# Prolog & Logic

- Why is Prolog a "Logic Programming Language?"
- Prolog is a *theorem prover*: Prolog will tell us if a certain theorem follows from a given set of facts.
- Our facts (or *hypothesis*) are given in a simplified form of predicate logic.

| Pred. Log. | Prolog |
|---|---|
| $\forall X \bullet P(X)$ | `all(X, p(X) => q(X))` |
| $\exists X \bullet P(X)$ | `exists(X, p(X) & q(X))` |
| $A \Rightarrow B$ | `a => b` |
| $A \iff B$ | `a <==> b` |
| $A \vee B$ | `a \| b` |
| $A \wedge B$ | `a & b` |
| $\neg A$ | `~ a` |

# Removing Implications

| $P$ | $Q$ | $P \Rightarrow Q$ | $\neg P \vee Q$ |
|---|---|---|---|
| T | T | T | T |
| T | F | F | F |
| F | T | T | T |
| F | F | T | T |

| $P$ | $Q$ | $P \iff Q$ | $(P \wedge Q) \vee (\neg P \wedge \neg Q)$ |
|---|---|---|---|
| T | T | T | T |
| T | F | F | F |
| F | T | F | F |
| F | F | T | T |

```
all(X, (bird(X) & ∼kiwi(X)) => flies(X))
```

$\Downarrow$

```
all(X, ∼(bird(X) & ∼kiwi(X)) | flies(X))
```

```
all(X, duck(X) <==> web_feet(X))
```

$\Downarrow$

```
all(X, (duck(X) & web_feet(X)) |
        (∼duck(X) & ∼web_feet(X))
```

# Moving Negation Inwards

| $P$ | $Q$ | $\neg(P \wedge Q)$ | $\neg P \vee \neg Q$ |
|---|---|---|---|
| T | T | F | F |
| T | F | T | T |
| F | T | T | T |
| F | F | T | T |

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$

$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$$

$$\neg(\neg P) \equiv P$$

$$\neg \forall X \bullet P(X) \equiv \exists X \bullet \neg P(X)$$

$$\neg \exists X \bullet P(X) \equiv \forall X \bullet \neg P(X)$$

```
~ all(X, ~(bird(X) & ~kiwi(X)) | flies(X))
                    ⇓
~ all(X, (~bird(X) | kiwi(X)) | flies(X))
                    ⇓
exists(X, ~((~bird(X) | kiwi(X)) | flies(X)))
                    ⇓
exists(X, ~(~bird(X) | kiwi(X)) & ~flies(X))
                    ⇓
exists(X, bird(X) & ~kiwi(X) & ~flies(X))
```

# Removing ∃ (Skolemizing)

- We remove all existential quantifiers by replacing variables with names (*Skolem constant*).
- Universally quantified variables are replaced with *Skolem functions*.

```
exists(X,bird(X) & ~kiwi(X) & ~flies(X))
                    ⇓
bird(g1) & ~kiwi(g1) & ~flies(g1)

all(X, bird(X) => exists(Y,
         is_egg(Y) & comes_from(X, Y)))
                  ⇓ (This is wrong!)
all(X, bird(X) => (is_egg(g2) &
         comes_from(X, g2)))
                  ⇓ (This is correct!)
all(X, bird(X) => (is_egg(g2(X)) &
         comes_from(X, g2(X))))
```

# Removing ∀

- We move all universal quantifiers to the beginning of the formula.
- We next remove all universal quantifiers, remembering that all variables are, in fact, introduced by ∀.

```
all(X, bird(X) =>
       all(Y, worm(Y) => likes(X, Y)))
```

⇓

```
all(X, ~bird(X) |
       all(Y, ~worm(Y) | likes(X, Y)))
```

⇓

```
all(X, all(Y, ~bird(X) |
       ~worm(Y) | likes(X, Y)))
```

⇓

```
~bird(X) | ~worm(Y) | likes(X, Y)
```

## Distributing & over |

- We transform our formulae to *conjunctive normal form*:

$$(a \mid b \mid c)\&(c \mid d \mid f)\&(f \mid r \mid t)$$

- We use the following identities:

$$(A\&B) \mid C \equiv (A \mid C)\&(B \mid C)$$
$$A \mid (B\&C) \equiv (A \mid B)\&(A \mid C)$$

```
(bird(X) & flies(X)) | kiwi(X)
```

$$\Downarrow$$

```
(bird(X) | kiwi(X)) &
        (flies(X) | kiwi(X))
```

## Putting Into Clauses

- Our formulae now look like this:

$$A\&B\&(C\&(D\&E))\&F$$

- Each variable represents a *literal* such as:

$$x \mid y \mid (z \mid v)$$

- We can remove brackets to get:

