LECTURE 12.
LOGIC PROGRAMMING

We deal now with the basic "data structure" in PROLOG – **A LIST**.

The elements of list may be
- constants
- variables
- other structures like lists.

List is either:
- an empty list (having no elements)
- or it has two components
  - the head
  - the tail

Both head & tail are variables passed to a functor "."

The empty list is denoted by [].

Remark: empty list has not got tail & head – this is important later for recursion.

LECTURE 12.
LOGIC PROGRAMMING

The list having one element a :

- .(a, []) or [a]

The list having three elements a, b, c

- .(a, .(b, .(c, []))) or [a, b, c]

The list notation [a] or [a, b, c] is more convenient.

From **list notation** we can locate head & tail:

List	Head	Tail
[a, b, c]	a	[b, c]
[ ]	none	none
[[the, cat], sat]	[the, cat]	[sat]
[the, [cat, sat]]	the	[[cat, sat]]
[the, [cat, sat], down]	the	[[cat, sat], down]

LECTURE 12
LOGIC PROGRAMMING

Since it is common to split a given list into **a Head** & **a Tail** we have such "split" operator:

$$[X \mid Y]$$

head — tail

Example 1: a)

$S = \{ p(1,2,3),\ p(a,b,c,[d,e]).$

one query for both!

>> $p([X \mid Y]).$

$X = 1 \qquad Y = [2,3]$

$X = a \qquad Y = [b,c,[d,e]]$

>> $p([\_,\_,\_,[\_ \mid X]]).$

$X = e.$

note that

$$q(\_). \equiv q(X). \equiv \forall x\, q(x).$$

LECTURE 12
LOGIC PROGRAMMING

b)

LIST 1		LIST 2	INSTANTIATION
$[X,Y,Z]$	$=$	$[a,b,c] \Rightarrow$	$X=a$, $Y=b$, $Z=c$
$[cat]$	$=$	$[X \mid Y] \Rightarrow$	$X=cat$, $Y=[\,]$
$\equiv$			
$[cat \mid [\,]]$			
$[X,Y \mid Z]$	$=$	$[a,b,c] \Rightarrow$	$X=a$, $Y=b$, $Z=[c]$
$\equiv$		$\equiv$	
$[X \mid [Y \mid Z]]$		$[a \mid [b,c]]$	
		$\equiv$	
		$[a \mid [b \mid [c]]]$	
$[[a,Y] \mid Z]$	$=$	$[[X,b],[c,d]]$	$X=a$, $Y=b$,
		$\equiv$	
		$[[X,b] \mid [[c,d]]]$	$Z=[[c,d]]$.

□

The are fundamental functions on lists:

- checking membership
- appending two lists
- adding or deleting elements to/from list.

LECTURE 12.
LOGIC PROGRAMMING

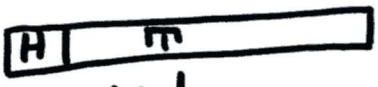
Note that there are in-built predicates in PROLOG (member/2, delete/2 etc).

So we define ours by adding e.g. a number.

(i) member1 (H, [H|T]).

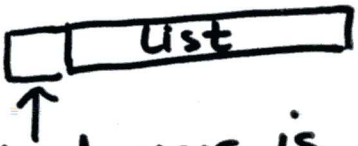
Δ (ii) member1 (H, [_|T]):- member1(H,T).

so for

(i) H ∈ 

list

(ii) H ∈ 

⇒ H ∈ 

whatever is added at forefront.

>> member1 (c, [a,b,c]). (*)

"is c a member of [a,b,c] ?"

yes

>> member1 (e, [a,b,c]).

"is e a member of [a,b,c]?"

No

Let us draw the corresponding SLD-tree for Δ & QUERY (*)

LECTURE 12.
LOGIC PROGRAMMING

$\equiv [a \mid [b,c]]$

← member1(c, [a, b, c]).

↓ only with (ii) {H/c, _/a, T/[b,c]}

← member1(c, [b, c]).

$\equiv [b \mid [c]]$

↓ only with (ii) {H/c, _/b, T/[c]}

← member1(c, [c]).

