Declarative Programming

3: logic programming and Prolog

Sentences in definite clause logic: *procedural and declarative meaning*

```
a :- b, c.
```

declarative meaning realized by model semantics

to determine whether `a` is a logical consequence of the clause, order of atoms in body is irrelevant

procedural meaning realized by proof theory

order of atoms may determine whether a can be derived

`a :- b, c.` to prove `a`, prove `b` and then prove `c`

`a :- c, b.` to prove `a`, prove `c` and then prove `b`

imagine c is false

and proof for b is infinite

Sentences in definite clause logic: *procedural meaning enables programming*

SLD-resolution refutation

procedural knowledge:
how the inference rules are applied to solve the problem

algorithm = logic + control

declarative knowledge:
the **what** of the problem

definite clause logic

SLD-resolution refutation:

also: an unwieldy theorem prover in effective programming language

turns resolution refutation into a proof procedure

left-most

determines how to select a literal to resolve upon

and which clause is used when multiple are applicable

top-down

definite clauses

selection rule

SLD

linear resolution

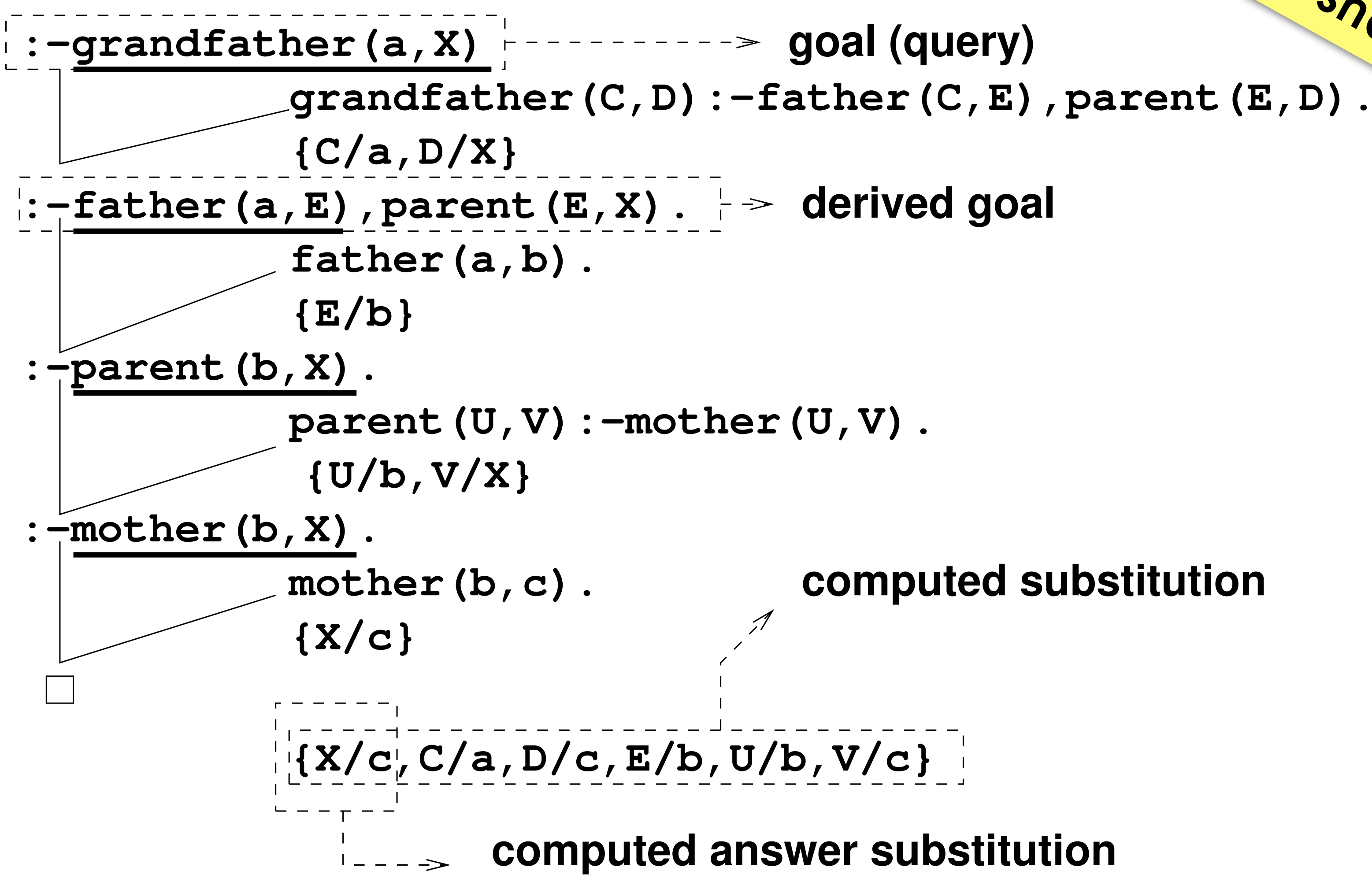
refers to the shape of the resulting proof trees

the clause obtained from a resolution step (the resolvent) is always resolved with a program clause in the next (and not with another resolvent)

SLD-resolution refutation:
refutation proof trees based on SLD-resolution

```
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(a,b).
mother(b,c).
```

linear shape!



