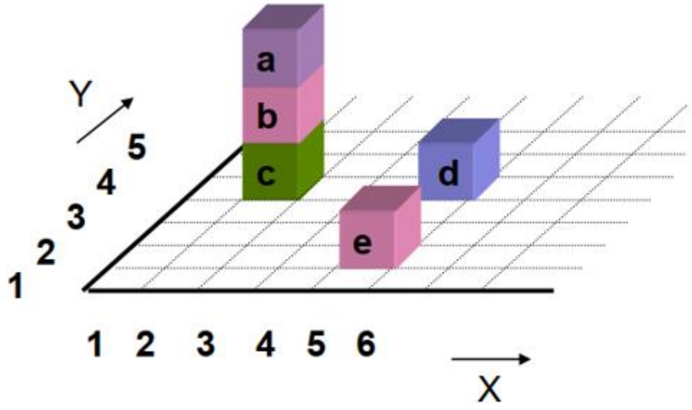
# Prolog

INTRODUZIONE 2

# AN EXAMPLE PROGRAM

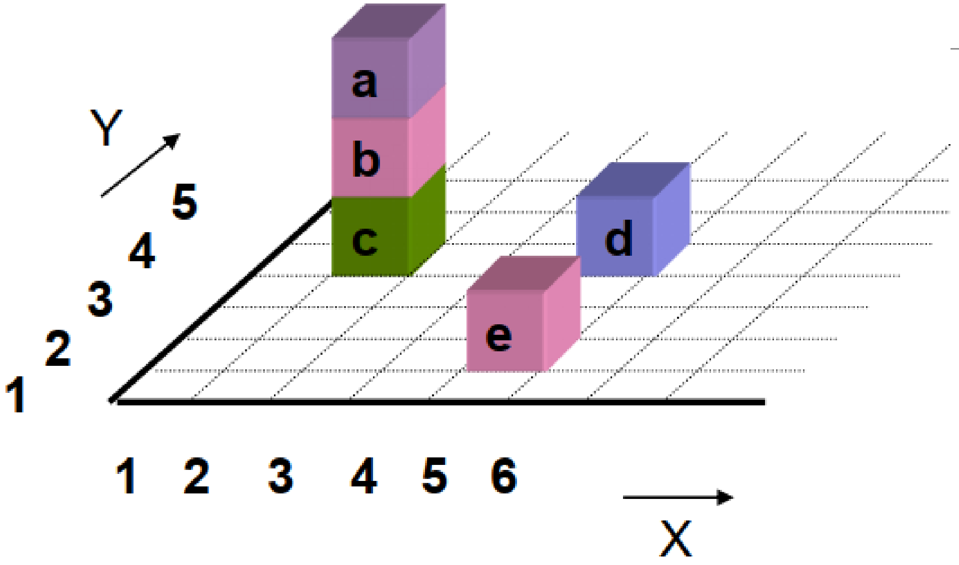
Esiste un robot che vuole manipulare dei blocchi su una tavola

- Robot **can see** blocks by a camera mounted on ceiling
- Robot wants to know **blocks' coordinates**, whether a block is graspable (nothing on top), etc.



# ROBOT'S WORLD

```
% see( Block, X, Y)
see( a, 2, 5).  % Block a seen at (2,5)
see( d, 5, 5).
see( e, 5, 2).
```



```
% on( Block, BlockOrTable)
on( a, b).
on( b, c).
on( c, table).
on( d, table).
on( e, table).
```

# INTERACTION WITH ROBOT PROGRAM

Start Prolog interpreter

```
?- [robot].                % Load file robot.pl
File robot consulted
?- see( a, X, Y).          % Where do you see block a
X = 2
Y = 5
?- see( Block, _, _).     % Which block(s) do you see?
Block = a;                 % More answers?
Block = d;
Block = e;
no
```

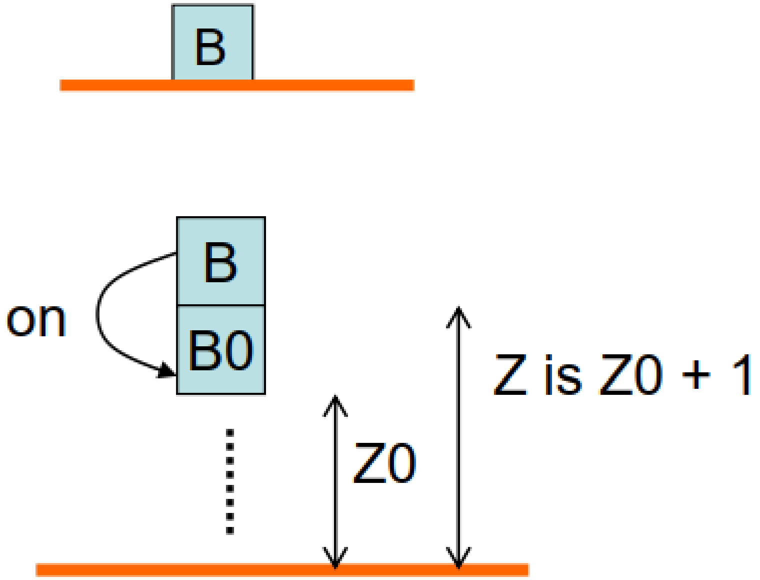
# INTERACTION, CTD.

```
?-  see( B1, _, Y),   see( B2, _, Y).     % Blocks at same Y?

% Prolog's answers may surprise!

% Perhaps this was intended:
?- see( B1, _, Y),   see( B2, _, Y),   B1 \== B2.
```

# Vogliamo definire la coordinata Z

```
z(Blocco,0):-on(Blocco,table).


z(Blocco,Z):-on(Blocco,AltroBlocco),
                        z(AltroBlocco,Zaltro),
                        Z is Zaltro+1.
```

