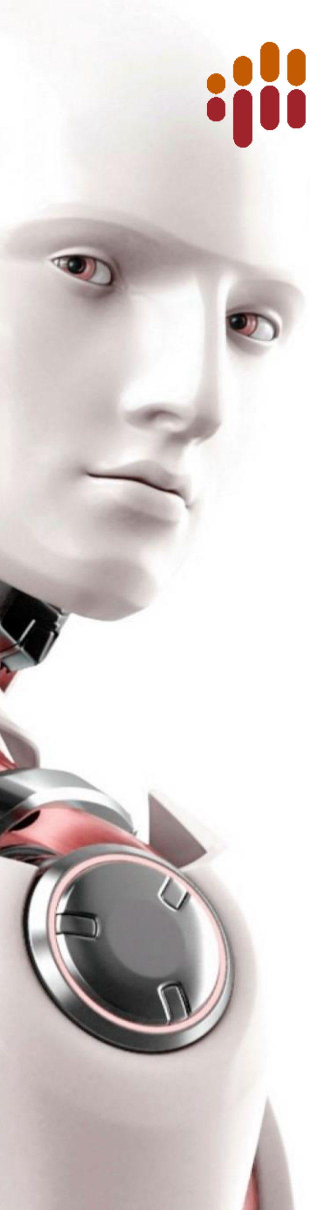
ISLab

Universidade do Minho
Escola de Engenharia
Departamento de Informática

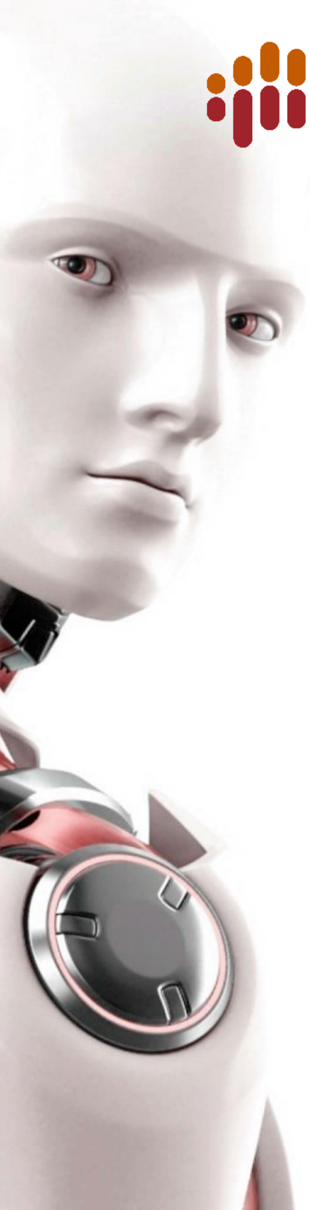
Prolog

Não , Cut, Fail

Mestrado Integrado em Engenharia Informática
Licenciatura em Engenharia Informática
Inteligência Artificial

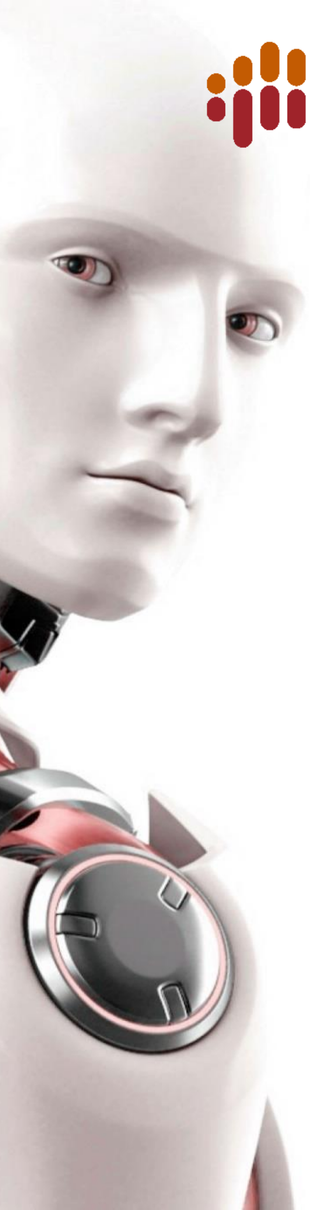


- Backtracking
- The cut operator !
- Negation-as-Failure
- **not**



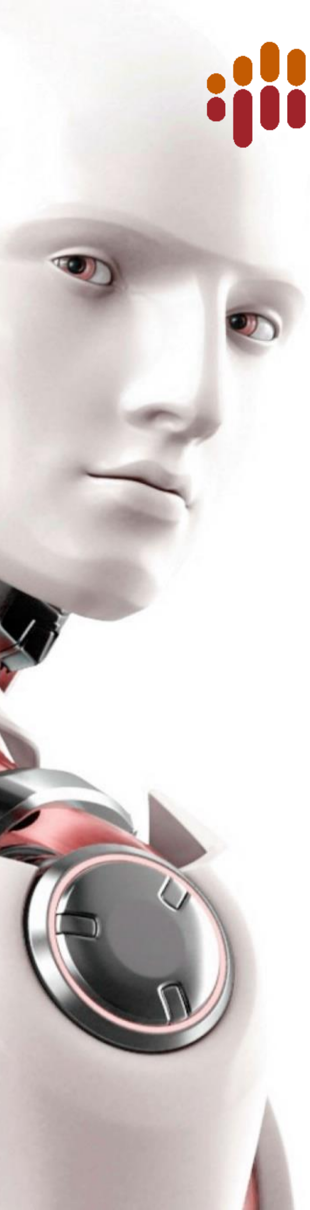
The Cut

- Backtracking is a characteristic feature of Prolog;
- But backtracking can lead to inefficiency:
  - Prolog can waste time and memory exploring possibilities that lead nowhere;



