Negation in Prolog

• Prolog uses negation as failure, i.e. if something cannot be proved true then it is assumed to be false.
• This is NOT logical negation but easy to implement.
• Negation may appear only in queries and in rule bodies.

Eg. $a \leftarrow b \wedge \sim c$
is written as
`a :- b, not c.`
meaning "infer a if b can be inferred and c *cannot be inferred*.

Example

```
loves(bill,X) :- pretty(X), female(X), not
loves(tom,X).
```
(Bill loves anyone pretty unless Tom loves her.)
```
loves(tom,X) :- famous(X), female(X), not dead(X
```
(Tom loves any famous living female.)

```
female(martha-stewart).   famous(martha-stewart).
female(cindy-crawford).   famous(cindy-crawford).
female(marilyn-monroe).   famous(marilyn-monroe).
dead(marilyn-monroe).     pretty(cindy-crawford).
pretty(marilyn-monroe).   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 (*) `hates(tom,X) :- not loves(tom,X).`

Why is this NOT what we want?

(1) the answer is infinite, i.e., for any person `p` not mentioned in the database, we cannot infer `loves(tom,p)`, so we can infer `hates(tom,p)`. (*) is therefore said to be unsafe.

(2) rule (*) does not require `X` to be a person. Since we cannot infer

```
loves(tom,hammer)
loves(tom,verbs)
loves(tom,green)
loves(tom,abc)
```
so, we can 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).
```
(tom hates every pretty female whom he does not love.)

Here `female` and `pretty` are ground literals, binding `X` to some values in the database.

A Note on Quantification

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

(A person, X, is happy if <u>everyone</u> loves him. Not Horn.)

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

(A person, X, is happy if <u>someone</u>, Y, loves him.)

Equivalent to the following Horn rule:

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

Why? (Left as an exercise)

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

So, in Horn logic, existential quantification can appear in the <u>premise</u> of a rule.

Three main ideas in Prolog

- Horn rules (done)
- Unification (done)
- Top-down Inference (next)

Reasoning

• <u>Forward</u> - generate everything you can from facts and rules and use that to answer queries
- inefficient (generates much more than necessary)
- might not terminate!

• <u>Backward</u> - start from the query, attempting only those inferences that are relevant to answering the query, e.g.
Let a query A and a rule A $\leftarrow$ B be given. To prove A, Prolog tries to prove B.

Bottom-up Inference

Given a fact, B and a rule, A $\leftarrow$ B, infer A, e.g.

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

infer A
↑ rule (1)
infer B
↑ rule (2)
infer C
↑ rule (3)
start

So, A is proved

Top-Down Inference

Given a query, A, and a rule, A $\leftarrow$ B, try to prove B, e.g.

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

goal A

$\downarrow$ rule (1)

goal B

$\downarrow$ rule (2)

goal C

$\downarrow$ rule (3)

success

So, A is proved

Notes on Inference

- Prolog uses top-down inference, although some other logic programming systems use bottom-up inference.
- Each has its own advantages and disadvantages:
  - Bottom-up may generate a huge number of new facts
  - Top-down may fail
- Top-down and bottom-up are equivalent. So, the choice is an implementation detail

Top-Down vs Bottom-up

Bottom-up inference derives many more facts.

Example 1:

```
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)
...
```


Top-down vs bottom-up (Cont'd)

Bottom-up inference derives infinitely many facts.

Example 2:

```
p(f(x)) <- p(x)

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

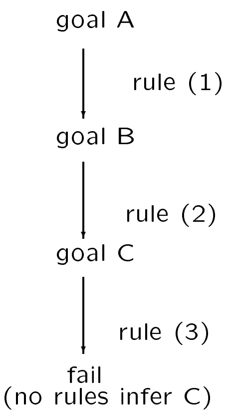
In contast, top-down inference derives only the facts requested by the user, e.g.

```
who does jane love?
what is john's telephone number?
```

Top-down vs bottom-up (Cont'd)

But... top-down inference may fail.
Example 3:

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



So, A is not proved

Dealing with Multiple Rules and Premises

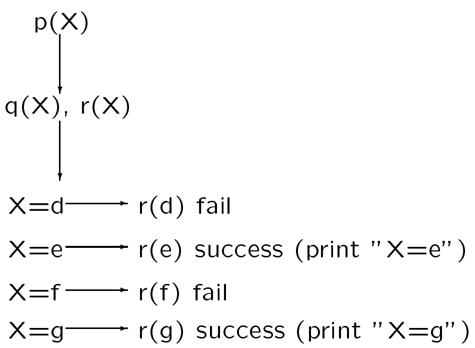
Example:

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

In top-down inference,
  - rules with multiple premises give rise to an inference <u>tree</u>
  - multiple rules that infer the same fact give rise to <u>backtracking</u>.

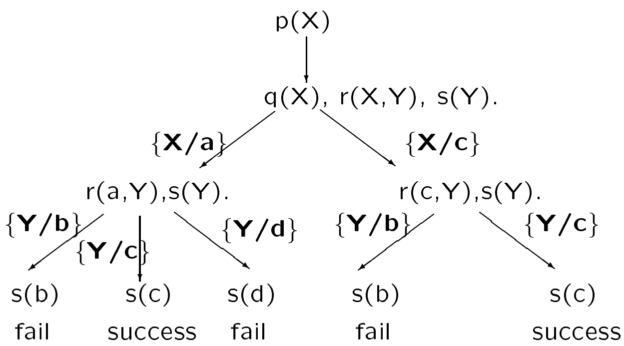
Inference Trees - Example 1

