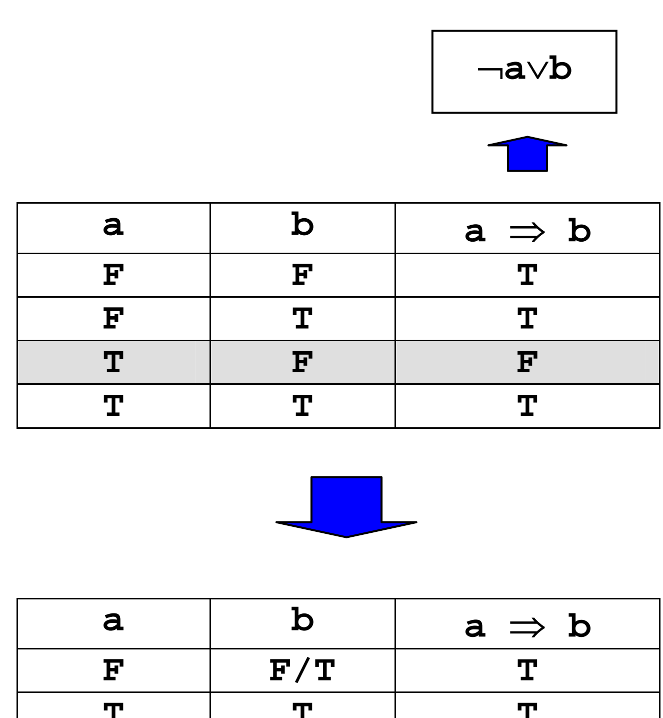
From FOL to Prolog

Implication

The meaning of the sentence **a ⇒ b** *results from the truth table of* ⇒*, considering that the sentence is true.*



a	b	a ⇒ b
F	F	T
F	T	T
T	F	F
T	T	T

a	b	a ⇒ b
F	F/T	T
T	T	T

Whenever **a** *is true* **b** *must be true. When* **a** *is false,* **b** *may be true or false. Therefore it cannot be that* **a** *is true and* **b** *is false.*

$$\neg(a \land \neg b) = \neg a \lor b$$

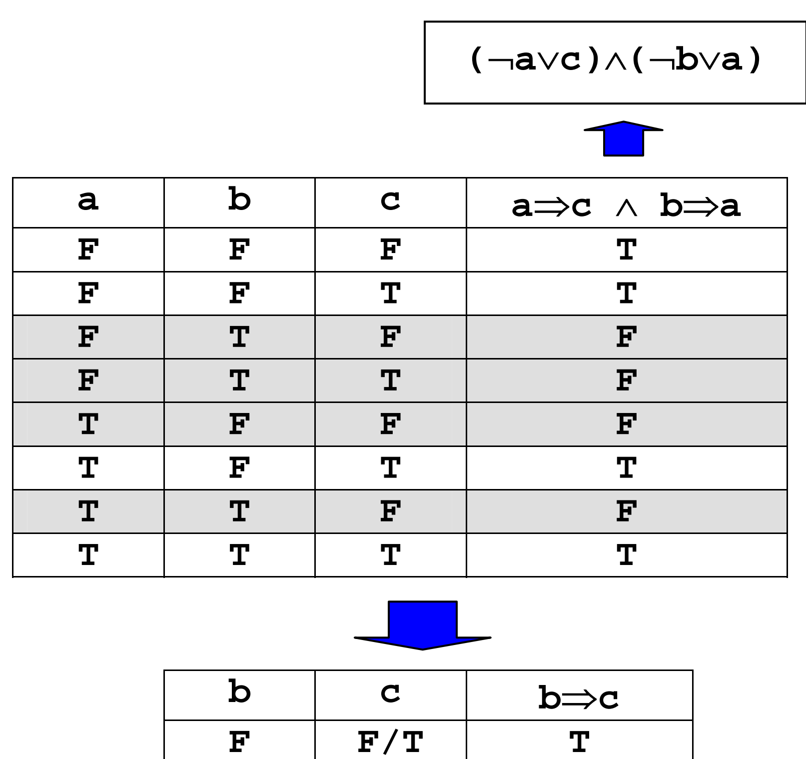
Resolution

$$\frac{a' \Rightarrow c \qquad b \Rightarrow a'' \qquad a' \; unify(\theta) \; a''}{[b \Rightarrow c]/\theta}$$

If both sentences $a' \Rightarrow c$ *and* $b \Rightarrow a''$ *are considered true, and* a' *unifies with* a'' *under substitution* θ*, then it logically follows that the sentence* $[b \Rightarrow c]/\theta$ *is true.*

Why?