Recursive predicate



**Blocco uguale B**
**AltroBlocco  uguale**
**Zaltro  uguale a Z0**

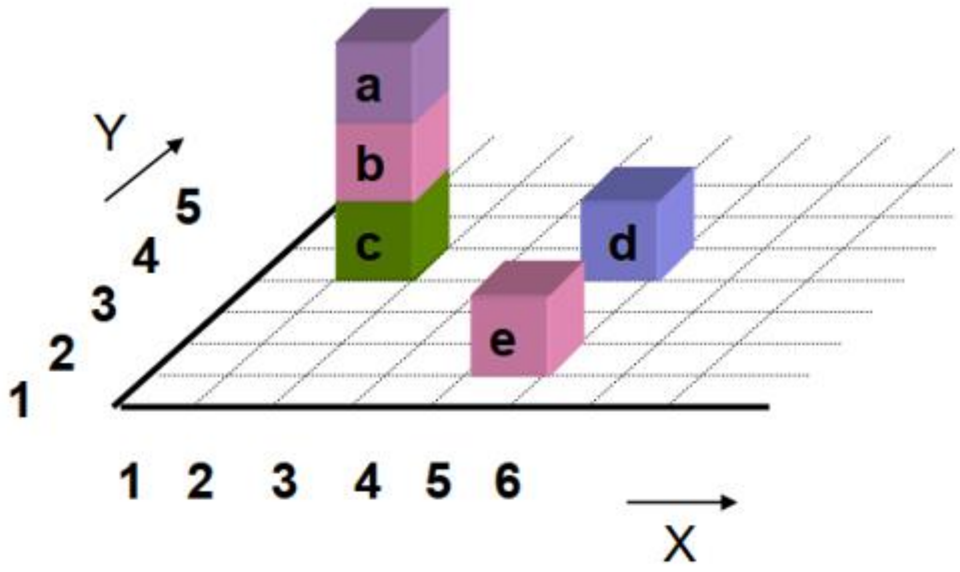
# Vogliamo definire la coordinata Z

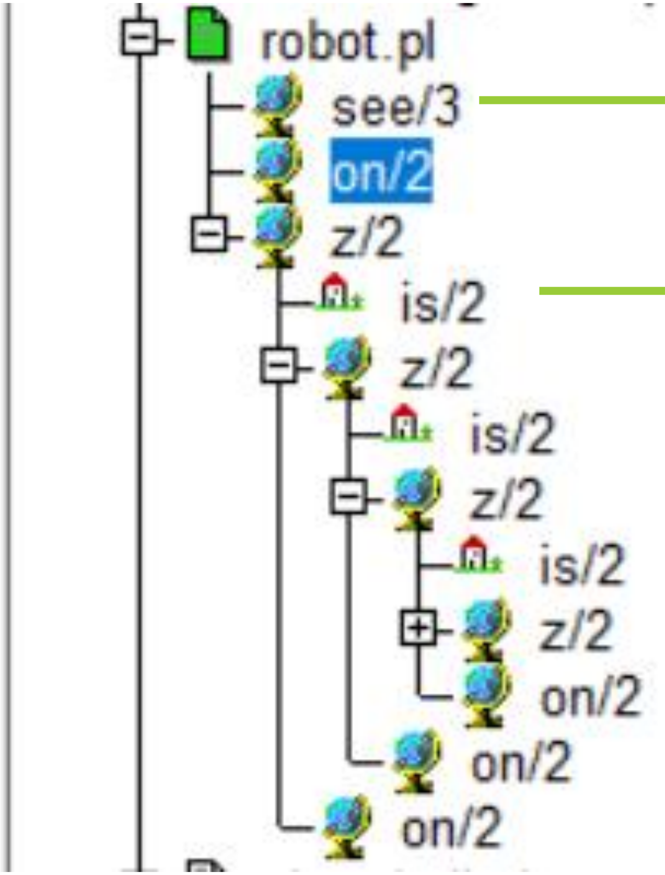
```
z(c,Altezza).

z(Blocco,Zaltro+1):-on(Blocco,AltroBlocco),
                         z(AltroBlocco,Zaltro).
```

Esempio  z(a,Altezza).
z=0+1+1    Result=2.    %Prolog constructs a formula

# Esercizi

- ?- z( c, Z).
- ?- z( a, Z).
- ?- z(c,1).
- ?- z(b,1).





Global predicate see/3

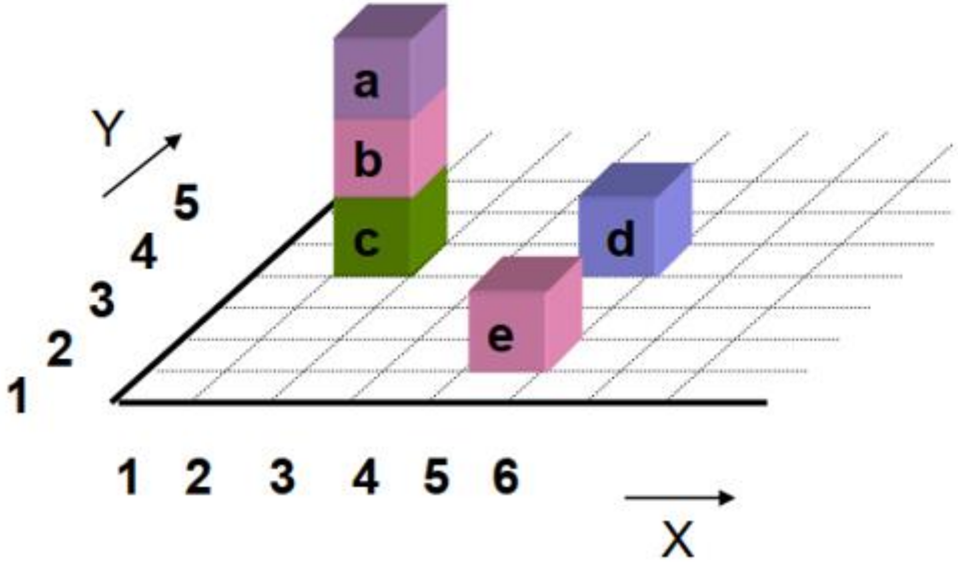
Built-in predicate is/2

Browse->prolog navigator

# X-Y coordinates of a block

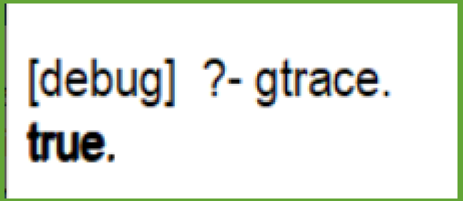
Il robot non puo vedere I blocchi **b,c** Sappiamo soltanto che **a** e sopra **b** e **b** e sopra **c** e questi due hanno le stesse coordinate come **a.**

```
xy(B,X,Y):-see(B,X,Y).

xy(B,X,Y):-on(B0,B),
                    xy(B0,X,Y).
```