$$(a \mid b \mid c)\&(c \mid d \mid f)\&(f \mid r \mid t)$$

- This is really just a collection (set) of literals:

$$\{(a \mid b \mid c), (c \mid d \mid f), (f \mid r \mid t)\}$$

- The formulae are now in *clausal form.*

# A Notation for Clauses

- Each literal looks like this:

$$\sim k \mid a \mid\sim l \mid b \mid c \mid\sim m$$

- Write unnegated literals first, negated ones second:

$$a \mid b \mid c \mid\sim k \mid\sim l \mid\sim m$$

- Extract negation (de Morgan's Law):

$$(a \mid b \mid c) \mid\sim (k \& l \& m)$$

- Use implication ($P \Rightarrow Q \equiv \neg P \vee Q$):

$$(k \& l \& m) \Rightarrow (a \mid b \mid c)$$

- Rewrite to use :- ($P \Rightarrow Q \equiv Q :- P$):

$$(a \mid b \mid c) :- (k \& l \& m)$$

# Predicate Logic $\Longrightarrow$ Clauses I

```
?- translate(all(x, p(x) => q(x))).
   q(x) :- p(x).
?- translate(exists(x, p(x) & q(x))).
   p(f1).
   q(f1).
?- translate(all(x,
         (bird(x) & ~kiwi(x)) => flies(x))).
   kiwi(x); flies(x) :- bird(x).
?- translate(exists(x,
         bird(x) & ~kiwi(x) & ~flies(x))).
   bird(f2).
   :- kiwi(f2).
   :- flies(f2).
?- translate(all(x, bird(x) =>
      exists(y, egg(y) & comes_from(x, y)))).
   egg(f3(x)) :- bird(x).
   comes_from(x,f3(x)) :- bird(x).
?- translate(~exists(x, bird(x) &
      ~exists(y, egg(y) & comes_from(x, y)))).
   egg(f4(x)) :- bird(x).
   comes_from(x,f4(x)) :- bird(x).
```

## Predicate Logic $\Longrightarrow$ Clauses II

```
:- op(30, fx, ~).
:- op(100, xfy, #).
:- op(100, xfy, &).
:- op(150, xfy, =>).
:- op(150, xfy, <==>).

translate(X) :-
    implout(X, X1),
    negin(X1, X2),
    skolem(X2, X3, []),
    univout(X3, X4),
    conjn(X4, X5),
    clausify(X5, Clauses, []),
    pclauses(Clauses).
```

# Predicate Logic $\Longrightarrow$ Clauses III

```
implout((P <==> Q),
      ((P1 & Q1) # (~P1 & ~Q1))) :-
   !, implout(P, P1), implout(Q, Q1).
implout((P => Q), (~P1 # Q1)) :-
   !, implout(P, P1), implout(Q, Q1).
implout(all(X,P), all(X, P1)) :- !,
   implout(P, P1).
implout(exists(X,P), exists(X, P1)):-
   !, implout(P, P1).
implout((P & Q), (P1 & Q1)) :- !,
   implout(P, P1), implout(Q, Q1).
implout((P # Q), (P1 # Q1)) :- !,
   implout(P, P1), implout(Q, Q1).
implout((~P), (~P1)) :- !,
   implout(P, P1).
implout(P, P).
```

# Predicate Logic $\Longrightarrow$ Clauses IV

```
negin((∼P), P1) :- neg(P, P1).
negin(all(X, P), all(X, P1)) :-
   !, negin(P, P1).
negin(exists(X, P), exists(X, P1)) :-
   !, negin(P, P1).
negin((P & Q), (P1 & Q1)) :- !,
   negin(P, P1), negin(Q, Q1).
negin((P # Q), (P1 # Q1)) :- !,
   negin(P, P1), negin(Q, Q1).
negin(P, P).

neg((∼P), P1) :- negin(P, P1).
neg(all(X, P), exists(X, P1)) :-
   !, neg(P, P1).
neg(exists(X, P), all(X, P1)) :-
   !, neg(P, P1).
neg((P & Q), (P1 # Q1)) :- !,
   neg(P, P1), neg(Q, Q1).
neg((P # Q), (P1 & Q1)) :- !,
   neg(P, P1), neg(Q, Q1).
neg(P, (∼P)).
```

# Predicate Logic $\Longrightarrow$ Clauses V

```
skolem(all(X, P), all(X, P1), Vars) :-
   !, skolem(P, P1, [X|Vars]).
skolem(exists(X, P), P2, Vars) :-
   !, mgensym(f, F),
   Sk =..  [F|Vars],
% subst(X, Sk, P, P1),
   subst(Sk, X, P, P1),
   skolem(P1, P2, Vars).
skolem((P # Q) , (P1 # Q1), Vars) :-
   !, skolem(P, P1, Vars),
   skolem(Q, Q1, Vars).
skolem((P & Q) , (P1 & Q1), Vars) :-
   !, skolem(P, P1, Vars),
   skolem(Q, Q1, Vars).
skolem(P, P, _).
```

