

☞ Default rules are typically true, but may have exceptions

`default((flies(X):-bird(X))).`	Birds **typically** fly
`rule((not flies(X):-penguin(X))).`	Penguins don't fly
`rule((bird(X):-penguin(X))).`	Penguins are birds
`rule((penguin(tweety):-true)).`	Tweety is a penguin
`rule((bird(opus):-true)).`	Opus is a bird

Default rules

```
explain(true,E,E):-!.
explain((A,B),E0,E):-!,
  explain(A,E0,E1),
  explain(B,E1,E).
explain(A,E0,E):-
  prove_e(A,E0,E).               % explain by rules only
explain(A,E0,[default((A:-B))|E]):-
  default((A:-B)),               % explain by default
  explain(B,E0,E),
  not contradiction(A,E).        % A consistent with E
contradiction(not A,E):-!,
  prove_e(A,E,_).
contradiction(A,E):-
  prove_e(not A,E,_).
```

Meta-interpreter for default rules


```
default(mammals_dont_fly(X),     (not flies(X):-mammal(X))).
default(bats_fly(X),             (flies(X):-bat(X))).
default(dead_things_dont_fly(X), (not flies(X):-dead(X))).

rule((mammal(X):-bat(X))).
rule((bat(dracula):-true)).
rule((dead(dracula):-true)).

% Cancellation rules:

% bats are flying mammals
rule((not mammals_dont_fly(X):-bat(X))).

% dead bats don't fly
rule((not bats_fly(X):-dead(X))).
```

Does Dracula fly or not?


```
explain(true,E,E):-!.
explain((A,B),E0,E):-!,
    explain(A,E0,E1),
    explain(B,E1,E).
explain(A,E0,E):-
    prove_e(A,E0,E).            % explain by rules only
explain(A,E0,[default(Name)|E]):-
    default(Name,(A:-B)),       % explain by default rule
    explain(B,E0,E),
    not contradiction(Name,E),  % default should be applicable
    not contradiction(A,E).     % A should be consistent with E
```

Extended meta-interpreter for named defaults

```
default(mammals_dont_fly(X),(not flies(X):-mammal(X))).
default(bats_fly(X),(flies(X):-bat(X))).
default(dead_things_dont_fly(X),(not flies(X):-dead(X))).
rule((mammal(X):-bat(X))).
rule((bat(dracula):-true)).
rule((dead(dracula):-true)).
rule((not mammals_dont_fly(X):-bat(X))).
rule((not bats_fly(X):-dead(X))).
```

```
?-explain(flies(dracula),[],E).
No

?-explain(not flies(dracula),[],E).
E = [ default(dead_things_dont_fly(dracula)),
      rule((dead(dracula):-true)) ]
```

Dracula doesn’t fly after all

- ☞ For each default name, introduce a predicate introducing the opposite ('abnormality predicate')

  `bats_fly(X)` becomes `nonflying_bat(X)`

- ☞ Add this predicate as a negative condition

```
default(bats_fly(X),(flies(X):-bat(X)))
```

  becomes

```
flies(X):-bat(X),not nonflying_bat(X)
```

- ☞ Introduce new predicate for negated conclusions

```
default(dead_things_don't_fly(X),(not flies(X):-dead(X)))
```

  becomes

```
notflies(X):-dead(X),not flying_deadthing(X)
```

Defaults using negation as failure

```
default(mammals_dont_fly(X),(not flies(X):-mammal(X))).
default(bats_fly(X),(flies(X):-bat(X))).
default(dead_things_dont_fly(X),(not flies(X):-dead(X))).
rule((mammal(X):-bat(X))).
rule((bat(dracula):-true)).
rule((dead(dracula):-true)).
rule((not mammals_dont_fly(X):-bat(X))).
rule((not bats_fly(X):-dead(X))).
```

