \ldots$ hold simultaneously.

Example (Rule)

```
taller_shorter(X, Y) :-
   person_height(X, (FX, IX)),
   person_height(Y, (FY, IY)), FX =:= FY, IX > IY.
```

Exercise

Example (Program)

```
fun(friday).
fun(water_slide).
boring(ira_roth_conversions).
boring(cleveland).
```

Results?

```
?- fun(X).
?- fun(Cleveland).
?- boring(cleveland).
```

Conjunction and Disjunction

Example (Conjunction between rules)

```
between(X, Smaller, Bigger) :-
    X > Smaller, X < Bigger.
% AND operator is a comma
```

Example (Disjunction between rules)

```
outside(X, Smaller, Bigger) :-
    X < Smaller; X > Bigger.
% OR operator is a semicolon
```

Example (Two separate rules for disjunction)

```
elem_list(Elem, [Elem|_]).
elem_list(Elem, [_|Tail]) :-
    elem_list(Elem, Tail).
```

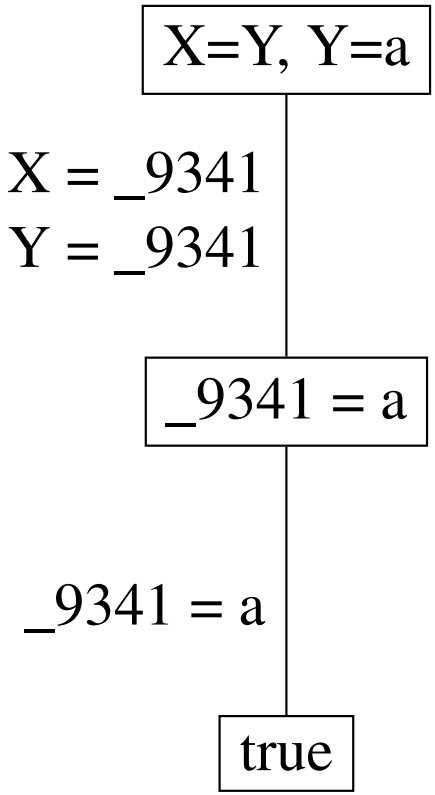
Unification

- A query term holds if it *unifies* with a term provable using the rules and facts in the program.
- Intuitively, two terms unify if the variables on both sides can be replaced with terms to make the two terms the same.
  - Every occurrence of a given variable needs to be replaced with the same term.
- Examples: (= tests whether two terms unify, \= tests whether they don't)
  - `X=X X=Y X=a(Y) a(X,y,z)=a(y,X,z) a\=b` all succeed (individually).
  - `X = a, X = b` fails because `X = a` forces X to equal a and then `a \= b`.