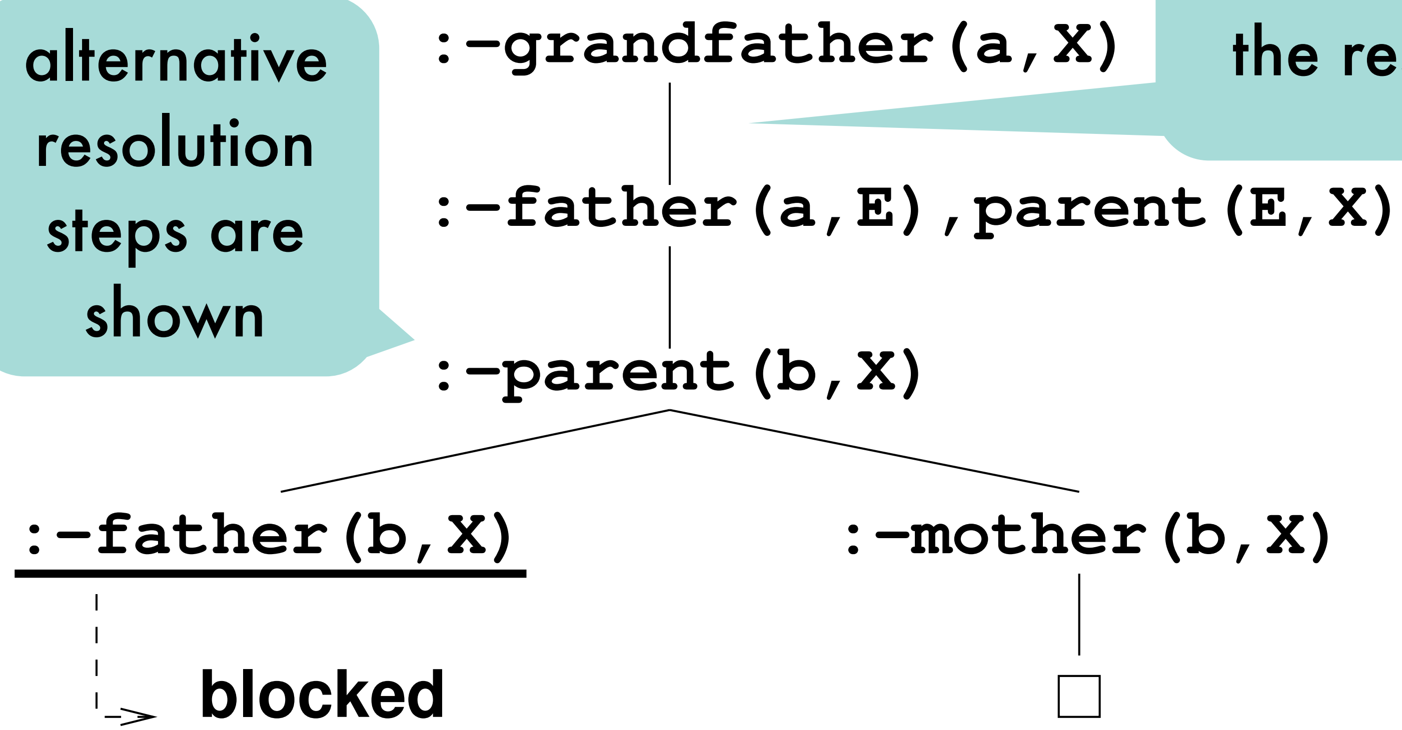
SLD-resolution refutation: *SLD-trees*

not the same as proof trees!

```
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(a,b).
mother(b,c).
```

program clauses resolved with are not shown, nor are the resulting substitutions

alternative resolution steps are shown



failure branch

success branch

Prolog traverses SLD-trees depth-first, backtracking from a blocked node to the last choice point (also from a success node when more answers are requested)

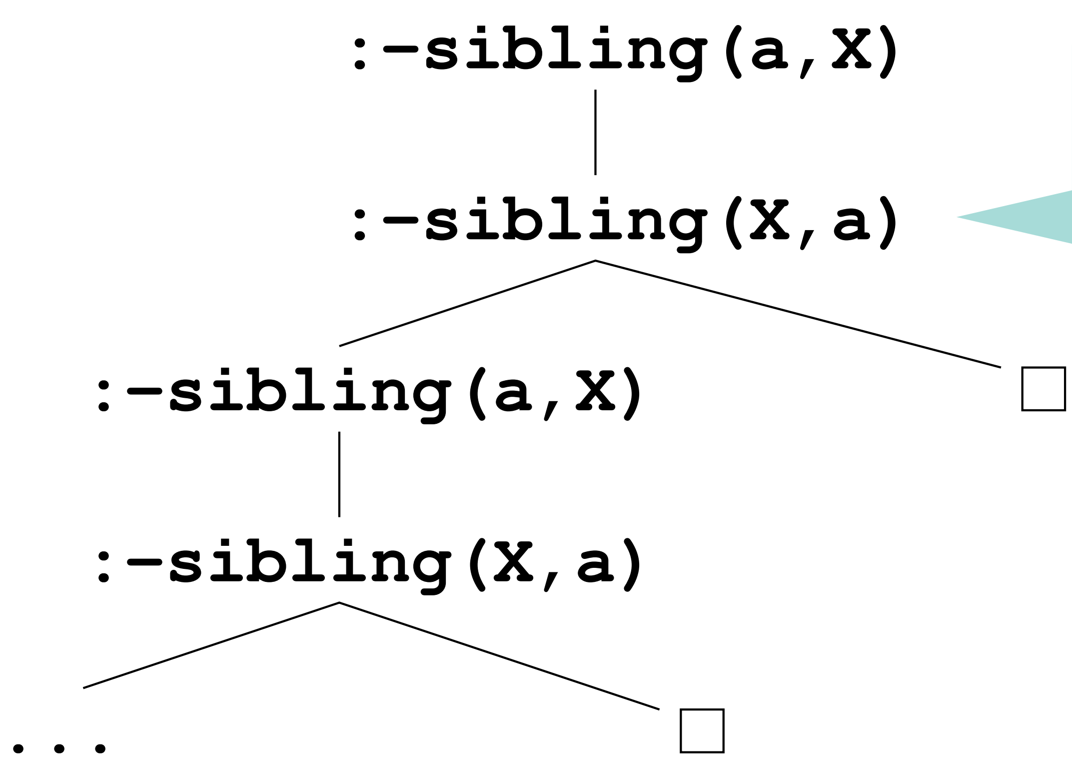
every path from the query root to the empty clause corresponds to a proof tree (a successful refutation proof)