The Cut

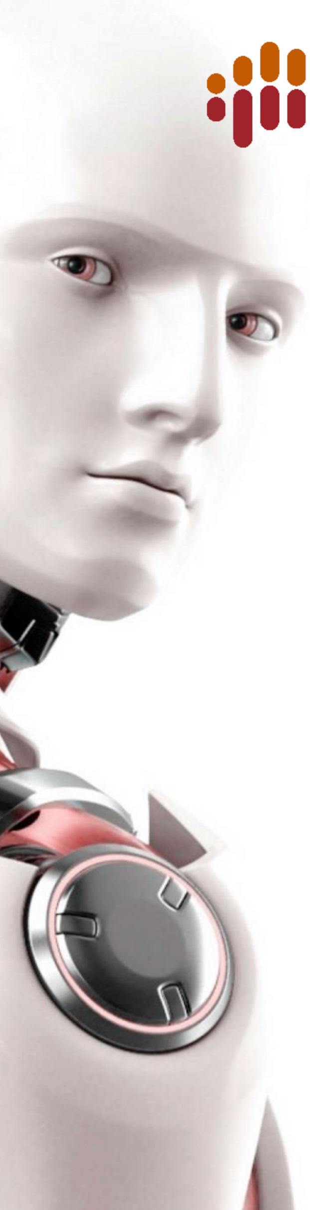
- The cut predicate (!) offers a way to control backtracking;
- The cut has no arguments, so we write (officially): !/0 .



Example of cut

- The cut is a Prolog predicate, we can add it to the body of rules:
  - – Example:
    - p(X):- b(X), c(X), !, d(X), e(X).

- Cut is a goal that always succeeds;
- Cut commits Prolog to the choices that were made since the parent goal was called.



Example of cut

- Cut tells the system that:
  - If you have come this far,
  - Do not backtrack,
  - Even if you fail subsequently.
  - ‘Cut’ written as ‘!’ always succeeds.

Backtracking and Nondeterminism

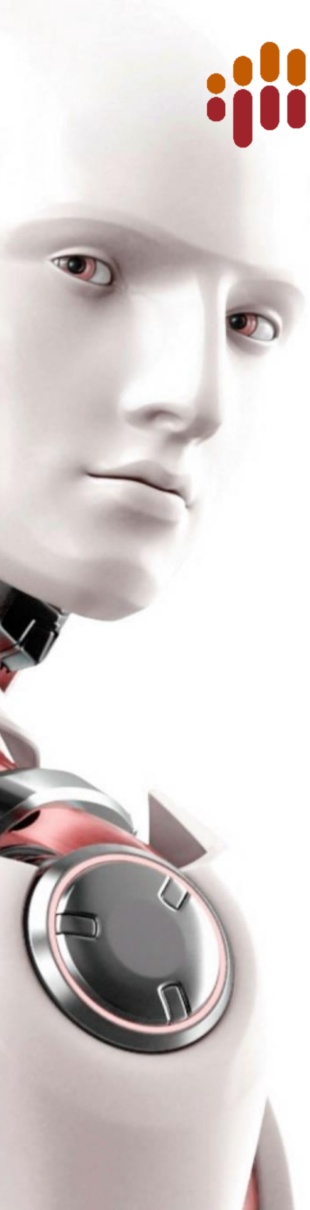
```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