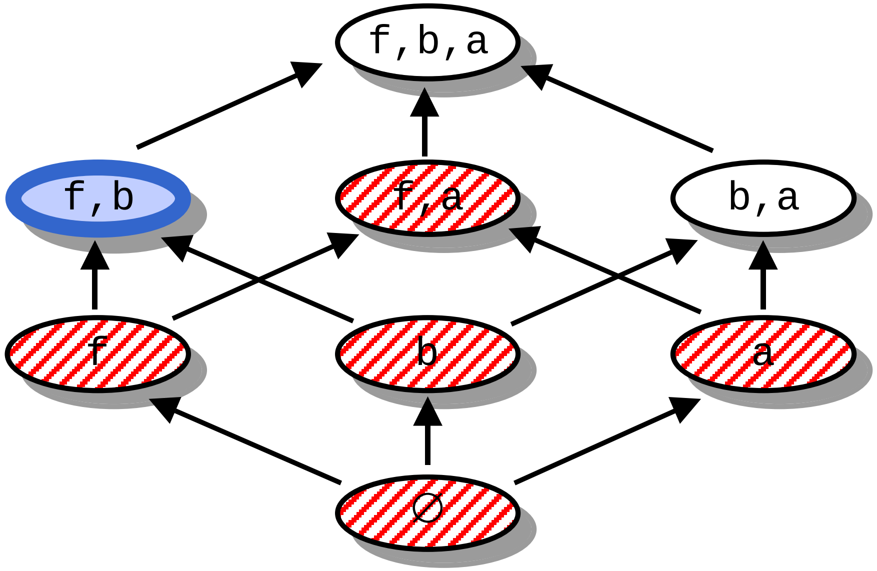
```
notflies(X):-mammal(X),not flying_mammal(X).
flies(X):-bat(X),not nonflying_bat(X).
notflies(X):-dead(X),not flying_deadthing(X).

mammal(X):-bat(X).
bat(dracula).
dead(dracula).

flying_mammal(X):-bat(X).
nonflying_bat(X):-dead(X).
```

Dracula again

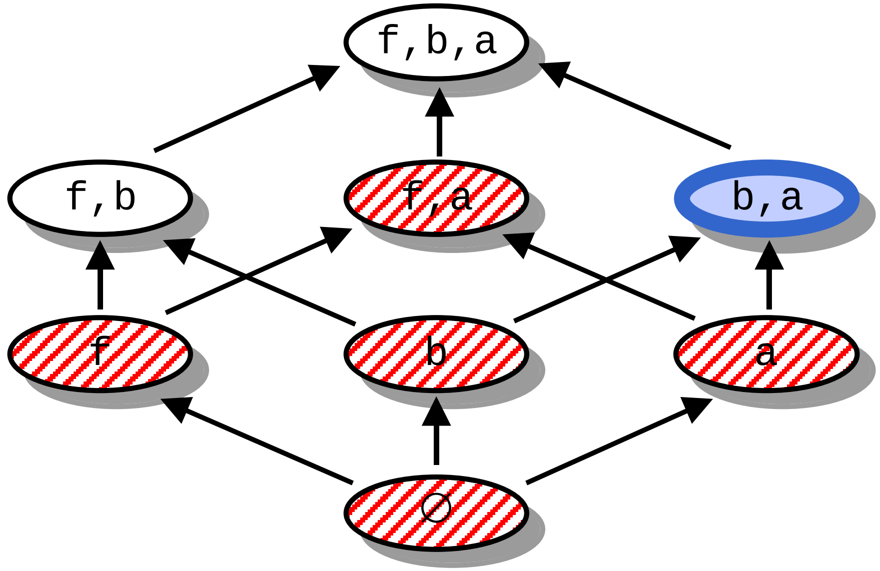
- ☞ Incompleteness arises when assumptions regarding a domain are not explicitly represented in a logic program ***P***.
- ☞ There are several ways to make these assumptions explicit:
  - ✓by selecting one of the models of ***P*** as the ***intended model***
  - ✓by transforming ***P*** into the ***intended program***
    - Closed World Assumption
    - Predicate Completion
- ☞ New information can invalidate previous conclusions if they were based on assumptions
  - ✓non-monotonic reasoning

Semantics of incomplete information


```
flies(X);abnormal(X):-bird(X).
bird(tweety).
```




```
flies(X):-bird(X),not abnormal(X).
```




```
abnormal(X):-bird(X),not flies(X).
```