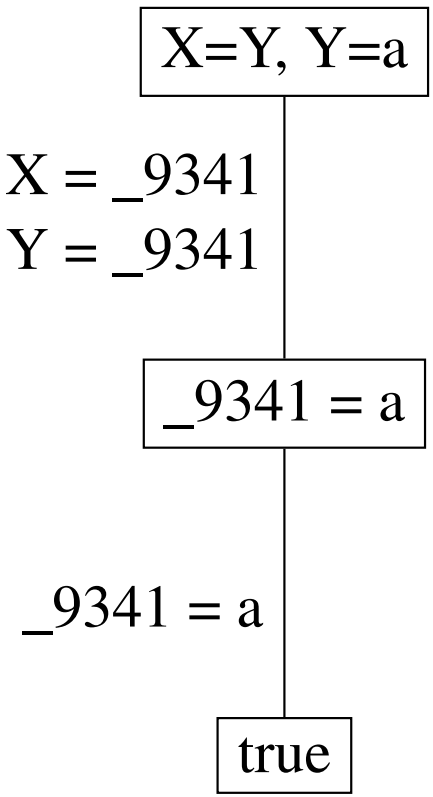
Formal Definition

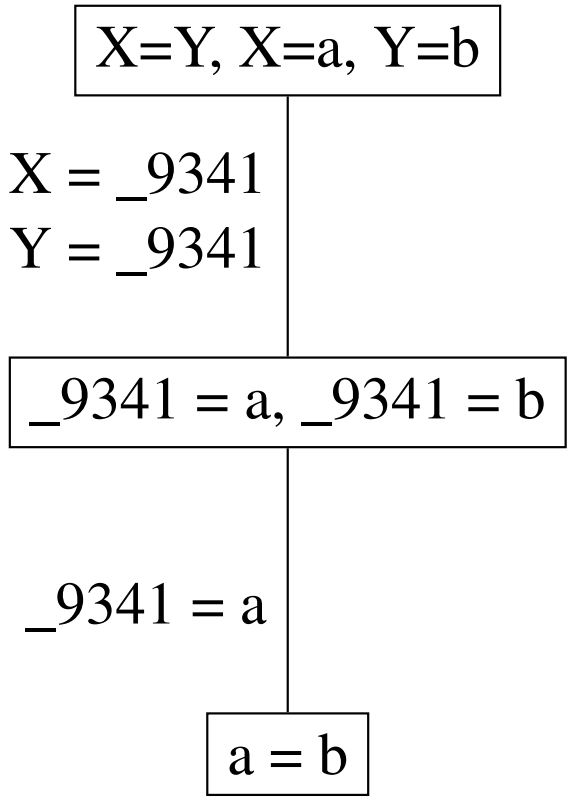
- Two identical terms unify.
- A variable unifies with any other term.
- If T_1 and T_2 are complex terms, they unify if
  - They have the same functor and arity,
  - Their corresponding arguments unify, and
  - The resulting variable instantiations are compatible.
- If none of the above rules applies to T_1 and T_2, then T_1 and T_2 do not unify.

Unification Diagrams



Unification Diagrams





Occurs Check

- What about the query `X = f(X)`?
- Logically, this should fail because there is no (finite) instantiation of X that makes the two sides equal.
- In Prolog, this query succeeds with the answer `X = f(X).`
- In the interest of efficiency, Prolog does not check whether a variable occurs in its own replacement.
- If you want to test for unification with occurs check, use **unify_with_occurs_check/2**

Example

```
?- unify_with_occurs_check(X, f(X)).
false.
```

Backtracking

To find the answers to a query, Prolog applies a depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the knowledge base from top to bottom.

Example

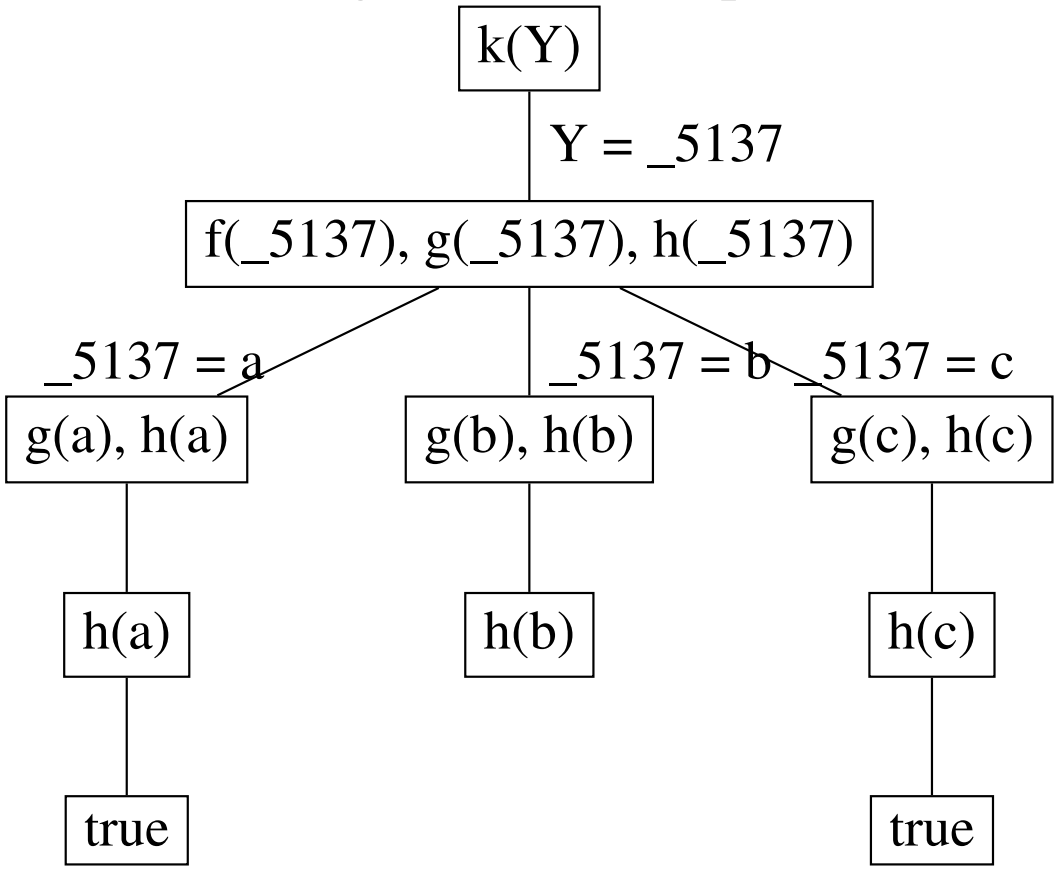
```
f(a).
f(b).
f(c).
g(a).
g(b).
g(c).
h(a).
h(c).
k(X) :- f(X),
  g(X), h(X).
```

Backtracking

To find the answers to a query, Prolog applies a depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the knowledge base from top to bottom.