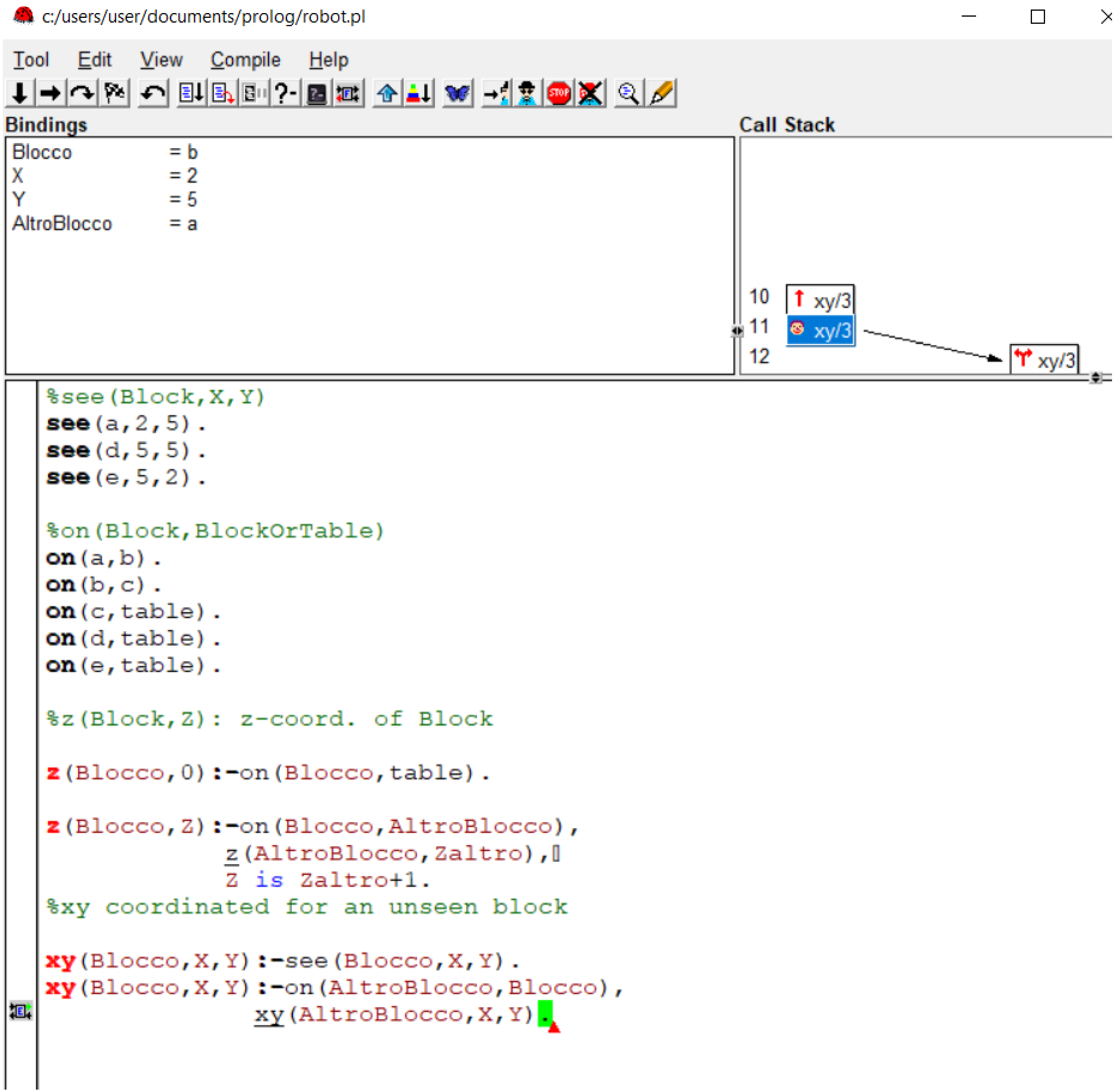


# gtrace command

# Graphical trace



```
[debug]  ?- gtrace.
true.
```



```
Blocco        = b
X             = 2
Y             = 5
AltroBlocco   = a
```

```
%see(Block,X,Y)
see(a,2,5).
see(d,5,5).
see(e,5,2).

%on(Block,BlockOrTable)
on(a,b).
on(b,c).
on(c,table).
on(d,table).
on(e,table).

%z(Block,Z): z-coord. of Block

z(Blocco,0):-on(Blocco,table).

z(Blocco,Z):-on(Blocco,AltroBlocco),
            z(AltroBlocco,Zaltro),▯
            Z is Zaltro+1.
%xy coordinated for an unseen block

xy(Blocco,X,Y):-see(Blocco,X,Y).
xy(Blocco,X,Y):-on(AltroBlocco,Blocco),
              xy(AltroBlocco,X,Y).
```

# Graphical trace

**Source code**

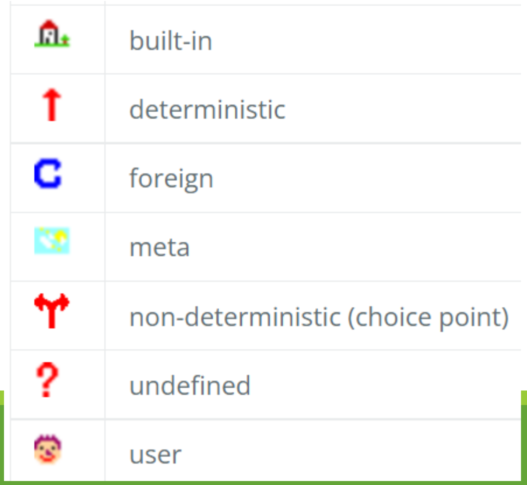
The current location in the source code is displayed in a window displaying the actual source code or, if the clause is asserted, in a window displaying the decompiled predicate. Colours are used to indicate the status, **green meaning normal forward calling, red failure, yellow redo and purple exception.**

**Bindings**

Local variables of the selected frame. Variables are indicated by their true name. A concise display, clearly indicating which variables share the same value and removing unbound variables, is provided. Values can be examined by double-clicking.

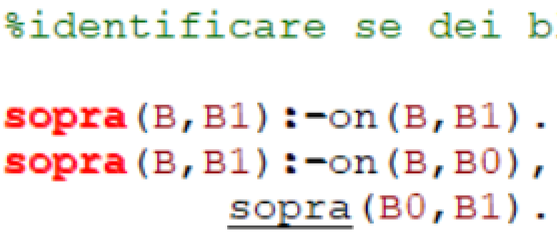
**Stack**

The stack-view not only provides the call-stack,
but also the choice-point chain.
The latter is notably useful to detect (undesired) non-determinism.

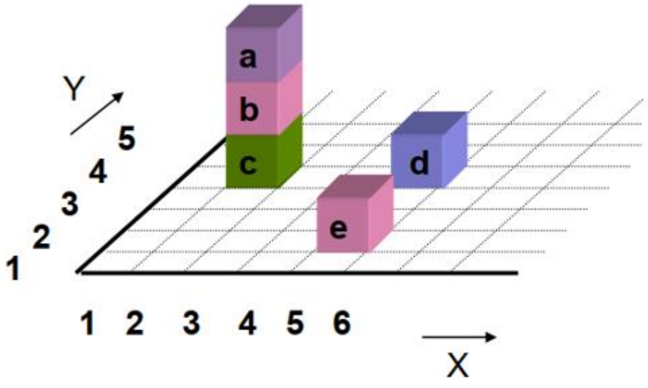


| | |
|---|---|
| 🏠 | built-in |
| ↑ | deterministic |
| C | foreign |
| ▨ | meta |
| ⑂ | non-deterministic (choice point) |
| ? | undefined |
| ☺ | user |

# Verticale Z



```
%identificare se dei blocchi sono stessa Z coordinate

sopra(B,B1):-on(B,B1).
sopra(B,B1):-on(B,B0),
          sopra(B0,B1).
```

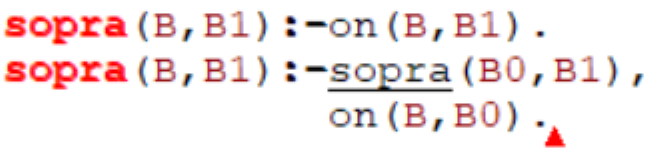


1.Trovate se c'i sonno dei blocchi sopra il blocco **c**?
**sopra(X,c). (usando la gtrace.)**

2.Trovate se c'i sonno dei blocchi sopra il blocco **e**?
**sopra(X,e). (usando la gtrace.)**

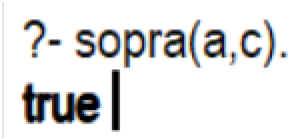
3.**sopra(e,X). sopra(a,Y).**

## Cambiare l’ordine dei goal su una clausa



```
sopra(B,B1):-on(B,B1).
sopra(B,B1):-sopra(B0,B1),
             on(B,B0).
```

Clause

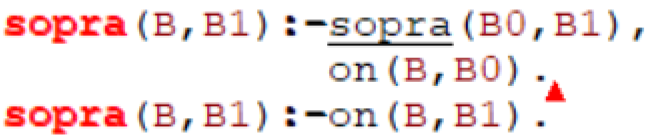


```
?- sopra(a,c).
true
```

Risultato stesso

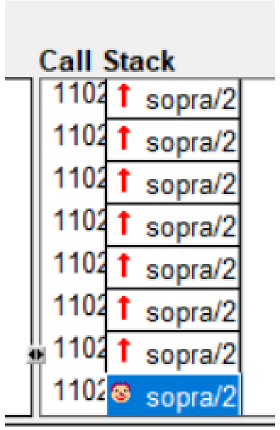
Anticipare la chiamata recursive all’interno della clausa recursive non e una buona idea.