For simplicity consider $a =_{def} a' =_{def} a''$
The substitution θ *is empty and* $[b \Rightarrow c]/\theta$ *is* $b \Rightarrow c$



a	b	c	a⇒c ∧ b⇒a
F	F	F	T
F	F	T	T
F	T	F	F
F	T	T	F
T	F	F	F
T	F	T	T
T	T	F	F
T	T	T	T

b	c	b⇒c
F	F/T	T
T	T	T

If we consider that both $a \Rightarrow c$ *and* $b \Rightarrow a$ *are true it follows that* $b \Rightarrow c$ *is true.*
The sentence $b \Rightarrow c$ *is logically implied by* $a \Rightarrow c$ **and** $b \Rightarrow a$

Prolog

FOL subject to (severe) restrictions + specific syntax + proof by refutation based on resolution + depth-first control of proof steps

Semantic restictions

a) *The accepted sentences are Horn clauses (Alfred Horn, 1951)*

$$a_1 \wedge a_2 \wedge \ . \ . \ . \wedge a_n \Rightarrow c$$

b) *Cyclic unification is not supported*

c) *The logical negation is not supported. It is replaced be a negation predicate based on closed world assumption.*

- **not(a)** *is true (succeeds) if the term* **a** *cannot be proved true according to the current contents of the program.*
- **not(a)** *is false (fails) if the term* **a** *can be proved true.*

Some Syntax

Sentence	*Prolog writing*	*Terminology*
$a_1 \wedge a_2 \wedge ... \wedge a_n \Rightarrow c$	c :- $a_1, a_2, ..., a_n$.	*Rule/clause*
true $\Rightarrow$ **c**	c.	*Unit rule/clause*
c $\Rightarrow$ **false**	:- c	*Goal*
The list (a b c ...)	[a,b,c,...]	
The list (cons X L)	[X | L]	
The empty list	[]	

Control

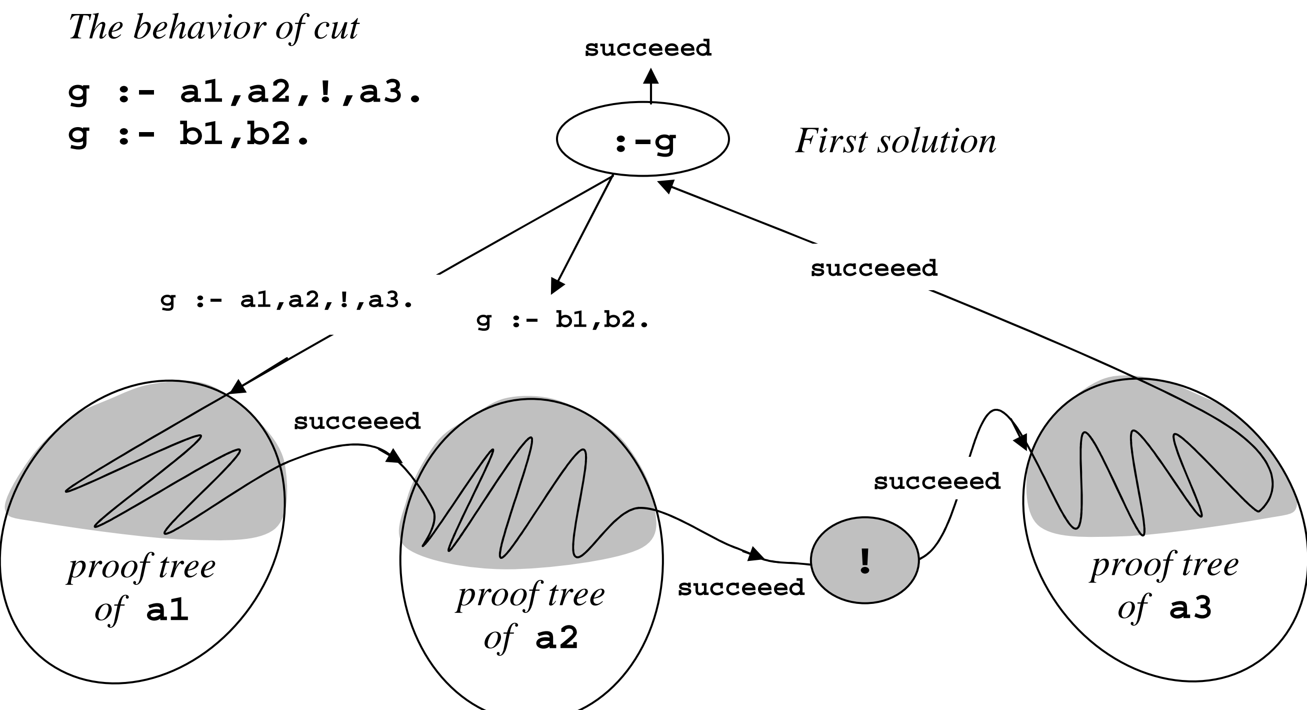
- *The order of rules is important.*
- *The order of terms in a rule is important.*