Example

```
f(a).
f(b).
f(c).
g(a).
g(b).
g(c).
h(a).
h(c).
k(X) :- f(X),
  g(X), h(X).
```



Exercise

Program

```
queen(anne).
queen(victoria).
king(george).
king(edward).

royal_couple(X,Y) :- queen(X), king(Y).
```

Results?

```
?- king(X).

?- royal_couple(X, george).
?- royal_couple(X,Y).
```

Lists

- Sequences and collections are represented as lists.
- Since list elements can themselves be lists, we can use lists to represent complicated data structures such as trees (even though they are often better represented as deeply nested complex terms).
  - Empty list: `[]`
  - Head and tail: `[a|[b,c,d]] = [a,b,c,d]` `[a|[]] = [a]`
  - Multiple heads: `[a,b|[c,d]] = [a,b,c,d]`

Using Prolog

Control Flow

- The notion of “control flow” is much weaker in Prolog than even in a functional language because we are (mostly) not concerned with the order in which the Prolog interpreter does things.
- What we need is a way to build up arbitrarily complex relations using recursion.
- Follow tail recursion patterns with an accumulator variable because there is not a concept of a returned value.

Recursion

Example (Summing a list of integers)

```
sum([], 0).
sum([X|Xs], Sum) :-
  sum(Xs, Sum1), Sum is Sum1 + X.
```

Example (Summing a list of integers (better))

```
sum([], 0).
sum([X|Xs], Sum) :-
  Sum #= Sum1 + X, sum(Xs, Sum1).
% finite domain constraint... not in scope of class
```

Mapping a Predicate Over a List

Example

```
odd(X) :- 1 is X mod 2.

?- maplist(odd,[1,3,5]).
true.
?- maplist(odd,[1,2,3]).
false.

?- maplist(<,[1,3,5],[2,7,8]).
true.
?- maplist(<,[1,3,9],[2,7,8]).
false.

add(X,Y,Sum) :- Sum is X+Y.
?- maplist(add,[1,3,5],[4,8,9],Sums).
Sums = [5,11,14].
```

Built-In Predicates

- Primitives:
  - true, false
  - fail (is the same as false)
- Unification:
  - = (arguments unify), \= (arguments do not unify)
- Arithmetic and numeric comparisons: (Use with caution.)
  - `+, -, *, /, //`
  - `<, >, >=, =<, =:=, =\=`
  - 5 \= 2+3 but X is 2+3, 5 = X
- Lots more

Goal Ordering

Example (Given)

```
f(e).  g(a).  g(b).  g(c).  g(d).  g(e).
```

Example (Two equivalent rules)

```
h1(X) :- f(X), g(X).
h2(X) :- g(X), f(X).
```

- Is one more efficient than the other?
  - h1 instantiates `X = e` and then succeeds because g(e) holds.
  - h2 instantiates `X = a, X = b, ...` and fails on all instantiations except `X = e`.

Goal Ordering with Recursion

Example (Given)

```
parent(anne,bridget).    parent(bridget,caroline).
parent(caroline,donna).  parent(donna,emily).
```

Example (Two equivalent relationships)

```
descend1(X,Y)
  :- parent(X,Z), descend1(Z,Y).
descend1(X,Y) :- parent(X,Y).
descend2(X,Y)
  :- descend2(Z,Y), parent(X,Z).
descend2(X,Y) :- parent(X,Y).
```

- `descend1(anne,bridget)` succeeds.
- `descend2(anne,bridget)` does not terminate.

Exercise 1

Example (Given)

```
parent(anne,bridget).    parent(bridget,caroline).
parent(caroline,donna).  parent(donna,emily).
```

Write a **grandparent/2** predicate that is true if the first argument is a grandparent of the second.

Exercise 2

Write an **a2b/2** predicate that is true if the first argument is a list of **a**'s, and the second argument is an equal-length list of **b**'s.

Example (Results)

```
?- a2b([a,a,a,a],[b,b,b,b]).
true
?- a2b([a,a,a,a],[b,b,b]).
false
?- a2b([a,c,a,a],[b,b,b,t]).
false
```

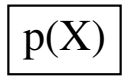
Cut

- ! (read “cut”) is a predicate that always succeeds, but with a side effect:
  - It commits Prolog to all choices (unification of variables) that were made since the parent goal was unified with the left-hand side of the rule.
  - This includes the choice to use this particular rule.