Problems with SLD-resolution refutation:

never reaching success branch because of infinite subtrees

```
sibling(X,Y) :- sibling(Y,X).
sibling(b,a).
```

rule of thumb: non-recursive clauses before recursive ones



had we re-ordered the clauses, we would have reached a success branch at the second choice point

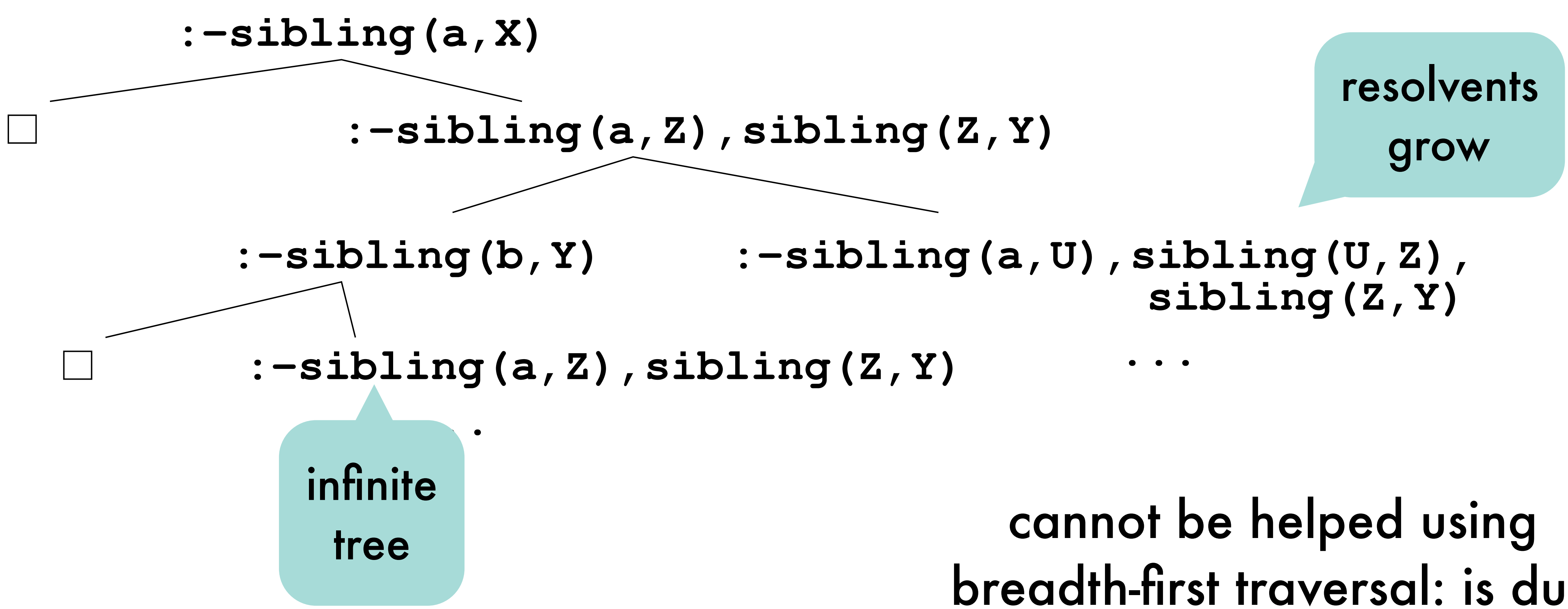
incompleteness of Prolog is a design choice: **breadth-first traversal** would require keeping all resolvents on a level in memory instead of 1

Prolog loops on this query; renders it incomplete!
only because of **depth-first traversal** and not because of resolution as all answers are represented by success branches in the SLD-tree

Problems with SLD-resolution refutation:

Prolog loops on infinite SLD-trees
when no (more) answers can be found

```
sibling(a,b).
sibling(b,c).
sibling(X,Y) :- sibling(X,Z), sibling(Z,Y).
```



cannot be helped using breadth-first traversal: is due to **semi-decidability** of full and definite clausal logic