$\equiv [c \mid []]$

↓ with (i) {H/c, T/[]}

□

success. YES

If we now QUERY:

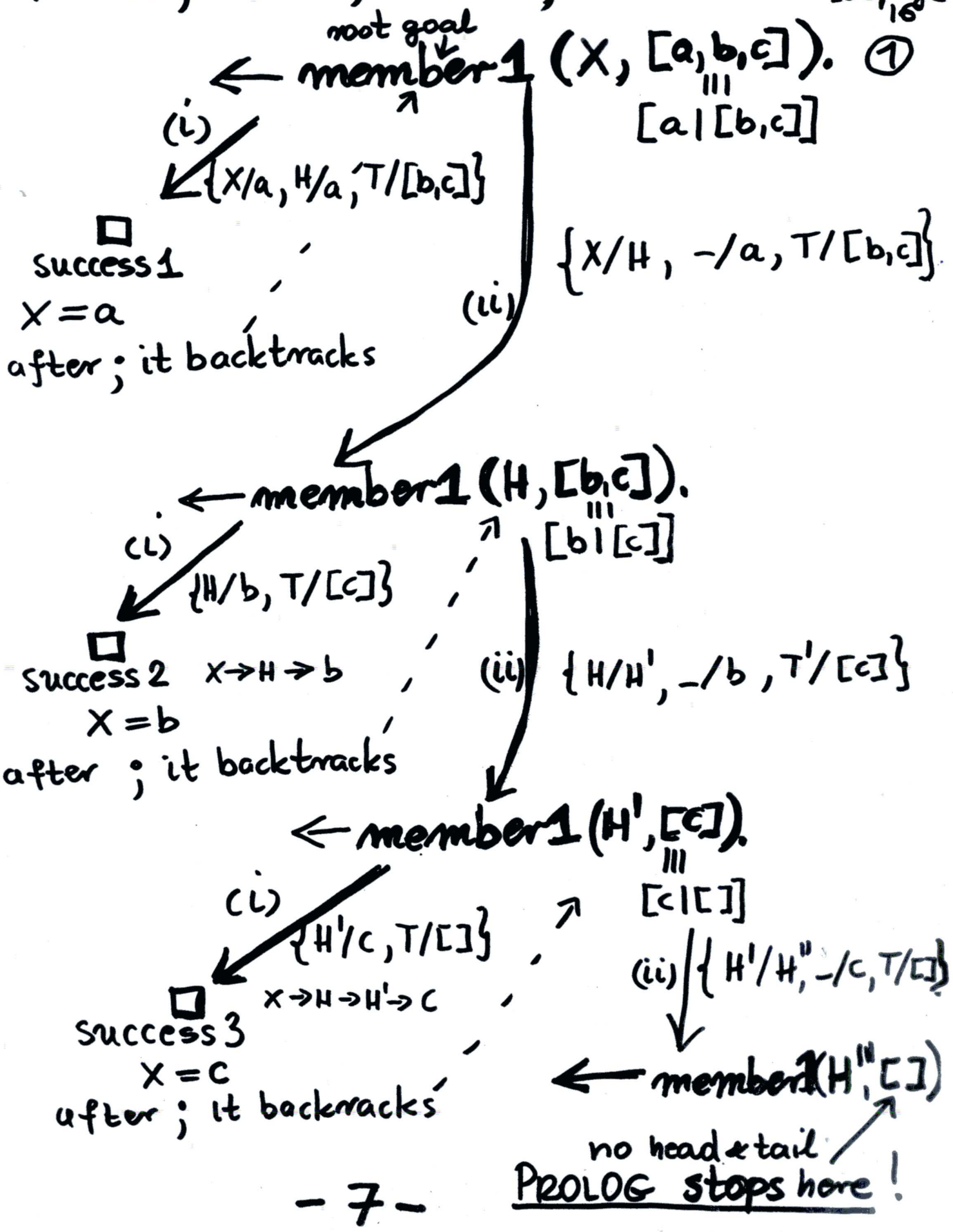
>> member1(X, [a, b, c]).

"what are the members of list [a,b,c]" ≡ ∃X member1(X, [a,b,c]).