Selecting an intended model

CWA(***P***) = ***P*** ∪ {`:-A` | `A` ∈ Herbrand base, `A` is not a logical consequence of ***P***}

```
likes(peter,S):-
      student_of(S,peter).
student_of(paul,peter).
```

CWA

```
:-student_of(paul,paul).
:-student_of(peter,paul).
:-student_of(peter,peter).
:-likes(paul,paul).
:-likes(paul,peter).
:-likes(peter,peter).
```

```
likes(peter,S):-
      student_of(S,peter).
student_of(paul,peter).
likes(paul,X).
```

CWA

```
:-student_of(paul,paul).
:-student_of(peter,paul).
:-student_of(peter,peter).
:-likes(peter,peter).
```

Closed World Assumption

- Step 1: rewrite clauses such that the head contains only distinct variables, by adding literals `Var=Term` to the body
- Step 2: for each head predicate, combine its clauses into a single universally quantified implication with disjunctive body
  - ✓take care of existential variables
- Step 3: turn all implications into equivalences
  - ✓undefined predicates `p` are rewritten to $\forall$`X:`$\neg$`p(X)`
- (Step 4: rewrite as general clauses)

Predicate Completion


```
likes(peter,S):-student_of(S,peter).
student_of(paul,peter).

likes(X,S):-X=peter,student_of(S,peter).
student_of(X,Y):-X=paul,Y=peter.

∀X∀Y:likes(X,Y) ← (X=peter ∧ student_of(Y,peter))
∀X∀Y:student_of(X,Y) ← (X=paul ∧ Y=peter)

∀X∀Y:likes(X,Y) ↔ (X=peter ∧ student_of(Y,peter))
∀X∀Y:student_of(X,Y) ↔ (X=paul ∧ Y=peter)

likes(peter,S):-student_of(S,peter).
X=peter:-likes(X,S).
student_of(S,peter):-likes(X,S).
student_of(paul,peter).
X=paul:-student_of(X,Y).
Y=peter:-student_of(X,Y).
```

Predicate Completion


```
ancestor(X,Y):-parent(X,Y).
ancestor(X,Y):-parent(X,Z),ancestor(Z,Y).
```

$\forall X \forall Y$: ancestor(X,Y) $\leftarrow$ (parent(X,Y) $\vee$ ($\exists Z$: parent(X,Z) $\wedge$ ancestor(Z,Y)))

$\forall X \forall Y$: ancestor(X,Y) $\leftrightarrow$ (parent(X,Y) $\vee$ ($\exists Z$: parent(X,Z) $\wedge$ ancestor(Z,Y)))

```
ancestor(X,Y):-parent(X,Y).
ancestor(X,Y):-parent(X,Z),ancestor(Z,Y).
parent(X,Y);parent(X,pa(X,Y)):-ancestor(X,Y).
parent(X,Y);ancestor(pa(X,Y),Y):-ancestor(X,Y).
```

Completion with existential variables


```
bird(tweety).
flies(X):-bird(X),not abnormal(X).
```

```
bird(X):-X=tweety.
flies(X):-bird(X),not abnormal(X).
```

$\forall X$:bird(X) $\leftarrow$ X=tweety
$\forall X$:flies(X) $\leftarrow$ (bird(X) $\wedge$ $\neg$abnormal(X))

$\forall X$:bird(X) $\leftrightarrow$ X=tweety
$\forall X$:flies(X) $\leftrightarrow$ (bird(X) $\wedge$ $\neg$abnormal(X))
$\forall X$:$\neg$abnormal(X)

```
bird(tweety).
X=tweety:-bird(X).
flies(X);abnormal(X):-bird(X).
bird(X):-flies(X).
:-flies(X),abnormal(X).
:-abnormal(X).
```

Completion with negation


```
wise(X):-not teacher(X).
teacher(peter):-wise(peter).
```

```
wise(X):-not teacher(X).
teacher(X):-X=peter,wise(peter).
```

$\forall X$:wise(X) $\leftarrow \neg$teacher(X)
$\forall X$:teacher(X) $\leftarrow$ (X=peter $\wedge$ wise(peter))

$\forall X$:wise(X) $\leftrightarrow \neg$teacher(X)
$\forall X$:teacher(X) $\leftrightarrow$ (X=peter $\wedge$ wise(peter))

```
wise(X);teacher(X).
:-wise(X),teacher(X).
teacher(peter):-wise(peter).
X=peter:-teacher(X).
wise(peter):-teacher(X).
```

inconsistent!