## Cambiare l’ordine delle clause.



```
sopra(B,B1):-sopra(B0,B1),
             on(B,B0).
sopra(B,B1):-on(B,B1).
```

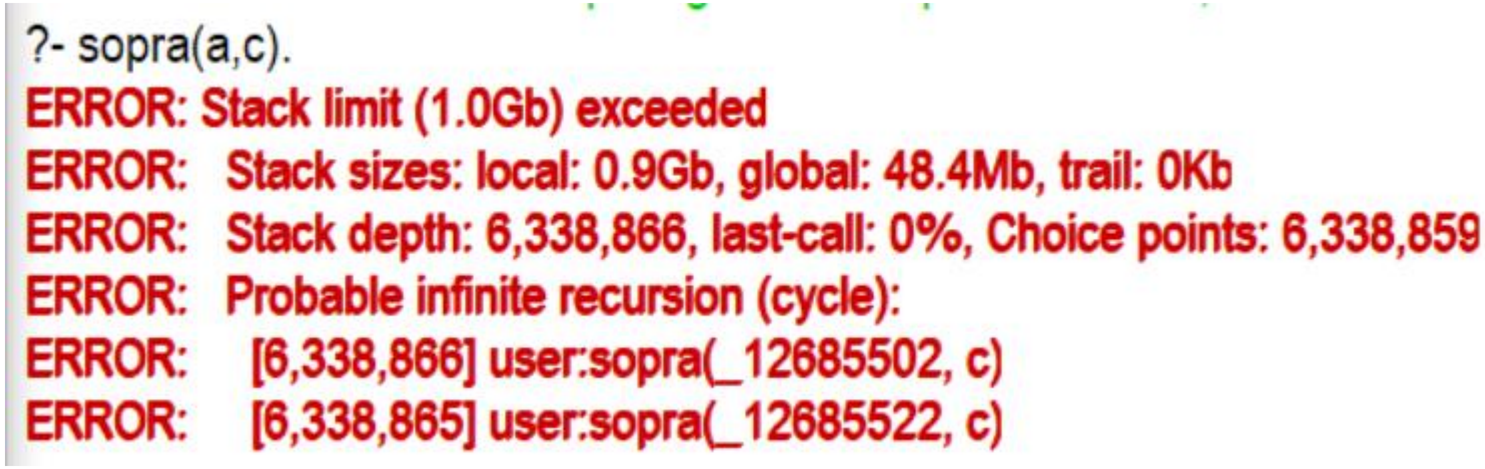
Clause



```
Call Stack
1102 sopra/2
1102 sopra/2
1102 sopra/2
1102 sopra/2
1102 sopra/2
1102 sopra/2
1102 sopra/2
1102 sopra/2
```

Ciclo infinito

# Cambiare l’ordine delle clause.-errore



```
?- sopra(a,c).
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.9Gb, global: 48.4Mb, trail: 0Kb
ERROR:   Stack depth: 6,338,866, last-call: 0%, Choice points: 6,338,859
ERROR:   Probable infinite recursion (cycle):
ERROR:     [6,338,866] user:sopra(_12685502, c)
ERROR:     [6,338,865] user:sopra(_12685522, c)
```

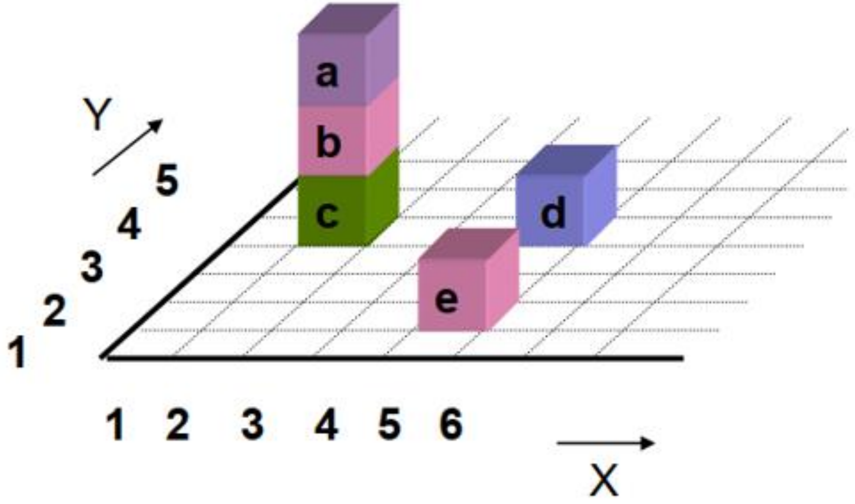
# DECLARATIVE vs PROCEDURAL MEANING

- A & B is logically equal to B & A
- Declarative meaning of Prolog program = logical meaning
- Order of goals in clauses does not affect declarative meaning

- Procedural meaning of Prolog = algorithm for searching for proof
- Order of goals and clauses does affect search for proof

# Progetto (prossima settimana)

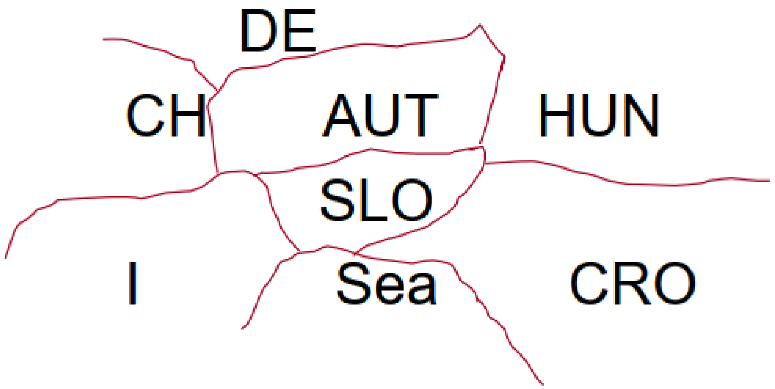
Create le clause necessarie
per **identificare il numero dei blocchi**
che sonno sopra un blocco esistente?