Example with no cut

Example

```
a(1). b(1). b(2). c(1). c(2). d(2). e(2). f(3).
p(X):-a(X). p(X):-b(X), c(X), d(X), e(X). p(X):-f(X).
```

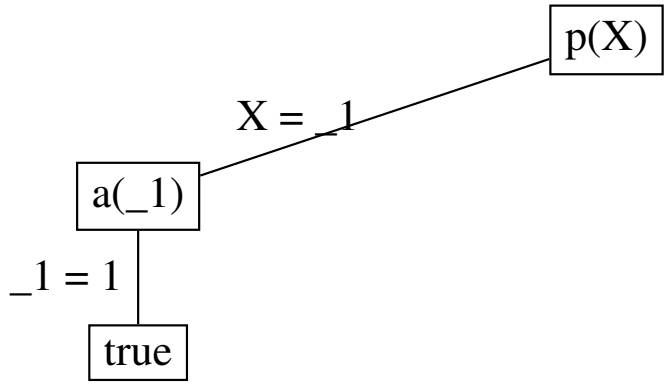


- ?- p(X).
- X = 1;
- X = 2;
- X = 3.

Example with no cut

Example

```
a(1). b(1). b(2). c(1). c(2). d(2). e(2). f(3).
p(X):-a(X). p(X):-b(X), c(X), d(X), e(X). p(X):-f(X).
```



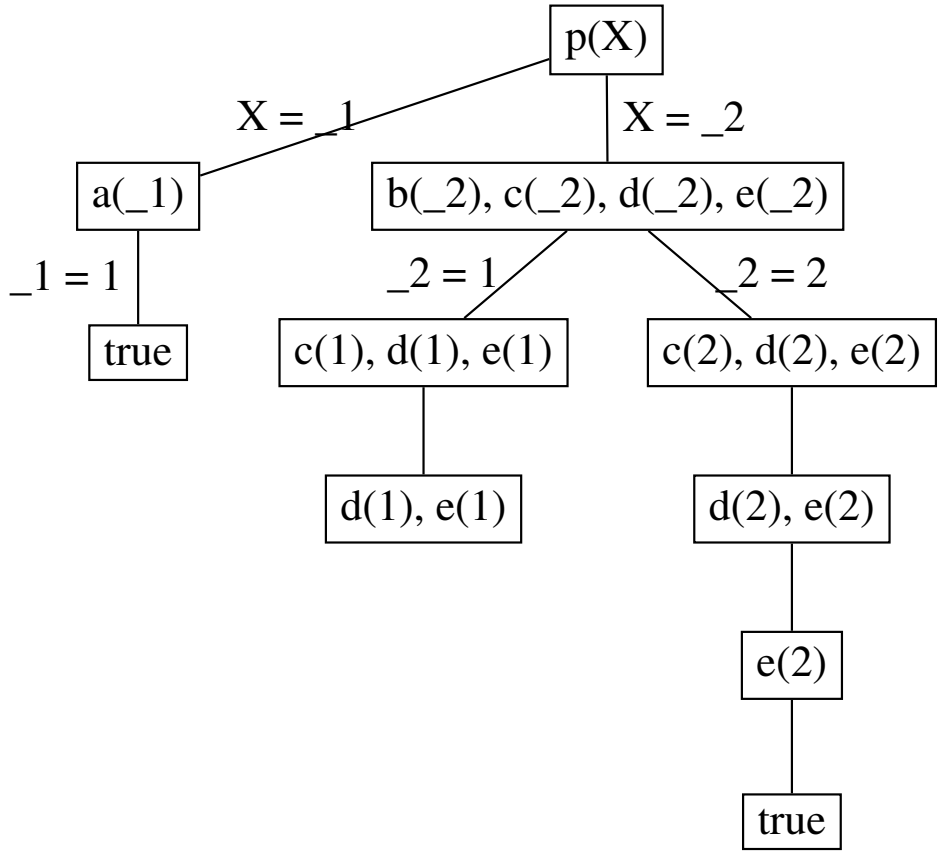
- `?- p(X).`
- `X = 1;`
- `X = 2;`
- `X = 3.`

Example with no cut

Example

```
a(1). b(1). b(2). c(1). c(2). d(2). e(2). f(3).
p(X):-a(X). p(X):-b(X), c(X), d(X), e(X). p(X):-f(X).
```

