COMP3411-9814- Artificial Intelligence

Prolog
Trace, Cut, Negation

2020 – Summer Term

Tatjana Zrimec

Outiline

- Trace
- Cut
- Negation

Adding rules from the console

- The most convenient way to add a few clauses is by consulting the pseudo file user.
- The input is ended using the system end-of-file character.

```
?- [user].
|: hello :- format('Hello world~n').
|: ^D
true.

?- hello.
Hello world
true.
```

Example - Investing the list

```
we_have([bread, wine, cheese, meat, beer]).
```

Ask prolog:

?- we_have ([H | T])?

H = bread

T = [wine, cheese, meat, beer]

How would Prolog check that the cheese is in the list?

?- we_have (List), member (cheese, List)?

%member(X,L) was defined in the previous lecture about Lists

Example - Investing the list

we_have([bread, wine, cheese, meat, beer]).

Ask prolog:

?- we_have ([H **|** T]).

```
?- we_have([H | T]).
H = bread,
T = [wine, cheese, meat, beer].
```

How would Prolog check that the cheese is in the list?

?- we_have (List), member (cheese, List).

%member(X,L) was defined in the previous lecture about Lists

Trace

- To trace the program execution, you can use the predicate trace

- Tracing the execution of a Prolog query allows you to see all of the goals that are executed as part of the query, in sequence, along with whether or not they succeed. Tracing also allows you to see what steps occur as Prolog backtracks.

SWI SWI Prolog Manual

- 4.39 Debugging and Tracing Programs
- http://www.swi-prolog.org/pldoc/man?section=debugger

Trace demo

```
?- member(X, [bread, wine, cheese, meat, beer]).
X = bread
X = wine
X = cheese
X = meat
X = beer
false.

?- trace, member(X, [bread, wine, cheese, meat, beer]).
   Call: (9) member(_4432, [bread, wine, cheese, meat, beer]) ? creep
   Exit: (9) member(bread, [bread, wine, cheese, meat, beer]) ? creep
X = bread .

[trace]  ?- member(X, [bread, wine, cheese, meat, beer]).
   Call: (8) member(_4710, [bread, wine, cheese, meat, beer]) ? creep
   Exit: (8) member(bread, [bread, wine, cheese, meat, beer]) ? creep
X = bread
   Redo: (8) member(_4710, [bread, wine, cheese, meat, beer]) ? creep
   Call: (9) member(_4710, [wine, cheese, meat, beer]) ? creep
   Exit: (9) member(wine, [wine, cheese, meat, beer]) ? creep
   Exit: (8) member(wine, [bread, wine, cheese, meat, beer]) ? creep
X = wine
   Redo: (9) member(_4710, [wine, cheese, meat, beer]) ? creep
   Call: (10) member(_4710, [cheese, meat, beer]) ? creep
   Exit: (10) member(cheese, [cheese, meat, beer]) ? creep
   Exit: (9) member(cheese, [wine, cheese, meat, beer]) ? creep
   Exit: (8) member(cheese, [bread, wine, cheese, meat, beer]) ? creep
X = cheese
   Redo: (10) member(_4710, [cheese, meat, beer]) ? creep
   Call: (11) member(_4710, [meat, beer]) ? creep
   Exit: (11) member(meat, [meat, beer]) ? creep
   Exit: (10) member(meat, [cheese, meat, beer]) ? creep
   Exit: (9) member(meat, [wine, cheese, meat, beer]) ? creep
   Exit: (8) member(meat, [bread, wine, cheese, meat, beer]) ? creep
X = meat
   Redo: (11) member(_4710, [meat, beer]) ? creep
   Call: (12) member(_4710, [beer]) ? creep
   Exit: (12) member(beer, [beer]) ? creep
   Exit: (11) member(beer, [meat, beer]) ? creep
   Exit: (10) member(beer, [cheese, meat, beer]) ? creep
   Exit: (9) member(beer, [wine, cheese, meat, beer]) ? creep
   Exit: (8) member(beer, [bread, wine, cheese, meat, beer]) ? creep
X = beer
   Redo: (12) member(_4710, [beer]) ? creep
   Call: (13) member(_4710, []) ? creep
   Fail: (13) member(_4710, []) ? creep
   Fail: (12) member(_4710, [beer]) ? creep
   Fail: (11) member(_4710, [meat, beer]) ? creep
   Fail: (10) member(_4710, [cheese, meat, beer]) ? creep
   Fail: (9) member(_4710, [wine, cheese, meat, beer]) ? creep
   Fail: (8) member(_4710, [bread, wine, cheese, meat, beer]) ? creep
false.
```

