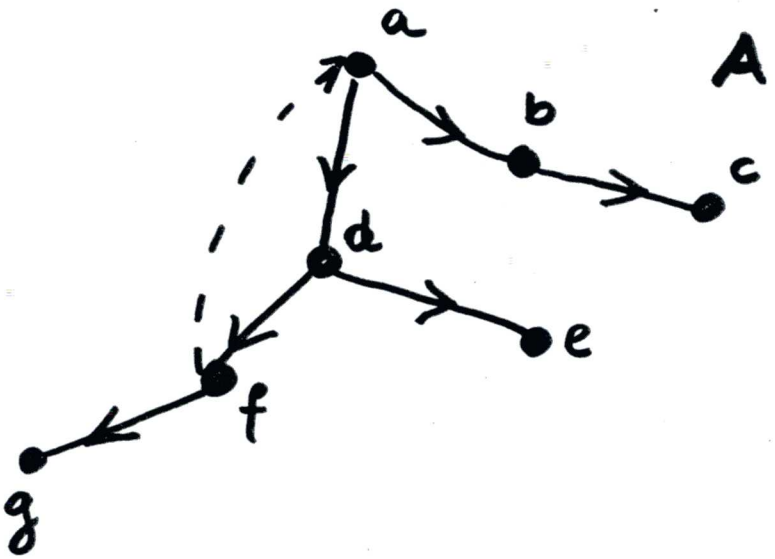
LECTURE 13

LOGIC PROGRAMMING

GRAPH WITH CYCLES & STRUCTURED INFORMATION

GRAPH : are networks of nodes & connected arcs.

Finding **a path** from **node1** to **node2** can be done by PROLOG



A GRAPH

There is **a path** from

(i) a to c

(ii) a to g

We add later arc from f to a.

LECTURE 13
LOGIC PROGRAMMING

PROLOG program:

(1) arc(a,b).
(2) arc(b,c).
(3) arc(a,d).
(4) arc(d,e).
(5) arc(d,f).
(6) arc(f,a).
(7) arc(f,g).

(8) go(X,X).
(9) go(X,Y):- arc(X,Z), go(Z,Y).

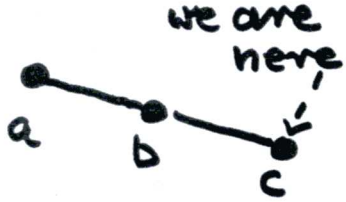
without (6) acyclic graph

>> go(a,c).

yes

SLD TREE

← go(a,c).
↓ (9) {X/a, Y/c}
← arc(a,Z), go(Z,c).
↓ (1) {Z/b}
← go(b,c).
↓ (9)
← arc(b,Z'), go(Z',c).
{Z'/c}
↓ (2)
← go(c,c)
↓ (8) {X/c}
□



LECTURE 13.
LOGIC PROGRAMMING

>> go(d,c).

No

>> go(a,g).

yes

>> go(f,a).

No.

Now we add the arc(f,a).

There is a loop in the graph.
cycle

cyclic graph

>> go(a,c).

