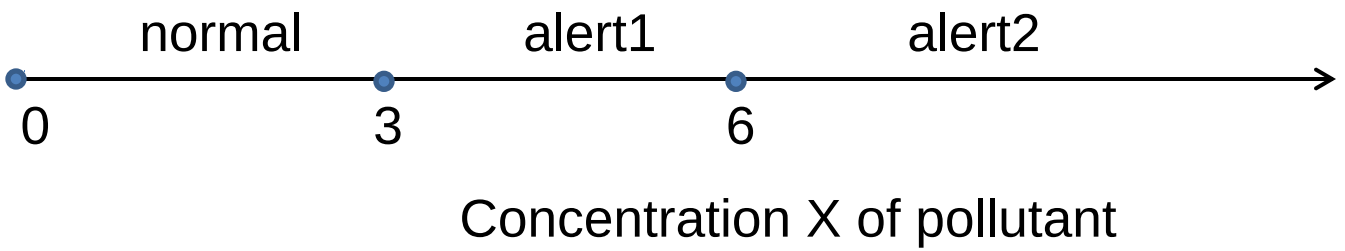
PROLOG: CONTROLLING BACKTRACKING, CUT AND NEGATION

Ivan Bratko
University of Ljubljana

These slides are meant to be used with a Prolog system to demonstrate the examples, and the book: I. Bratko, Prolog Programming for Artificial Intelligence, 4th edn., Pearson Education 2011. The slides are not self-sufficient.

EXAMPLE: STATE OF POLLUTION ALERT



HOW ALERT DEPENDS ON POLLUTION?

- *Rule 1*: if $X < 3$ then Y = normal
- *Rule 2*: if $3 \leq X$ and $X < 6$ then Y = alert1
- *Rule 3*: if $6 \leq X$ then Y = alert2

f(Concentration, State_of_alert)

f(X, normal) :- X < 3. % Rule 1

f(X, alert1) :- 3 =< X, X < 6. % Rule 2

f(X, alert2) :- 6 =< X. % Rule 3

EXPERIMENT 1

?- f(2, Y), Y = alert1.

no

- Study execution trace; at some points backtracking occurs when it obviously makes no sense?

VERSION 2

f(X, normal) :- X < 3, !.

f(X, alert1) :- 3 =< X, X < 6, !.

f(X, alert2) :- 6 =< X.

- “!” is read as “cut” because it cuts alternatives
- Cut prevents pointless backtracking

- Version 2 is more efficient than version 1,
- Here, the cuts do not affect the logical meaning

EXPERIMENT 2

?- f(7, Y).

Y = **alert2**

- Study execution trace, Prolog again does some unnecessary work

VERSION 3

```
f( X, normal)  :-  X < 3, !.
f( X, alert1)  :-  X < 6, !.
f( X, alert2).
```

- This is the most efficient version

- But unfortunatly, the logical meaning has changed. Try this:

```
?-  f( 2, alert1).
    yes              % Not as intended!
```

- Study why Prolog now answered “yes”

- A more careful formulation of the question is:

?- f(2, Y), Y = alert1.

no

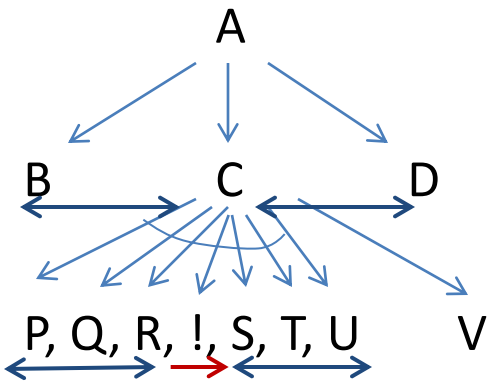
THE SCOPE OF CUT

C :- P, Q, R, !, S, T, U.

C :- V.

A :- B, C, D.

?- A.



The cut is not “visible” from A (cut is nested too deep from point of view of A)

MAXIMUM

```
max( X, Y, X)  :-  X >= Y.
max( X, Y, Y)  :-  X < Y.

% More efficient with cut
max( X, Y, X)   :-   X >= Y, !.
max( X, Y, Y).

% But note again!!!
?- max( 3, 1, 1).
yes                    % Not as intended!
```

MORE CAREFUL FORMULATION OF MAX

```
max( X, Y, Max) :-
  X >= Y, !, Max = X
  ;
  Max = Y.

?-  max( 3, 1, 1).
no                    % As intended
```

CUT AFFECTS DECLARATIVE MEANING

p :- a, b.

p :- c.

- This means: p <===> (a & b) v c

p :- a, !, b.

p :- c.

- Means: p <===> (a & b) v (~a & c)

- If we change the order of clauses:

p :- c.

p :- a, !, b.

- The meaning also changes:

p <===> c v (a & b)

“Mary likes all animals but snakes”

How can we express this in Prolog?

If X is a snake then “Mary likes X” is not true,

otherwise if X is an animal then Mary likes X.

```
likes( mary, X) :-
    snake( X), !, fail.        % “fail” is built-in predicate that always fails

likes( mary, X) :-
    animal( X).
```

NEGATION

- In Prolog, negation is defined as:

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

- This is called *negation as failure*

- **not** can be written as a prefix operator: **not P**

MARY & ANIMALS: FORMULATION WITH NEGATION

```
likes( mary, X) :-
    animal( X),
    not snake( X).
```

- This is more readable than the formulation with cut + fail

NEGATION AS FAILURE

- Not exactly the same as negation in logic (mathematics)

- Negation as failure makes the “closed world assumption”
- That is: Everything that Prolog cannot derive from the program is assumed to be false
- Standard abbreviation: CWA = Closed World Assumption

- Alternative, more standard but less pretty, notation for **not P** is:

  \+ P

CLOSED WORLD ASSUMPTION

- What yes/no means under CWA? Consider this single line programe:

  round(ball).

- How should Prolog's answers be understood in the following?

?- round(ball).

yes % Yes, round(ball) logically follows from program

?- round(earth).

no % "no" means: I don't know, can't be derived from program

?- not round(earth).

yes % It follows from the program, but only under CWA

PROBLEMS WITH NEGATION

- Negation as failure is defined through cut, so we can expect some difficulties. Consider this example about restaurants:

good_standard(jeanluis).

expensive(jeanluis).

good_standard(francesco).

reasonable(Restaurant) :- % A restaurant is reasonably priced if

not expensive(Restaurant). % it is not expensive

ASKING ABOUT RESTAURANTS

% Ask for good and reasonable restaurant:

?- good_standard(X), reasonable(X).

X = francesco % As expected

% Ask for reasonable and good reastaurant:

?- reasonable(X), good_standard(X).

no % Surprize! What happened?

- Under negation, Prolog's usual quantification of variables changes

- Safe use of negation as failure: variables in negated goals are instantiated at the time of the execution of such goals