LECTURE 12
LOGIC PROGRAMMING

X=a ; Y=b ; X=c ; No.

see page 16

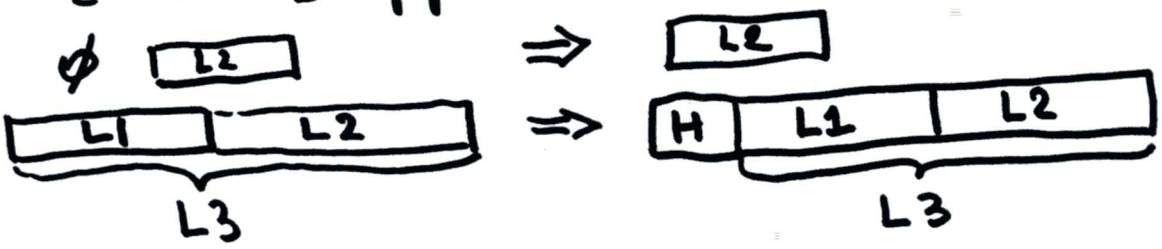


LECTURE 12
LOGIC PROGRAMMING

Note: since it is assumed that empty list [ ] has no tail & head the above program does not loop into ∞ but terminates after all three solutions are found.

" if L1 = empty appended to L2 then L3=L2."

" if L1 appended to L2 yield L3 ⇒
[H|L1] appended to L2 yield [H|L3] "



(j) append1([ ], L2, L2).

(jj) append1([H|T], X, [H|Y]) :-
append1(T, X, Y).

We can test now this predicate by the following QUERIES:

LECTURE 12
LOGIC PROGRAMMING

```
>> append1([a,b], X, [a,b,c,d]).
   X = [c, d].
>> append1(X, Y, [a,b,c,d]).
   X = [ ]      Y = [a,b,c,d];
   X = [a]      Y = [ b, c, d];