# Map coloring

Problem: Given a map, color the countries in the map

- Theorem: Four colors suffice to color any map
- Example map:



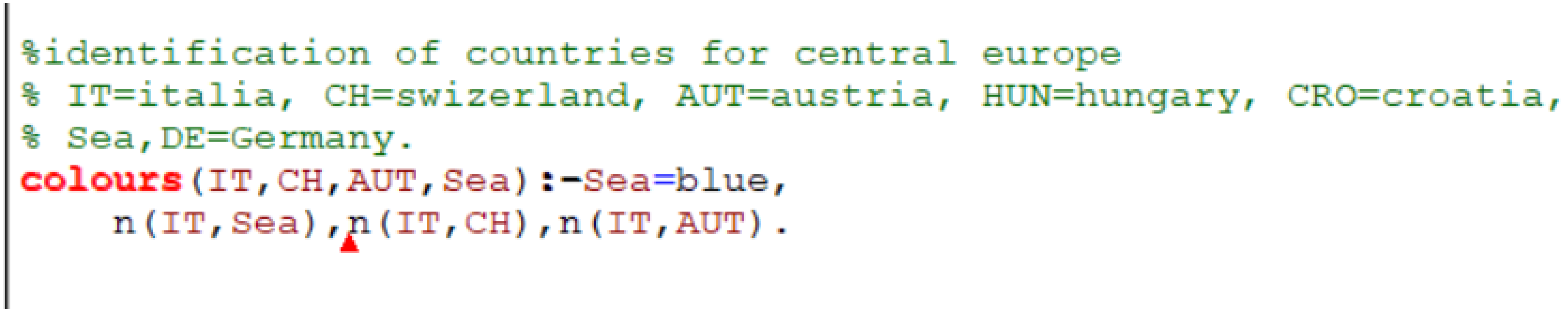
% Possible pairs of colors of neighbour countries

```
n( red, green).     n( red, blue).      n( red, yellow).
n( green, red).     n( green, blue).    n( green, yellow).
n( blue, red).      n( blue, green).    n( blue, yellow).
n( yellow, red).    n( yellow, green).  n( yellow, blue).
```

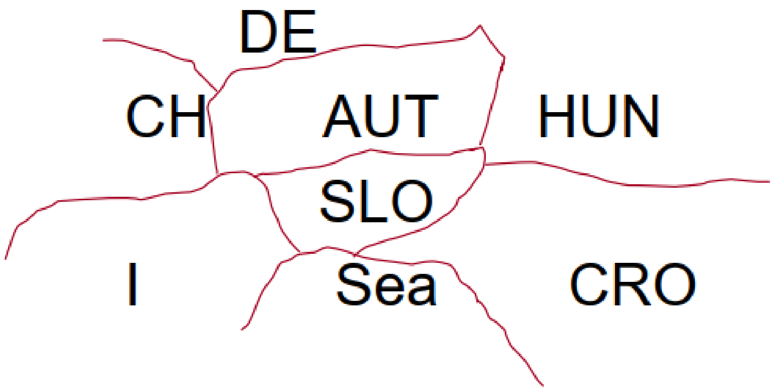
- Neighbour countries:

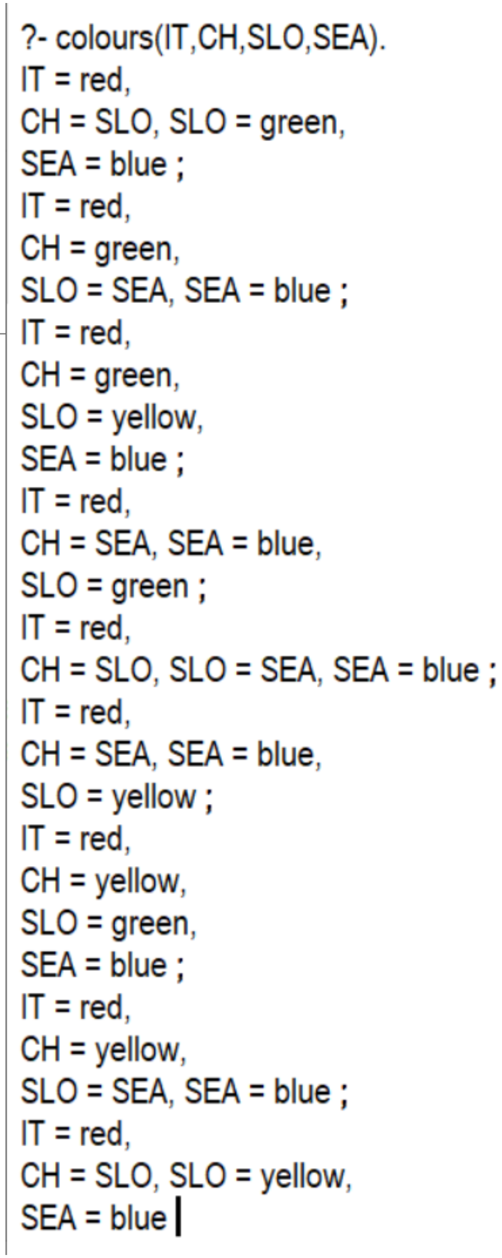
n( I, SLO). n( I, Sea). n( Sea, SLO). …

# Map coloring



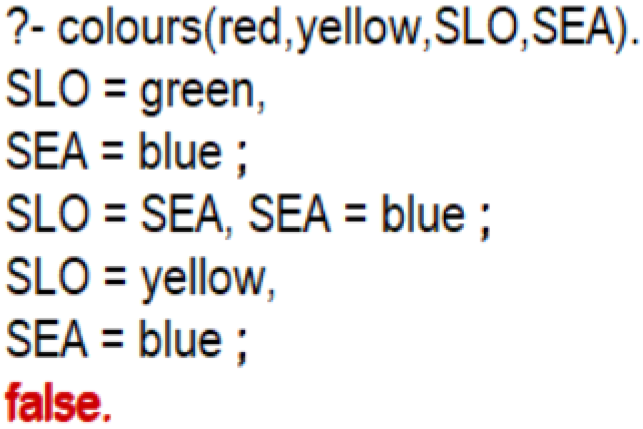
```
%identification of countries for central europe
% IT=italia, CH=swizerland, AUT=austria, HUN=hungary, CRO=croatia,
% Sea,DE=Germany.
colours(IT,CH,AUT,Sea):-Sea=blue,
    n(IT,Sea),n(IT,CH),n(IT,AUT).
```





```
?- colours(IT,CH,SLO,SEA).
IT = red,
CH = SLO, SLO = green,
SEA = blue ;
IT = red,
CH = green,
SLO = SEA, SEA = blue ;
IT = red,
CH = green,
SLO = yellow,
SEA = blue ;
IT = red,
CH = SEA, SEA = blue,
SLO = green ;
IT = red,
CH = SLO, SLO = SEA, SEA = blue ;
IT = red,
CH = SEA, SEA = blue,
SLO = yellow ;
IT = red,
CH = yellow,
SLO = green,
SEA = blue ;
IT = red,
CH = yellow,
SLO = SEA, SEA = blue ;
IT = red,
CH = SLO, SLO = yellow,
SEA = blue
```