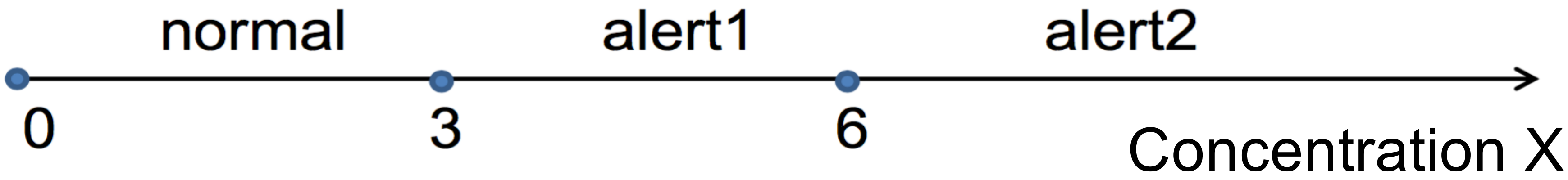
Automatic backtracking is not always desirable

- Automatic return is built into prolog
- It is often useful and significantly reduces the program
- However, sometimes automatic backtracking is unnecessary or even undesirable
- Backtracking is prevented by calling "!" (Cut)

Cut !

- The cut “!” is used to prevent unwanted backtracking, for example, to prevent extra solutions being found by Prolog.

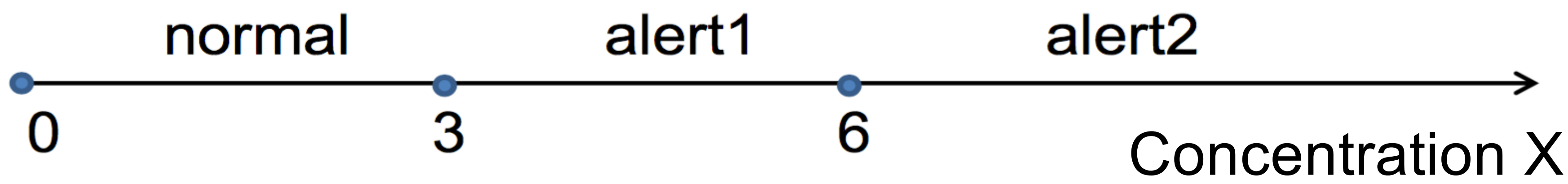
Example: Pollution degree



- State of alert due to pollution.
- The degree of pollution Y is a concentration function of X:

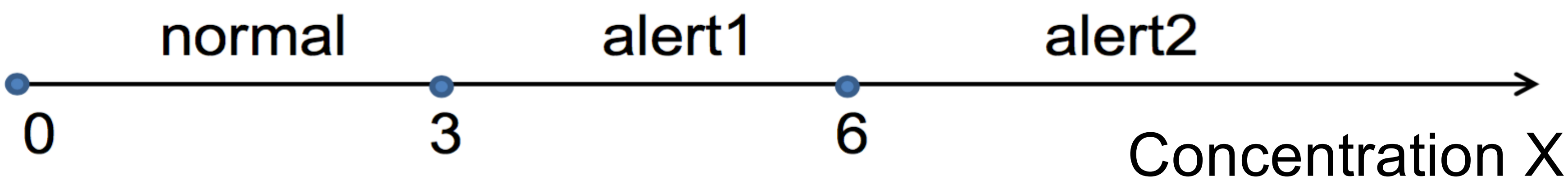
$$Y = f(X)$$

Example: Pollution degree



- Rules for determining the degree of pollution
  - Rule 1: if $X < 3$ then Y = normal
  - Rule 2: if $3 \le X$ and $X < 6$ then Y = alert1
  - Rule 3: if $6 \le X$ then Y = alert2