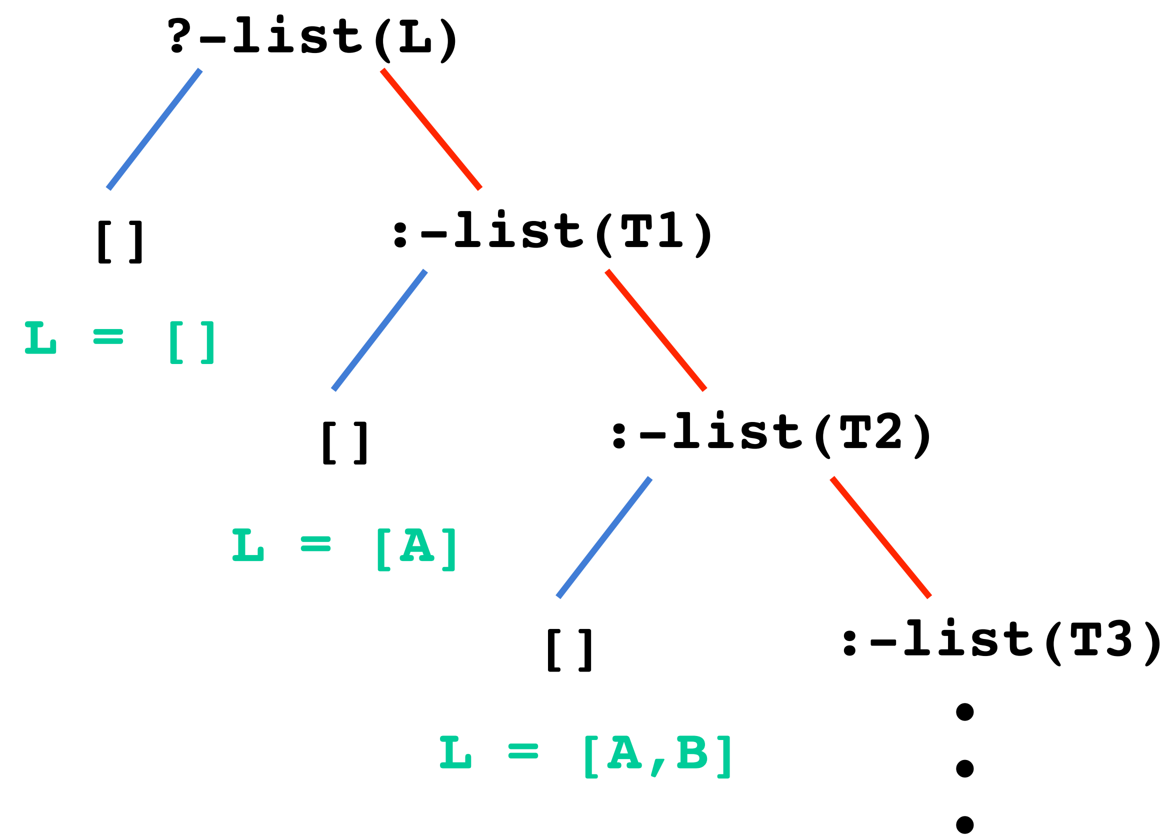
Problems with SLD-resolution refutation: *illustrated on list generation*

```
list([]).
list([H|T]):-list(T).
```

Prolog would loop without finding answers if clauses were reversed!

```
?-list(L).
L = [];
L = [A];
L = [A,B];
…
```