- `?- p(X).`
- `X = 1;`
- `X = 2;`
- `X = 3.`

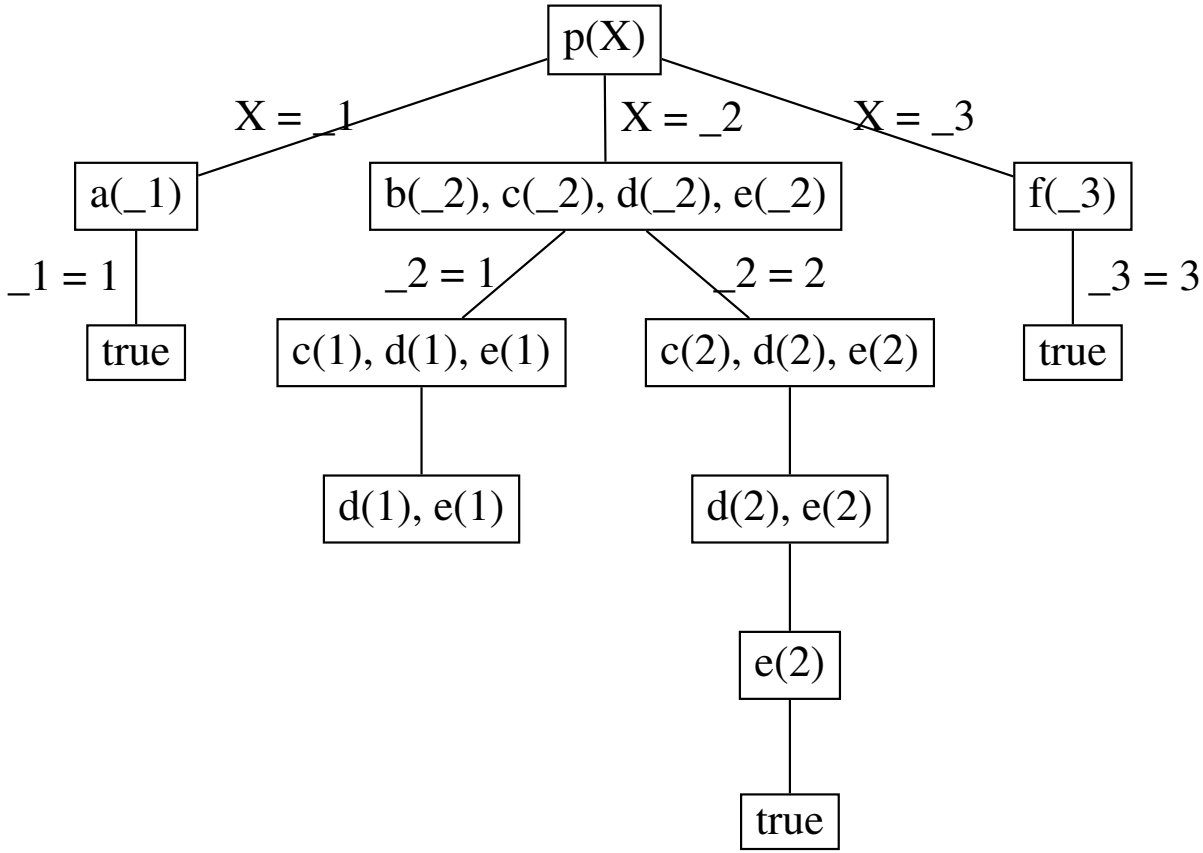


Example with no cut

Example

```
a(1). b(1). b(2). c(1). c(2). d(2). e(2). f(3).
p(X):-a(X). p(X):-b(X), c(X), d(X), e(X). p(X):-f(X).
```

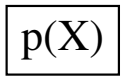
- `?- p(X).`
- `X = 1;`
- `X = 2;`
- `X = 3.`



Example with cut

Example

```
a(1). b(1). b(2). c(1). c(2). d(2). e(2). f(3).
p(X):-a(X). p(X):-b(X), c(X), !, d(X), e(X). p(X):-f(X).
```

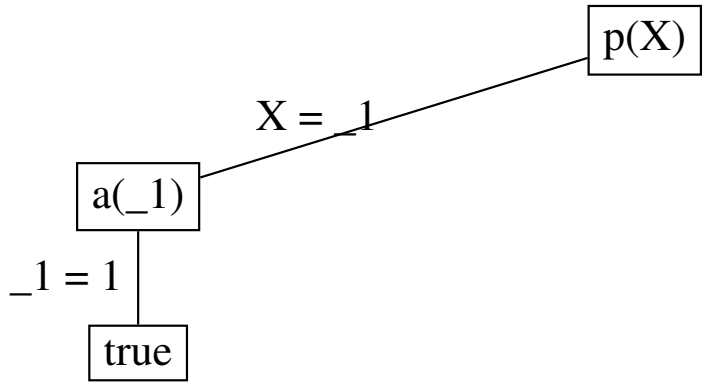


- ?-
  p(X).
- X = 1;
- false.

