

Logic Programming with Prolog

Prolog is based on three main ideas:

- Logical Horn rules (day before last)
- Unification (last day)
- Top-down reasoning (today)

Reasoning

- Bottom-up (or forward) reasoning: starting from the given facts, apply rules to infer everything that is true.

e.g., Suppose the fact B and the rule $A \leftarrow B$ are given. Then infer that A is true.

- Top-down (or backward) reasoning: starting from the query, apply the rules in reverse, attempting only those lines of inference that are relevant to the query.

e.g., Suppose the query is A , and the rule $A \leftarrow B$ is given. Then to prove A , try to prove B .

Bottom-up Inference

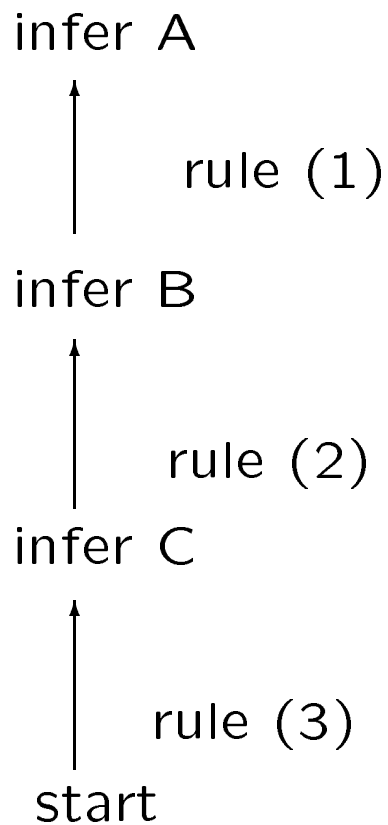
A rule base:

A \leftarrow B (1)

B \leftarrow C (2)

C (3)

A bottom-up proof:



So, A is proved

Top-Down Inference

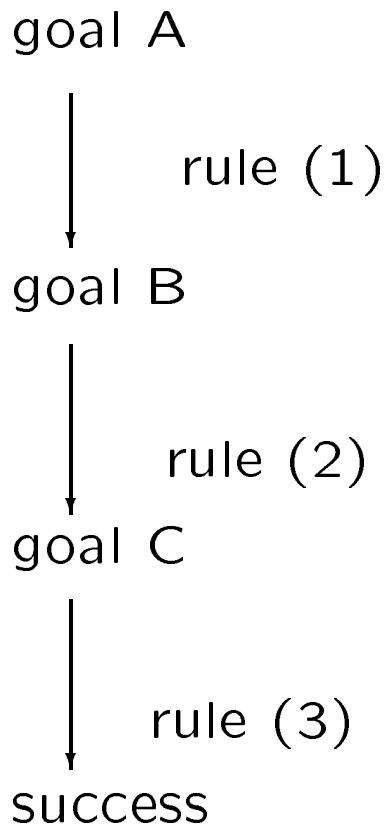
A rule base:

A \leftarrow B (1)

B \leftarrow C (2)

C (3)

A top-down proof:



So, A is proved

Top-down vs Bottom-up Inference

- Prolog uses top-down inference, although some other logic programming systems use bottom-up inference (*e.g.*, Coral).
- Each has its own advantages and disadvantages:
 - Bottom-up may generate many irrelevant facts.
 - Top-down may explore many lines of reasoning that fail.
- Top-down and bottom-up inference are logically equivalent.
i.e., they both prove the same set of facts.
- However, only top-down inference simulates program execution.
i.e., execution is inherently top down, since it proceeds from the main procedure downwards, to subroutines, to sub-subroutines, etc.

Example 1

Bottom-up inference can derive
many facts.

Rule base:

$p(X, Y, Z) \leftarrow q(X), q(Y), q(Z).$

$q(a_1).$

$q(a_2).$

...

$q(a_n).$

Bottom-up inference derives n^3 facts of the
form $p(a_i, a_j, a_k)$:

$p(a_1, a_1, a_1)$

$p(a_1, a_1, a_2)$

$p(a_1, a_2, a_3)$

...

Example 2

Bottom-up inference can derive
infinitely many facts.

Rule base:

```
p(f(x)) ← p(x).  
p(a).
```

Derived facts:

```
p(f(a))  
p(f(f(a)))  
p(f(f(f(a))))  
...
```

In contrast, top-down inference derives only the facts requested by the user, e.g.

who does jane love?

what is john's telephone number?