benign:
infinitely many lists of arbitrary length are generated

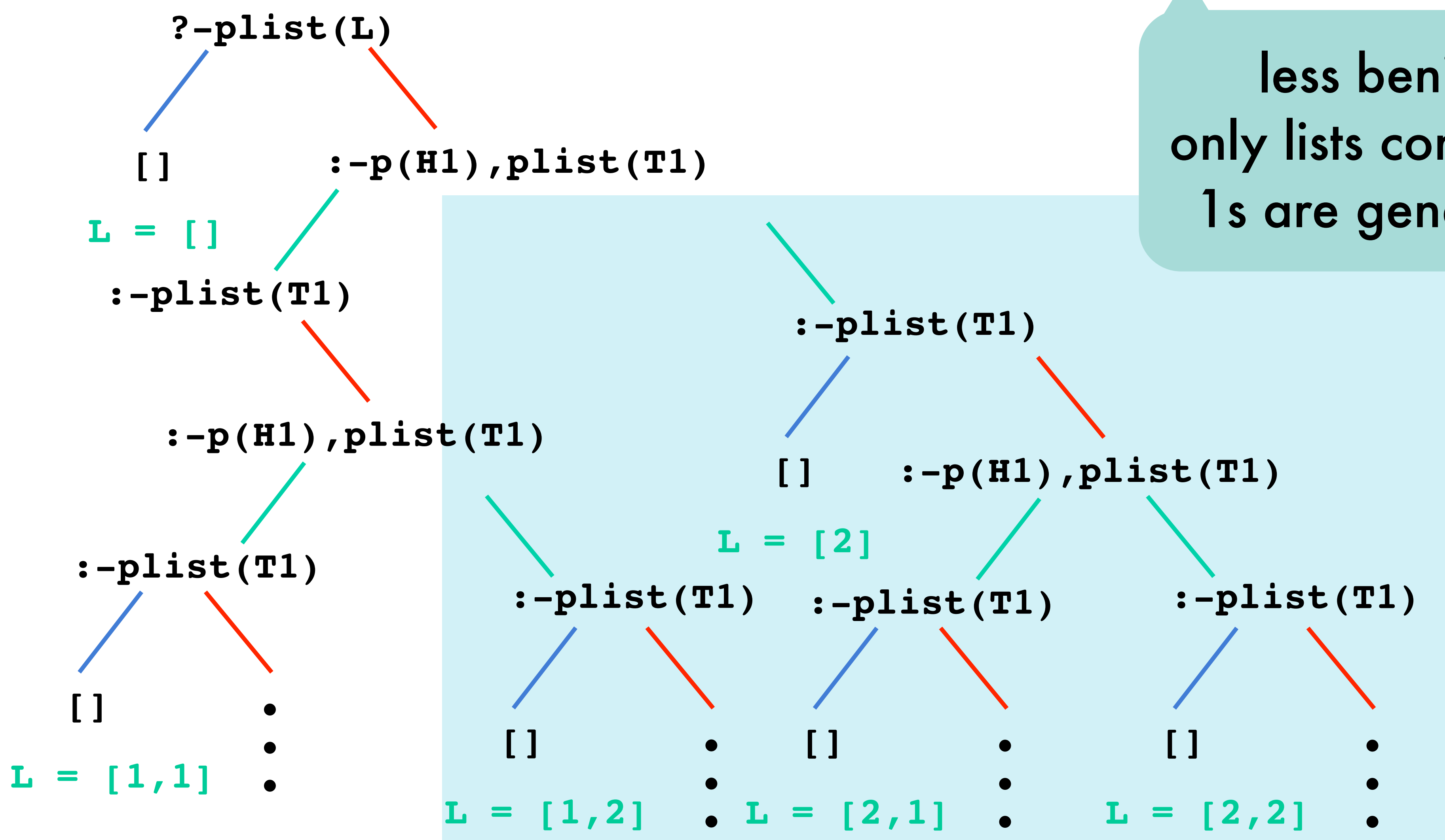


Problems with SLD-resolution refutation: *illustrated on list generation*

```
plist([]).
plist([H|T]):-p(H),plist(T).
p(1).
p(2).
```

```
?-plist(L).
L=[];
L=[1];
L=[1,1];
...
```

less benign:
only lists containing 1s are generated



explored by Prolog   success branches that are never reached

SLD-resolution refutation: *implementing backtracking*

amounts to going up one level in SLD-tree and descending into the next branch to the right

when a failure branch is reached (non-empty resolvent which cannot be reduced further), next alternative for the last-chosen program clause has to be tried

requires remembering previous resolvents for which not all alternatives have been explored together with the last program clause that has been explored at that point

backtracking=
popping resolvent from stack and
exploring next alternative

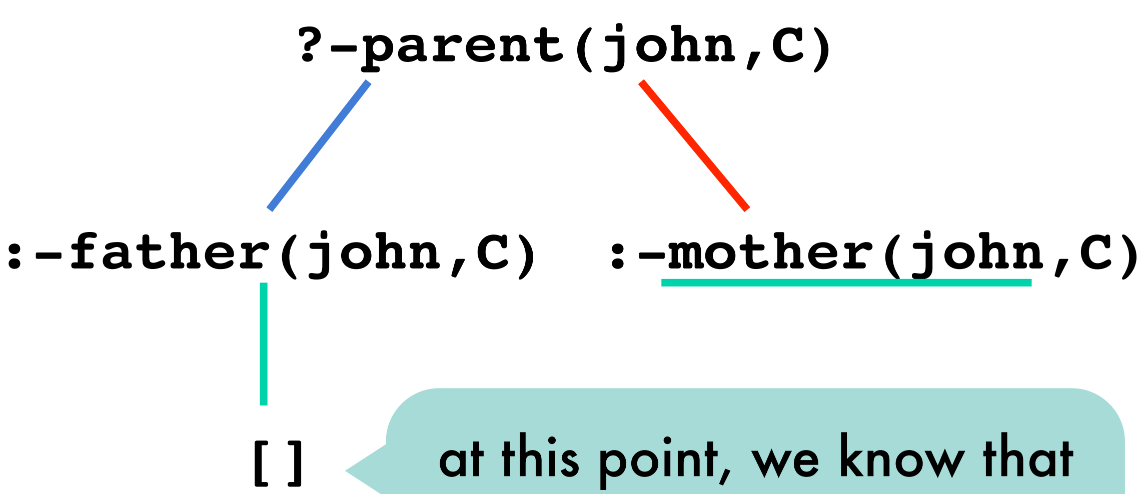
Pruning the search by means of cut: *cutting choice points*

need to be **remembered** for all resolvents for which not all alternatives have been explored