   X = [a,b]    Y = [ c, d];
   X = [a,b,c]  Y = [ d];
   X = [a,b,c,d] Y = [ ];
no.
```

There are many other predicates on lists.
Consider the following one:

```
(K)   count(X, [ ], 0).
(KK)  count(X, [X|T], N) :- count(X, T, M), N is M+1.
(KKK) count(X, [_|T], N) :- count(X, T, N).
```

```
>> count(a, [b,a], X).
X = 1;          correct      | Prolog success1
X = 0;          incorrect    • Prolog success2
no
```

This is very common in PROLOG!
Only the 1st solution is correct.

It is clear that if _ = X in (KKK) then this case is incorrectly defined - only if _ ≠ X it is correct.

By the time we get however to (KKK) (in such a case) we first execute clause (KK) which:

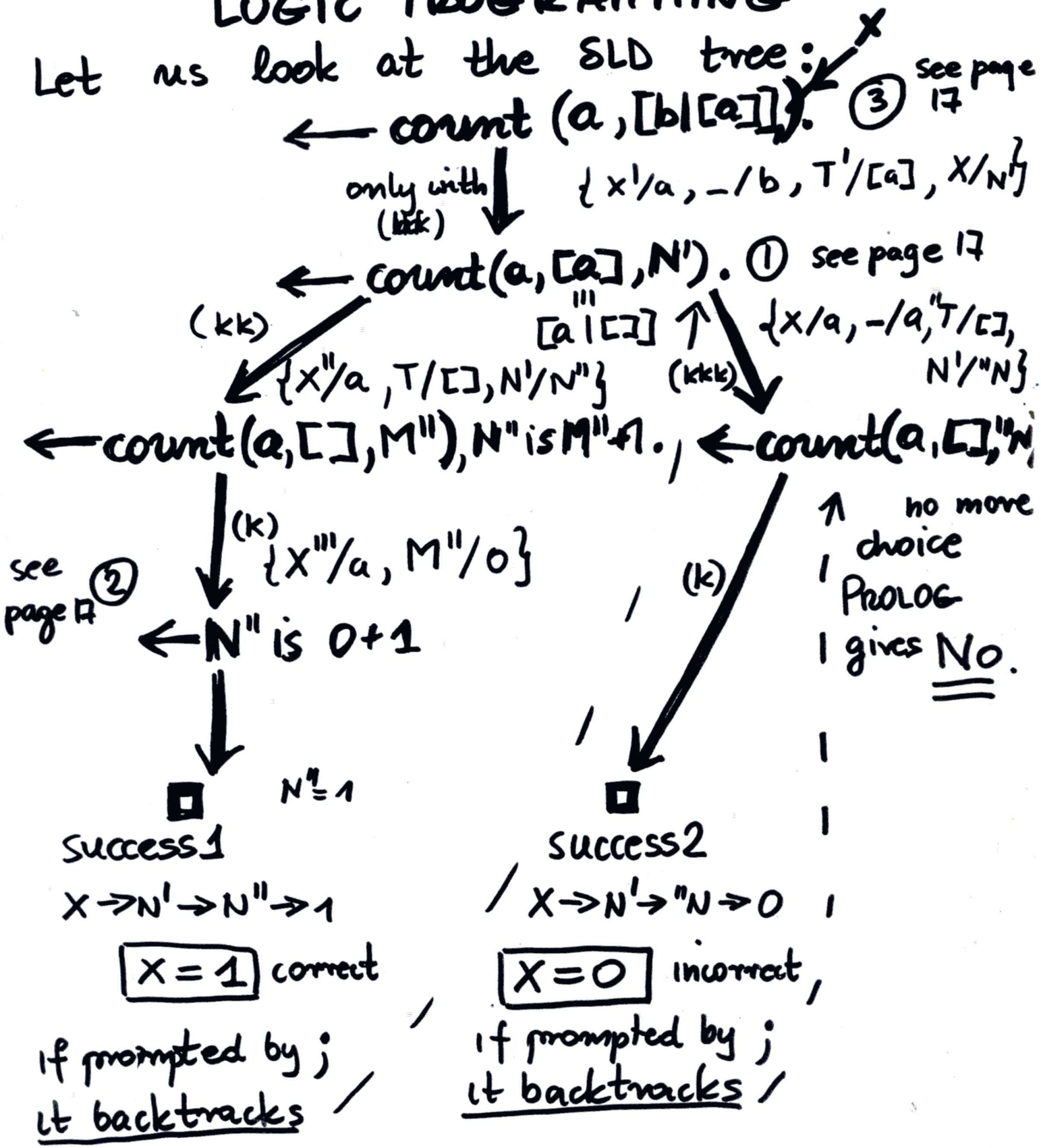
covers this case correctly!