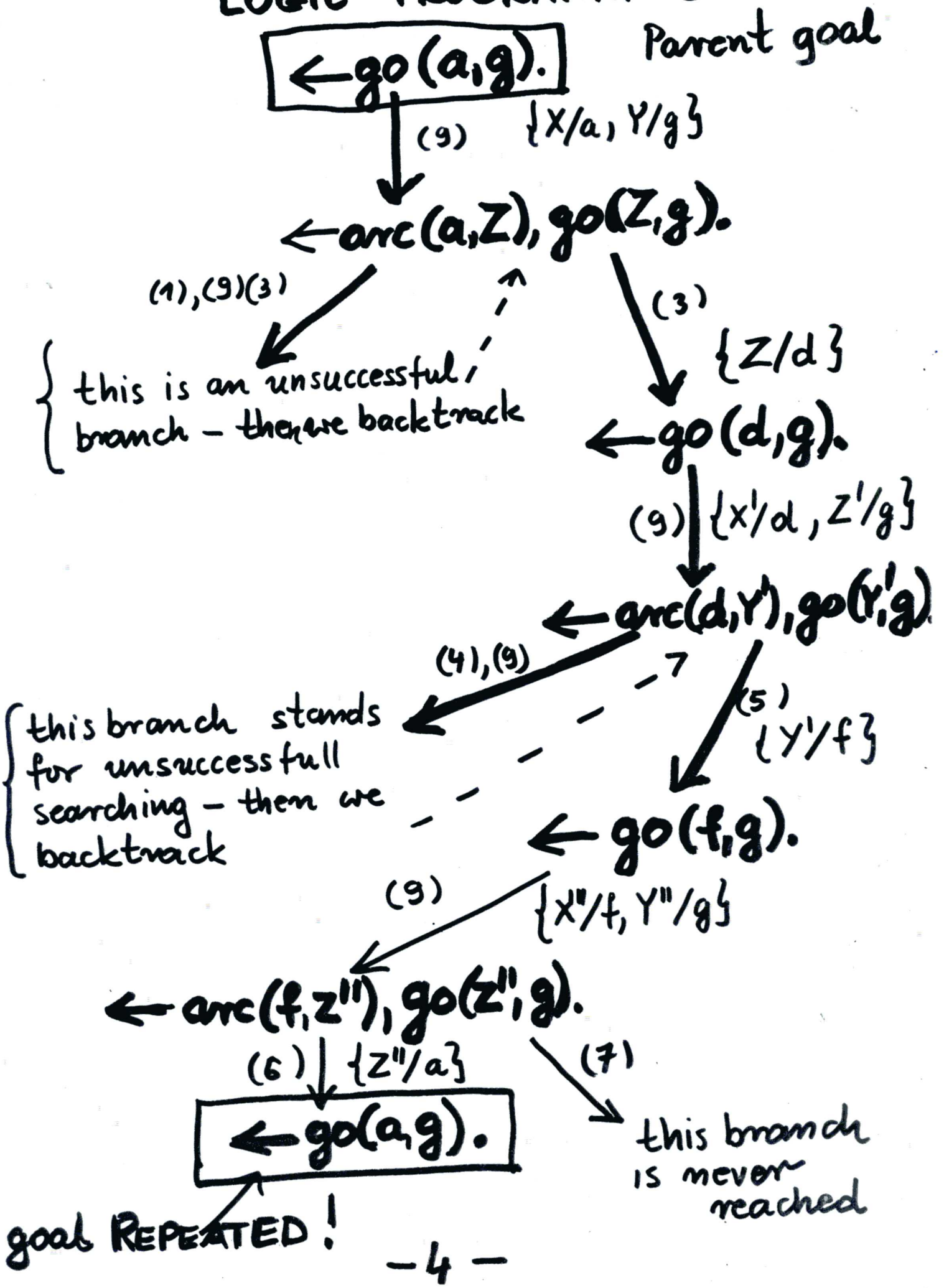
yes

>> go(a,g).

" out of local stack during
[execution aborted].

This time though there is a path from a to g we have not detected it.

LECTURE 13
LOGIC PROGRAMMING



So the loop happens as goal is repeated.

What is a remedy?

① Naive method

to rearrange order of clauses.
e.g. if 7) is put in front of 6) then:

```
>> go(a,g).
   Yes
```

this approach **is not viable** as

- one has to see potential loops (this can be hard for complex programs)
- clause rearrangement may delete some loops but also may add new ones.