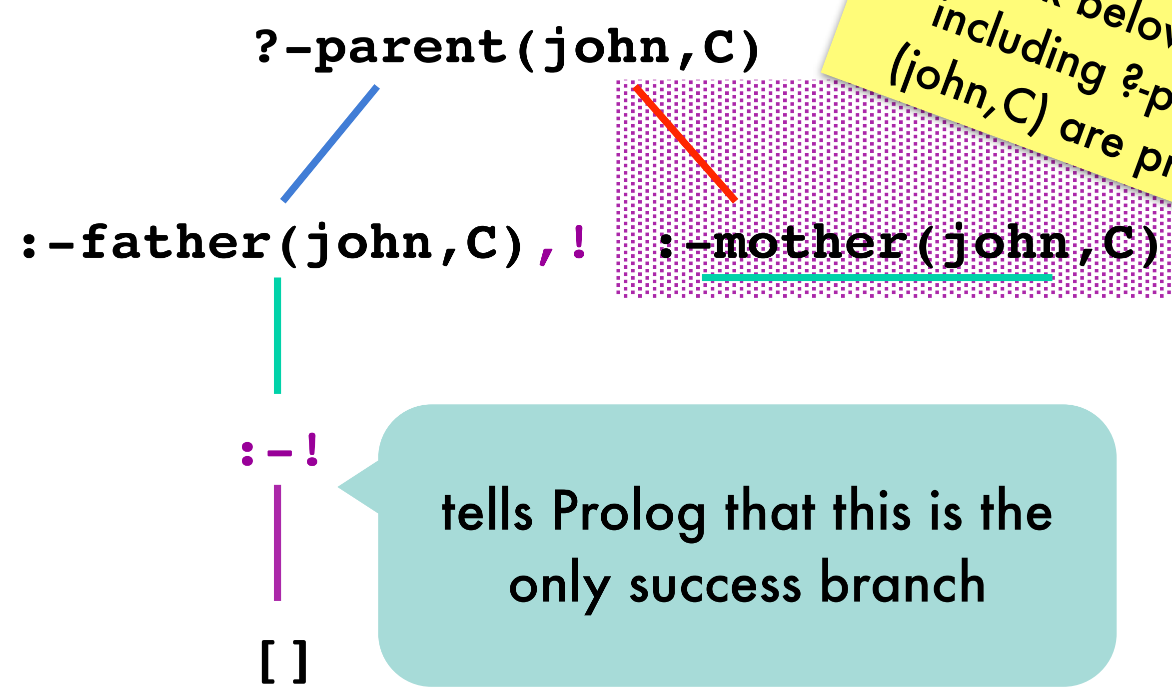
unnecessary alternatives **will eventually be explored**

```
parent(X,Y):-father(X,Y).
parent(X,Y):-mother(X,Y).
father(john,paul).
mother(mary,paul).
```

```
parent(X,Y):-father(X,Y),!.
parent(X,Y):-mother(X,Y).
father(john,paul).
mother(mary,paul).
```



at this point, we know that exploring the alternative clause for parent/2 will fail



choice points on the stack below and including ?-parent (john,C) are pruned

tells Prolog that this is the only success branch

Pruning the search by means of cut: *operational semantics*

"Once you've reached me, stick with all variable substitutions you've found after you entered my clause"

Prolog won't try alternatives for:

literals left to the cut

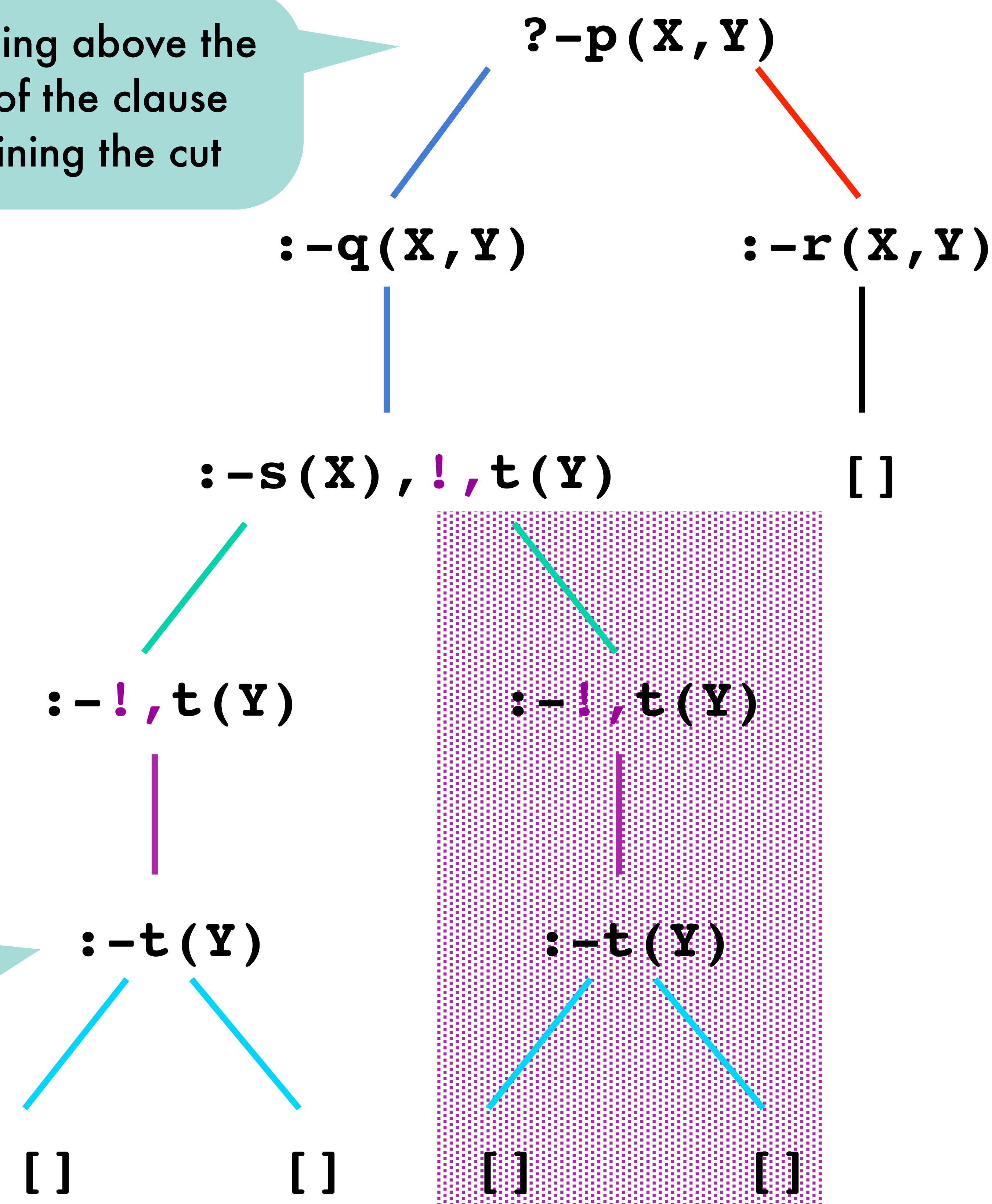
nor the clause in which the cut is found

A cut evaluates to true.