```
?- member(ivo, [joao, ivo, paulo,ivo]).
yes
```

Deterministic query

```
?- member(X, [joao, ivo, paulo, ivo]).
X = joao;
X = ivo;
X = paulo;
X = ivo;
no
```

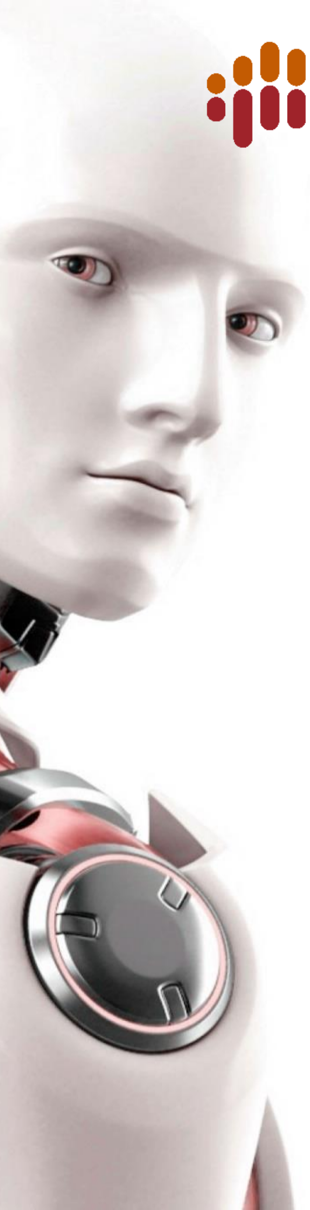
Nondeterministic query



Controlling Backtracking

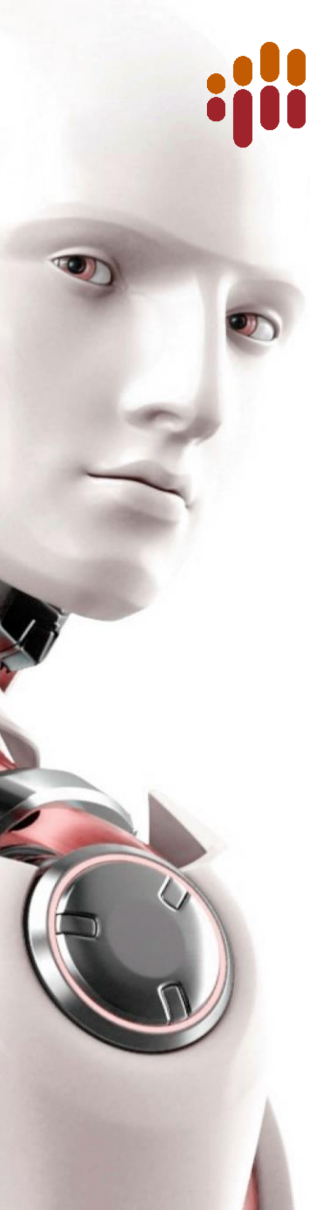
```
cor(cereja, vermelha).
cor(banana, amarela).
cor(maça, vermelha).
cor(maça, verde).
cor(laranja, laranja).
cor(X, desconhecido).
```

```
?- cor(banana, X).
X = amarelo

?- cor(physalis, X).
X = desconhecido

?- cor(cereja, X).
X = vermelho;
X = desconhecido;
no
```



The CUT

- The cut is a built-in predicate written as !
- The cut always succeeds
- When backtracking over a cut, the goal that caused the current procedure to be used fails
- Not used for its logical properties, but
  to control backtracking.



How CUT works

- Suppose goal H is called, and has two clauses:

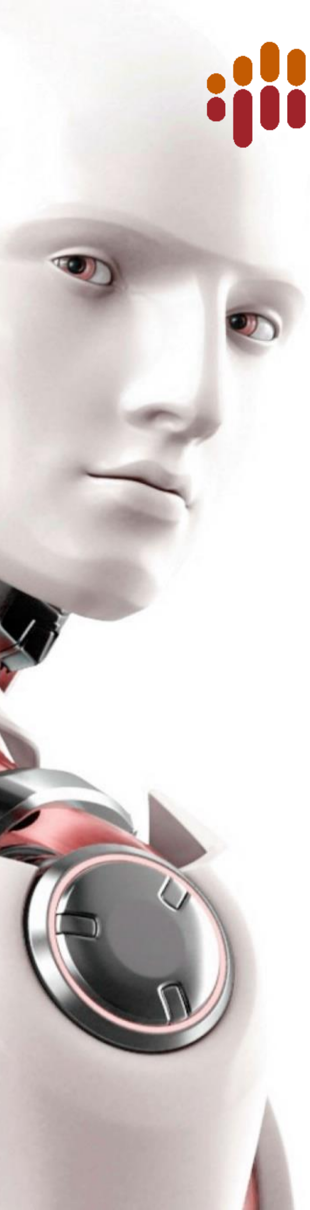
  H1 :- B1,... Bi, !, Bk,... Bm.

  H2 :- Bn,... Bp.

- If H1 matches goals B1...Bi are attempted and may backtrack among themselves

- If B1 fails, H2 will be attempted

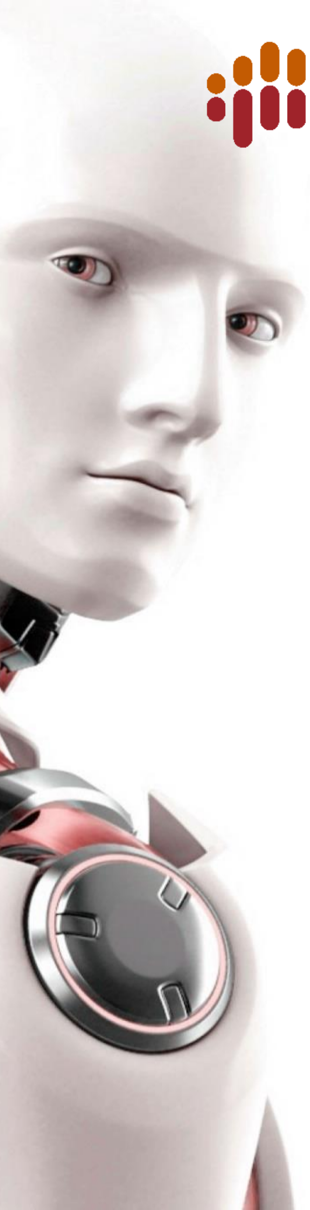
- But as soon as ! is crossed, Prolog commits to the current choice.
  All other choices are discarded.



Commitment to the Clause

H1 :- B1,... Bi, !, Bk,... Bm.
H2 :- Bn,... Bp.

- Goals Bk...Bm may backtrack amongst themselves, but
- If goal Bk fails, then the predicate fails and the subsequent clauses are not matched



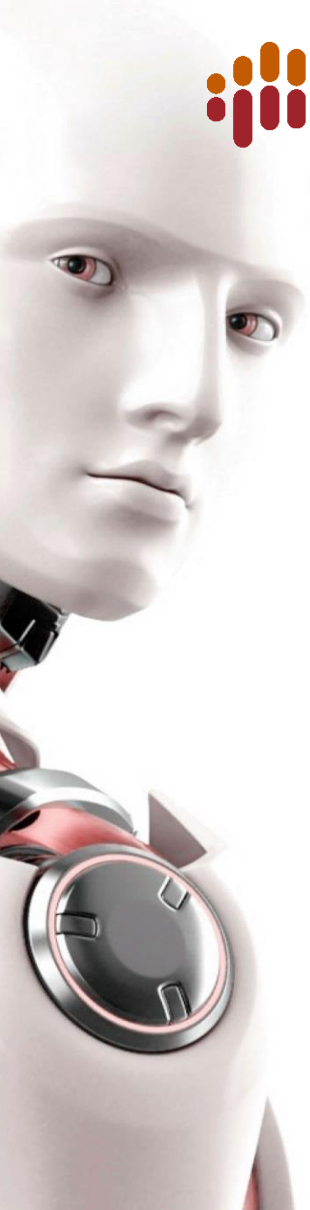
Using Cut

- Consider the following predicate max/3 that succeeds if the third argument is the maximum of the first two

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