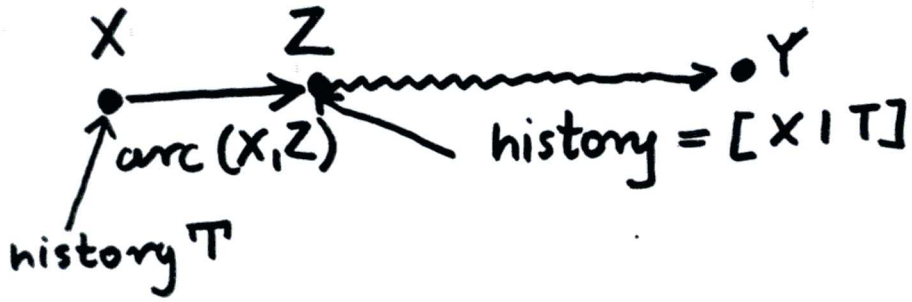
LECTURE 15
LOGIC PROGRAMMING

(2) **An advance approach:**
to **let computer check** the potential cycles and eliminate them by disallowing PROLOG to repeat the same current goal.

We modify

(i) go1(X, X, _). (a)

(ii) go1(X, Y, T) :- arc(X, Z), (b) not (member(Z, [X|T])), go1(Z, Y, [X|T]). (c)



(a) if there is an edge from X to Z

(b) to avoid cycles – if Z is not a member of the already visited vertices

(c) if there is a path from Z to Y with history [X|T]

(d) then there is a path from X to Y with the decremented list of already visited vertices (history).

Now if we check

```
>> go1(a,g,[]).
```

" if there is a path from node a to g with initial history []"

Yes

So this time there is no

∞ loop !

Let us look at the SLD-tree:
(only a success path from ↑)

LECTURE 13
LOGIC PROGRAMMING

$\leftarrow go1(a, g, [\,])$. (vertex a; history [ ])

$\{X/a, Y/g, T/[\,]\}$

(ii) ↓

$\leftarrow arc(a,Z), not(member(Z,[a|[\,]])), go1(Z, g, [a|[\,]])$.

(3) ↓ $\{Z/d\}$

$\leftarrow not(member(d,[a])), go1(d, g, [a])$.

↓ as $d \notin [a]$ we have no converted not(no) = yes

$\leftarrow go1(d, g, [a])$. (vertex d; history [a]).

(ii) ↓ $\{X'/d, Y'/g, T'/[a]\}$

$\leftarrow arc(d,Z'), not(member(Z',[d|[a]])), go1(Z', g, [d|[a]])$.

(5) ↓ $\{Z'/f\}$

$\leftarrow not(member(f, [d,a])), go1(f, g, [d,a])$

↓ as $f \notin [d,a]$ no is converted not(no) = yes

$\leftarrow go1(f, g, [d,a])$. (vertex f, history [d,a])