Built-in inference control

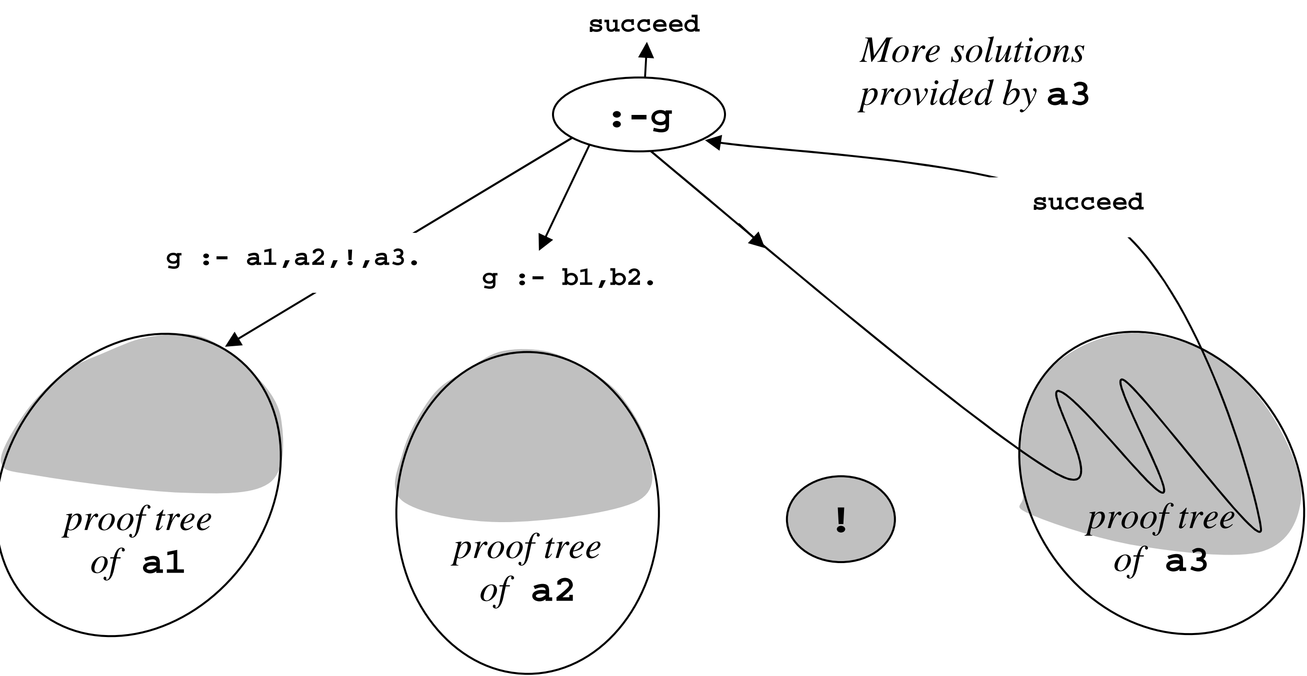
```
// Proof by refutation using depth-first,
// left-to-right traversal of the proof tree.
// Rules = the list of program rules
// Goals = the list of current goals to prove
// Subst = the current bindings of variables.
//         Initially Subst = ∅

backward-chaining(Rules,Goals,Subst) {
   if(Goals == empty) return success;

   goal = head(Goals);
   Goals = rest(Goals);

   for each rule ∈ Rules, according to the
       order of the rules in the program

      if(unify(goal,conclusion(rule),Subst,Bindings))
      {
        NewSubst = Subst ∪ Bindings;
        NewGoals = append(antecedents(rule),Goals);
        // The antecedents are proved depth-first,
           following their order in the rule

        if(backward-chaining(Rules,NewGoals,NewSubst))
          return success;
      }

   return failure;
}
```