Example with cut

Example

```
a(1). b(1). b(2). c(1). c(2). d(2). e(2). f(3).
p(X):-a(X). p(X):-b(X), c(X), !, d(X), e(X). p(X):-f(X).
```



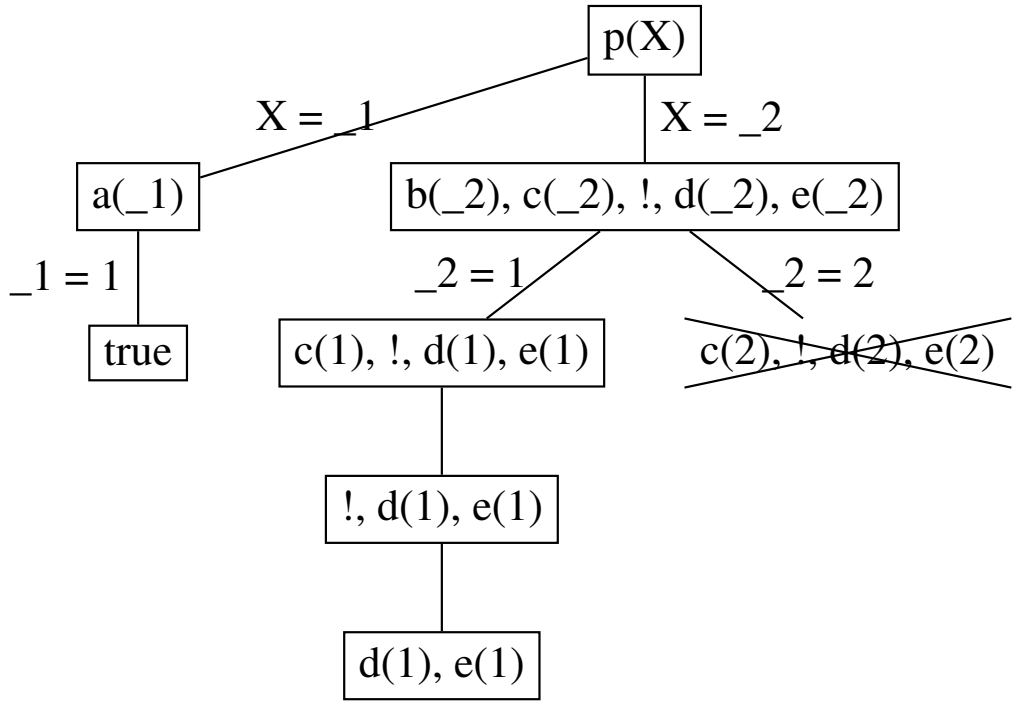
- `?- p(X).`
- `X = 1;`
- `false.`

Example with cut

Example

```
a(1). b(1). b(2). c(1). c(2). d(2). e(2). f(3).
p(X):-a(X). p(X):-b(X), c(X), !, d(X), e(X). p(X):-f(X).
```