Example 3

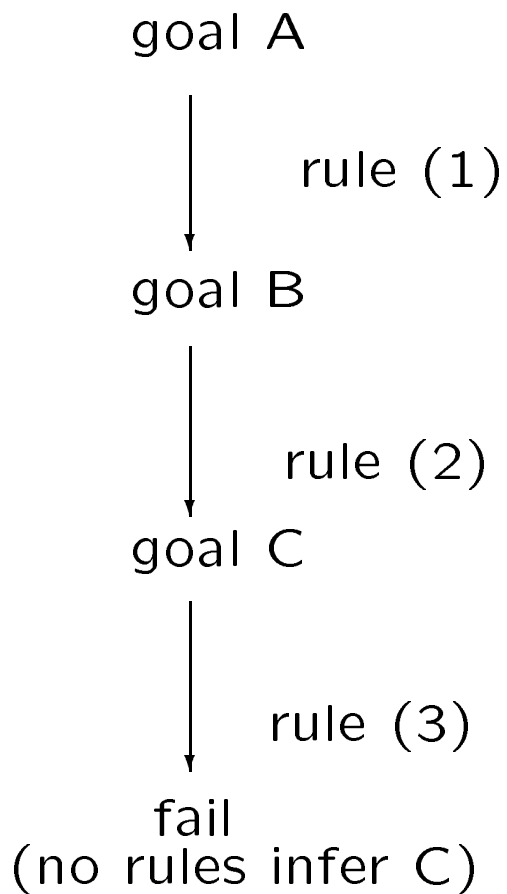
Top-down inference may fail.

Rule base:

$A \leftarrow B$ (1)

$B \leftarrow C$ (2)

Failed line of inference:



So, A is not proved

Multiple Rules and Premises

A fact may be inferred by many rules. *e.g.*,

$E \leftarrow B$

$E \leftarrow C$

$E \leftarrow D$

A rule may have many premises. *e.g.*,

$E \leftarrow B \wedge C \wedge D$

In top-down inference, such rules give rise to

- inference trees
- backtracking

Example 1: Multiple Premises

Rule base:

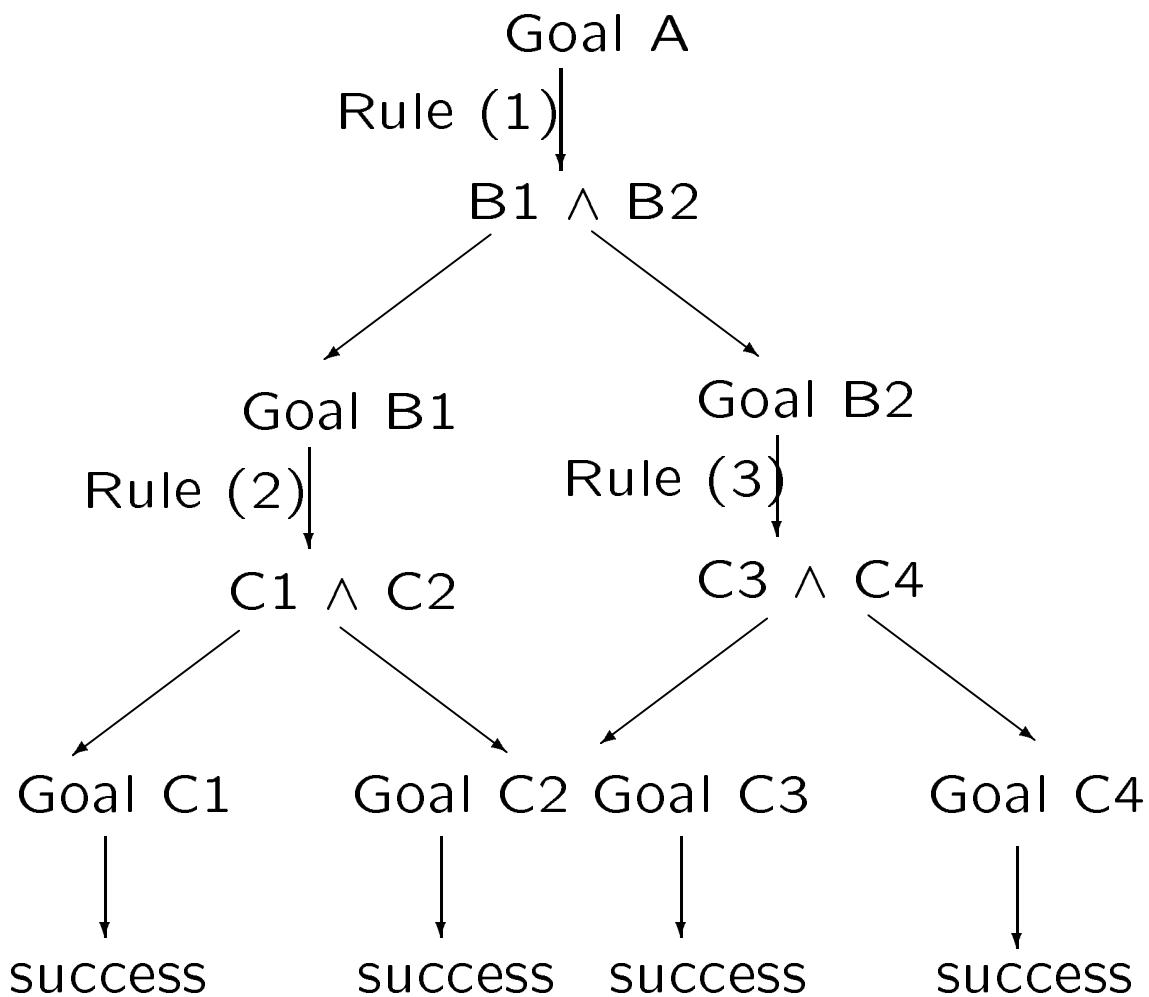
(1) $A \leftarrow B1 \wedge B2$

(2) $B1 \leftarrow C1 \wedge C2$

(3) $B2 \leftarrow C3 \wedge C4$

C1 C2 C3 C4

Query: Is A true?



So, goal A is proved. (all paths must succeed)

Example 2: Multiple Rules

Rule base:

A ←- B1

B1 ←- C1

B2 ←- C3

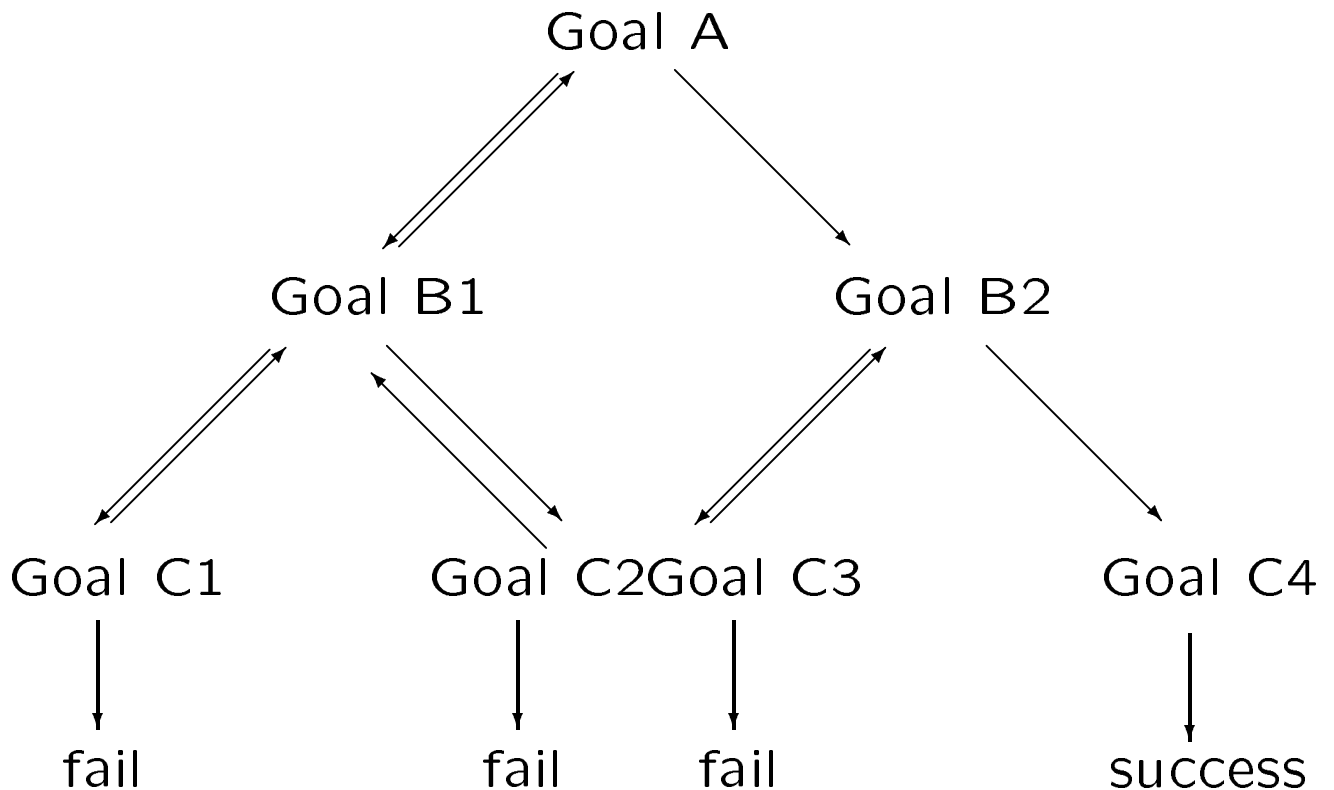
A ←- B2

B1 ←- C2

B2 ←- C4

C4

Query: Is A true?