so the 1st success is a correct one.

Our program works modulo 1st success!

LECTURE 12.
LOGIC PROGRAMMING

Let us look at the SLD tree:



So what can we do with multiple solutions (only 1st one is correct!)?

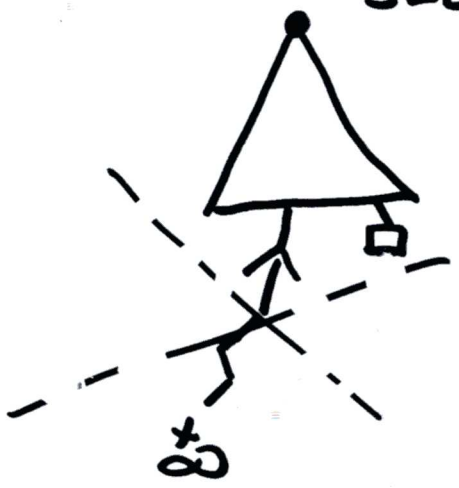
LECTURE 12
LOGIC PROGRAMMING

We can re-write a program or use CUT facility. "!"

- it prunes some branches in SLD-tree
- this is important if we anticipate

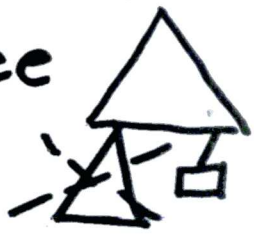
  - infinite branches

SLD-tree



  - fake success branches

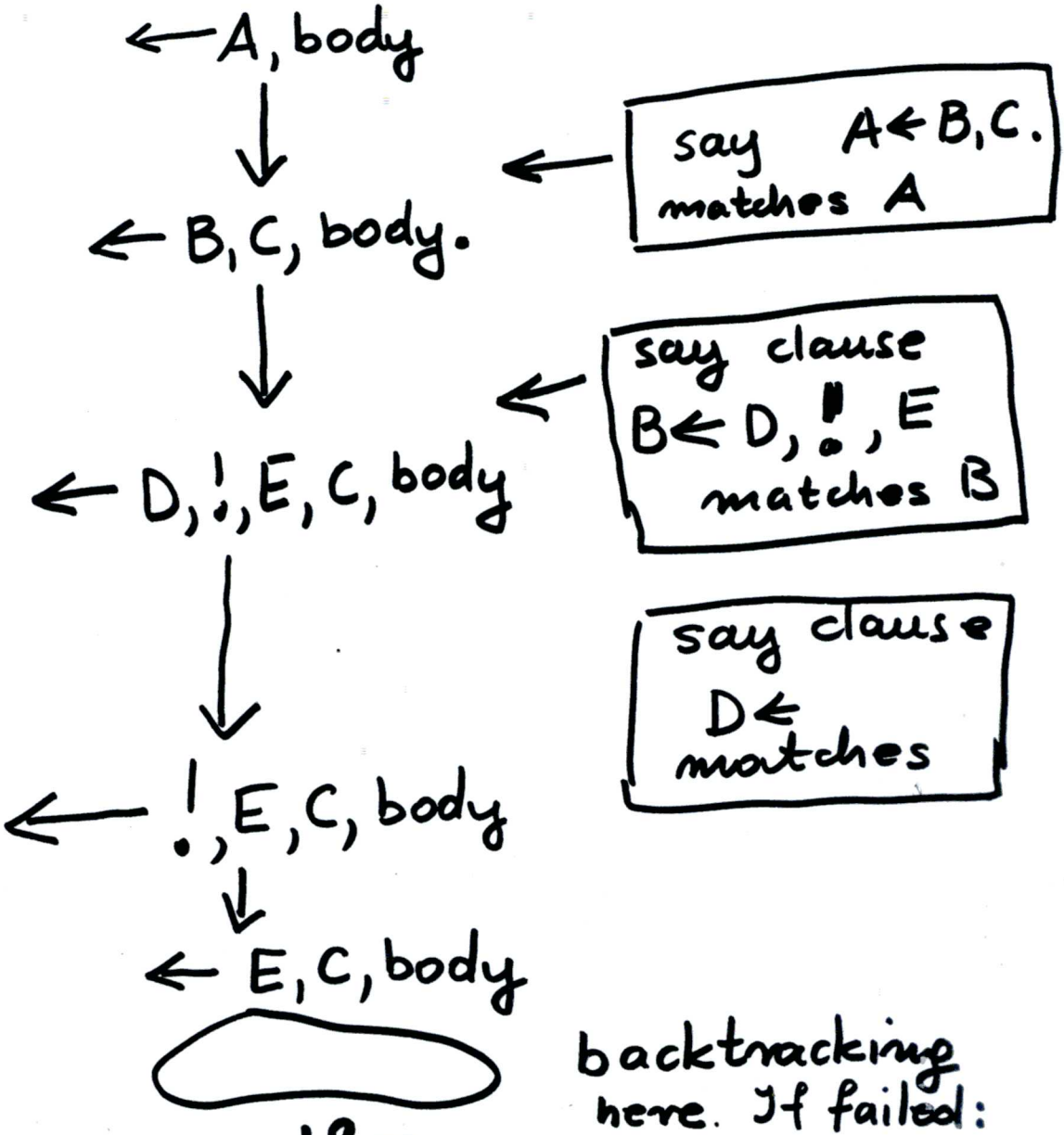
  - no success branches in some part of SLD tree