Example: Pollution degree



Rules for determining the degree of pollution

Rule 1: if X < 3 then Y = normal

Rule 2: if 3 ≤ X and X < 6 then Y = alert1

Rule 3: if 6 ≤ X then Y = alert2

- In Prolog: f(Concentration, Pollution_Alert)

```
f(X, normal) :-  X <  3.              %Rule1
f(X, alert1) :-  3  =<  X,  X < 6.    %Rule2
f(X, alert2) :-  6  =<  X.            %Rule3
```

Program version 1

%determining the degree of pollution

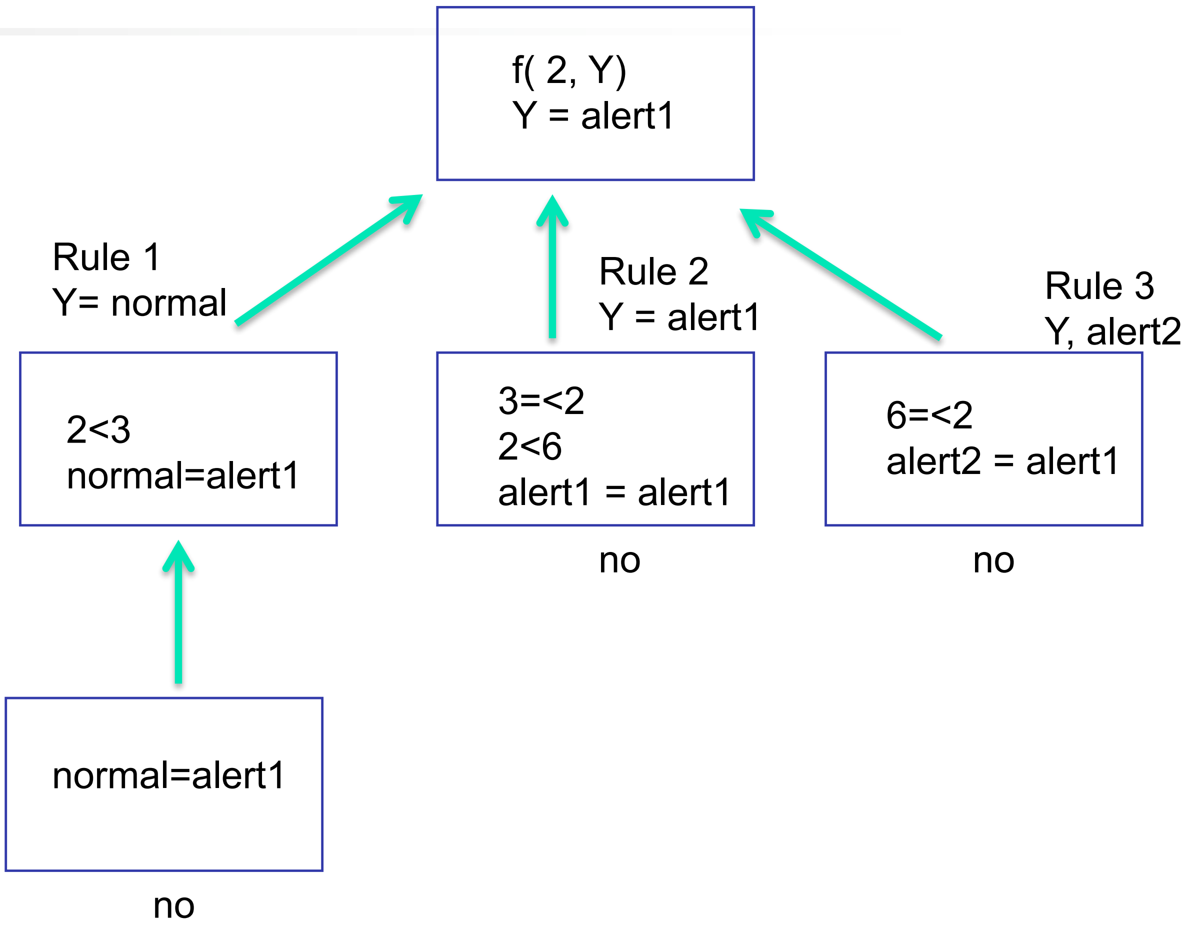
%   f(Concentration, Pollution_Alert)

```
f(X, normal) :-  X <  3.                    %Rule1
f(X, alert1) :-   3 =<  X,  X  < 6.         %Rule2
f(X, alert2) :-   6 =<  X.                  %Rule3
```