So, goal A is proved. (only one path must succeed)

Example 3: Variables

Rule base:

```
student(1234,sam).      enrolled(1234,csc324).
student(3456,joe).     enrolled(1234,csc364).
student(5678,lisa).    enrolled(1234,csc378).
student(6789,bart).    enrolled(3456,csc324).
                       enrolled(3456,csc364).
                       enrolled(5678,csc378).

takes(Name, Course) :- student(Number, Name),
                       enrolled(Number, Course).

% i.e., view course enrollment in terms of
% student names, instead of student numbers.
```

Query:

Find N and C such that takes(N,C) is true.

Answer:

```
N=sam,   C=csc324;
N=sam,   C=csc364;
N=sam,   C=csc378;
N=joe,   C=csc324;
N=joe,   C=csc364;
N=lisa,  C=csc378;
no
```

Example 3 (continued)

Same rule base:

```
student(1234,sam).      enrolled(1234,csc324).
student(3456,joe).     enrolled(1234,csc364).
student(5678,lisa).    enrolled(1234,csc378).
student(6789,bart).    enrolled(3456,csc324).
                       enrolled(3456,csc364).
                       enrolled(5678,csc378).
```

```
takes(Name, Course) :- student(Number, Name),
                       enrolled(Number, Course).
```

Query:

Find N such that takes(N,csc324) is true.

Answer:

N=sam;

N=joe;

no

Example 4: Backtracking

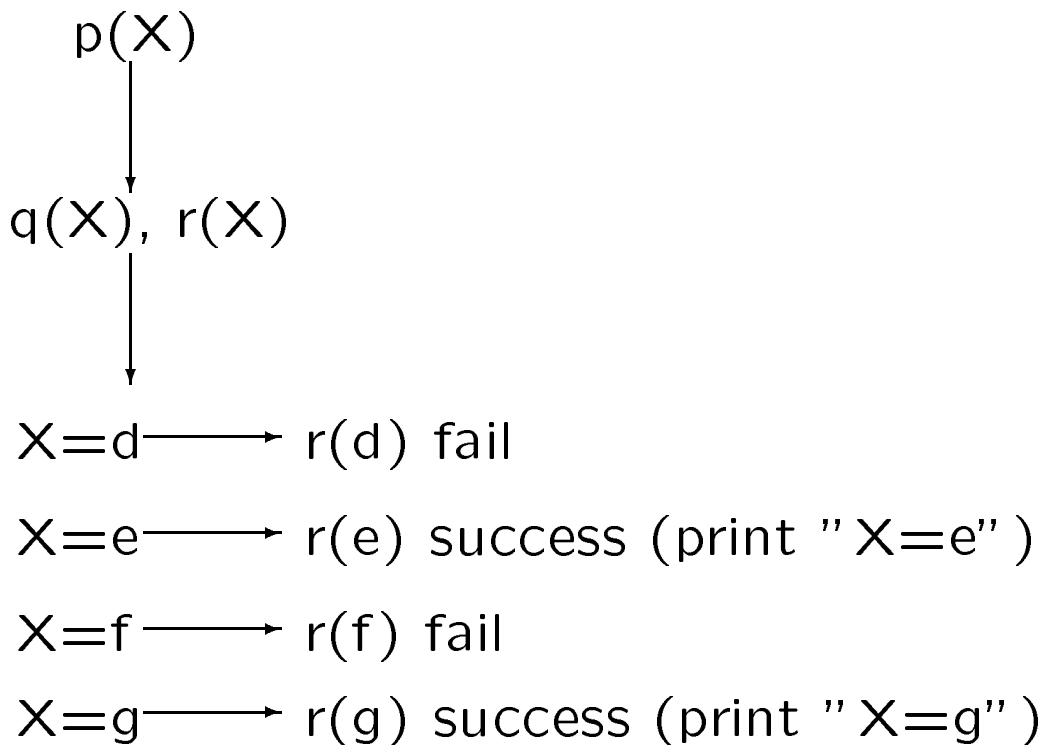
Rule base:

$p(X) \text{ :- } q(X), r(X).$

$q(d).$ $q(e).$ $q(f).$ $q(g).$

$r(e).$ $r(g).$

Query: Find x such that $p(x)$ is true.