# Predicate Logic $\Longrightarrow$ Clauses VI

```
univout(all(X, P), P1) :-
   !, univout(P, P1).
univout((P & Q), (P1 & Q1)) :-
   !, univout(P, P1), univout(Q, Q1).
univout((P # Q), (P1 # Q1)) :-
   !, univout(P, P1), univout(Q, Q1).
univout(P, P).

conjn((P # Q), R) :- !,
   conjn(P, P1), conjn(Q, Q1),
   conjnl((P1 # Q1), R).
conjn((P & Q), (P1 & Q1)) :- !,
   conjn(P, P1), conjn(Q, Q1).
conjn(P, P).

conjnl(((P & Q) # R), (P1 & Q1)) :- !,
   conjn((P # Q), P1), conjn((Q # R), Q1).
conjnl((P # (Q & R)), (P1 & Q1)) :- !,
   conjn((P # Q), P1), conjn((P # R), Q1).
conjnl(P, P).
```

# Predicate Logic $\Longrightarrow$ Clauses VII

```
clausify((P & Q), C1, C2) :- !,
   clausify(P, C1, C3),
   clausify(Q, C3, C2).
clausify(P, [cl(A, B)|Cs], Cs) :-
   inclause(P, A, [], B, []), !.

inclause((P # Q), A, A1, B, B1) :- !,
   inclause(P, A2, A1, B2, B1),
   inclause(Q, A, A2, B, B2).
inclause((~P), A, A, B1, B) :- !,
   notin(P, A), putin(P, B, B1).
inclause(P, A1, A, B, B) :-
   notin(P, B), putin(P, A, A1).

notin(X, [X|_]) :- !, fail.
notin(X, [_|L]) :- !, notin(X, L).
notin(X, []).

putin(X, [], [X]) :- !.
putin(X, [X|L], L) :- !.
putin(X, [Y|L], [Y|L1]) :- putin(X, L, L1).
```

# Predicate Logic $\Longrightarrow$ Clauses VIII

```
pclauses([]) :- !, nl, nl.
pclauses([cl(A, B)|Cs]) :-
   pclause(A, B), nl, pclauses(Cs).
pclause(L, []) :- !, pdisj(L), write('.').
pclause([], L) :- !, write(' :- '),
   pconj(L), write('.').
pclause(L1, L2) :-
   pdisj(L1), write(' :- '), pconj(L2),
   write('.').

pdisj([L]) :- !, write(L).
pdisj([L|Ls]) :-
   write(L), write('; '), pdisj(Ls).

pconj([L]) :- !, write(L).
pconj([L|Ls]) :-
   write(L), write(', '), pconj(Ls).
```

# Resolution I

- 1965 J. Alan Robinson discovered *the resolution principle*.
- Resolution is a *rule of inference*: what conclusions follow from our premises?

  If the same formula is to the left in one clause and to the right in another clause, then we can merge the two clauses, omitting the identical term.

| | | |
|---|---|---|
| A; B; C | :- | D, $\boxed{\text{E}}$, F |
| G; $\boxed{\text{E}}$ | :- | H, I |
| A; B; C; G | :- | D, F, H, I |

# Resolution II

1. If Clark has wings, then Clark is a Kiwi or Clark flies.
2. If Clark flies and has a cape, then Clark is Superman.
3. Therefore, if Clark has wings and a cape, then he is either a Kiwi or Superman.

| | | |
|---|---|---|
| kiwi(clark); flies(clark) | :- | has_wings(clark) |
| superman(clark) | :- | flies(clark), has_cape(clark) |
| kiwi(clark); superman(clark) | :- | has_wings(clark), has_cape(clark) |

# Resolution III – Unification

- Actually, the formulae don't have to be identical, they just have to *match* (unify).

(1) person(f1(X)); king(X) :-
(2) king(Y) :- respects(f1(Y), Y)
(3) respects(Z, john) :- person(Z)

(4) ← (2), (3): king(john) :- person(f1(john)).
(5) ← (1), (4): king(john); king(john) :- .

# Proving with Resolution I

- How can we prove theorems using Resolution?
- Resolution can always detect when a set of clauses is *inconsistent*.
- If a set of clauses is inconsistent Resolution will derive the *empty* clause (:-).