Exercise 8.3

Abduction: given a ***Theory*** and an ***Observation***, find an ***Explanation*** such that the ***Observation*** is a logical consequence of ***Theory*** ∪ ***Explanation***

```
% abduce(O,E0,E) <-   E is abductive explanation of O, given E0
abduce(true,E,E):-!.
abduce((A,B),E0,E):-!,
  abduce(A,E0,E1),
  abduce(B,E1,E).
abduce(A,E0,E):-
  clause(A,B),
  abduce(B,E0,E).
abduce(A,E,E):-         % already assumed
  element(A,E).
abduce(A,E,[A|E]):-     % A can be added to E
  not element(A,E),     % if it's not already there,
  abducible(A).         % and if it's abducible
abducible(A):-not clause(A,_).
```

Abductive meta-interpreter


```
likes(peter,S):-student_of(S,peter).
likes(X,Y):-friend(Y,X).
```

```
?-abduce(likes(peter,paul),[],E).
E = [student_of(paul,peter)] ;
E = [friend(paul,peter)]
```

```
flies(X):-bird(X),not abnormal(X).
abnormal(X):-penguin(X).
bird(X):-penguin(X).
bird(X):-sparrow(X).
```

```
?-abduce(flies(tweety),[],E).
E = [not abnormal(tweety),penguin(tweety)] ; % WRONG!!!
E = [not abnormal(tweety),sparrow(tweety)]
```

Abduction: examples

```
abduce(true,E,E):-!.
abduce((A,B),E0,E):-!,
    abduce(A,E0,E1),
    abduce(B,E1,E).
abduce(A,E0,E):-
    clause(A,B),
    abduce(B,E0,E).
abduce(A,E,E):-
    element(A,E).               % already assumed
abduce(A,E,[A|E]):-             % A can be added to E
    not element(A,E),           % if it's not already there,
    abducible(A),               % if it's abducible,
    not abduce_not(A,E,E).      % and E doesn't explain not(A)
abduce(not(A),E0,E):-           % find explanation for not(A)
    not element(A,E0),          % should be consistent
    abduce_not(A,E0,E).
```

Abduction with negation


```
% abduce_not(O,E0,E) <- E is abductive explanation of not(O)
abduce_not((A,B),E0,E):-!,
    abduce_not(A,E0,E);            % disjunction
    abduce_not(B,E0,E).
abduce_not(A,E0,E):-
    setof(B,clause(A,B),L),
    abduce_not_l(L,E0,E).
abduce_not(A,E,E):-
    element(not(A),E).             % not(A) already assumed
abduce_not(A,E,[not(A)|E]):-      % not(A) can be added to E
    not element(not(A),E),         % if it's not already there,
    abducible(A),                  % if A is abducible
    not abduce(A,E,E).             % and E doesn't explain A
abduce_not(not(A),E0,E):-          % find explanation for A
    not element(not(A),E0),        % should be consistent
    abduce(A,E0,E).
```

Explaining negative literals


```
flies(X):-bird(X),not abnormal(X).
flies1(X):-not abnormal(X),bird(X).
abnormal(X):-penguin(X).
abnormal(X):-dead(X).
bird(X):-penguin(X).
bird(X):-sparrow(X).
```