Example 5: Backtracking

Rule base:

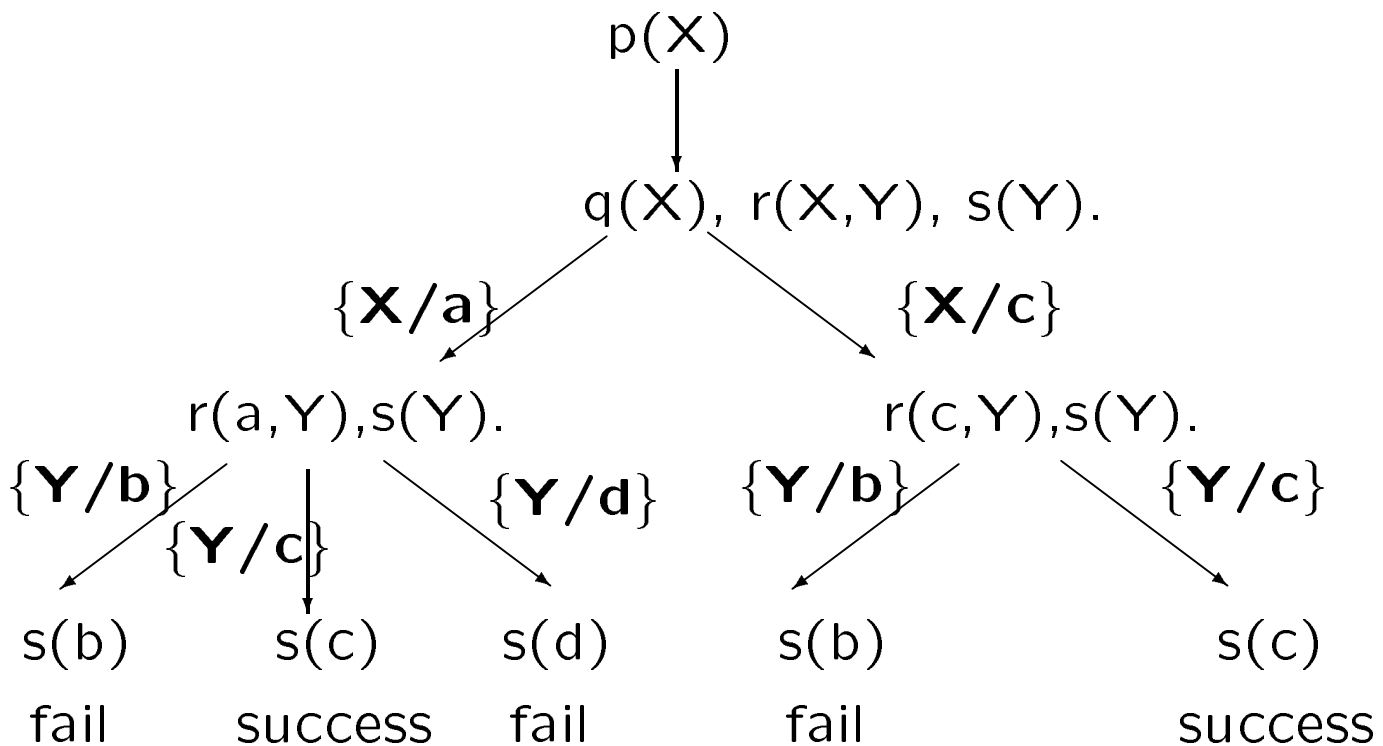
$p(X) \text{ :- } q(X), r(X, Y), s(Y).$

$q(a). \quad r(a, b). \quad r(c, b). \quad s(c).$

$q(c). \quad r(a, c). \quad r(c, c).$

$r(a, d).$

Query: Find x such that $p(x)$ is true.



Hints on Debugging

We can follow the execution of Prolog programs with `write` statements. *e.g.*,

Rule base:

```
p(X) :- q(X), write(X), r(X).  
q(a).  q(b).  q(c).  q(d).  q(e).  
r(a).  r(d).
```

Query: Find `X` such that `p(X)` is true.

Then Prolog prints:

```
a  
X = a  
bcd  
X = d  
e  
no
```

Recursion in Prolog

If a predicate symbol occurs in both the head and body of a rule, then the rule is *recursive*.

For example,

```
a(X) :- b(X,Y), a(Y).
```

i.e., to prove $a(X)$, Prolog must prove $a(Y)$.