- `?- p(X).`
- `X = 1;`
- `false.`

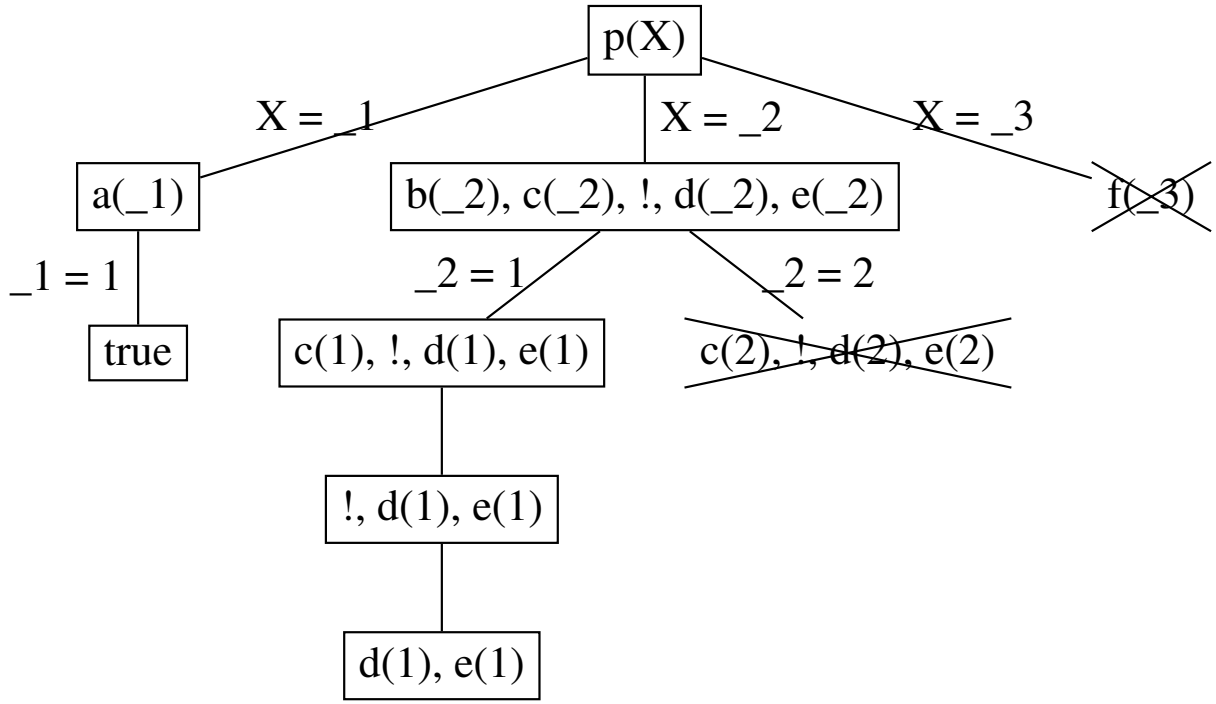


Example with cut

Example

```
a(1). b(1). b(2). c(1). c(2). d(2). e(2). f(3).
p(X):-a(X). p(X):-b(X), c(X), !, d(X), e(X). p(X):-f(X).
```

- `?-`
  `p(X).`
- `X = 1;`
- `false.`



Second Cut Example

In the q branch of the tree, X is unified with 1 at the cut, so results like `X = 2, Y = 4` are no longer possible.

Example

```
p(X,Y) :- q(X,Y).
p(3,6).
q(X,Y) :- a(X), !, b(Y).
q(4,7).
a(1).  a(2).
b(4).  b(5).

?- p(X,Y).
X = 1, Y = 4 ;
X = 1, Y = 5 ;
X = 3, Y = 6.
```

Exercise

Example (Given)

```
teaches(dr_fred, history).  studies(alice, english).
teaches(dr_fred, english).  studies(angus, english).
teaches(dr_fred, drama).    studies(amelia, drama).
teaches(dr_fiona, physics). studies(alex, physics).
```

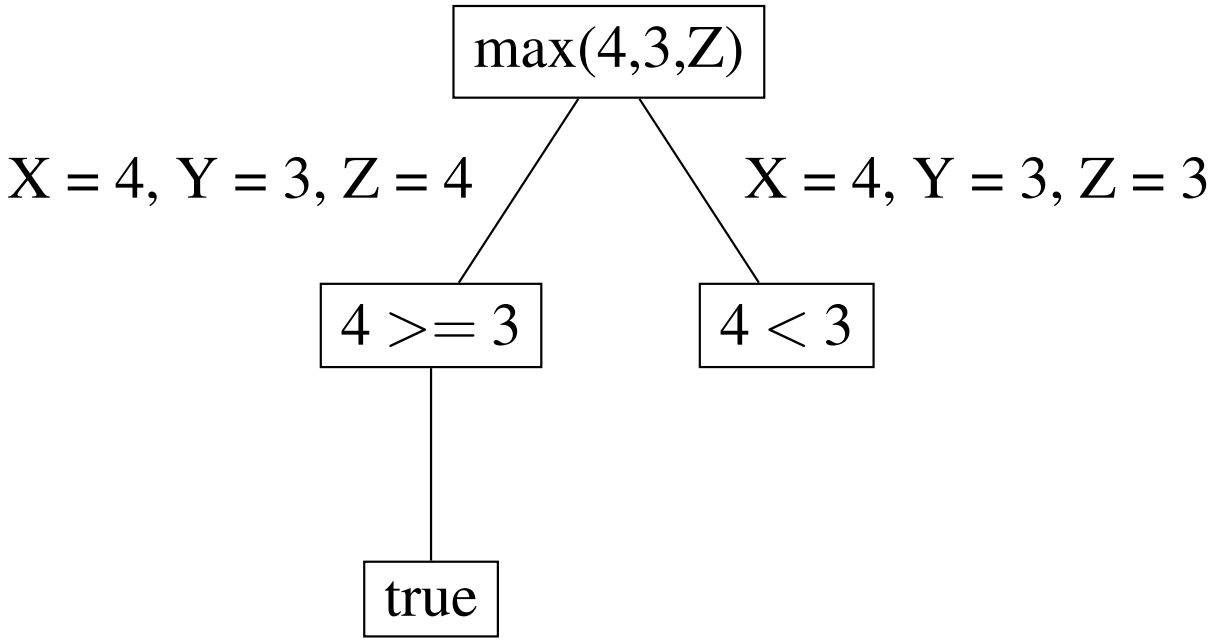
Results?

```
?- teaches(dr_fred, Course), !, studies(Student, Course).
?- teaches(dr_fred, Course), studies(Student, Course), !.
?- !, teaches(dr_fred, Course), studies(Student, Course).
```