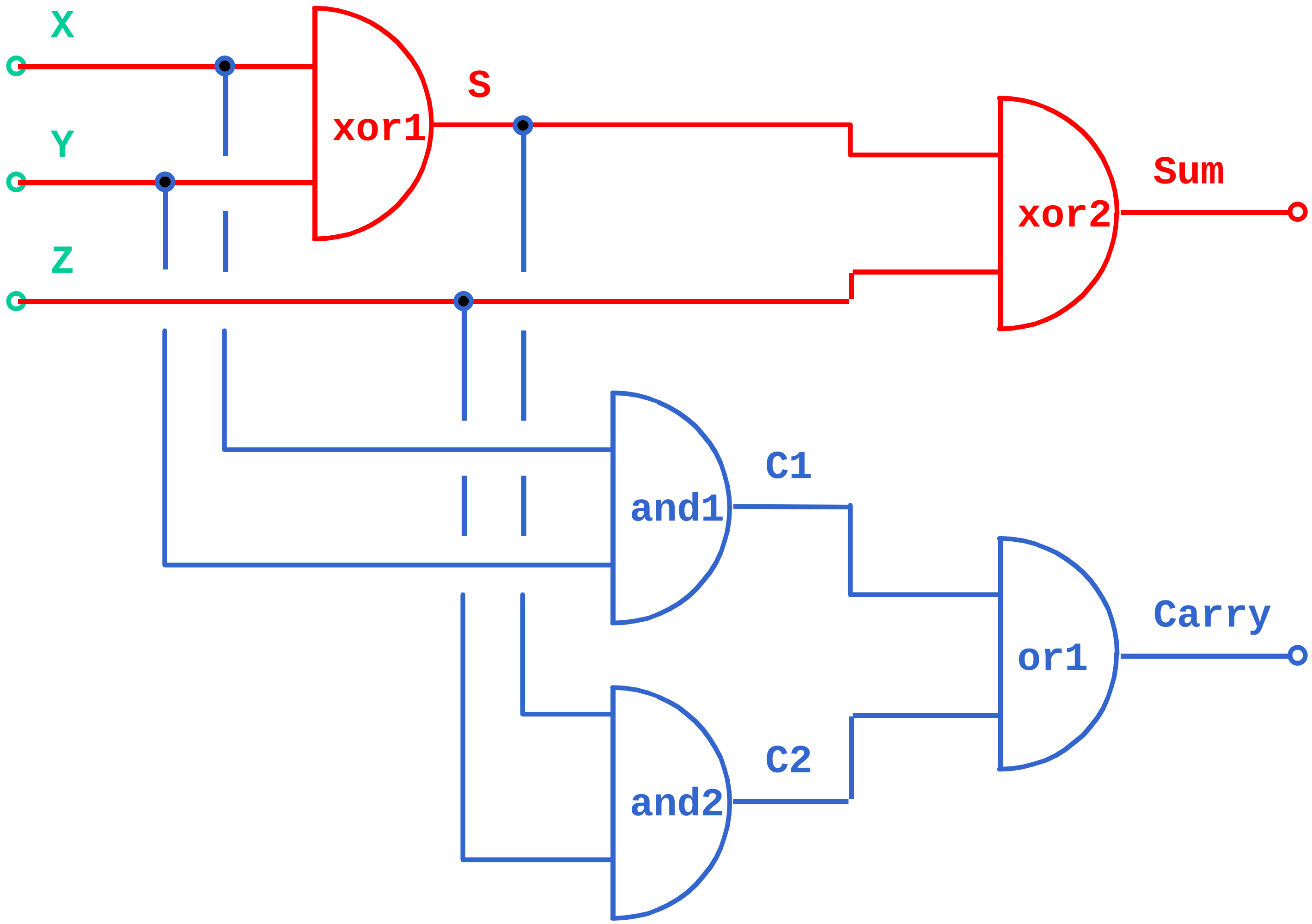
```
?-abduce(flies(tweety),[],E).
E = [not penguin(tweety),not dead(tweety),sparrow(tweety)]

?-abduce(flies1(tweety),[],E).
E = [sparrow(tweety),not penguin(tweety),not dead(tweety)]
```

Abduction with negation: example


```
notflies(X):-mammal(X),not flying_mammal(X).
flies(X):-bat(X),not nonflying_bat(X).
notflies(X):-dead(X),not flying_deadthing(X).
mammal(X):-bat(X).
bat(dracula).
dead(dracula).
flying_mammal(X):-bat(X).
nonflying_bat(X):-dead(X).
```