To prove that formula $B$ follows from formulae

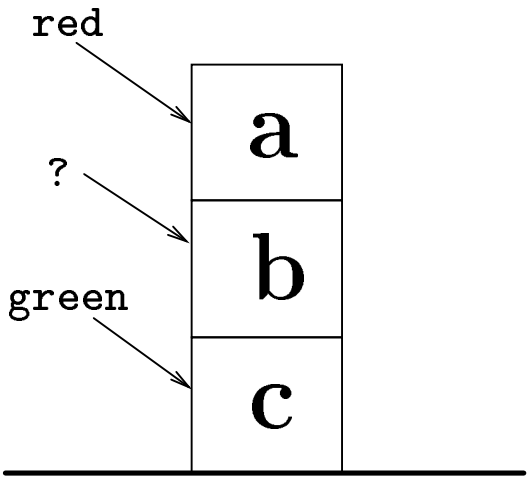
$$\{A_1, A_2, \cdots, A_n\}$$

we show that

$$\{A_1, A_2, \cdots, A_n, \sim B\}$$

is inconsistent.

# Resolution Example I




- Block **a** is on top of block **b** (1).
- Block **b** is on top of block **c** (2).
- Block **a** is red (3), **c** is green (4).
- Blocks are either green or red (5).
- Does some red block rest directly on top of a green block (6)?

```
?- translate(on(a,b) & on(b,c) & red(a) &
     green(c) & all(X, red(X) # green(X))).
   on(a,b).  on(b,c).
   red(a).  green(c).
   red(_9124); green(_9124).
```

# Resolution Example II

```
(1)   on(a, b) :- .
(2)   on(b, c) :- .
(3)   red(a) :- .
(4)   green(c) :- .
(5)   red(X), green(X) :- .
(6)   :- on(X, Y), red(X), green(Y)
```

(7) $\leftarrow$ (1), (6): :- red(a), green(b).
(8) $\leftarrow$ (7), (3): :- green(b).
(9) $\leftarrow$ (8), (5): red(b) :- .
(10) $\leftarrow$ (9), (6): :- on(b, Y), green(Y).
(11) $\leftarrow$ (10), (2): :- green(c).
(12) $\leftarrow$ (11), (4): :- .

# Prolog & Resolution I

- Prolog allows clauses (*Horn Clauses*) with *at most* one unnegated term (to the left of `:-`).

Our premises:

```
(1)   a :- b, c.
(2)   a :- d, c.
(3)   c :-.
(4)   d :-.
(5)   e :-.
```

Our (negated) hypothesis:

```
(6)   :- a, e.
```

Resolution:

(7) ← (1), (6): :- b, c, e. (backtrack!)
(8) ← (2), (6): :- d, c, e.
(9) ← (8), (4): :- c, e.
(10) ← (9), (3): :- e.
(11) ← (10), (5): :- . (contradiction)

# Prolog & Resolution II

## General Resolution:

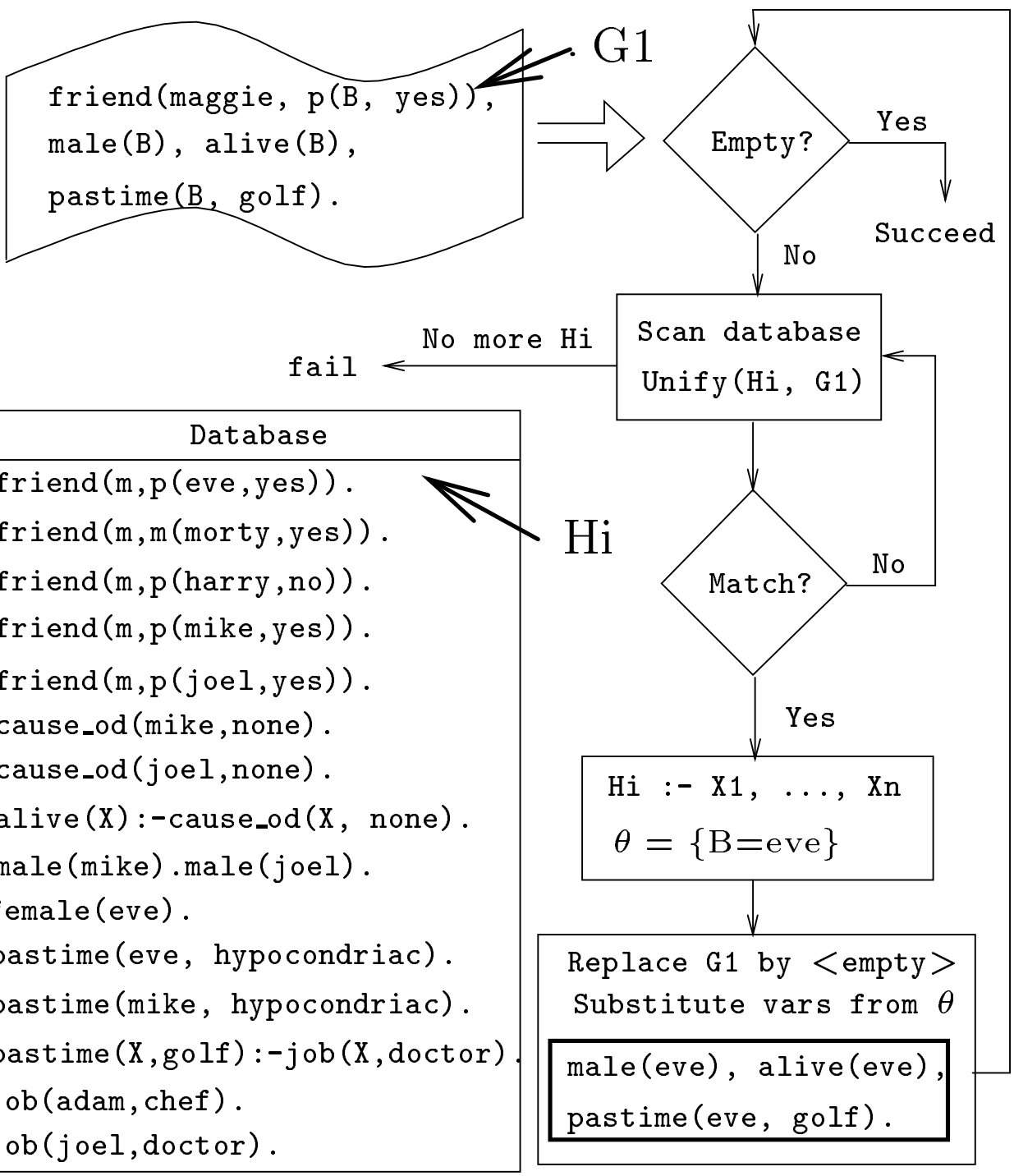
1. $A; B; C; \cdots : -K, L, M, \cdots$
2. Many *headless* clauses: $: -K, L, \cdots$
3. Algorithm: ($C$ is set of clauses)
   (a) Select any two clauses $A$ & $B$.
   (b) $N \leftarrow$ `Resolve`$(A, B)$. Add $N$ to $C$.
   (c) Repeat, unless derived empty clause.