The predicate a acts like a recursive subroutine.

It is called a recursively defined predicate, or simply a recursive predicate.

Mutual Recursion

Recursion might be indirect, involving several rules. For example,

$$\begin{aligned} a(X) &:- b(X,Y), c(Y). \\ c(Y) &:- d(Y,Z), a(Z). \end{aligned}$$

Thus, to prove $a(X)$,

Prolog tries to prove $c(Y)$ (and $b(X,Y)$)
so it tries to prove $a(Z)$ (and $d(Y,Z)$).

i.e., to prove $a(X)$, Prolog tries to prove $a(Z)$.

The predicates a and c are said to be mutually recursive.

Non-Linear Recursion

When the head predicate appears multiple times in the body of a rule, then the recursion is said to be *non-linear*.

For example,

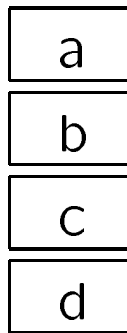
$$a(X) \text{ :- } b(X,Y), a(Y), c(Y,Z), a(Z).$$

i.e., to prove $a(X)$, Prolog tries to prove *both* $a(Y)$ and $a(Z)$.

This generates a recursive proof tree.

Example (Linear Recursion)

A stack of 4 toy blocks.



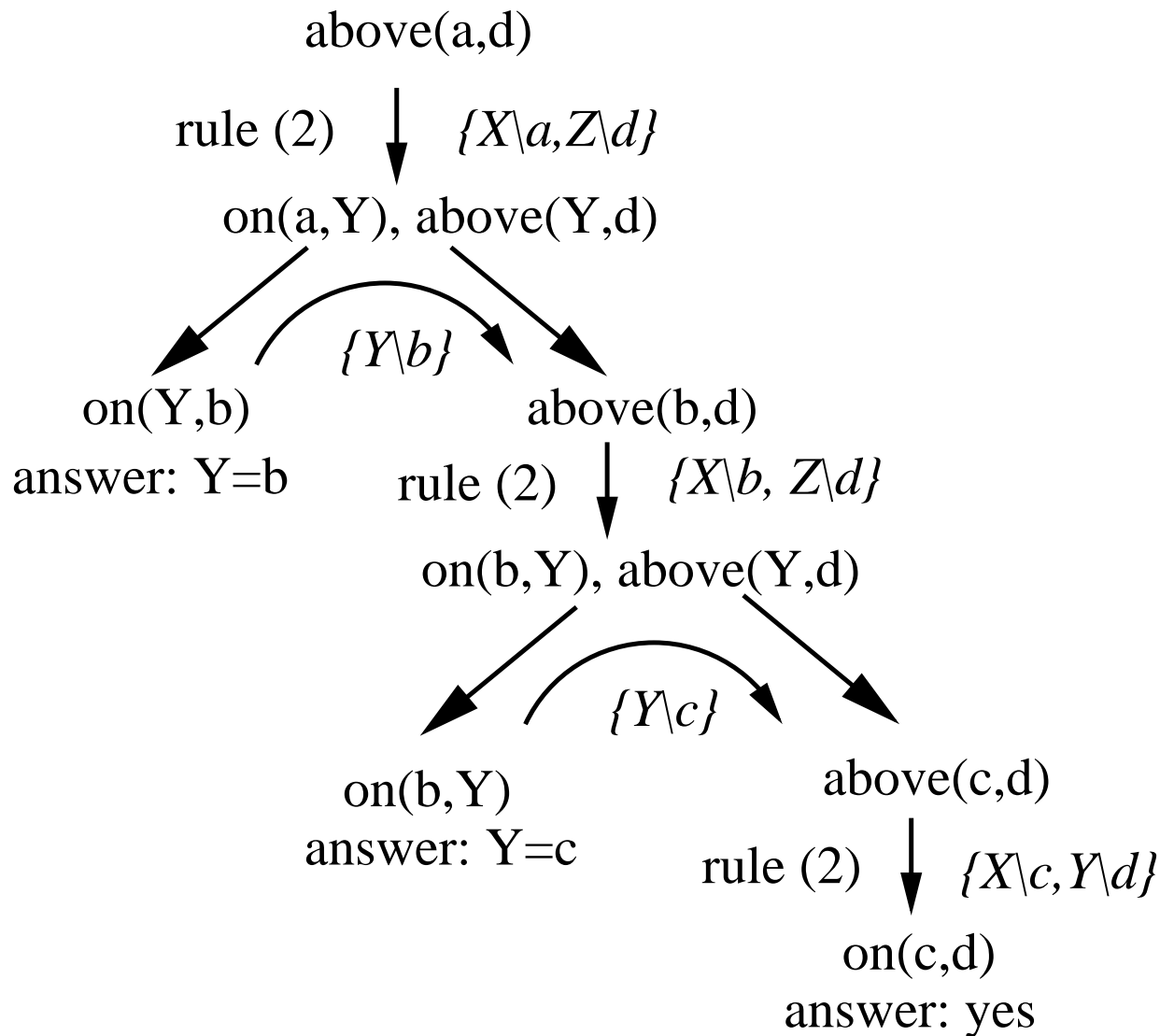
Rules:

- (1) `above(X,Y) :- on(X,Y).`
- (2) `above(X,Z) :- on(X,Y), above(Y,Z).`
- (3) `on(a,b).`
- (4) `on(b,c).`
- (5) `on(c,d).`

Query: `? - above(a,d)`

Use top-down inference.

Tree



All leaves are true, so the root is true,
i.e., above(a,d) is true.

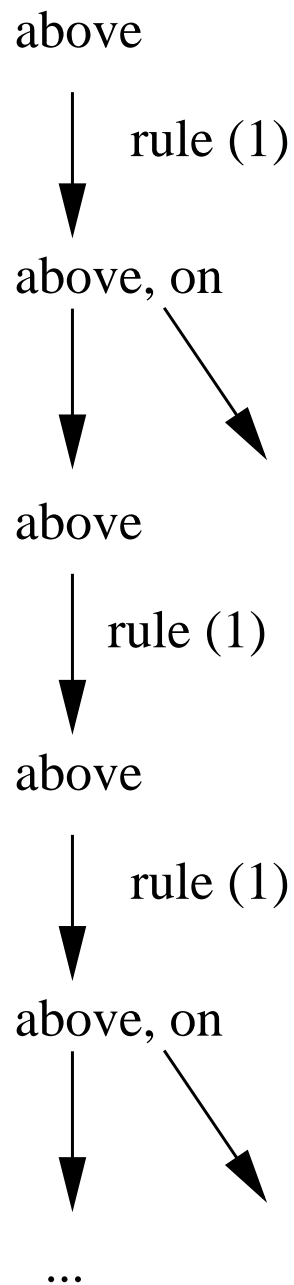
Observation

Changing the order of rules and/or rule premises can cause problems for Prolog. Example:

- (1) `above(X,Z) :- above(Y,Z), on(X,Y).`
- (2) `above(X,Y) :- on(X,Y).`

Because Prolog processes premises from left to right, and rules from first to last, rule (1) causes an infinite loop.

Tree



This is a flaw in Prolog.

Beyond Horn Logic

- So far, we have studied what is known as *pure* logic programming, in which all the rules are Horn.
- For some applications, however, we need to go beyond this.
- For instance, we often need
 - Negation
 - Existential quantification
 - Arithmetic
- Fortunately, these can easily be accommodated by simple extensions to the logic-programming framework,

Negation in Prolog

- Prolog uses negation as failure.
- *i.e.*, if you cannot prove something is true, then assume it is false. *e.g.*, unless we have reason to believe otherwise, we assume the sun will rise tomorrow.
- This is NOT logical negation, but it is easy to implement, and it is typical of much common-sense reasoning.
- In Prolog, negation may appear only in queries and in rule bodies.
- For example, the rule

$$a \leftarrow b \wedge \sim c$$

is written in Prolog as

$$a \text{ :- } b, \text{ not } c.$$

and it means, "*infer a if b can be inferred and c cannot be inferred.*"

Example

```
loves(bill,X) :- pretty(X), female(X),
                not loves(tom,X).
```

i.e., Bill loves any pretty female, unless Tom loves her.

```
loves(tom,X) :- famous(X), female(X),
                not dead(X).
```

i.e., Tom loves any famous living female.

```
female(marilyn-monroe).  famous(marilyn-monroe).
female(cindy-crawford). famous(cindy-crawford).
female(martha-stewart). famous(martha-stewart).
female(girl-next-door).
```

```
pretty(marilyn-monroe).  dead(marilyn-monroe).
pretty(cindy-crawford).
pretty(girl-next-door).
```

```
| ?- loves(tom,X).
   X = cindy-crawford;
   X = martha-stewart;
   no
```

```
| ?- loves(bill,X).
   X = marilyn-monroe;
   X = girl-next-door;
   no
```

Safety

Consider the following rule:

(*) `hates(tom,X) :- not loves(tom,X).`

This may NOT be what we want, for several reasons:

- The answer is *infinite*, since for any person *p* not mentioned in the database, we cannot infer `loves(tom,p)`, so we must infer `hates(tom,p)`.