```
?- max(2,3,3).
yes

?- max(7,3,7).
yes
```

```
?- max(2,3,2).
no

?- max(2,3,5).
no
```



The max/3 predicate

- What is the problem?
- There is a potential inefficiency
  - Suppose it is called with ?- max(3,4,Y).
  - It will correctly unify Y with 4
  - But when asked for more solutions, it will try to satisfy the second clause. This is completely pointless!

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

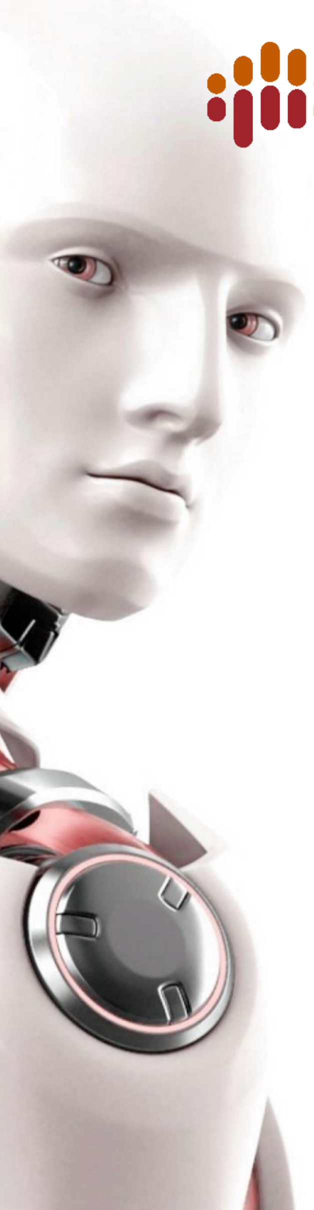
Synthetic Intelligence Lab

max/3 with cut

- With the help of cut this is easy to fix:

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

- Note how this works:
- If the X =< Y succeeds, the cut commits us to this choice, and the second clause of max/3 is not considered
- If the X =< Y fails, Prolog goes on to the second clause



Uses of Cut: Deterministic Predicates

Deterministic (functional) predicate.

Example:

a deterministic version of member, which is more efficient for doing ‘member checking’ because it doesn't need to give multiple solutions:

```
membercheck(X, [X|_]) :- !.
membercheck(X, [_|L]) :- membercheck(X, L).
```

```
?- membercheck(francisco, [joao, jose, francisco, paulo]).
yes.
?- membercheck(X, [a, b, c]).
X = a;
no.
```

ISLab
Synthetic Intelligence Lab

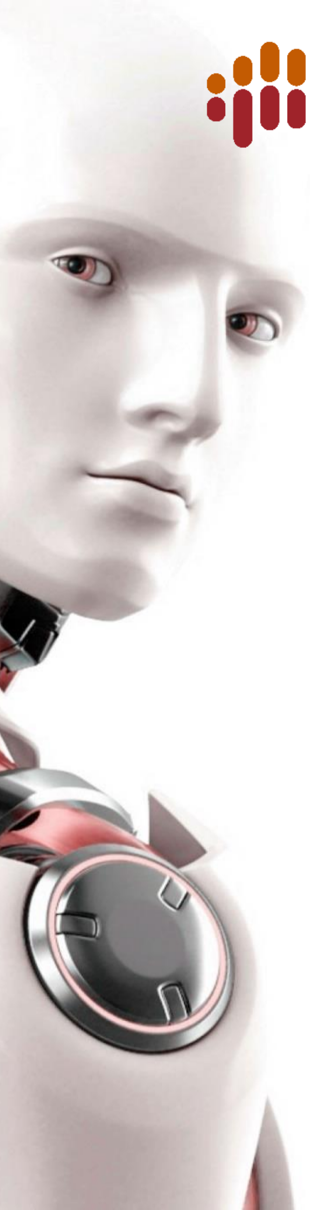
- Using cut together with the built-in predicate fail defines a kind of negation.

- Examples:
  - Maria likes any animals except reptiles:

```
gosta(maria,X) :- reptil(X), !, fail.
gosta(maria,X) :- animal(X).
```

- A utility predicate meaning something like “not equals”:

```
diferente(X, X) :- !, fail.
diferente(_, _).
```



Negation-as-Failure: not

- We can use the idea of “cut fail” to define the predicate not, which takes a term as an argument;
- not “calls” the term, evaluating as if it was a goal:

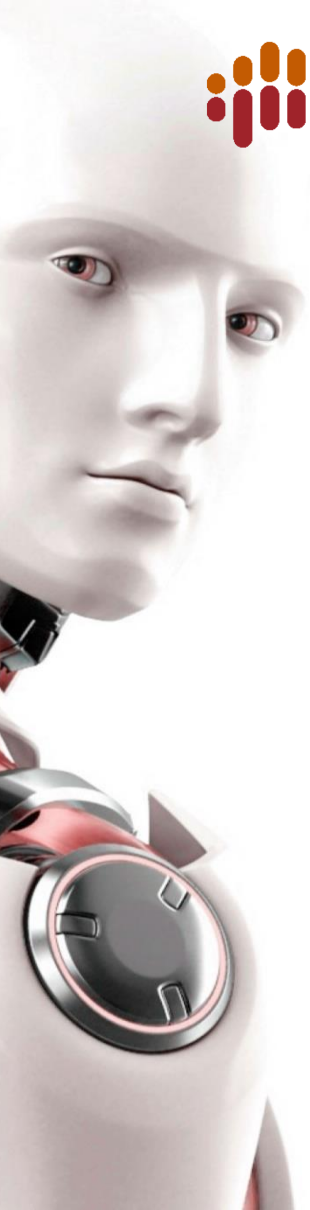
  not(G) fails if G succeeds

  not(G) succeeds if G does not succeed.

- In Prolog,

```
not(G) :- call(G), !, fail.
not(_).
```

- call is a built-in predicate.



Negation-as-Failure: not (cont.)

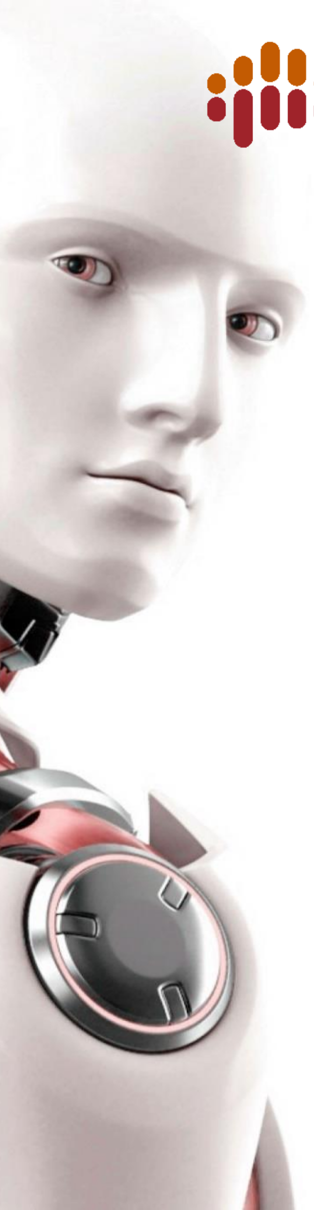
- Most Prolog systems have a built-in predicate not. SWI Prolog calls it \+.

- not does not correspond to logical negation, because it is based on the success/failure of goals.

- It can, however, be useful

  gosta(maria, X) :- not(reptil(X)).

  diferente(X, Y) :- not(X = Y).



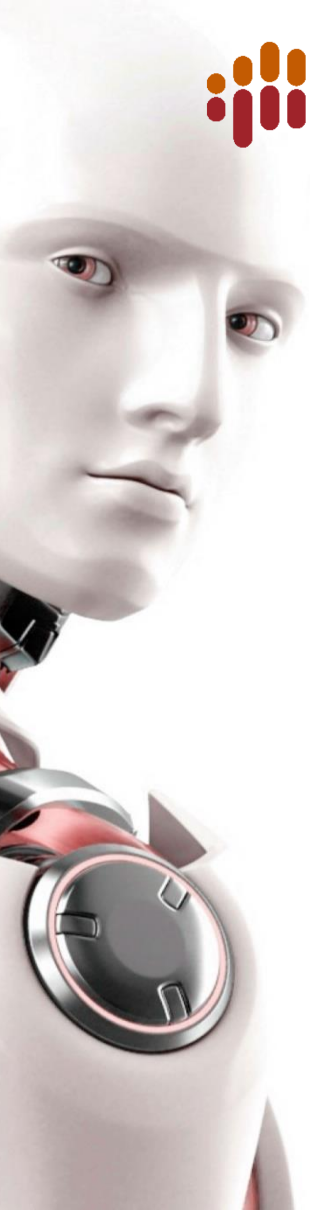
Misleading Negation-as-Failure

- The following database held the names of members of the public, marked by whether they are innocent or guilty of some offence:
- Suppose the database contains the following:

```
inocente(peter_pan).
inocente(X) :- ocupacao(X, freira).
inocente(winnie_the_pooh).
inocente(julie_andrews)
culpado(X) :- ocupacao(X, ladrao).
culpado(joao_facas).
culpado(rosa_carteiras).
```

- Consider the following dialogue:

```
?- inocente(s_francisco).
no.
```



ISLab
Synthetic Intelligence Lab

Problem – No may not mean False

- This can't be right – we know that S. Francisco is innocent;
- Why does this happen?
- Prolog produces no, because S. Francisco is not in the database;
- The user will believe it because the computer says so and the database is hidden from the user;
- How to solve this?

- Using not doesn't help

  culpado(X) :- not(inocente(X)).

- This makes matters even worse

  ```
  ?- culpado(s_francisco).
  yes
  ```

- It is one thing to show that s_francisco cannot be demonstrated to be innocent, but it is very bad to incorrectly show that he is guilty.

Negation-as-Failure can be Non-Logical

- More subtle than the inocente/culpado problem, not can lead to some extremely obscure programming errors.
- An example using a restaurant database:
    - boa_pontuacao(boa_mesa).
    - bom_standard(tia_carla).
    - caro(boa_mesa).
    - razoavel(R) :- not(caro(R)).
- Consider the query:

```
?- bom_standard(X), razoavel(X).
X = tia_carla
```

- But let's ask the logically equivalent question:

```
?- razoavel(X), bom_standard(X).
no.
```

Why Different Answers?

- Why different answers for logically equivalent queries?

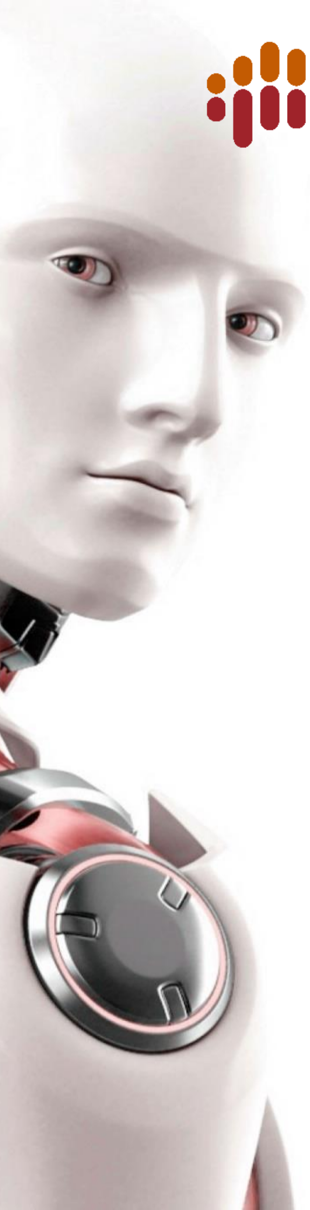
  ?- bom_standard(X), razoavel(X).

  ?- razoavel(X), bom_standard(X).

- In the 1st query, X is always instantiated when razoavel(X) is executed;

- In the 2nd query, X is not instantiated when razoavel(X) is executed;

- The semantics of razoavel(X) differ depending on whether its argument is instantiated!



Bad Programming Practice

- It is bad to write programs that destroy the correspondence between the logical and procedural meaning of a program without any good reason;

- Negation-as-failure does not correspond to logical negation, and so requires special care.

ISLab
Synthetic Intelligence Lab

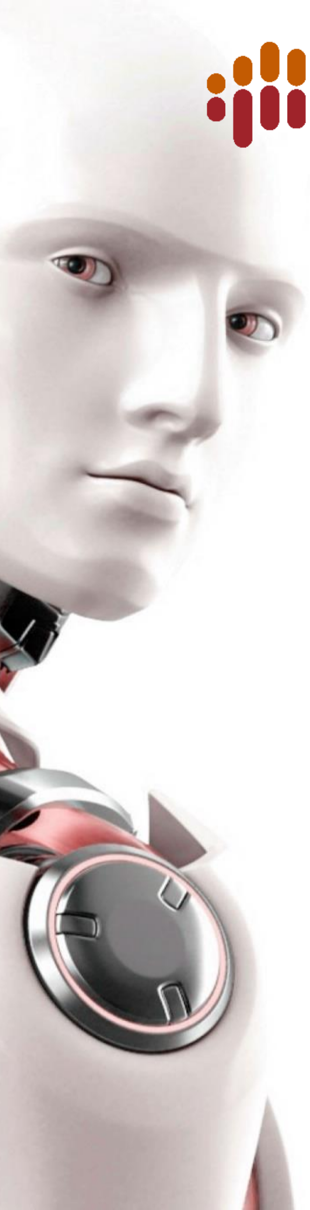
How to fix it?

- One way is to specify that:

  Negation of a non-ground formula is undefined

- A formula is ground if it has no unbound variables;

- Some Prolog systems issue a run-time exception when a non-ground goal is negated .



What the cut does

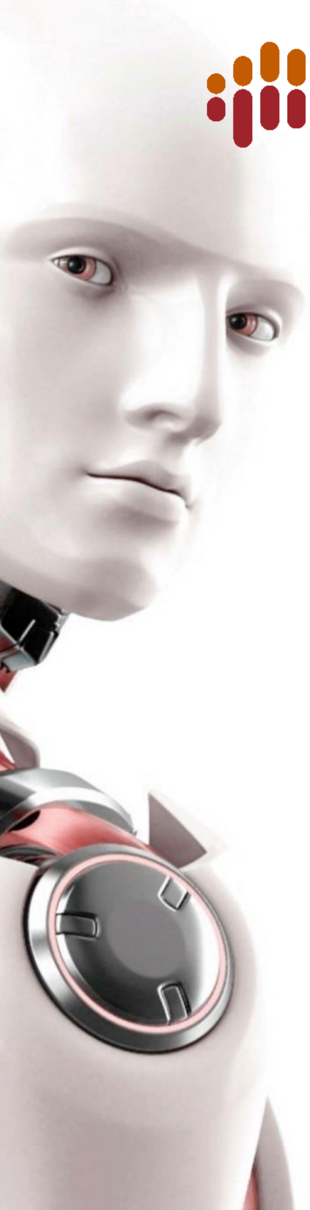
- The cut only commits us to choices made since the parent goal was unified with the left-hand side of the clause containing the cut;
- For example, in a rule of the form

  q:- p1, ... , pm, !, r1, ... , rn.

  when we reach the cut it commits us:
  - to this particular clause of q
  - to the choices made by p1, ... , pm
  - NOT to choices made by r1, ... , rn

Green Cuts

- Cuts that do not change the meaning of a predicate are called green cuts;
- The cut in max/3 is an example of a green cut:
  - the new code gives exactly the same answers as the old version,
  - but it is more eficient.



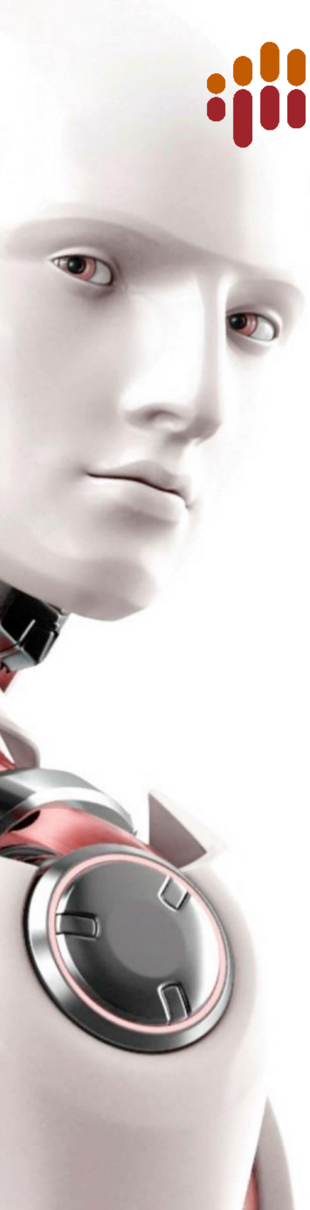
ISLab
Synthetic Intelligence Lab

Another max/3 with cut

- Why not remove the body of the second clause? After all, it is redundant.

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

- How good is it?



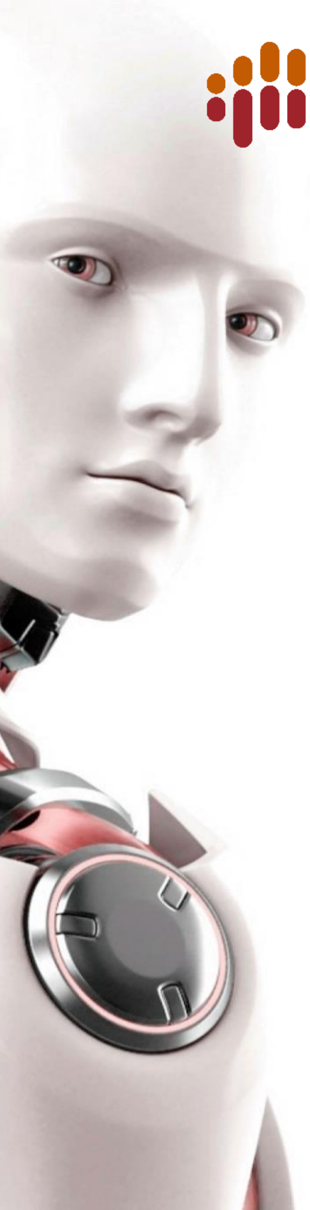
Another max/3 with cut

- Why not remove the body of the second clause? After all, it is redundant.

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

- How good is it?
  - ok