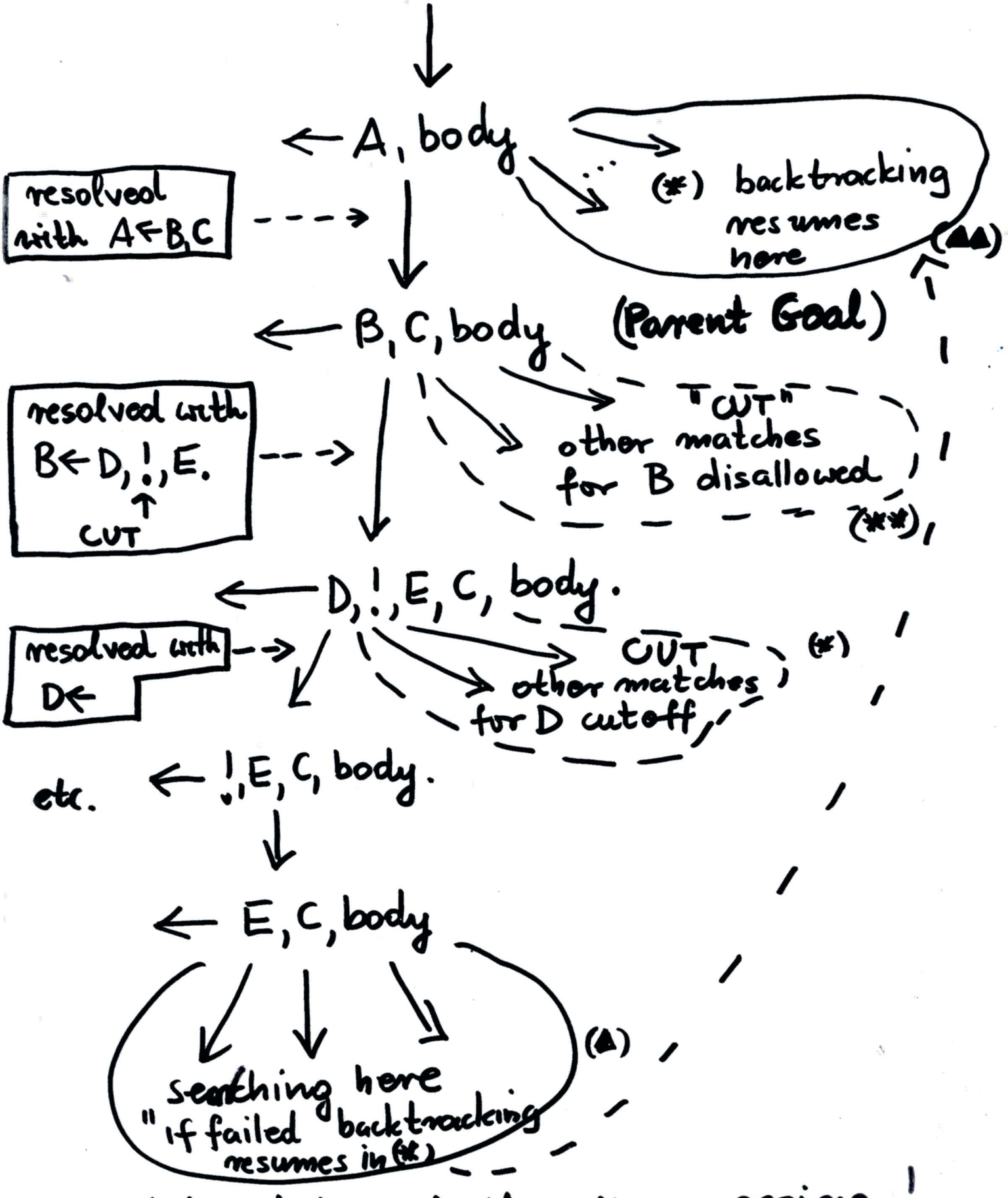
LECTURE 12.
LOGIC PROGRAMMING

- or we want only one correct solutions

We see the example now:



LECTURE 12
LOGIC PROGRAMMING



If backtracking fails after passing !
Then we resume it one level over Parent Goal

LECTURE 1
LOGIC PROGRAMMING

- parent goal: the goal that matched the head of the program clause containing cut i.e. !.
- when the cut ! is encountered as a goal it succeeds, but it commits the system to all choices made between the time parent goal was invoked & the time CUT was met.
- all remaining alternatives (*) between the parent goal & the parent goal are discarded.
- Backtracking may only occur between (▲) $B_1, .., B_n$ for goal $\leftarrow H_1, \ldots, H_n, !, B_1, \ldots, B_m$.
- all goals for $H_1, .. H_n$ are committed (*)

- if backtracking between $B_1, \ldots, B_m$ fails (▲) then it resumes not from other choices in parent goal (see **) but from one goal up to a parent goal (▲▲).

We can now remove multiple (but correct) solutions for QUERY

```
>> member1(X, [a,b,c]).
   X = a ; No
```

if

```
member1(H, [H|T]) :- !.
```

the second one the same.

here there is no "a level up goal" to the root goal (see page 7) & therefore after finding X=a. backtracking can also resume from one level up to root goal – so no more solutions!

Similarly if we correct:

(a) count(X, [X|T], N) :- !, count(X, T, M), N is M+1.

or

(b) count(X, [X|T], N) :- count(X, T, M), !, N is M+1.

then

```
>> count(a, [b,a], X).
   X = 1 ;
   No
```

This time the second success (X=0, which is incorrect) is precluded!

This is because for (a) { for (b) }
① page 11 is committed (no higher goal ③) { parent goal above
② which is ③ does not give more alternatives }.

REMARK: Note that in practice one knows which clause should contain !.

Exact placing requires a full SLD-tree. Faster is to place it in different possible spots & test experimentally whether it works.