Built-in Predicates in Prolog

Module of Logics and Artificial Intelligence course
a.a. 2016/2017

Arithmetic and Logical Operators

Classic notation	Prolog Notation
6+2=8	8 is 6+2.
6*2=12	12 is 6*2.
6-2=4	4 is 6-2.
6-8=-2	-2 is 6-8.
6 /2 =3	3 is 6/2.
7 mod 2 = 1	1 is mod(7,2).

Arithmetic and Logical Operators

?- 8 is 6+2.

?- 12 is 6*2.

? -2 is 6-8.

?- 3 is 6/2.

?- 1 is mod(7,2).

?- X is 6+2.

?- X is 6*2.

?- R is mod(7,2).

Arithmetic and Logical Operators

```
?- 8 is 6+2.
true
?- 12 is 6*2.
true
? -2 is 6-8.
true
?- 3 is 6/2.
true
?- 1 is mod(7,2).
true
?- X is 6+2.
X = 8
?- X is 6*2.
X = 12
?- R is mod(7,2).
R = 1
```

Examples

Sum 3 and double

sum3AndDouble(X,Y) :-

List length

Examples

Sum 3 and double

```
sum3AndDouble(X,Y) :- Y is (X+3)*2

?- somma_3_e_raddoppia(1,Y).
Y=8;
```

List length

```
len([],0).
len([_|T],N) :- len(T,X), N is X+1.

?- len([a,b,c,d,e,[a,b],g],X).
X = 7;
```

Arithmetic and Logical Operators

Notazione classica	Notazione Prolog
x<y	X<Y.
x<=y	X=<Y.
x>=y	X>=Y.
x=y	X=:=Y.
x!=y	X=\=Y.

```
4=:=4.
true
4=\=5.
true
4=\=4.
false
4 >= 4.
true
```

```
4 > 2.
true
2 < 4.
true
2 =< 4.
true
4 =< 4.
true
```

Example: Max of a List

Write a rule that finds the maximum of a list: **maxList(List,Max).**

Example: Max of a List

Write a rule that finds the maximum of a list: **maxList(List,Max).**

```
maxList([1,0,5,4,2], Max)
maxList([H|T],A,Max) :- H > A, maxList(T,H,Max).
maxList([H|T],A,Max) :- H =< A, maxList(T,A,Max).
maxList([],A,A).
```

```
?- maxList([1,0,5,4,2], Max).
Max = 5.
```

N.B. In this implementation two relations with different arities are defined:
maxList/2 and maxList/3

Built-in Predicates

Built-in predicates are already defined in Prolog implementation:

I/O predicates:

- **Write/1**: prints the argument value to the standard output
- **Nl/0**: breaks the line and returns a new line.
- **Tab/1**: inserts a number of tab spaces equal to the argument value.

When those predicate are called for the first time they success, if called more times they fail.

Not

It checks the **negation** of a predicate:

- **Not/1**: takes as argument a goal, Prolog tries to prove it and then negates the result.

Example:

?- not(is_cointained_in(key,office)).
false

Fail

- **fail/0**: It causes the predicate to fail.

It is useful to force backtracking and find more solutions.
It can be used to implement cycles, i.e. checking all the instantiations (by unification) that make a rule true.

Example:
Print all the elements within a list:

print_all(List):- is_contained_in(O,List),write(O),nl,fail.

Assert

- **assert/1**: adds the argument to the knowledge base.
- **asserta/1**: inserts the argument at the beginning of the knowledge base.
- **assertz/1**: appends the argument at the end of the knowledge base.

Assert and **assertz** actually behave the same.

Example:

```
insert_if_mortal(X):-person(X),assert(mortal(X)).
person(socrates).

?- insert_if_mortal(socrates).
true.
?- mortal(X).
X = socrates.
```

Retract

- **retract/1**: removes from the knowledge base the first fact or clause unifying with the argument.

- **retractall/1**: removes from the knowledge base any fact or clause unifying with the argument.

Findall

- **findall/3 (+Template, :Goal, -Bag)**: Create a list of the instantiations Template gets successively on backtracking over Goal and unify the result with Bag. Succeeds with an empty list if Goal has no solutions.

