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 <- B        (1)
B <- C        (2)
C             (3)
```

A bottom-up proof:

infer A

rule (1)

infer B

rule (2)

infer C

rule (3)

start

So, A is proved

Top-Down Inference

A rule base:

```
A <- B        (1)
B <- C        (2)
C             (3)
```

A top-down proof:

goal A

$\downarrow$ rule (1)

goal B

$\downarrow$ rule (2)

goal C

$\downarrow$ rule (3)

success

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) <- q(X),q(Y),q(Z).
q(a1).
q(a2).
...
q(an).
```

Bottom-up inference derives n^3 facts of the form p(a$_i$,a$_j$,a$_k$):

```
p(a1, a1, a1)
p(a1, a1, a2)
p(a1, a2, a3)
...
```

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 contast, 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 <- B   (1)
B <- C   (2)
```

Failed line of inference:

$$\begin{tikzcd}
\text{goal A} \arrow[d, "\text{rule (1)}"] \\
\text{goal B} \arrow[d, "\text{rule (2)}"] \\
\text{goal C} \arrow[d, "\text{rule (3)}"] \\
\text{fail}
\end{tikzcd}$$

(no rules infer C)

So, A is not proved

Multiple Rules and Premises

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

```
E <- B
E <- C
E <- D
```

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

```
E <- B /\ C /\ D
```

In top-down inference, such rules give rise to

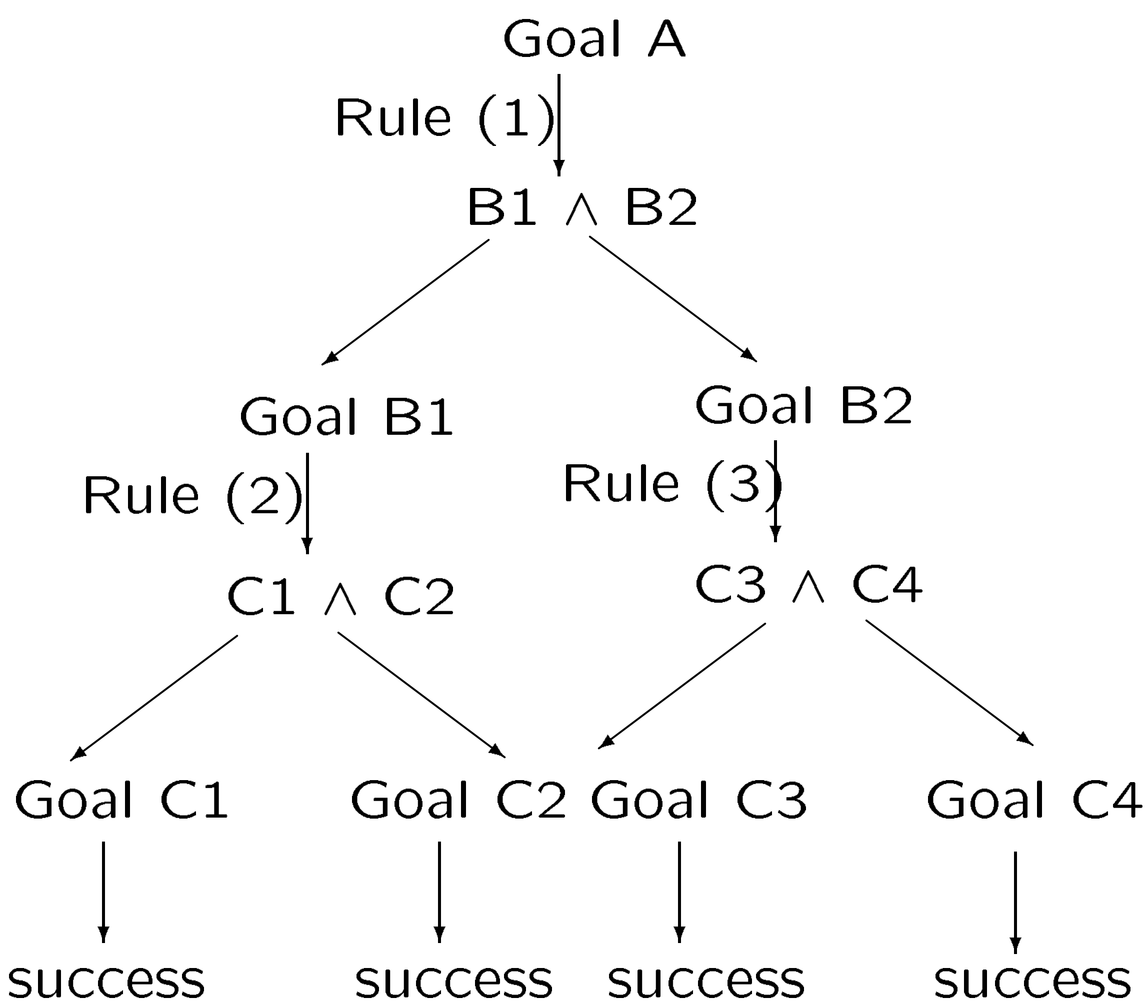
- inference trees
- backtracking

Example 1: Multiple Premises

Rule base:

```
(1)   A  <- B1 /\ B2
(2)   B1 <- C1 /\ C2
(3)   B2 <- C3 /\ C4
      C1    C2    C3    C4
```

Query: Is `A` true?



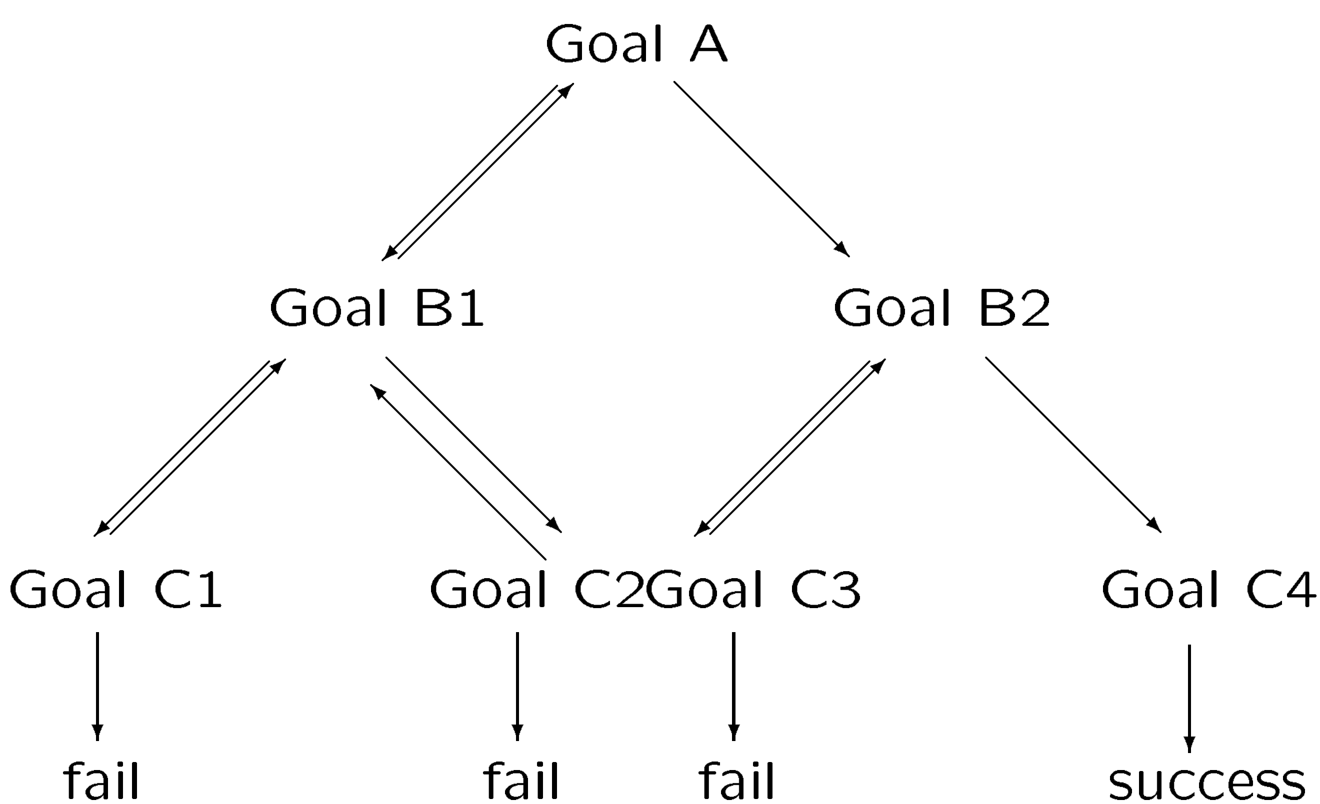
So, goal A is proved. (all paths must succeed)

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.

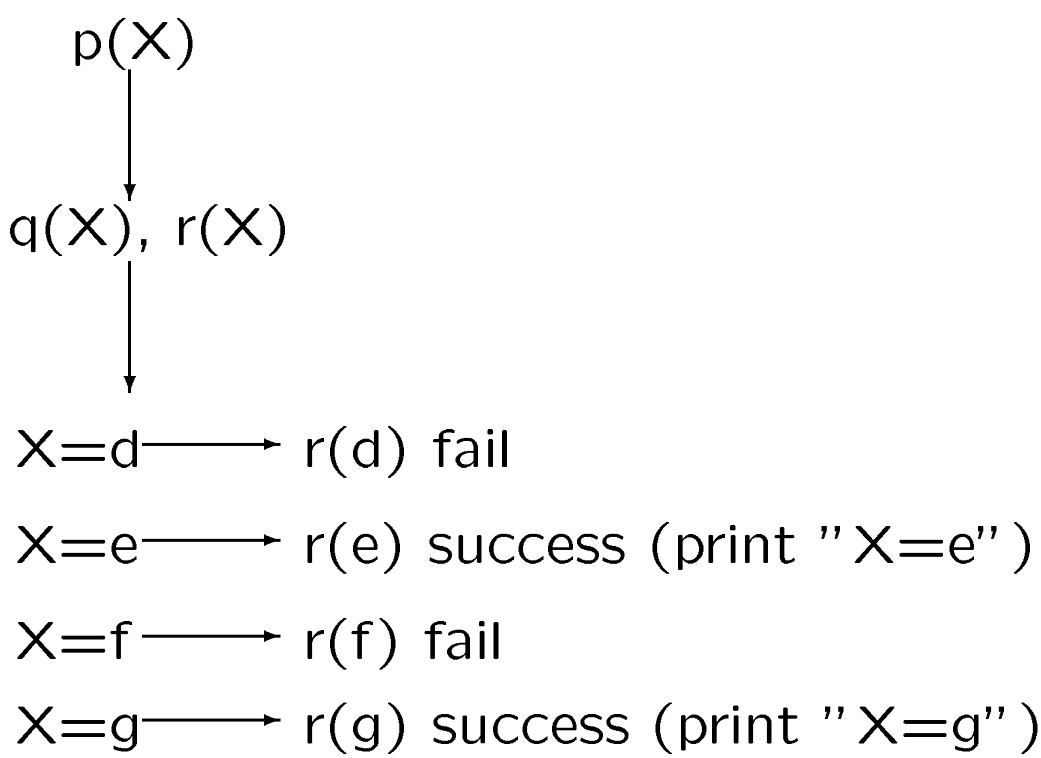
```
Answer:
  N=sam;
  N=joe;
  no
```

Example 4: Backtracking

Rule base:

```
p(X) :- 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.

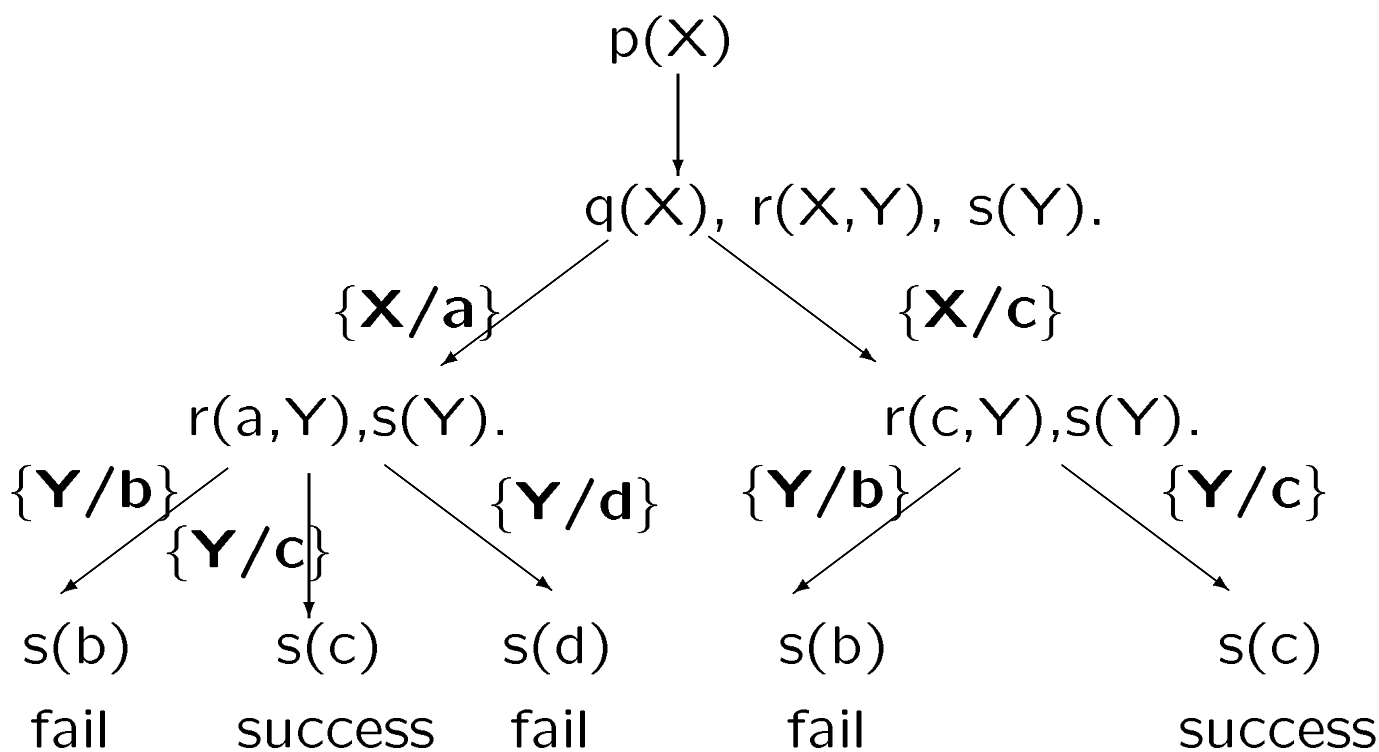


Example 5: Backtracking

Rule base:

```
p(X) :- q(X),r(X,Y),s(Y).
q(a).    r(a,b).    r(c,b).    s(c).
q(c).    r(a,c).    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 <u>both</u> 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 <u>recursive subroutine</u>.

It is called a <u>recursively defined predicate</u>, or simply a <u>recursive predicate</u>.