# Map coloring

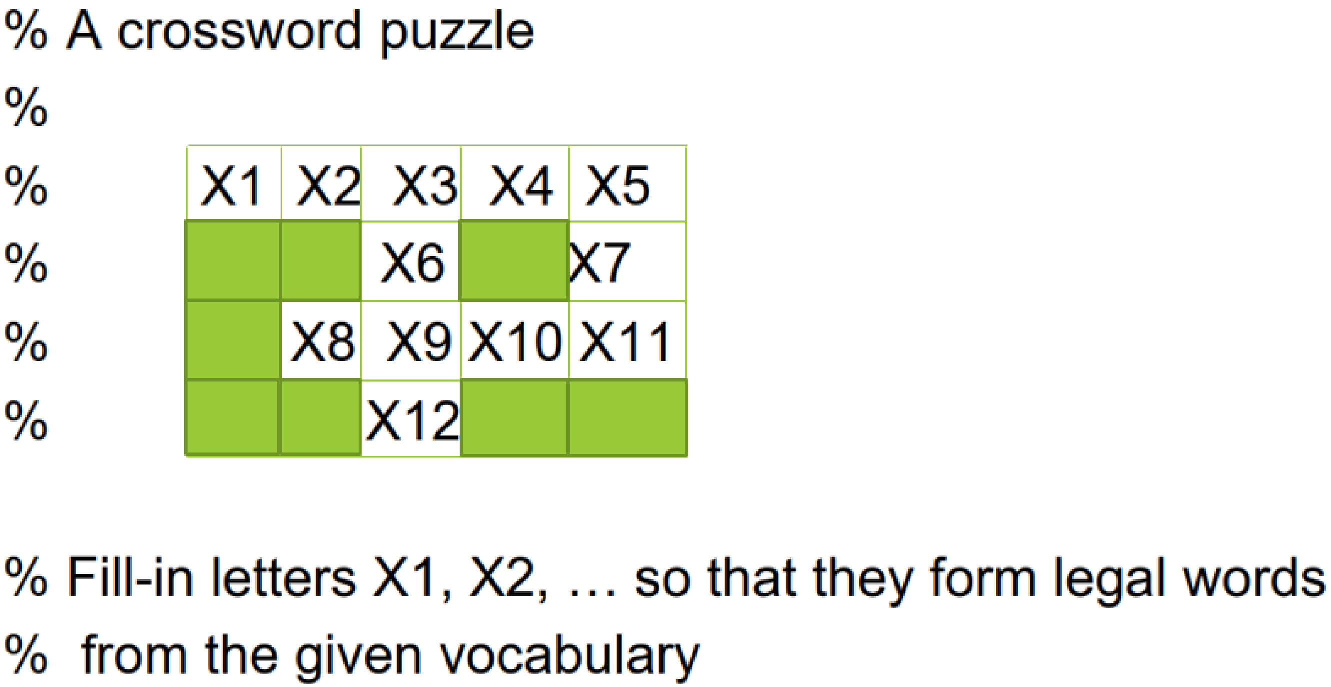


```
?- colours(red,yellow,SLO,SEA).
SLO = green,
SEA = blue ;
SLO = SEA, SEA = blue ;
SLO = yellow,
SEA = blue ;
false.
```

Spiegatte perche **SLO** non prende il colore **red**?

Se sul knowledge base aggiungiamo anche **n(AUT,SLO)** che cosa cambiera nel risultato?

# Crossword Puzzle



% A crossword puzzle

%

%

%

%

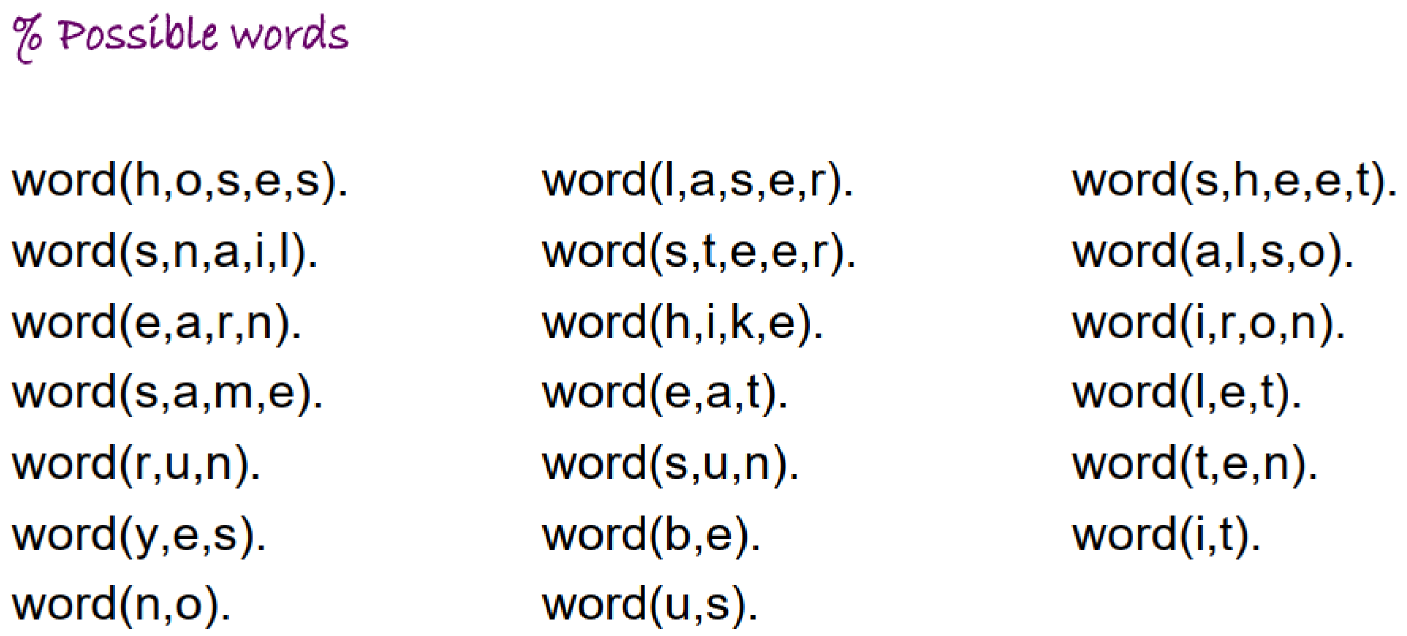
%

| | | | | |
|---|---|---|---|---|
| X1 | X2 | X3 | X4 | X5 |
| 🟩 | 🟩 | X6 | 🟩 | X7 |
| 🟩 | X8 | X9 | X10 | X11 |
| 🟩 | 🟩 | X12 | 🟩 | 🟩 |

% Fill-in letters X1, X2, … so that they form legal words

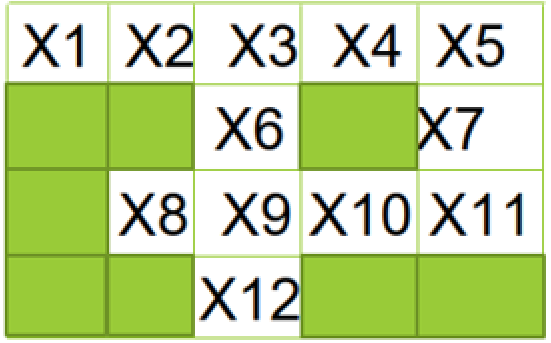
% from the given vocabulary

# Crossword Puzzle



```
% Possible words

word(h,o,s,e,s).
word(s,n,a,i,l).
word(e,a,r,n).
word(s,a,m,e).
word(r,u,n).
word(y,e,s).
word(n,o).
word(l,a,s,e,r).
word(s,t,e,e,r).
word(h,i,k,e).
word(e,a,t).
word(s,u,n).
word(b,e).
word(u,s).
word(s,h,e,e,t).
word(a,l,s,o).
word(i,r,o,n).
word(l,e,t).
word(t,e,n).
word(i,t).
```