Experiment 1

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

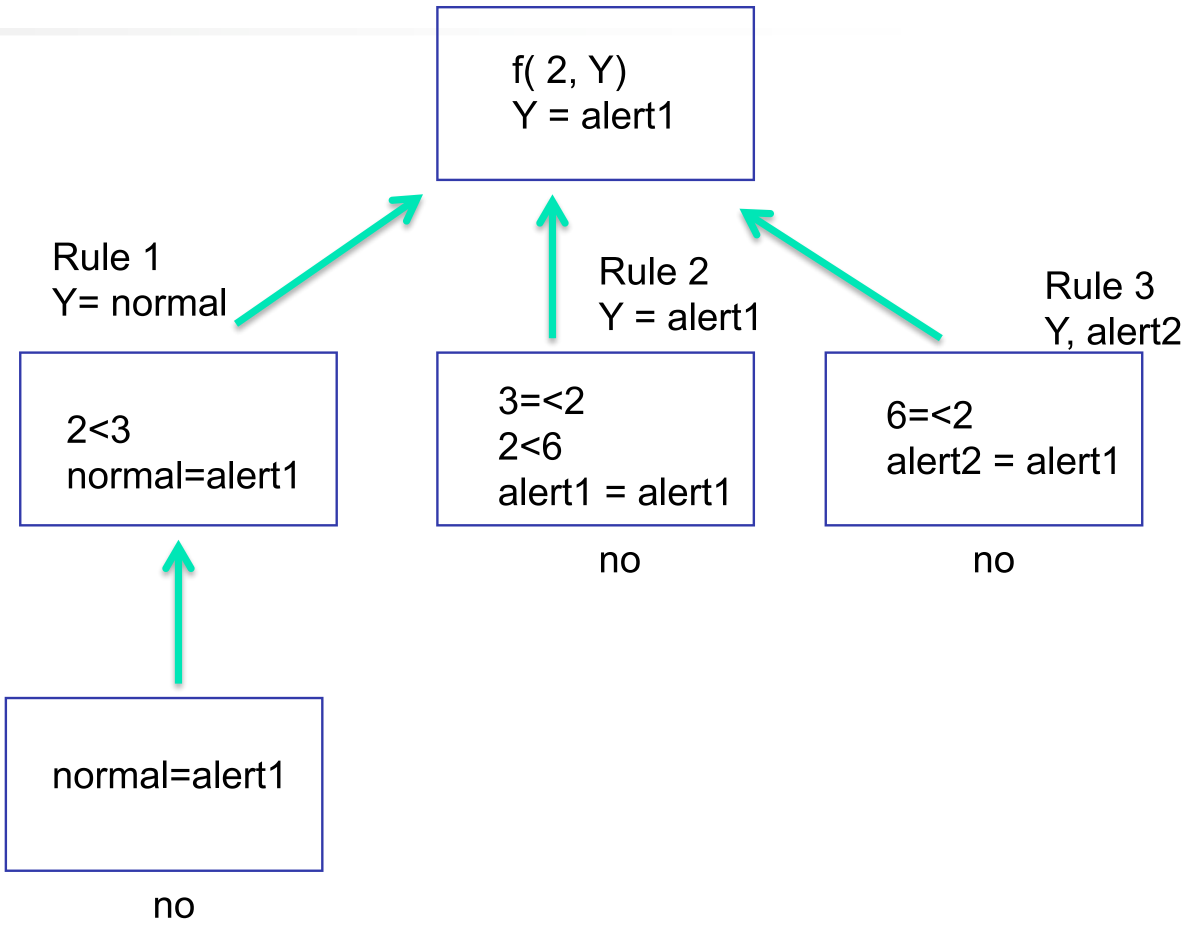
no



Experiment 1

?- f(2, Y), Y = alert1.
no

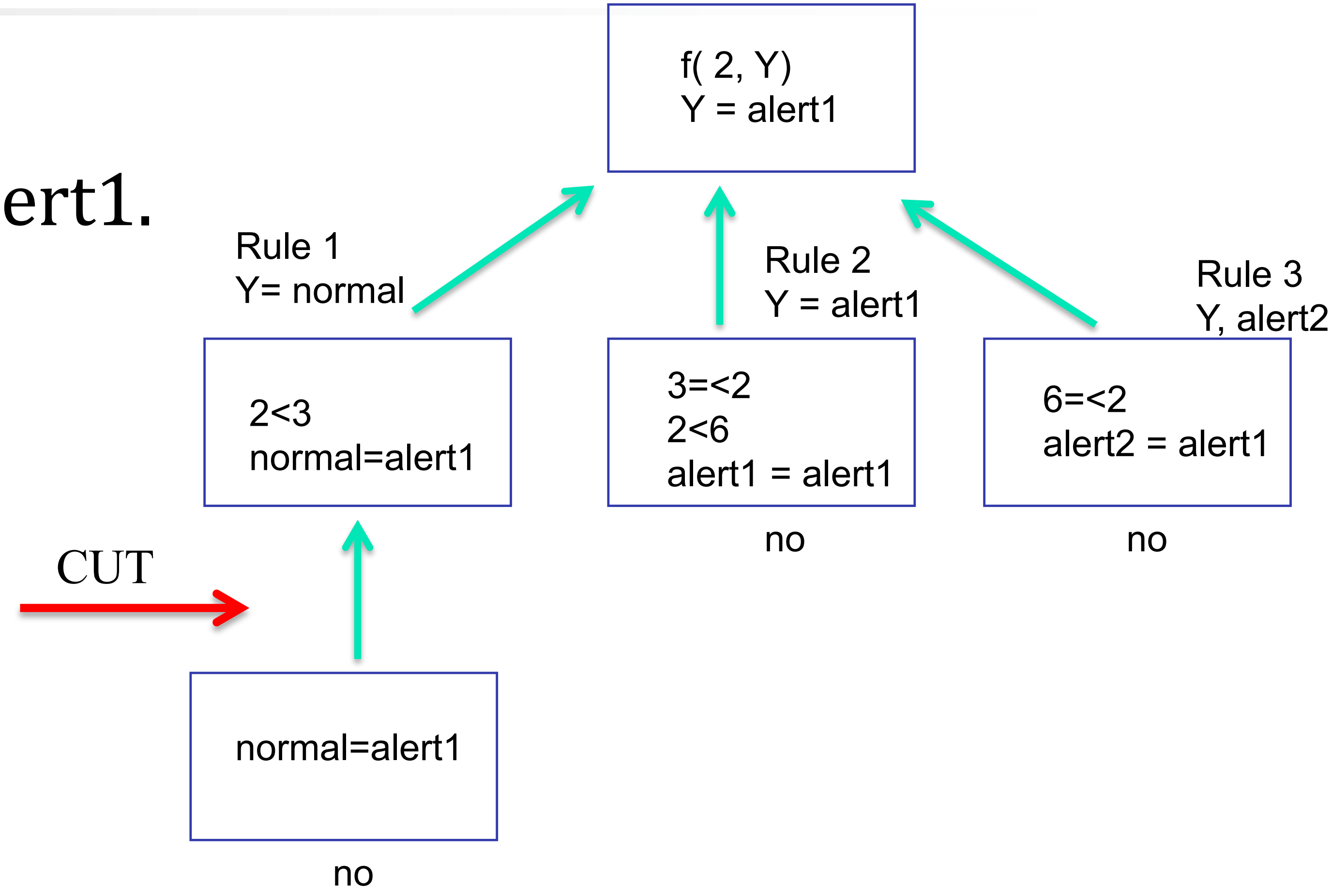
After running the program, the trace shows unnecessary backtracking, once we know that alternatives will fail.



Experiment 1

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

no



Second version with "!"

```
f( X, normal) :-  X < 3, !.          % Cut prevents backtracking

f( X, alert1) :-  3 =< X,  X < 6, !. % Cut prevents backtracking

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

More effective than version 1, calls do not affect the logical meaning.

That is: If the cats (!) are deleted, the results of the program will be the same.

Experiment 2 with the second version

?- f(7, Y).

Y = alert2

Tracing shows again some extra work (checking the complementary conditions)

Program version 3

```
f( X, normal) :-  X <  3, !.        % Cut prevents backtracking re
f( X, alert1) :-   X <  6, !.       % Cut prevents backtracking
f( X, alert2).
```

The most effective ...
But: the logical meaning has changed!