## Prolog:

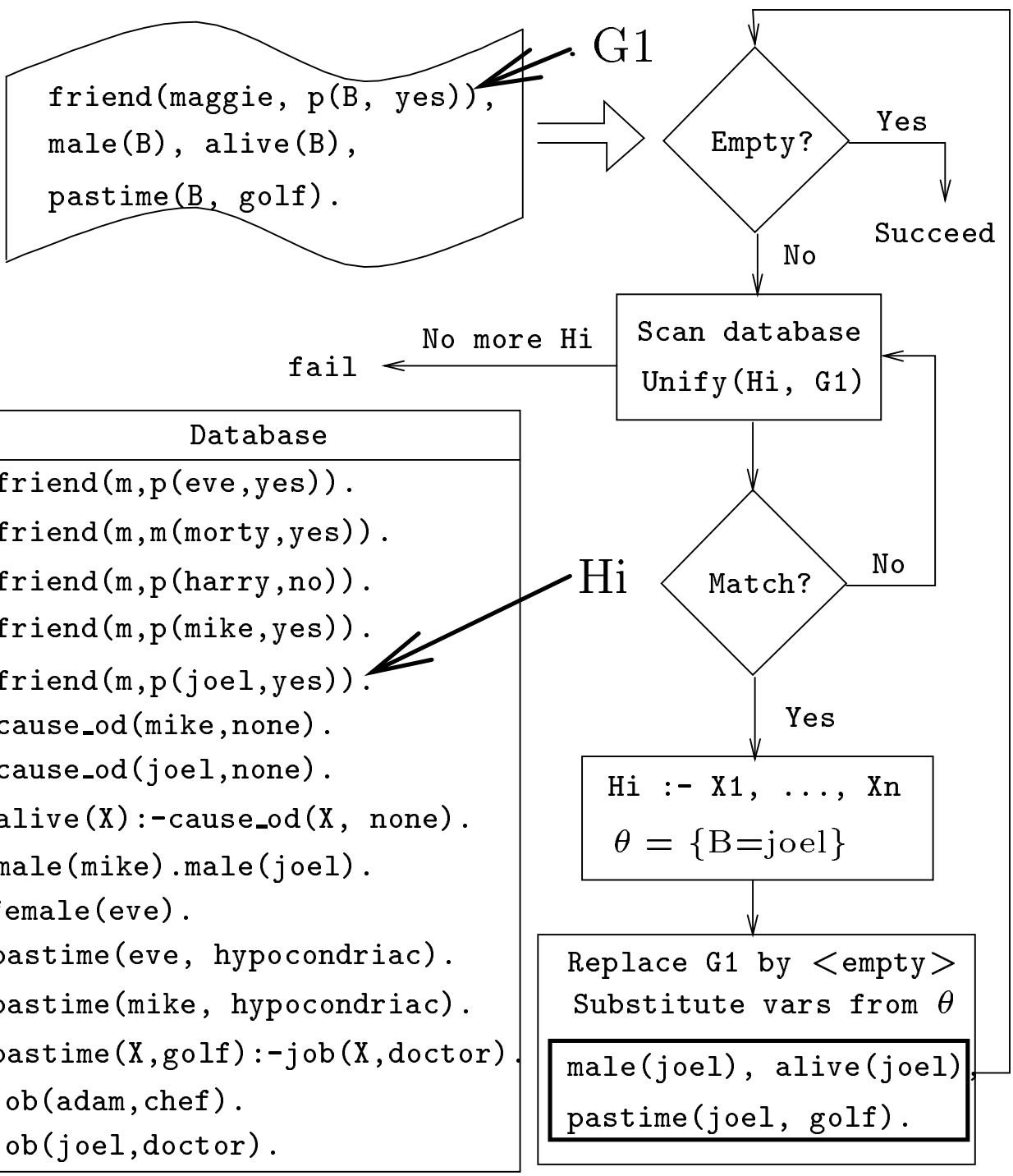
1. Horn clauses: $A$: $-K, L, M, \cdots$
2. One *headless* clause: $?-K, L, \cdots$
3. Algorithm: ($G = A_1, \cdots A_n$ is the goal.)
   (a) Select $B$, first clause that matches $A_1$
   (b) $N \leftarrow$ `Resolve`$(G, B); G \leftarrow N$.
   (c) Repeat, unless $G$ is empty.