Pruning the search by means of cut: *an example*

no pruning above the head of the clause containing the cut

```
p(X,Y):-q(X,Y).
p(X,Y):-r(X,Y).
q(X,Y):-s(X),!,t(Y).
r(c,d).
s(a).
s(b).
t(a).
t(b).
```



Are not yet on the stack when cut is reached.

no pruning for literals right to the cut

Pruning the search by means of cut: *different kinds of cut*

green cut

does not prune away success branches

stresses that the conjuncts to its left are deterministic and therefore do not have alternative solutions

and that the clauses below with the same head won't result in alternative solutions either

red cut

prunes success branches

some logical consequences of the program are not returned

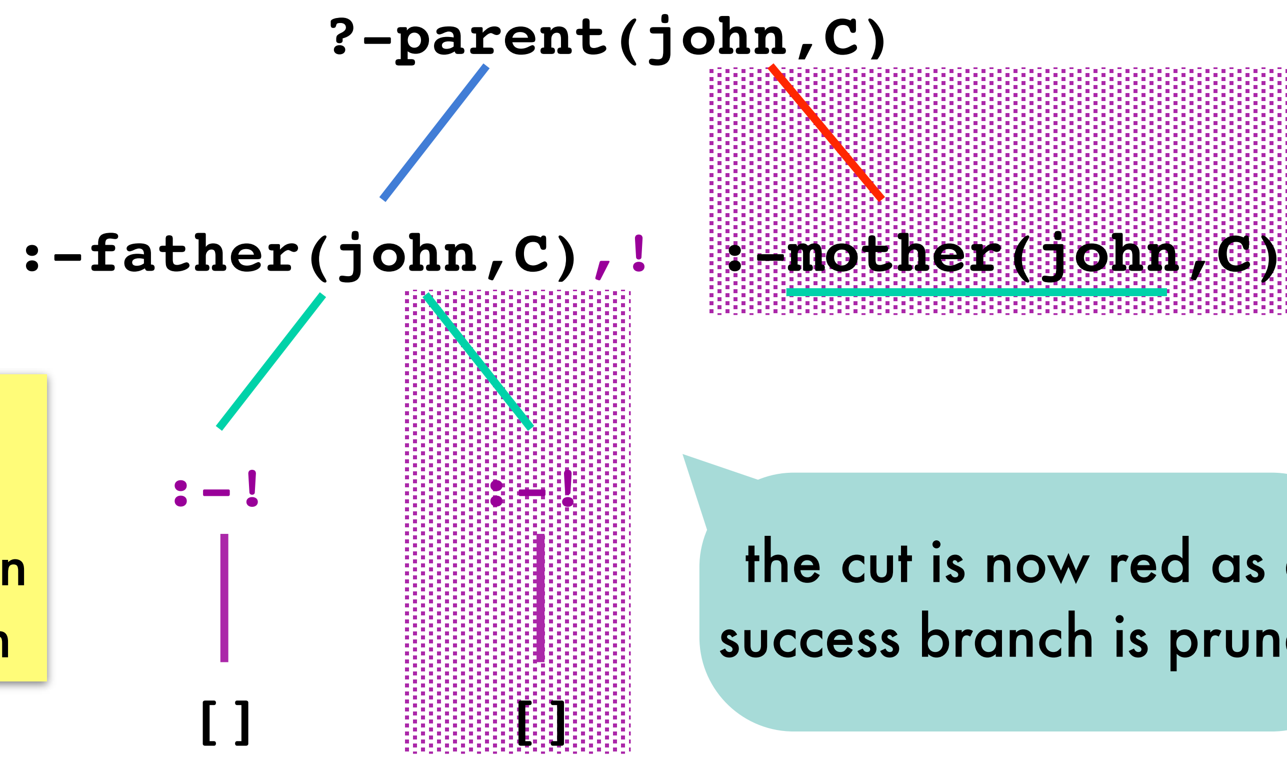
has the declarative and procedural meaning of the program diverge

Pruning the search by means of cut: *red cuts*

```
parent(X,Y):-father(X,Y),!.
parent(X,Y):-mother(X,Y).
father(john,paul).
father(john,peter).
mother(mary,paul).
mother(mary,peter).
```