```
?- f( 2, alert1).
yes
```

Program version 3

```
f( X, normal) :-  X <  3, !.          % Cut prevents backtracking re
f( X, alert1) :-   X <  6, !.          % Cut prevents backtracking
f( X, alert2).
```

Try :

?- f(1, Y).

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

Program version 3

```
?- f( 2, alert1).
yes
```

This is not correct, but with a more careful formulation of the question:

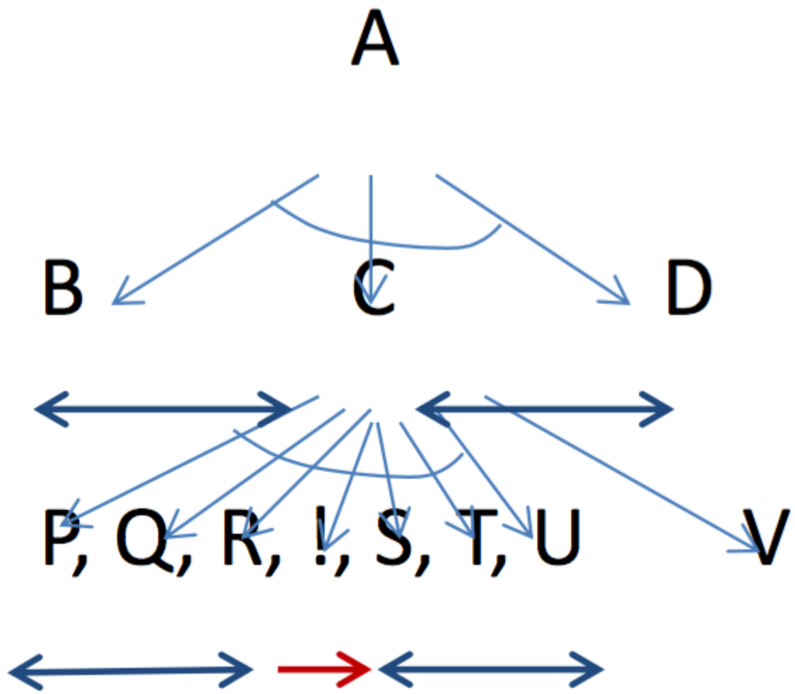
```
?- f(2,Y),Y=alert1.
No
```

The third version of the program with cut, cuts affect the procedural behavior and changed the result of the program.

We have to be careful when using “!”.

The effect of the "!" on the execution

C :- P, Q, R, !, S, T, U.
C :- V.
A :- B, C, D.
?- A.



The call affects the execution of the Goal C;
Backtracking is still valid between B, C, D (the call is "not visible" from Goal A)

Examples using cut -Maximum

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

% More effective with the  cut  "!"
max(X, Y, X) :- X >= Y, !.
max( X, Y, Y).
```

Be careful!

?- max(3, 1, 1).

Yes   -- >   Not correct answer!

Careful Formulation

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

?- max( 3, 1, 1).
no
```

"Mary likes all animals but snakes"

- 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.

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

Cut affects the declarative meaning

p :- a, b.

p :- c.

Declarative meaning : p <===> (a & b) v c

p :- a, ! ,b.

p :- c.

Declarative meaning : p <===> (a & b) v (~a & c)

If we change the order of sentences:

p :- c.

p :- a, !, b.

The meaning is changed, it becomes : p <===> c v (a & b)

Negation

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

- Negation as failure

- not we also write as a prefix operator:

not P   or as an operator \ +:   \ + P

- "\ +" is more standard and more often embedded in the prolog

"Mary likes all animals but snakes" Formation with Negation

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

It looks better than with ! + fail.

Negation as failure

- It is not exactly the same as the negation in logic (mathematics)
- It applies only under the assumption of a closed world (Closed World Assumption, CWA)

Closed World Assumption

Program:

```
round( ball).
?- round( ball).
yes   % Yes, it logically follows from program

?- round( earth).
no      % I don't know; it doesn't logically follow from program

?- \+ round( earth).              % \+ = not
yes          % It follows from program, but only under CWA
```

Interpreter for If-then-else

```
% Appropriate operator declaration
:- op( 500, fx, if).
:- op( 400, xfx, then).
:- op( 300, xfx, else).

% Interpreter
if CONDITION then S1 else S2 :-      % This is true if
     CONDITION, S1                   % is true :CONDITION in S1
     ;                               % or
      \+ CONDITION, S2.              % is true : not CONDITION in S2
```