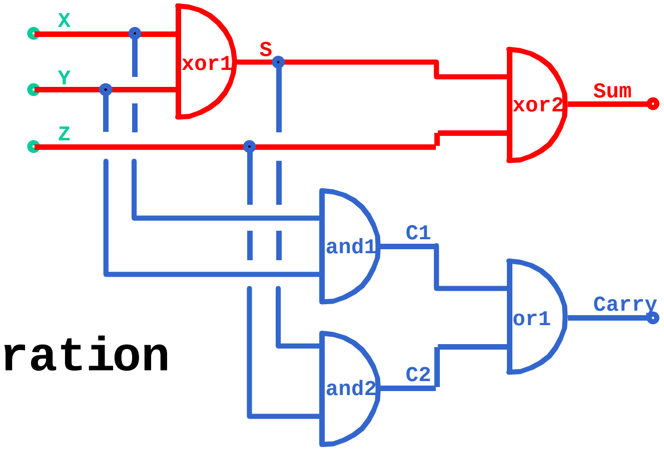
```
?-abduce(flies(X),[],E).
No

?-abduce(notflies(X),[],E).
E = [not flying_deadthing(dracula)]
```

Abduction generalises negation as failure



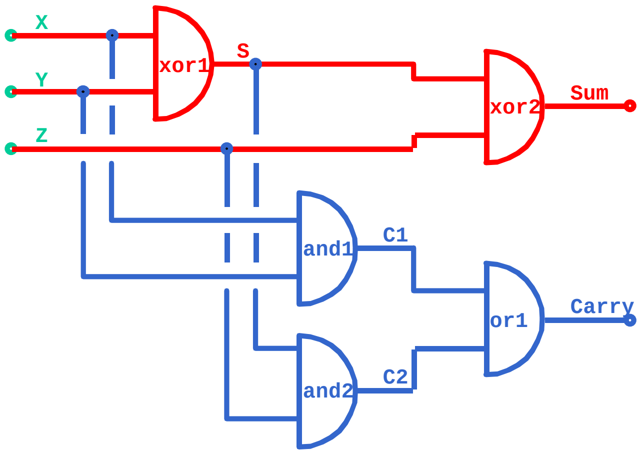
3-bit adder

```
adder(N,X,Y,Z,Sum,Carry):-
    xorg(N-xor1,X,Y,S),        % N-xor1 is the name of this gate
    xorg(N-xor2,Z,S,Sum),
    andg(N-and1,X,Y,C1),
    andg(N-and2,Z,S,C2),
    org(N-or1,C1,C2,Carry).
```




```
% fault model (similar for andg, org)
xorg(N,X,Y,Z):-xor(X,Y,Z).   % normal operation
xorg(N,1,1,1):-fault(N=s1).  % stuck at 1
xorg(N,0,0,1):-fault(N=s1).  % stuck at 1
xorg(N,1,0,0):-fault(N=s0).  % stuck at 0
xorg(N,0,1,0):-fault(N=s0).  % stuck at 0

% gates (similar for and, or)
xor(1,0,1).
xor(0,1,1).
xor(1,1,0).
xor(0,0,0).
```

3-bit adder in Prolog




```
?-abduce(adder(a,0,0,1,0,1),[],D).
D = [fault(a-or1=s1),fault(a-xor2=s0)];
D = [fault(a-and2=s1),fault(a-xor2=s0)];
D = [fault(a-and1=s1),fault(a-xor2=s0)];
D = [fault(a-and2=s1),fault(a-and1=s1),fault(a-xor2=s0)];
D = [fault(a-xor1=s1)];
D = [fault(a-or1=s1),fault(a-and2=s0),fault(a-xor1=s1)];
D = [fault(a-and1=s1),fault(a-xor1=s1)];
D = [fault(a-and2=s0),fault(a-and1=s1),fault(a-xor1=s1)];
No more solutions
```

Abductive diagnosis