Third Cut Example

Example (A predicate to compute the maximum)

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.
```

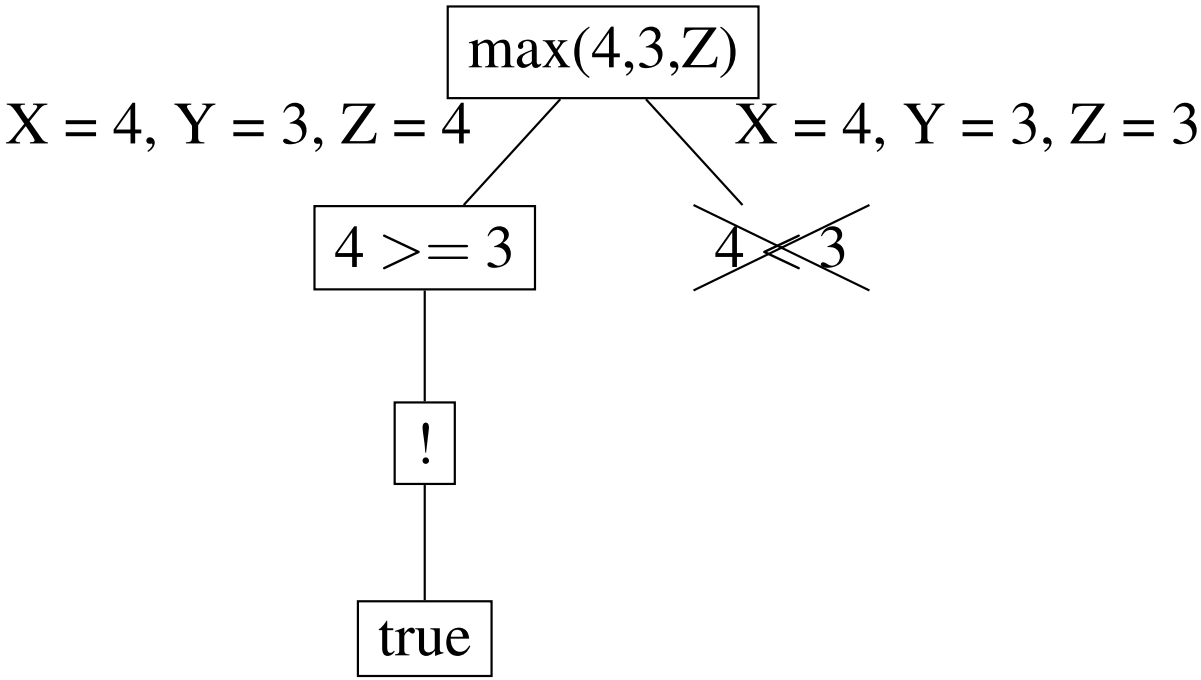


Correct but inefficient – branch truths are mutually exclusive.

Third Cut Example

Example (A predicate to compute the maximum)

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) :- X < Y.
```



Negation

- In general, Prolog has no notion of a predicate not being true! It can only decide whether it can prove the predicate using the information in the database.
- This is called "negation as failure".
- It is useful to be able to ask the question: "Are you unable to prove this predicate?" (Is this predicate false?)

Example

```
neg(P) :- P, !, fail.
neg(_).
Example:
?- neg(true).
false.
?- neg(false).
true.
```

Prolog has a built-in unary operator \+ that does exactly what neg does. Thus, these two queries become `\+ true.` and `\+ false.`

Once Predicate

- Sometimes, we know that a predicate can match only once or we never need more than one solution.
- In these cases, we would like to prevent Prolog from searching for additional solutions, in the interest of efficiency.
- `once(P)`:
  - Fails if P fails.
  - Succeeds if P succeeds but finds only one solution.

Example

```
a(1). a(2).
?- a(X).
X = 1 ;
X = 2.
```

Example

```
a(1). a(2).
?- once(a(X)).
X = 1.
```