Language level control mechanisms

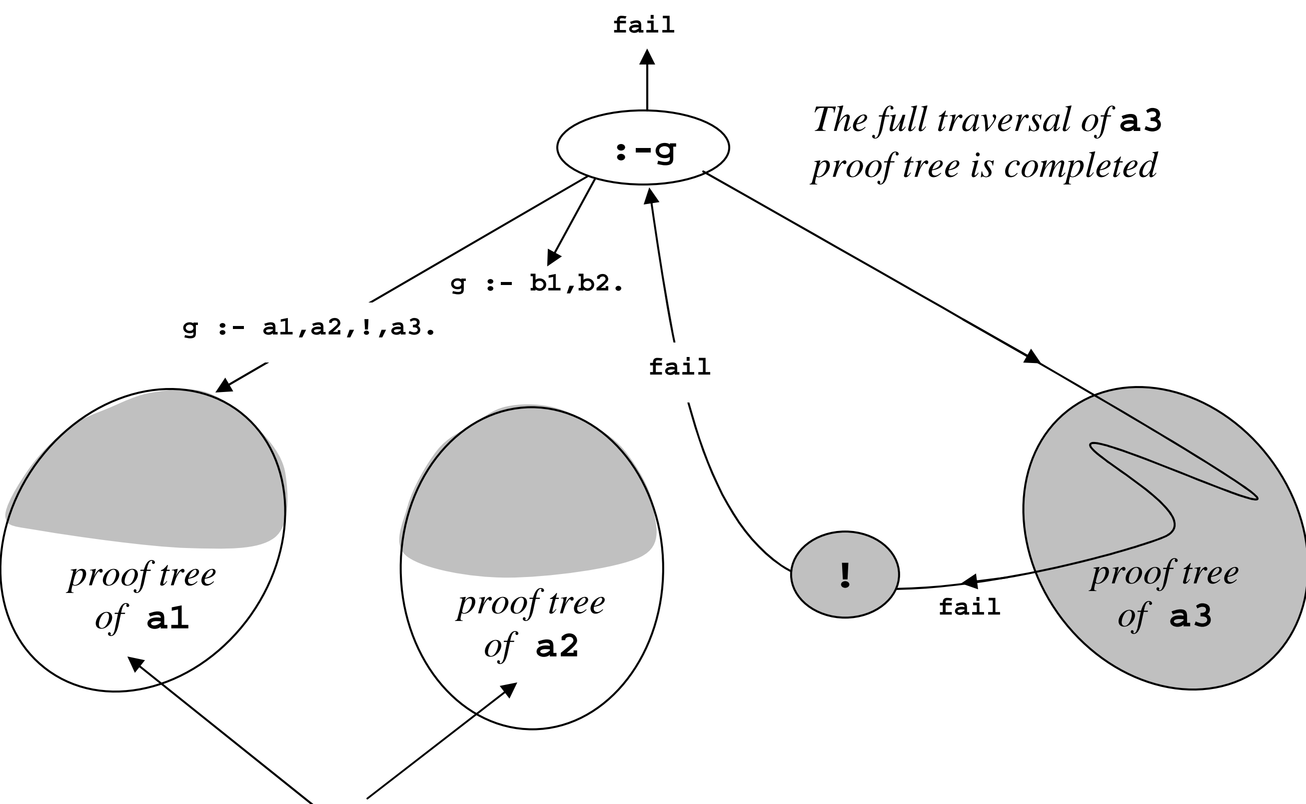
- **`true`** – *always succeeds*
- **`fail`** – *always fails*
- **`!`** – *the cut symbol*
    - *When first processed, cut succeeds*
    - *On backtrack the cut fails and the currently proved goal fails completely (the remaining clauses whose head unifies the goal are ignored)*



The proof trees **a1** and **a2** are partially explored and succeed. **When first processed the cut term ! succeeds.** The proof tree **a3** is partially explored and succeeds. The first solution of proving the goal **g** is computed.



Resuming the traversal of **a3**. The traversal of the proof tree of **a3** succeeds. New solutions of proving **g** are obtained.