Mutual Recursion

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

```
a(X) :- b(X,Y), c(Y).
c(Y) :- d(Y,Z), a(Z).
```

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 <u>mutually recursive</u>.

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,

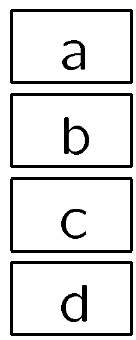
```
a(X) :- b(X,Y), a(Y), c(Y,Z), a(Z).
```

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.

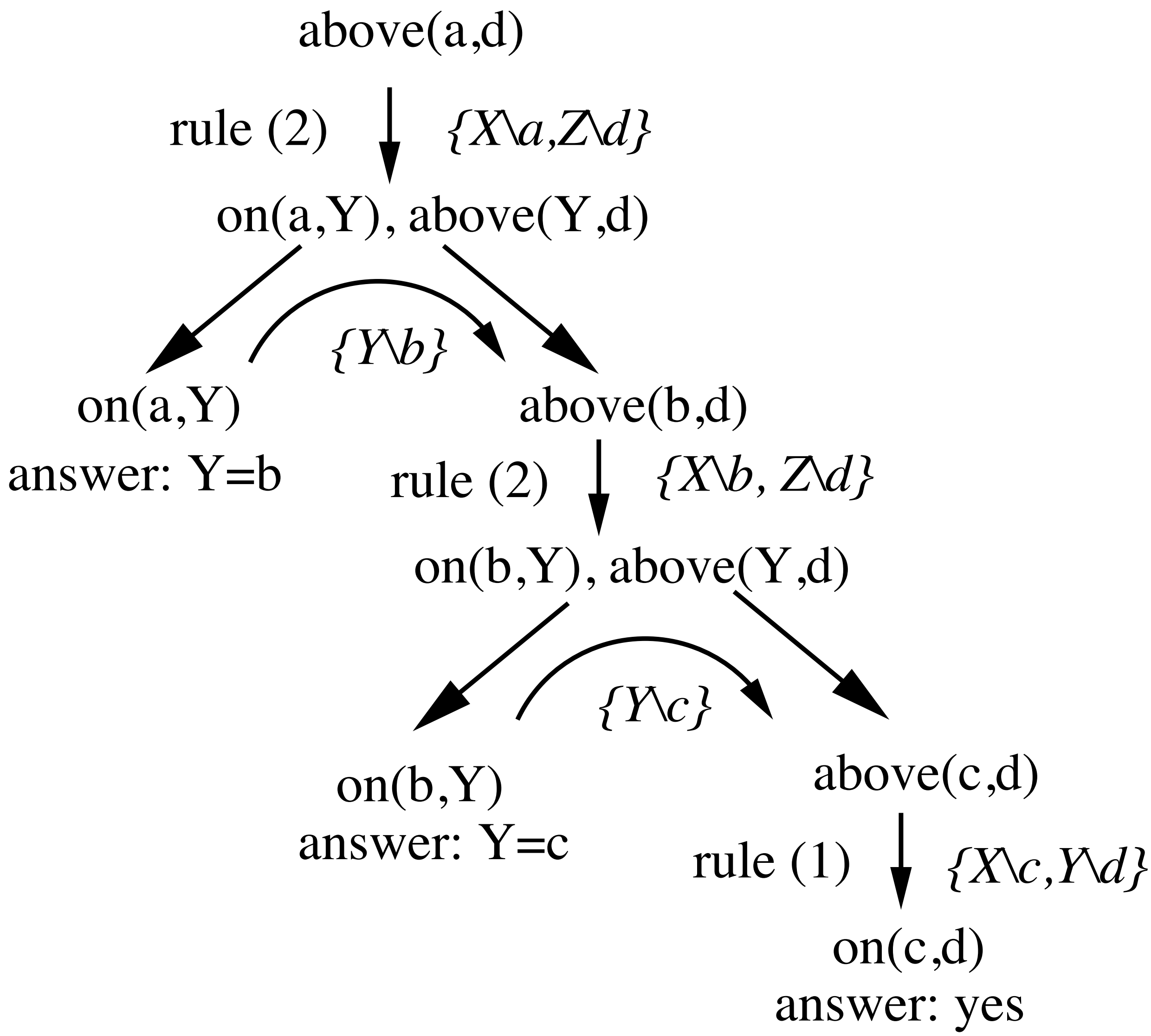


Rules:

```
(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.

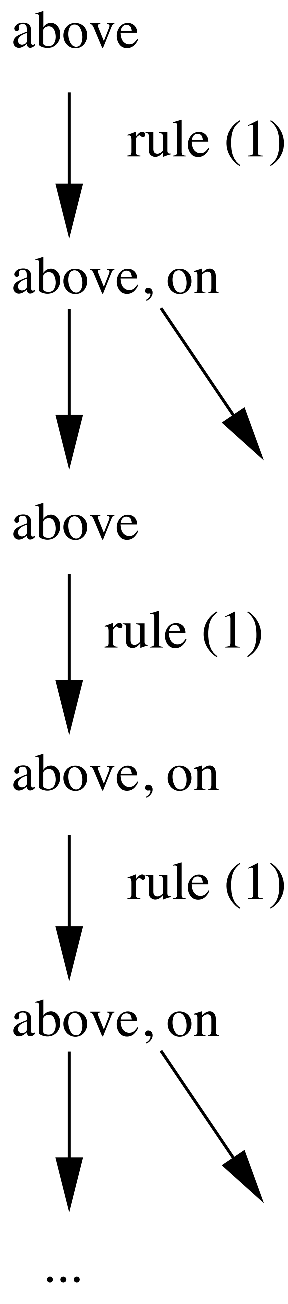
Observation

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 <u>infinite loop</u>.

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 accomodated by simple extensions to the logic-programming framework,

Negation in Prolog

- Prolog uses <u>negation as failure</u>.

- *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 tommorrow.

- 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

  $\texttt{a} \leftarrow \texttt{b} \wedge \sim\texttt{c}$

  is written in Prolog as

  ```
  a :- b, not c.
  ```

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

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 <u>everyone</u> loves him. This rule is not Horn.

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

i.e., A person is happy if <u>someone</u> 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:

$$loves(bill, mary) \Rightarrow happy(mary) \quad \{X \backslash mary, Y \backslash bill\}$$
$$loves(bill, cindy) \Rightarrow happy(cindy) \quad \{X \backslash cindy, Y \backslash bill\}$$
$$loves(tom, cindy) \Rightarrow happy(cindy) \quad \{X \backslash cindy, Y \backslash tom\}$$

So, in Horn logic, existential quantifiers can appear in the <u>premise</u> 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

eq. query: $XY + XY^2 + Y^2X = 10.$
$(XY)^2 + X^2 + Y^2 = 6.$

answers: ??

These are really problems in <u>numerical analysis</u>, 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

  $\mathtt{X}_1$ `is` ϕ_1, $\mathtt{X}_2$ `is` ϕ_2, ... $\mathtt{X}_n$ `is` ϕ_n,

where each ϕ_i is an arithmetic expression and each $\mathtt{X}_i$ is a variable or a constant.

This query is interpreted to mean

$$(\mathtt{X}_1 = \phi_1) \wedge (\mathtt{X}_2 = \phi_2) \wedge \ldots \wedge (\mathtt{X}_n = \phi_n).$$

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

First $\mathtt{X}_1$ is set to the value of ϕ_1
then $\mathtt{X}_2$ is set to the value of ϕ_2
...
$\mathtt{X}_n$ 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.