```
?- max(200,300,X).
X=300
yes
```



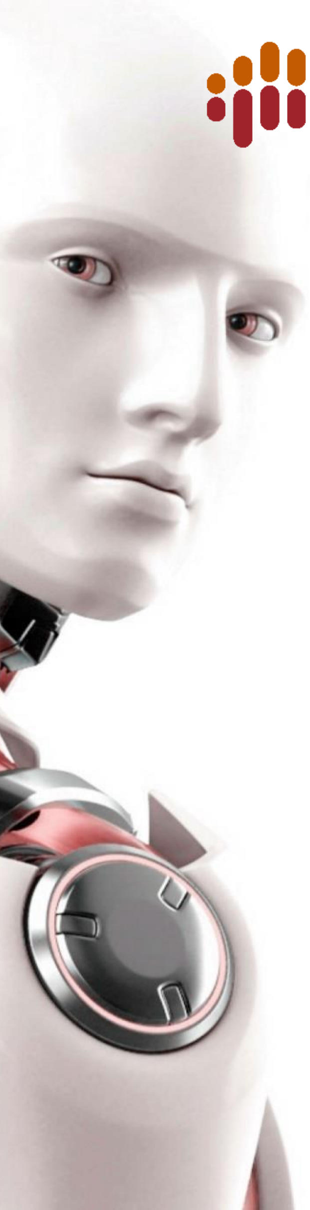
Another max/3 with cut

- Why not remove the body of the second clause? After all, it is redundant.

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