Discarded regions of the proof trees

Resuming the traversal of the proof three of **`a3`**. *The traversal fails: the* **`a3`** *proof tree is fully explored and has no more solutions to offer. The backtrack process returns to the cut symbol.* ***When subsequently processed, the cut fails.*** *The remainder of the proof trees of* **`a1`** *and* **`a2`** *is not explored. The clause* **`g:-b1,b2`** *is ignored and the goal* **`g`** *fails completely.*

Using the cut symbol for defining the **`not`** *predicate:*

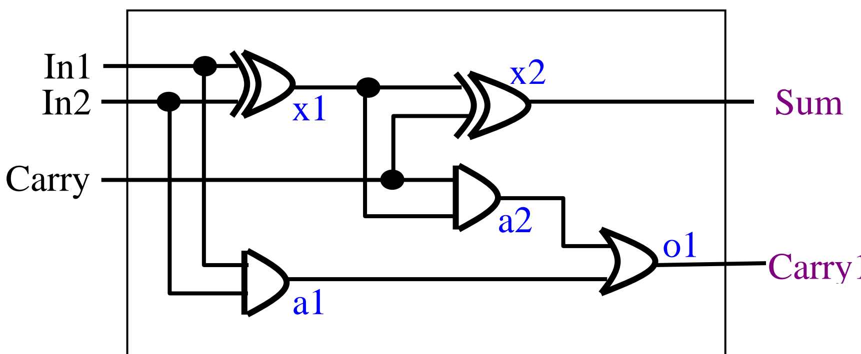
```
not(P) :- P, !, fail.
not(P) :- true.
```

If **`P`** *succeeds, then* **`not(P)`** *fails completely.*
If **`P`** *cannot be proved,* **`not(P)`** *suceeds.*

`not` ***is not the logical negation.*** *Truth is assumed as a consequence of absent evidence.*

Simple Prolog

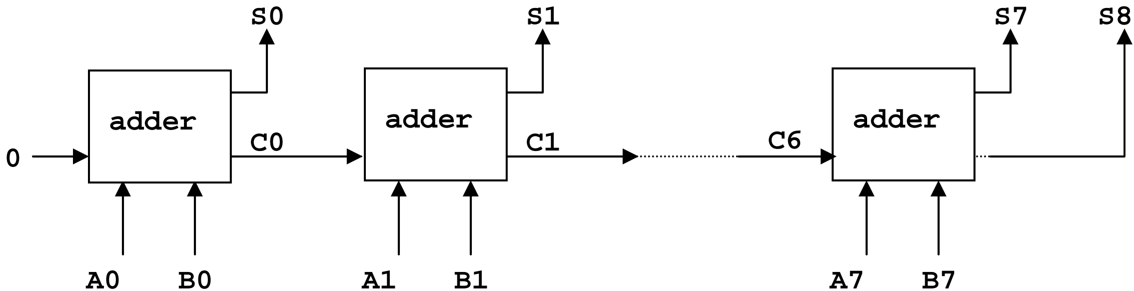
type(x_1)=xor type(x_2)=xor
type(a_1)=and type(a_2)=and
type(o_1)=or




```
:-op(600,fx,[in,out]).

adder(in[In1,In2,Carry],out[Sum,Carry1]) :-
   gate(xor,in[In1,In2],out[X1]),
   gate(and,in[In1,In2],out[A1]),
   gate(xor,in[X1,Carry],out[Sum]),
   gate(and,in[Carry,X1],out[A2]),
   gate(or,in[A1,A2],out[Carry1]).

gate(Type,in In, out Out) :-   % gate(Type,in[X,Y],out[Z])
   append(In,Out,L),           % synthesizes and solves the goal
   Term =..[Type | L],         % Type(X,Y,Z)
   call(Term).

and(0,0,0).
and(0,1,0).
and(1,0,0).
and(1,1,1).

xor(0,0,0).
xor(0,1,1).
xor(1,0,1).
xor(1,1,0).

or(0,0,0).
or(0,1,1).
or(1,0,1).
or(1,1,1).
```

Don't worry. These are unworthy tricks. Everything can be programmed in Basic

The holiday is too near to pay attention to such obscure tricks.




```
sum(in[A7,A6,A5,A4,A3,A2,A1,A0],
    in[B7,B6,B5,B4,B3,B2,B1,B0],
    out[S8,S7,S6,S5,S4,S3,S2,S1,S0])
:-
  adder(in[A0,B0, 0],out[S0,C0]),
  adder(in[A1,B1,C0],out[S1,C1]),
  adder(in[A2,B2,C1],out[S2,C2]),
  adder(in[A3,B3,C2],out[S3,C3]),
  adder(in[A4,B4,C3],out[S4,C4]),
  adder(in[A5,B5,C4],out[S5,C5]),
  adder(in[A6,B6,C5],out[S6,C6]),
  adder(in[A7,B7,C6],out[S7,S8]).
```