Rule (*) is therefore said to be unsafe.

- The rule does not require *X* to be a person.
e.g., since we cannot infer

`loves(tom,hammer)`

`loves(tom,verbs)`

`loves(tom,green)`

`loves(tom,abc)`

we must infer that tom hates all these things.

Safety (Cont'd)

To avoid these problems, rules with negation should be guarded:

```
hates(tom,X) :- female(x), pretty(X),  
                not loves(tom,X).
```

i.e., Tom hates every pretty female whom he does not love.

Here, `female` and `pretty` are called guard literals. They guard against safety problems by binding `X` to specific values in the database.

Quantified Rule Bodies

$$\forall X [happy(X) \leftarrow \forall Y loves(Y, X)]$$

i.e., A person is happy if everyone loves him.
This rule is not Horn.

$$\forall X [happy(X) \leftarrow \exists Y loves(Y, X)]$$

i.e., A person is happy if someone loves him.
This rule is not Horn either, but it is equivalent to the following Horn rule:

$$\forall X \forall Y [happy(X) \leftarrow loves(Y, X)]$$

Why? (Left as an exercise)

Examples:

$$\begin{aligned} loves(bill, mary) &\Rightarrow happy(mary) && \{X \setminus mary, Y \setminus bill\} \\ loves(bill, cindy) &\Rightarrow happy(cindy) && \{X \setminus cindy, Y \setminus bill\} \\ loves(tom, cindy) &\Rightarrow happy(cindy) && \{X \setminus cindy, Y \setminus tom\} \end{aligned}$$

So, in Horn logic, existential quantifiers can appear in the premise of a rule.

They can also appear in queries, since a rule premise is just a query placed inside a rule.

Declarative Arithmetic

What we would like:

- Given a set of equations with variables, find values of the variables that satisfy the equations.

eg,. query: $X + 3 = 5.$

answer: $X = 2$

query: $X + Y = 1, X - Y = 2.$

answer: $X = 3/2, Y = -1/2$

query: $X^2 = 4.$

answers: $X = 2$

$X = -2$

query: $X + Y = 0, 2X + 2Y = 1.$

answer: no

(no solutions since equations are contradictory)

query: $X = 1, X = 2.$

answer: no

Declarative Arithmetic (Cont'd)

There are two problems with this ideal.

(1) There may be infinitely many answers

eg. query: $X + Y = 0.$
answers: $X = 0, Y = 0$
 $X = 1, Y = -1$
 $X = 2, Y = -2$
etc.

(2) The solutions may be difficult (or impossible) to compute

eg. query: $XY + XY^2 + Y^2X = 10.$
 $(XY)^2 + X^2 + Y^2 = 6.$
answers: ??

These are really problems in numerical analysis, not logic programming.

Dealing with These Problems

Prolog takes a simple, but practical approach (though somewhat procedural and non-logical).

- Require that queries have the form

X_1 is ϕ_1 , X_2 is ϕ_2 , ... X_n is ϕ_n ,

where each ϕ_i is an arithmetic expression and each X_i is a variable or a constant.

This query is interpreted to mean

$(X_1 = \phi_1) \wedge (X_2 = \phi_2) \wedge \dots \wedge (X_n = \phi_n)$.

This is processed from left to right (as usual):

First X_1 is set to the value of ϕ_1

then X_2 is set to the value of ϕ_2

...

X_n is set to the value of ϕ_n .

Note: once a variable is assigned a value, it is fixed, i.e., it cannot change.

Examples

|? - X is 5+7.

X = 12

|? - X is 5+7, Y is X-2.

X = 12

Y = 10

(* left-to-right evaluation *)

|? - Y is X-2, X is 5+7.

no

(* X is unbound here *)

|? - 7 is 4+3.

yes

|? - 8 is 4+3.

no

A variable can only be given one value. *e.g.*,

|? - X is 4, X is 5.

no

i.e., there is no value of X such that

$$X = 4 \wedge X = 5.$$

Arithmetic in Rule Bodies

```
square(X,Y) :- Y is X*X.
```

```
e.g. |? - square(5,Y).
```

```
      Y = 25
```

```
      |? - square(5,25).
```

```
      yes
```

```
      |? - square(5,13).
```

```
      no
```

```
      |? - square(X,25)
```

```
      Error: X is unbound.
```

i.e., The query `square(X,25)` becomes the subquery `25 is X*X`, in which `X` is unbound.