```
?- max(400,300,X).
X=400
yes
```

- How good is it?
- – ok



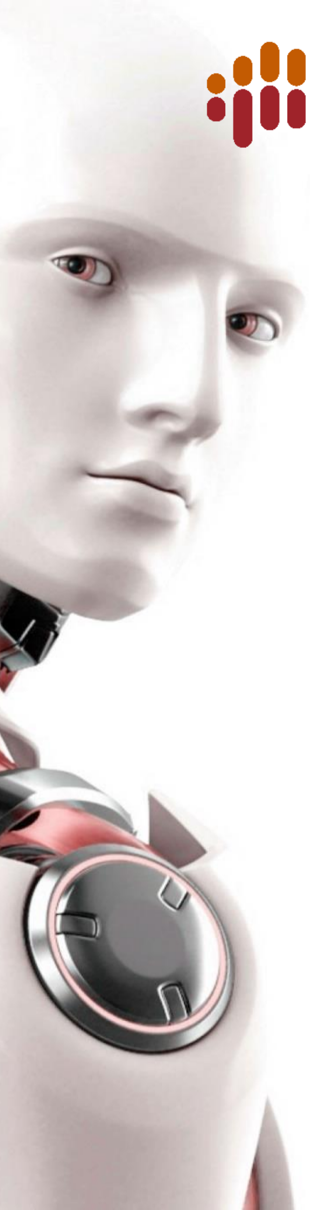
Another max/3 with cut

- Why not remove the body of the second clause? After all, it is redundant.

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

- How good is it?
  - – oops....

```
?- max(200,300,200).
yes
```

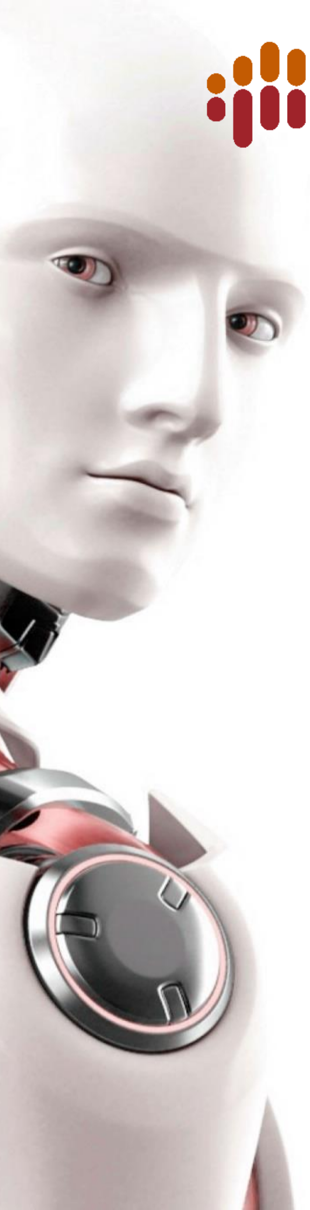


ISLab
Synthetic Intelligence Lab

Revised max/3 with cut

- Unification after crossing the cut
  - max(X,Y,Z):- X =< Y, !, Y=Z.
  - max(X,Y,X).
- This does work

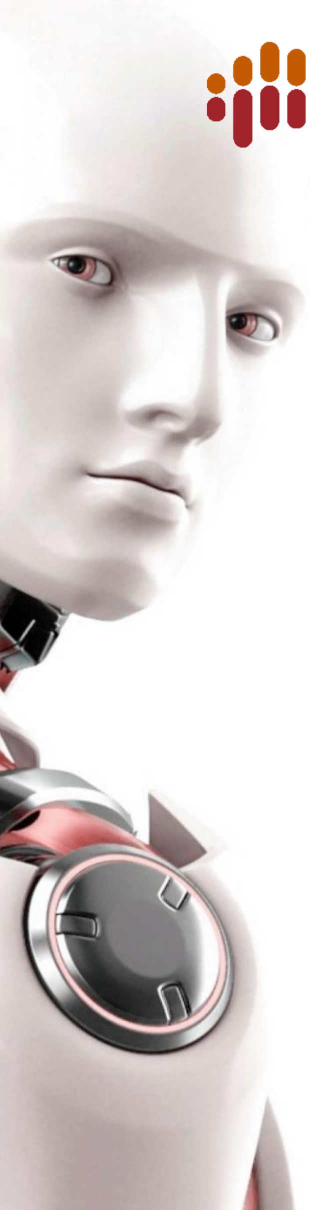
```
?- max(200,300,200).
no
```



ISLab
Synthetic Intelligence Lab

Red Cuts

- Cuts that change the meaning of a predicate are called red cuts;
- The cut in the revised max/3 is an example of a red cut:
  - If we take out the cut, we don’t get an equivalent program;
- Programs containing red cuts
  - Are not fully declarative;
  - Can be hard to read;
  - Can lead to subtle programming mistakes.



Another build-in predicate: fail/0

- As the name suggests, this is a goal that will immediately fail when Prolog tries to proof it;
- That may not sound too useful…
- But remember:
  - when Prolog fails, it tries to backtrack.

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Prolog

Não , Cut, Fail

Mestrado Integrado em Engenharia Informática
Licenciatura em Engenharia Informática
Inteligência Artificial