(ii) ↓

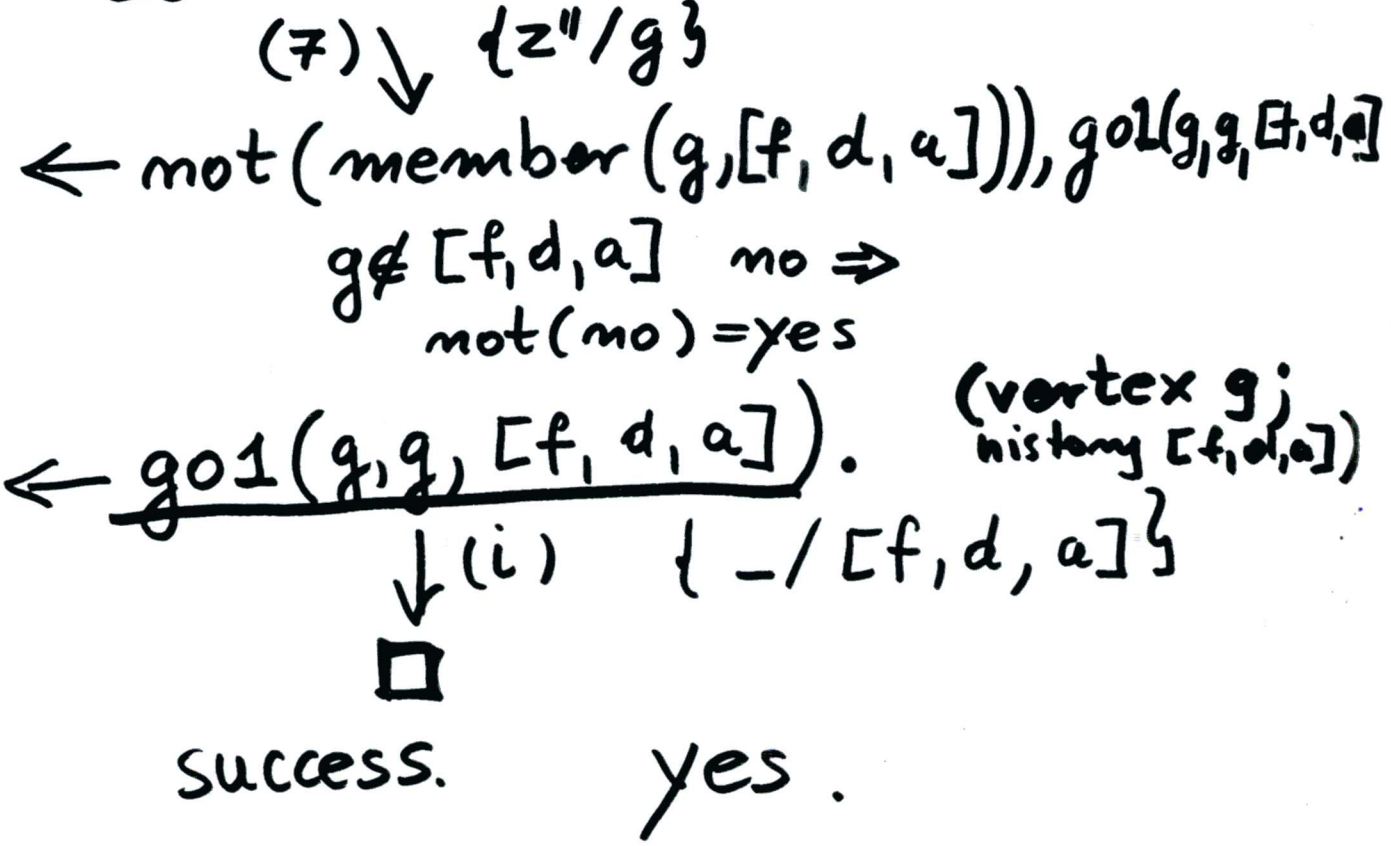
$\leftarrow arc(f, Z''), not(member(Z'', [f|[d,a]])), go1(g, g, [f,d,a])$.

$\{Z''/e\}$ ↙ (6) ↘ (7) $\{Z''/g\}$

as member(a, [f,d,a]) is yes ⇒ not(yes) no ⇢ backtracks

next page

LECTURE 13
LOGIC PROGRAMMING



So we did not reach (as in the case of predicate go) recurrance of the initial goal!

However if we want also to get the **history** of the **visited nodes** from a to g:

>> go1(a,g,X).

the answer is **"No"** !!!

The reason is that in SLD-tree at some stage QUERY member(d,[a|X]) succeeds & not(YES) → no.

LECTURE 13
LOGIC PROGRAMMING

So uninstantiated history makes a problem.

We extend go1/3 ⟶ go2/4 or go3/4.

```
go2(X,X,T,[X|T]).
```

history*

```
go2(X,Y,T,P):- arc(X,Z),
               not(member(Z,[X|T])),
               go2(Z,Y,[X|T],P).
```

If we query now:

```
>> go2(a,g,[],X).

   X=[g,f,d,a]
```

For inverting the history list:

- either we apply the predicate to reverse the order of the list or:

* - A is a history; P is the number of nodes to go.