Example:

```
foo(a, b, c).
foo(a, b, d).
foo(b, c, e).
foo(b, c, f).
foo(c, c, g).

?- findall(C, foo(A, B, C), Cs).
Cs = [c, d, e, f, g]
```

Bagof

- **bagof/3 (+Template, :Goal, -Bag)**: Unify **Bag** with the alternatives of **Template**. If Goal has free variables besides the one sharing with Template, **bagof/3** will **backtrack over the alternatives** of these free variables, unifying Bag with the corresponding alternatives of Template. The construct **+Var^Goal** tells **bagof/3** not to bind Var in Goal. **bagof/3 fails** if Goal has no solutions.

Example:

```
foo(a, b, c).
foo(a, b, d).
foo(b, c, e).
foo(b, c, f).
foo(c, c, g).
```

```
?- bagof(C, foo(A, B, C), Cs).
A = a, B = b, C = G308, Cs = [c, d] ;
A = b, B = c, C = G308, Cs = [e, f] ;
A = c, B = c, C = G308, Cs = [g].

4 ?- bagof(C, A^foo(A, B, C), Cs).
A = G324, B = b, C = G326, Cs = [c, d] ;
A = G324, B = c, C = G326, Cs = [e, f, g].
```

Setof

- **setof/3 (+Template, :Goal, -Bag)**: it behaves as bagof but sorts the result to get a sorted list of alternatives without duplicates.

Example:

```
foo(a, b, c).
foo(a, b, d).
foo(b, c, e).
foo(b, c, f).
foo(c, c, g).
```

```
?- setof(C, foo(A, B, C), Cs).
A = a,
B = b,
Cs = [c, d] ;
A = b,
B = c,
Cs = [e, f] ;
A = B, B = c,
Cs = [g].
```

Cut

- **!/0**: Discards all alternatives since entering the predicate in which the cut appears. It **COMMITS** to the clause in which the cut appears and **FORCES** Prolog interpreter to not use backtracking to find alternatives for the goals to the left of cut in the current clause.

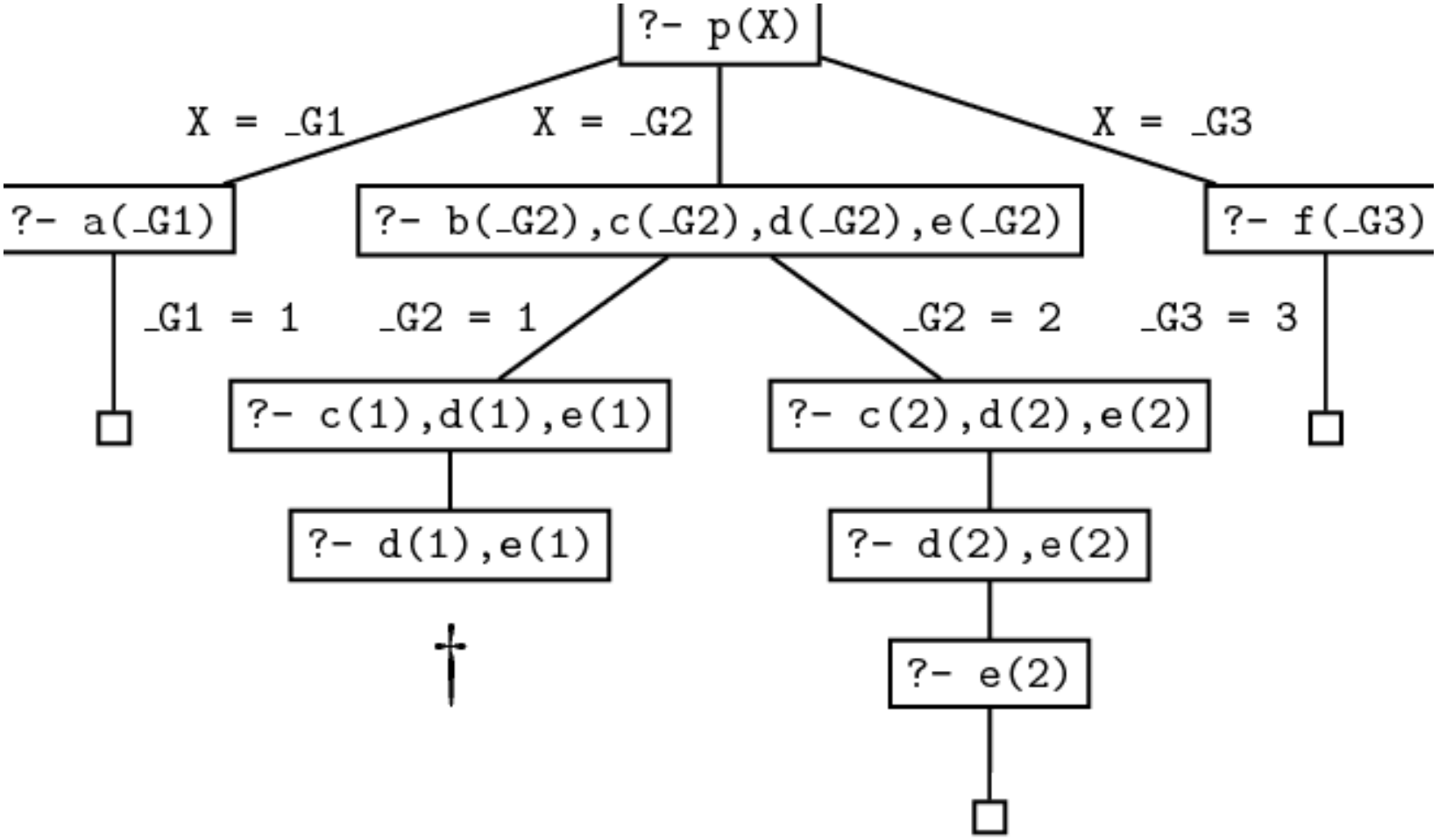
Example:

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).
c(1).
d(2).
e(2).
f(3). b(2). c(2).
```

```
?- p(X).
X = 1 ;
X = 2 ;
X = 3 .
```

it has to backtrack once, namely when it enters the second clause for p/1 and decides to unify the first goal withb(1) instead of b(2) .

Cut


?- p(X)
X = _G1
X = _G2
X = _G3
?- a(_G1)
?- b(_G2),c(_G2),d(_G2),e(_G2)
?- f(_G3)
_G1 = 1
_G2 = 1
_G2 = 2
_G3 = 3
?- c(1),d(1),e(1)
?- c(2),d(2),e(2)
?- d(1),e(1)
?- d(2),e(2)
?- e(2)

Cut

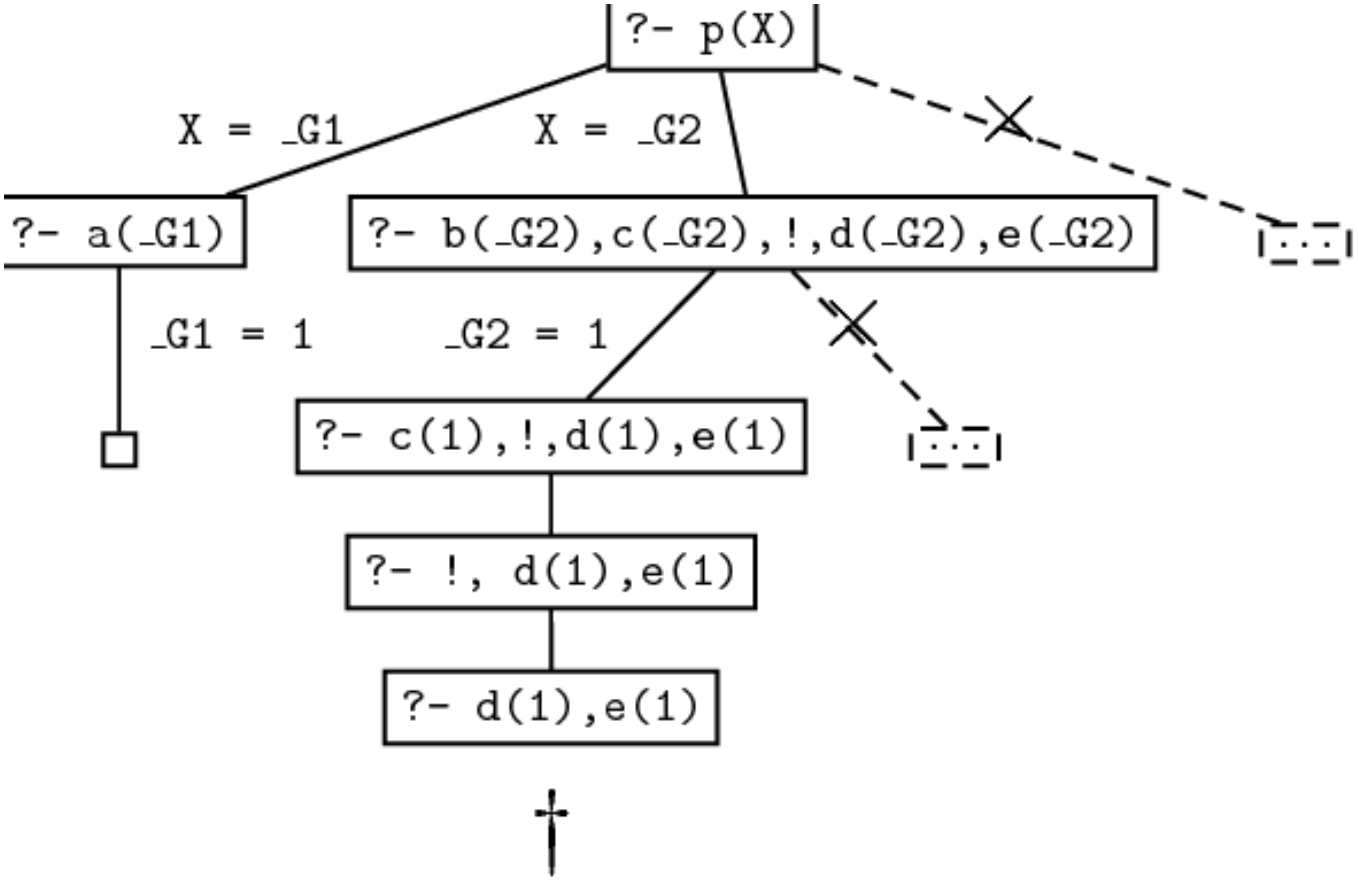
Now we modify the second clause inserting a cut:

Example:

```
p(X):- a(X).
p(X):- b(X), c(X), !, d(X), e(X).
p(X):- f(X).
a(1).
b(1).
c(1).
d(2).
e(2).
f(3). b(2). c(2).
```

```
?- p(X).
X = 1.
```

Cut



Cut

1. **p(X)** is first unified with the first rule, so we get a new goal **a(X)** . By instantiating X to 1 , Prolog **unifies a(X)** with the fact **a(1)** and we have found a solution. So far, this is exactly what happened in the first version of the program.
2. We then go on and look for a second solution. **p(X) is unified with the second rule**, so we get the new goals **b(X),c(X),!,d(X),e(X)** . By instantiating X to 1 , Prolog unifies **b(X) with the fact b(1)** , so we now have the goals c(1),!,d(1),e(1) . But **c(1) is in the database** so this simplifies to !,d(1),e(1) .
3. Now for the big change. The ! goal succeeds (as it always does) and **commits us to the choices made so far**. In particular, **we are committed to having X = 1 , and we are also committed to using the second rule.**
4. But **d(1) fails**. And there's **no way we can re-satisfy the goal p(X)** . Sure, if we were allowed to try the value X=2 we could use the second rule to generate a solution (that's what happened in the original version of the program). But we can't do this: **the cut has removed this possibility from the search tree**. And sure, if we were allowed to try the third rule, we could generate the solution X=3 . But we can't do this: once again, the cut has removed this possibility from the search tree.

Cut

N.B. The **cut only commits us to choices made since the parent goal was unified with the left hand side of the clause** containing the cut. For example, in a rule of the form

q:- p1,...,pn, !, r1,...,rm

when we reach the cut it commits us to using this particular clause for q and it commits us to the choices made when evaluating p1,...,pn .

However, **we are free to backtrack among the r1,...,rm** and **we are also free** to backtrack among **alternatives** for choices that were **made before reaching the goal q** .

Cut – More complete Example

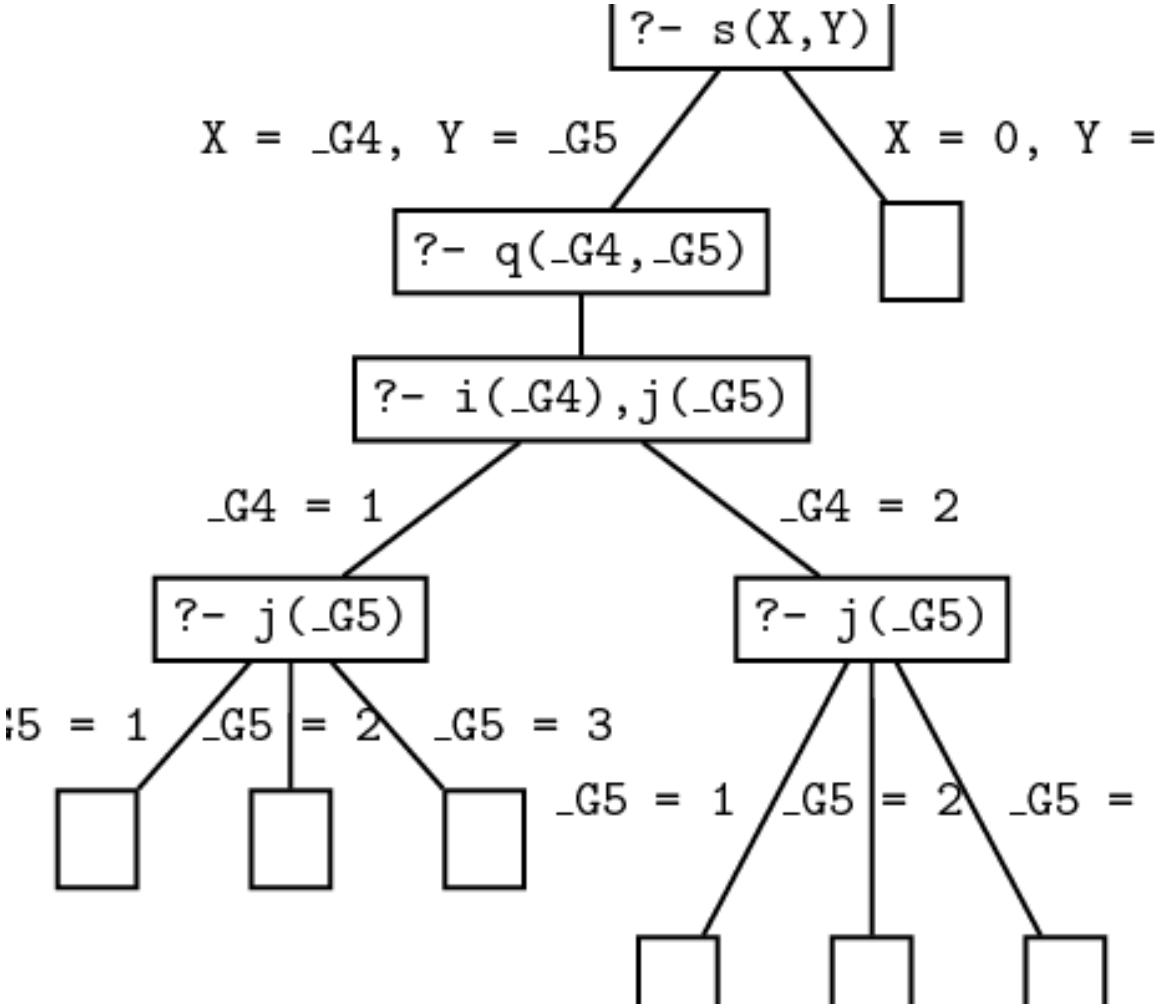
Consider the following cut-free program:

```
s(X,Y):- q(X,Y).
s(0,0).

q(X,Y):- i(X), j(Y).

i(1).
i(2).

j(1).
j(2).
j(3).
```

Cut – More complete Example

?- s(X,Y).

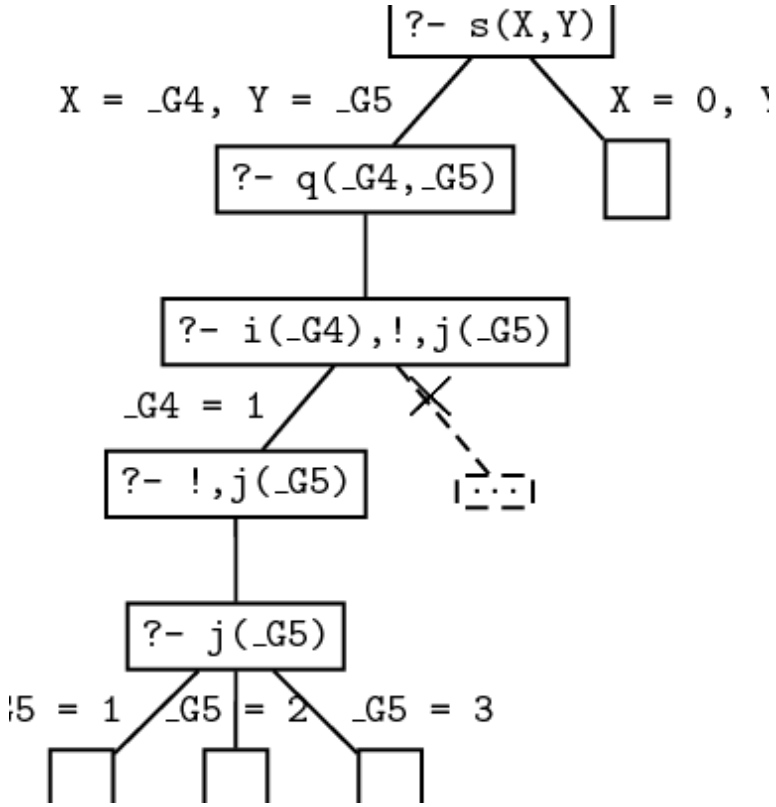
X = 1
Y = 1 ;
X = 1
Y = 2 ;
X = 1
Y = 3 ;
X = 2
Y = 1 ;
X = 2
Y = 2 ;
X = 2
Y = 3 ;
X = 0
Y = 0.



Cut – More complete Example

Consider the following program:

```
s(X,Y):- q(X,Y).
s(0,0).

q(X,Y):- i(X), !, j(Y).

i(1).
i(2).

j(1).
j(2).
j(3).
```




```
?- s(X,Y).

X = 1
Y = 1 ;

X = 1
Y = 2 ;

X = 1
Y = 3 ;

X = 0
Y = 0.
```

Cut – More complete Example

1. s(X,Y) is **first unified with the first rule**, which gives us a new goal q(X,Y) .
2. q(X,Y) is **then unified with the third rule**, so we get the new goals i(X),!,j(Y) . **By instantiating X to 1 , Prolog unifies i(X) with the fact i(1)** . This leaves us with the goal !,j(Y) . The cut succeeds, and **commits us to the choices made so far**.
3. **But what are these choices? These: that X = 1 , and that we are using this clause. But note: we have not yet chosen a value for Y .**
4. Prolog then goes on, and by instantiating Y to 1 , Prolog unifies j(Y) with the fact j(1) . So we have found a solution.
5. But we can find more. Prolog is free to try another value for Y . So it backtracks and sets Y to 2 , thus finding a second solution. And in fact it can find another solution: on backtracking again, it sets Y to 3 , thus finding a third solution.
6. **But those are all alternatives for j(X) . Backtracking to the left of the cut is not allowed**, so it can't reset X to 2 , so it won't find the next three solutions that the cut-free program found. **Backtracking over goals that were reached before q(X,Y) is allowed however, so that Prolog will find the second clause for s/2 .**

Cut - Example

Write a program, f/2 which take an integer (the level of pollutant in the air) as argument and returns the degree of alert:

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

Suppose we want to ask the query:

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

We will see that prolog tries all the clauses with backtracking, before returning *false*, which is correct but inefficient.

Cut - Example

But the three rules are mutually exclusive, we can prevent backtracking after the body succeed:

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

All alternative solutions for Y will be discarded after we reach the first CUT.

These type of CUT are called **GREEN CUTS**: they do not change the result of the program, but only the *procedural meaning*. They make the program more efficient.

Cut - Example

Let's try another query:

?- f(7,Y).

If rule 1 fails (7 < 3), the next goal will succeed (3 =< 7), since they are alternatives. And this is true also for X < 6, and 6 =< X.

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

If we remove the cut and ask the query **f(1,Y)**, we will have some incorrect solutions. This CUT changes the result of the program. CUTS like this are called **RED CUTS**.

Cut – Another Example

Write a program, max/3, which takes integers as arguments and succeeds if the third argument is the maximum of the first two:

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

The clauses are mutually exclusive, so we rewrite the program:

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

This is a GREEN CUT.

Cut – Another Example

Somebody could point out that the second clause is redundant, so she could propose a more compact (and **wrong**) version:

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

This program is wrong for simple queries like **max(2,3,2).**

```
max(X,Y,Z) :- X =< Y,!, Y = Z.
max(X,_,X).
```

This is a **RED CUT**.

Other Useful Built-in

Tuples

We can indicate pairs of values with the notation:

X/Y

They can be two variables, two values, a variable and a value, etc.

\=

Equivalent to \+ Term1 = Term2, means *Term1 cannot be unified with Term2*