# Crossword Puzzle

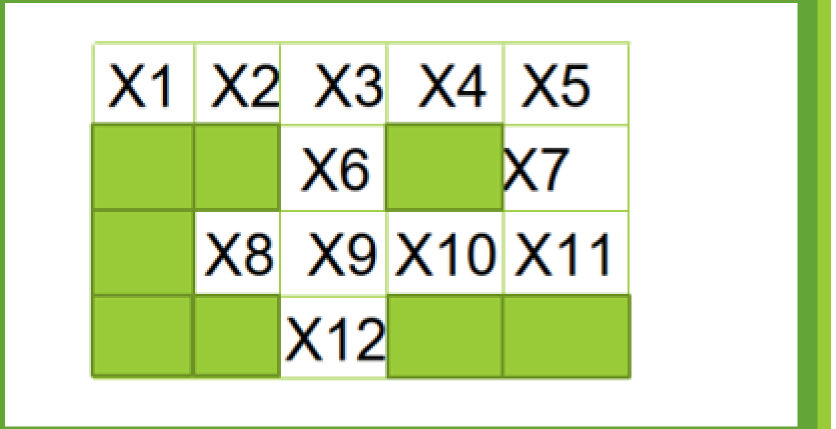


|  |  |  |  |  |
|---|---|---|---|---|
| X1 | X2 | X3 | X4 | X5 |
| 🟩 | 🟩 | X6 | 🟩 | X7 |
| 🟩 | X8 | X9 | X10 | X11 |
| 🟩 | 🟩 | X12 | 🟩 | 🟩 |

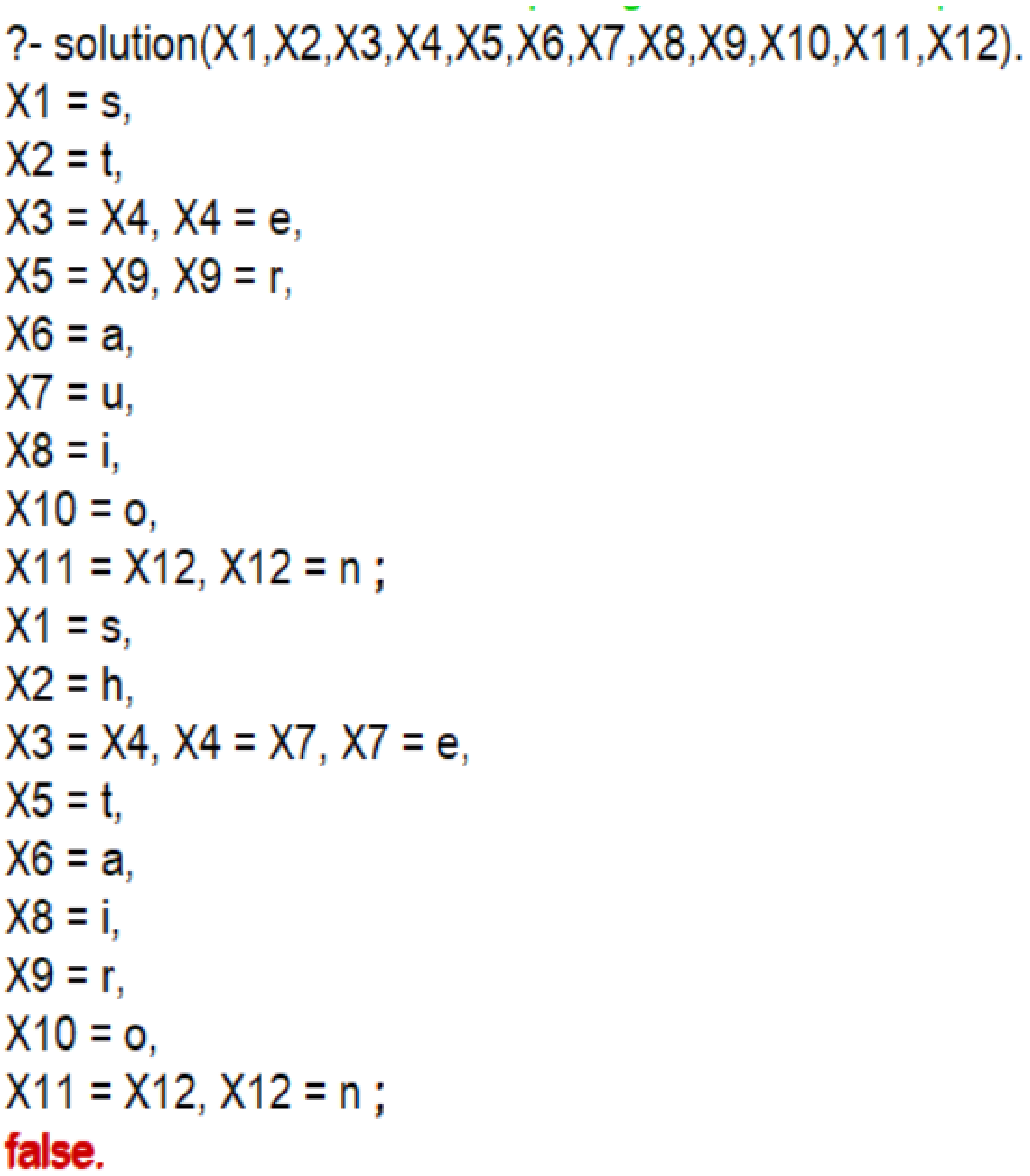
% Problem statement

```
solution( X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12)  :-
  word( X1, X2, X3, X4, X5),
  word( X3, X6, X9, X12),
  word( X5, X7, X11),
  word( X8, X9, X10, X11).
```

# Crossword Puzzle

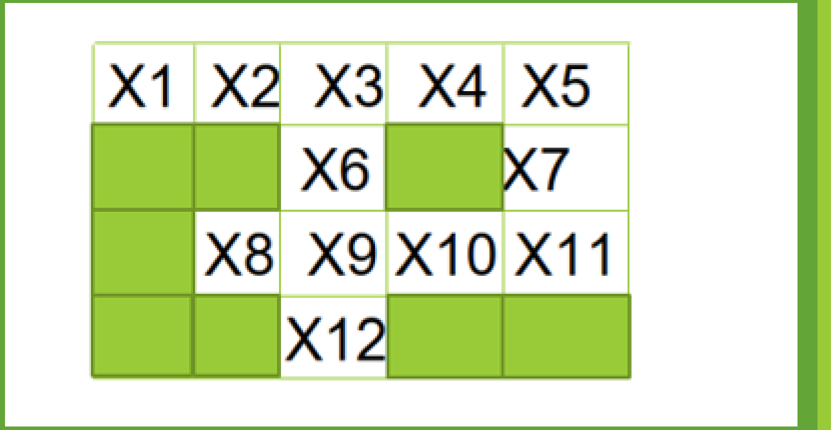


|  |  |  |  |  |
|---|---|---|---|---|
| X1 | X2 | X3 | X4 | X5 |
| 🟩 | 🟩 | X6 | 🟩 | X7 |
| 🟩 | X8 | X9 | X10 | X11 |
| 🟩 | 🟩 | X12 | 🟩 | 🟩 |