LECTURE 13
LOGIC PROGRAMMING

```
go3(X,X,A,[X]).
go3(X,Y,A,[X|P]):-arc(X,Z),
                  not(member(Z,[X|A])),
                  go3(Z,Y,[X|A],P).
```

Then

```
>> go3(a,g,[],X).
   X=[a,d,f,g].
```

SUMMARY

(a) **go(X,Y)**: fails for cyclic graph.
2 variables are not sufficient!

(b) **go1(X,Y,Z)**: solves the problem of cyclic graph path
but go1(node1, node2, X) yields no path! – incorrect answer.

(c) both **go2(X,Y,Z,T)**, **go3(X,Y,Z,T)** solve problems appearing in a) & b).

LECTURE 13
LOGIC PROGRAMMING

REMARK: If not needed use history as [ ]; otherwise false answer may be given by PROLOG. For example if initial history contains a node which is on the path between node1 & node2.

PROLOG can be used to retrieve

"**structured information**"

by the power of unification algorithm.

The specific application goes to

expert systems

&

different logic puzzles

We present 2 examples below:

LECTURE 13
LOGIC PROGRAMMING

Example 1: logic puzzle

" Three children ran the race.
Peter did better than the person
who ran in red.
Jack wearing gold, did better
than a child in green.
Who did win the race ? "

We need functors: child(Name, Colour).
Order(X, Y, Z).

Prolog program

```
did better(X, Y, order(X, Y, _)).
did better(X, Y, order(X, _, Y)).
did better(X, Y, order(_, X, Y)).
clue1(S):- did better(child(peter, _), child(_, red), S).
clue2(S):- did better(child(jack, gold), child(_, green), S).
```

```
>> clue1(S), clue2(S).
```

LECTURE 13
LOGIC PROGRAMMING

```
S= order( child(jack, gold), child(peter, green),
          child(_15, red))
```

if prompted for more solutions ;

No.

So we have only one winner, the second person & the last one.
We know the colour of their shirts.

The data we had are sufficient to solve the logic puzzle. But insufficient to get exactly the name of all racing participants.

_15 – looks strange! But it **is correct**.

The token that starts with underscore **is a variable** for which PROLOG has not been given a nam
So internal binding was used □

LECTURE 13
LOGIC PROGRAMMING

REMARK: strictly speaking
functor child/2 should

$$child : D \times D \rightarrow D$$

↑ ↑ ↗

domain of discourse
for a given interpretation

Since we pass to functor child

name & colour

D = { jack, peter, unknownname, gold, green, red }

Similarly for order : $D \times D \times D \rightarrow D$

Note we do not have to specify what are the values of both functors!

A power of unification algorithm
via symbolic matching
finds arguments for both functors!

LECTURE 13
LOGIC PROGRAMMING

REMARK: if predicate has bigger arity then 2 i.e p/n with $n > 2 \Rightarrow$ we can group variables $(X_1, \ldots, X_k)$ & $(X_{k+1} \ldots X_n)$ & introduce 2 functors f/k & $g/n-k$ such that new predicate $\tilde{p}(f(X_1, \ldots, X_k), g(X_{k+1}, \ldots X_n))$ is equivalent to $p(X_1, X_2, \ldots, X_n)$.

This allows to use graph representation.

Example 2: retrieving structured information from database.

We represent a book in the following fashion:

LECTURE 13
LOGIC PROGRAMMING

```
book(title(c++), author([a1, a2, a3]),
     published(mit, 80), isbn(number1)).

book(title(pascal), author([b1]),
     published(oxford, 89), isbn(number2)).

book(title(prolog), author([c1, c2]),
     published(springer, 81), isbn(number3)).
```

predicate book is really not used here to represent

true or false

but more to be

a place-holder for structured information

Now we can QUERY PROLOG for all titles of springer in our database:

LECTURE 13
LOGIC PROGRAMMING

```
>> book(title(X),_,published(springer,_),
                                    _).
   X = prolog.
```

Note that for variables with underscore "_" PROLOG does not substitute.

```
>> book(title(X),_,published(springer,_),
                              isbn(Y)).
   X = prolog.
   Y = number3.
```

□