   Prolog searches *depth-first*. Cheap, but can lead to looping.

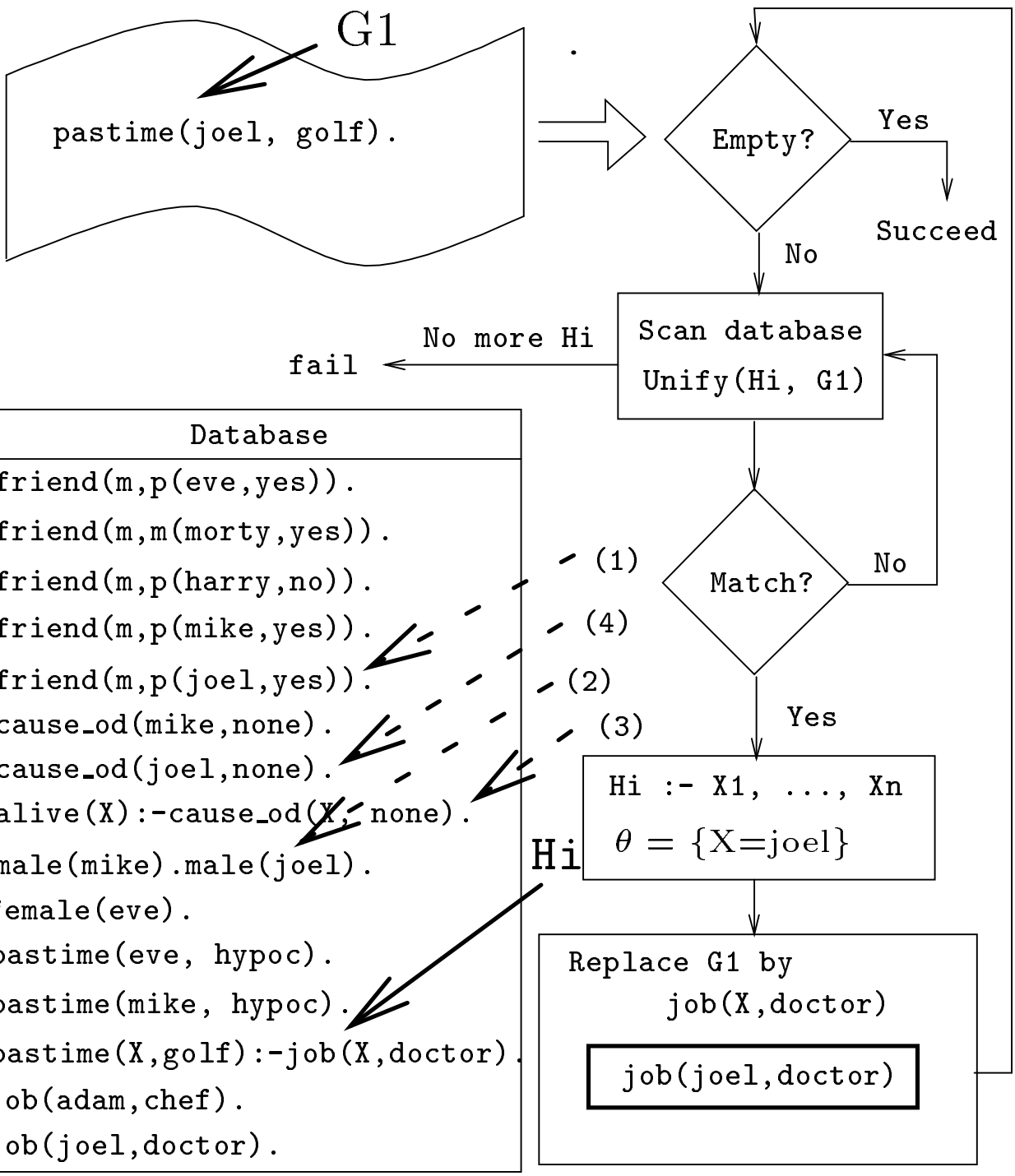
# Prolog Execution Example I



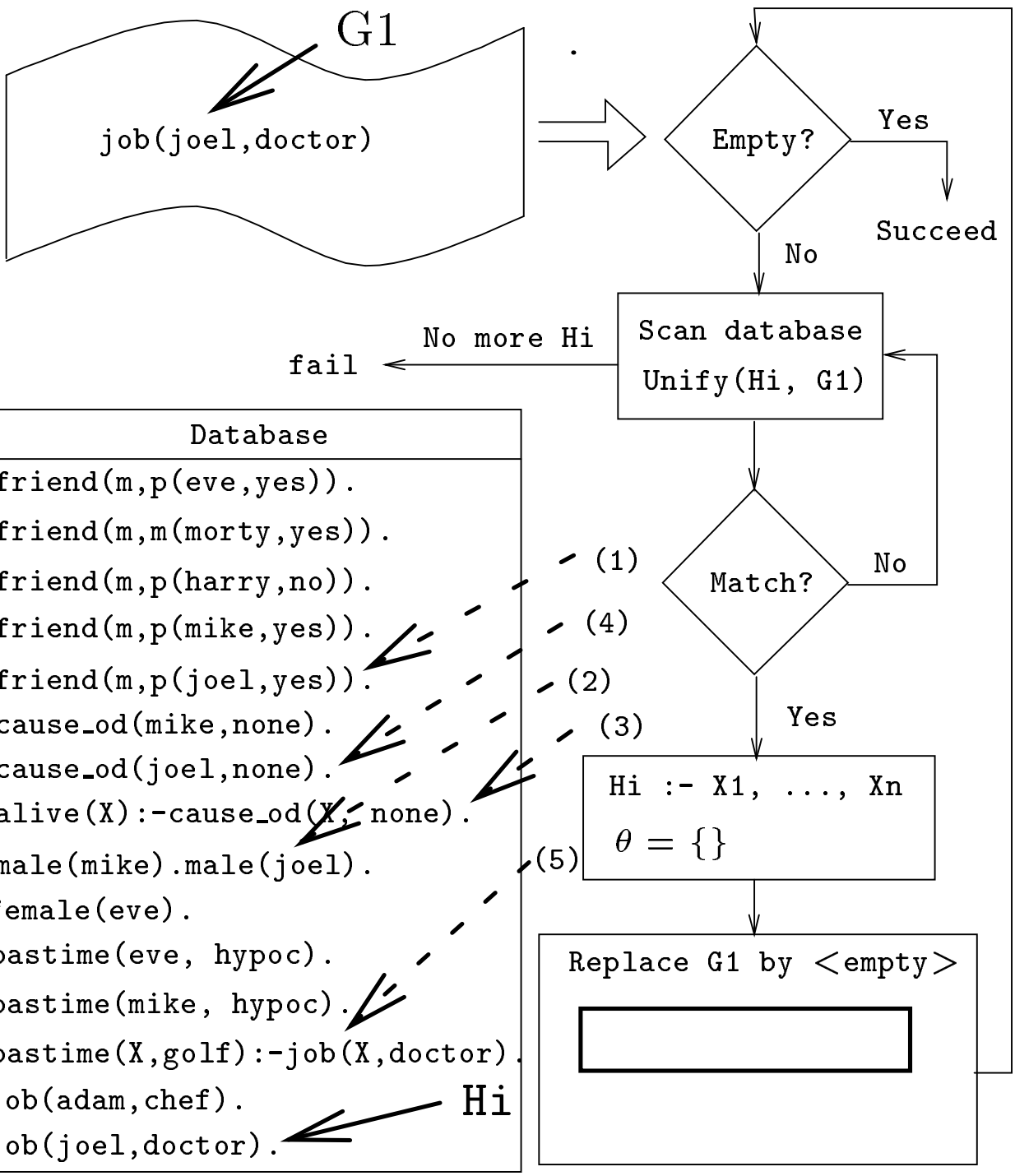
# Prolog Execution Example II



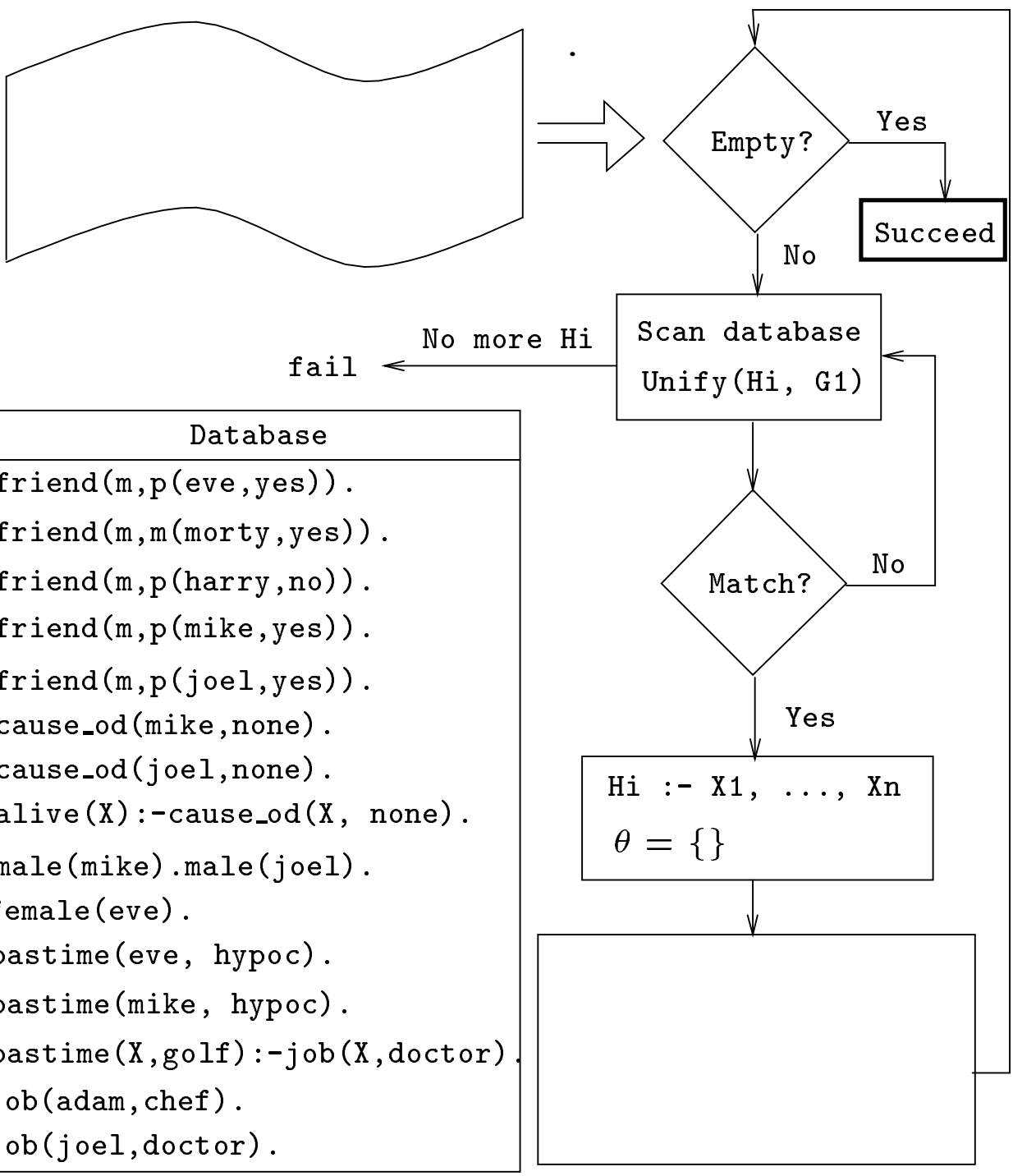
# Prolog Execution Example III



# Prolog Execution Example IV



# Prolog Execution Example V




| Database |
|---|
| friend(m,p(eve,yes)). |
| friend(m,m(morty,yes)). |
| friend(m,p(harry,no)). |
| friend(m,p(mike,yes)). |
| friend(m,p(joel,yes)). |
| cause_od(mike,none). |
| cause_od(joel,none). |
| alive(X):-cause_od(X, none). |
| male(mike).male(joel). |
| female(eve). |
| pastime(eve, hypoc). |
| pastime(mike, hypoc). |
| pastime(X,golf):-job(X,doctor). |
| job(adam,chef). |
| job(joel,doctor). |