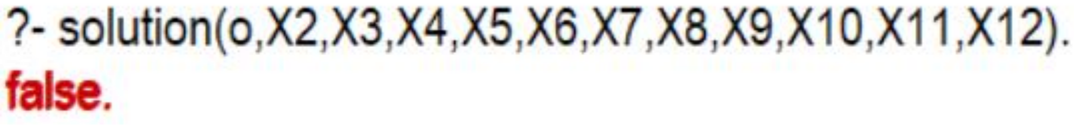


```
?- solution(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12).
X1 = s,
X2 = t,
X3 = X4, X4 = e,
X5 = X9, X9 = r,
X6 = a,
X7 = u,
X8 = i,
X10 = o,
X11 = X12, X12 = n ;
X1 = s,
X2 = h,
X3 = X4, X4 = X7, X7 = e,
X5 = t,
X6 = a,
X8 = i,
X9 = r,
X10 = o,
X11 = X12, X12 = n ;
false.
```

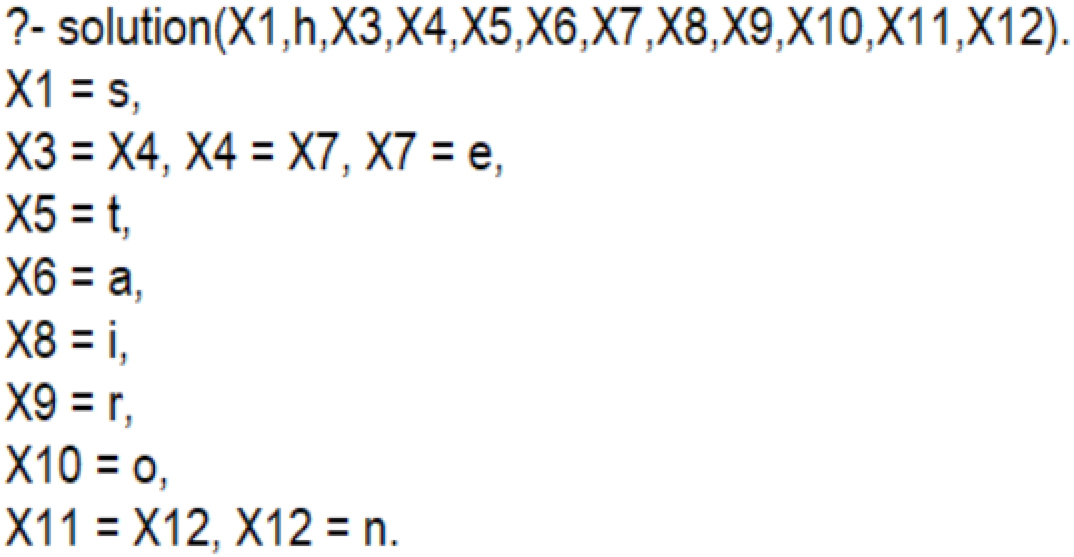
# Crossword Puzzle



|  |  |  |  |  |
|---|---|---|---|---|
| X1 | X2 | X3 | X4 | X5 |
| 🟩 | 🟩 | X6 | 🟩 | X7 |
| 🟩 | X8 | X9 | X10 | X11 |
| 🟩 | 🟩 | X12 | 🟩 | 🟩 |



```
?- solution(o,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12).
false.
```



```
?- solution(X1,h,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12).
X1 = s,
X3 = X4, X4 = X7, X7 = e,
X5 = t,
X6 = a,
X8 = i,
X9 = r,
X10 = o,
X11 = X12, X12 = n.
```

# Scheduling a meeting

Organising a project meeting according to these specifications

- ❑The meeting is organised in 3 sessions: artificial intelligence, bioinformatics, and databases
- ❑Each session takes half a day, morning or afternoon
- ❑Session on bioinformatics takes place before session on databases
- ❑Each session concerns a topic, and at least two participants of a session have to be experts in the session's topic

# Scheduling a meeting

Problem is to assign **times** and **experts** to sessions
*session(Time,Topic,P1,P2)* where P1 and P2 are partecipants.

*time(morning).*
*time(afternoon).*
*before(morning,afternoon).* %la regola

# Scheduling a meeting

```
%Experts for topics

expert( bioinformatics, barbara).
expert( bioinformatics, ben).
expert( artificial_intelligence, adam).
expert( artificial_intelligence, ann).
expert( artificial_intelligence, barbara).
expert( databases, adam).
expert( databases, danny).
```

# Scheduling a meeting

***no_conflict( Time1, P1, P2, Time2, Q1, Q2):***
There is no time conflict between two sessions at Time1 and Time2 and experts P1, P2, and Q1, Q2, respectively

***no_conflict( Time1, _, _, Time2, _, _) :-Time1 \= Time2***.
OK, sessions at differet times

***no_conflict( Time, P1, P2, Time, Q1, Q2) :- P1 \= Q1, P1 \= Q2,*** % Parallel sessions

***P2 \= Q1, P2 \= Q*** % No overlap between experts

# Scheduling a meeting

```
% schedule( TimeA, A1, A2, TimeB, B1, B2, TimeD, D1, D2):
%    TimeA and expertsA1, A2 assigned to session on Artificial Intelligence,
%    TimeB, B1, B2 assigned to session on bioinformatics, etc.

schedule( Ta, A1, A2, Tb, B1, B2, Td, D1, D2)  :-
   session( Ta, artificial_intelligence, A1, A2),
   session( Tb, bioinformatics, B1, B2),
   session( Td, databases, D1, D2),
   before( Tb, Td),                      % Bioinformatics happens before Databases
   no_conflict( Ta, A1, A2, Tb, B1, B2),    % No conflict between AI and Bioinfo
   no_conflict( Ta, A1, A2, Td, D1, D2),    % No conflict between Databases and AI
   no_conflict( Tb, B1, B2, Td, D1, D2).    % No conflict between Bioinfo and Data.
```

# Scheduling a meeting

```
?- schedule( Ta, A1, A2, Tb, B1, B2, Td, D1, D2).
A1 = adam,
A2 = ann,
B1 = barbara,
B2 = ben,
D1 = adam,
D2 = donald,
Ta = morning,
Tb = morning,
Td = afternoon ;
...
```

# Esercizio (homework)

- How many schedules are possible? We can ask Prolog with this question:

```
?- findall( 1,
       schedule( Ta, A1, A2, Tb, B1, B2, Td, D1, D2),L),
     length(L,N).

 L = [1,1,1,1,1,1,1,1,1,1|...],
 N = 16 ?
```

# Conclusioni

- Prolog programming consiste con le definizioni relations e querying per le relazioni.
- Un programa consiste di clause. Tre tipi: facts, rules e questions.
- Una relazione si puo specificare dai facts, semplicemente identificando n-tuples of objects che soddisfano la relazione
- Una procedura e una set of clauses about the same relation.
- Querying about relations, by means of questions, resembles querying a database.

# Conclusioni

Prolog's answer to a question consists of a set of objects that satisfy the question.

- In Prolog, to establish whether an object satisfies a query is often a complicated process that involves logical inference, exploring among alternatives and possibly backtracking. All this is done automatically by the Prolog system and is, in principle, hidden from the user.
- Two types of meaning of Prolog programs are distinguished: declarative and procedural. The declarative view is advantageous from the programming point of view. Nevertheless, the procedural details often have to be considered by the programmer as well.

# Domande?