```
p(X) :- q(X), r(X).
q(d)     q(f)     r(e)
q(e)     q(g)     r(g)
p(X) % query "find X s.t. p(X) is true"
```



Inference Trees - Example 2

```
p(X) :- q(X), r(X,Y), s(Y).
q(a)     r(a,b)     r(a,d)   r(c,c)
q(c)     r(a,c)     r(c,b)   s(c).
Query p(X) %"find X s.t. p(X) is true"
```



Hints on Analyzing Output

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

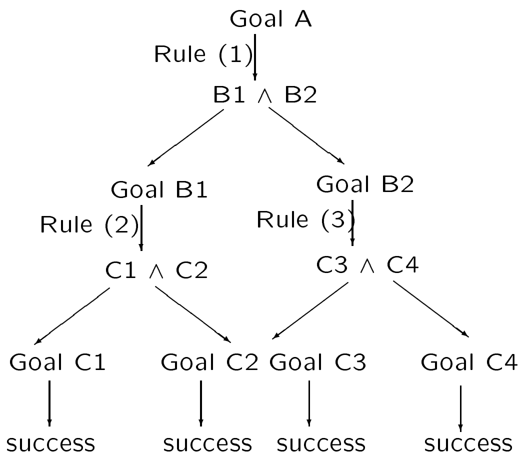
```
a(X) :- b(X), write(X), c(X).
b(a)    b(c)  b(e)
c(a)    b(d)
b(b)    c(d)
Query: a(X).
```

Then Prolog prints

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


Inference Trees - Example 3

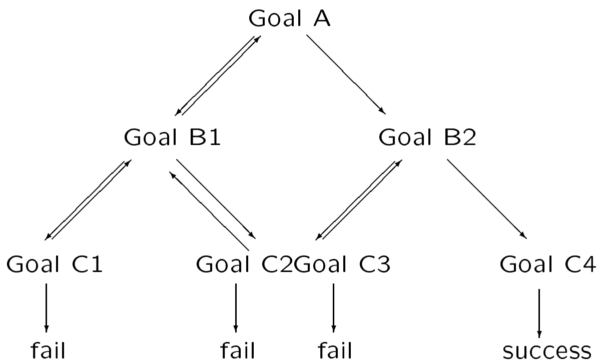
```
(1)  A  <- B1 /\ B2  (4) C1
(2)  B1 <- C1 /\ C2  (5) C2
(3)  B2 <- C3 /\ C4  (6) C3
(7)  C4           A  % query
```



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

Inference Trees - Example 4

```
(1)  C4          (5)  B1 <- C2
(2)  A  <- B1    (6)  B2 <- C3
(3)  A  <- B2    (7)  B2 <- C4
(4)  B1 <- C1
          A  % query
```



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

Recursion in Prolog

If a predicate symbol occurs both in the head and body of a rule, then the rule is *recursive*, e.g.
`a(X) :- b(X,Y),a(Y).`,
where we want to prove `a(X)`.
The predicate `a` acts like a recursive subroutine and is called recursively defined predicate.

Recursion might be indirect, involving several rules:

```
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)`.
Predicates `a` and `c` are said to be mutually recursive.

Recursion (Cont'd)

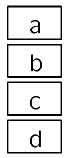
Recursion can be non-linear:

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

So, to prove `a(x)`, Prolog tries to prove both `a(Y)` and `a(Z)`.
This generates a recursive proof tree.

Example

A stack of 4 toy blocks.

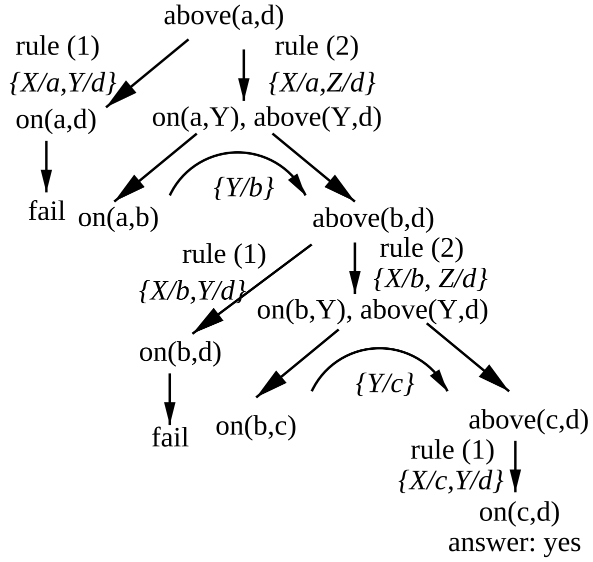


Rules:

```
(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 of one branch are true, so the root is true, i.e., above(a,d) is true.

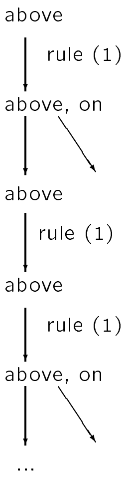
Observation

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 flow in Prolog.

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 deal.
(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:	??

There 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
  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
$(X_1 = \phi_1) \wedge (X_2 = \phi_2) \wedge \ldots \wedge (X_n = \phi_n)$.
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.
eg. `|?- X is 4, X is 5.`
   (meaning $(X = 4) \wedge (X = 5)$)
   `no`
i.e., there is no value of `X` that is <u>both</u> 4 and 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)
```

i.e., subquery is `25 is X*X`. `X` is unbound. So, error.

food for slide eater

ood for slide eater