same query, but John has multiple children in this program

{~~C/peter~~}

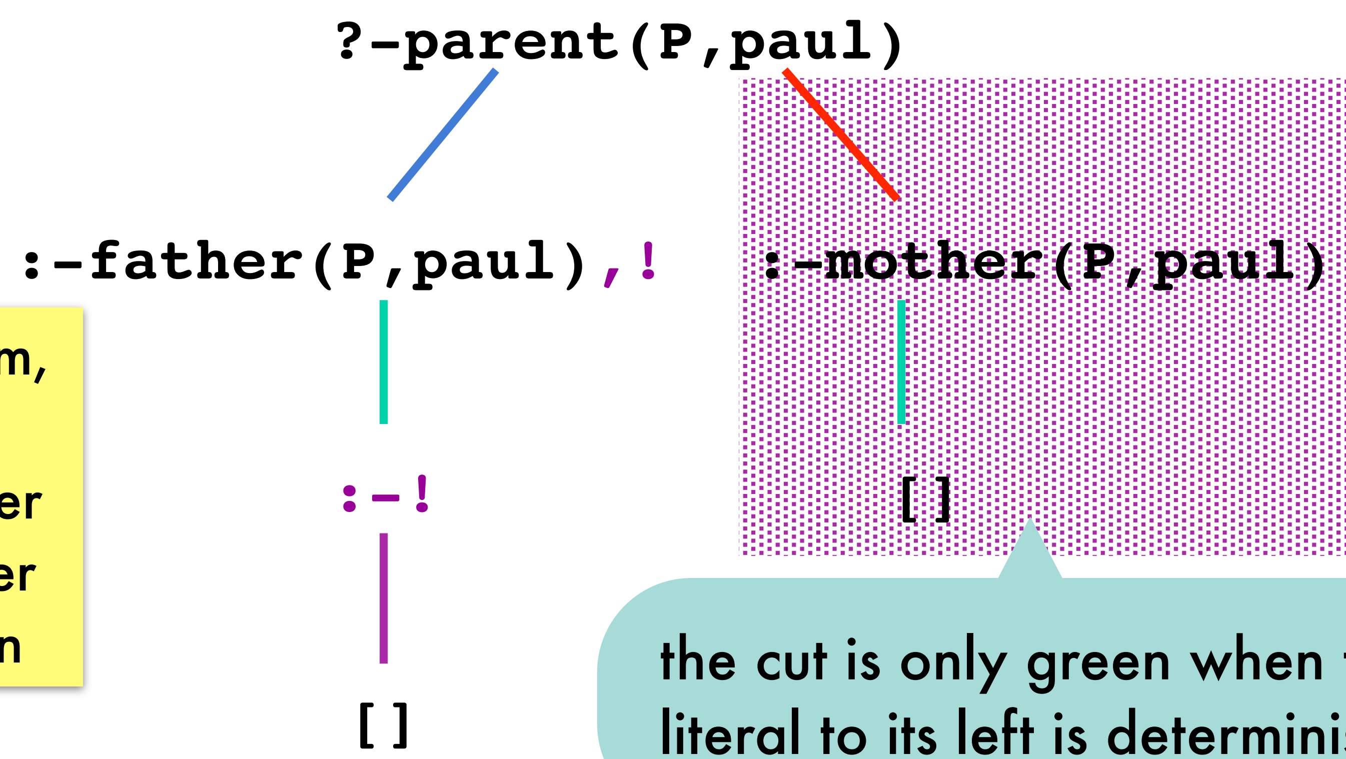


the cut is now red as a success branch is pruned

```
parent(X,Y):-father(X,Y),!.
parent(X,Y):-mother(X,Y).
father(john,paul).
mother(mary,paul).
```

same program, but query quantifies over parents rather than children

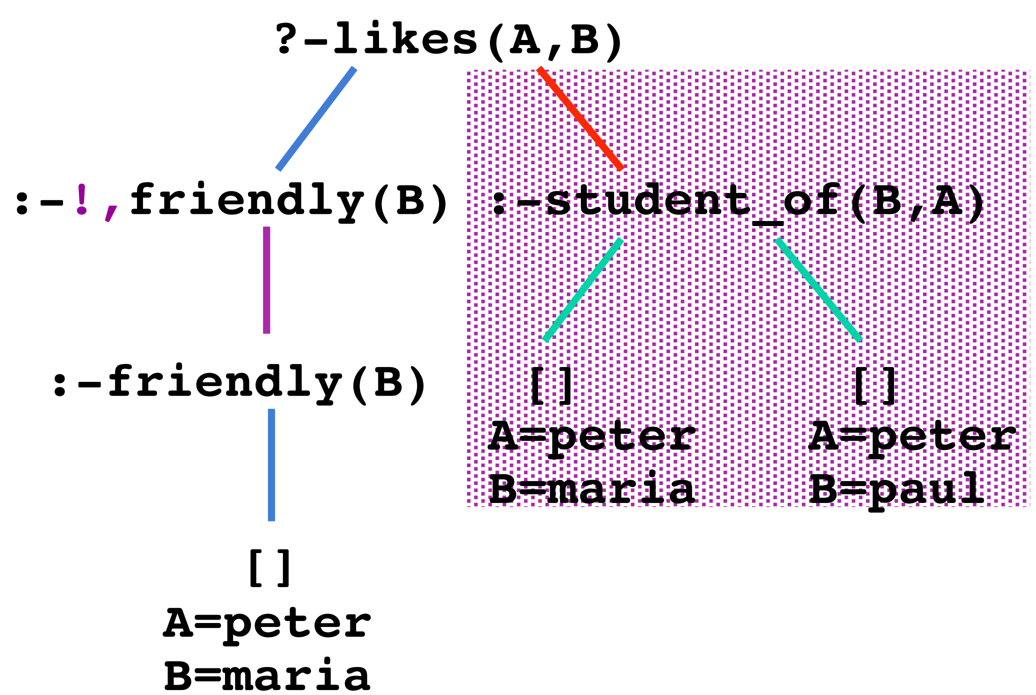
{~~P/mary~~}



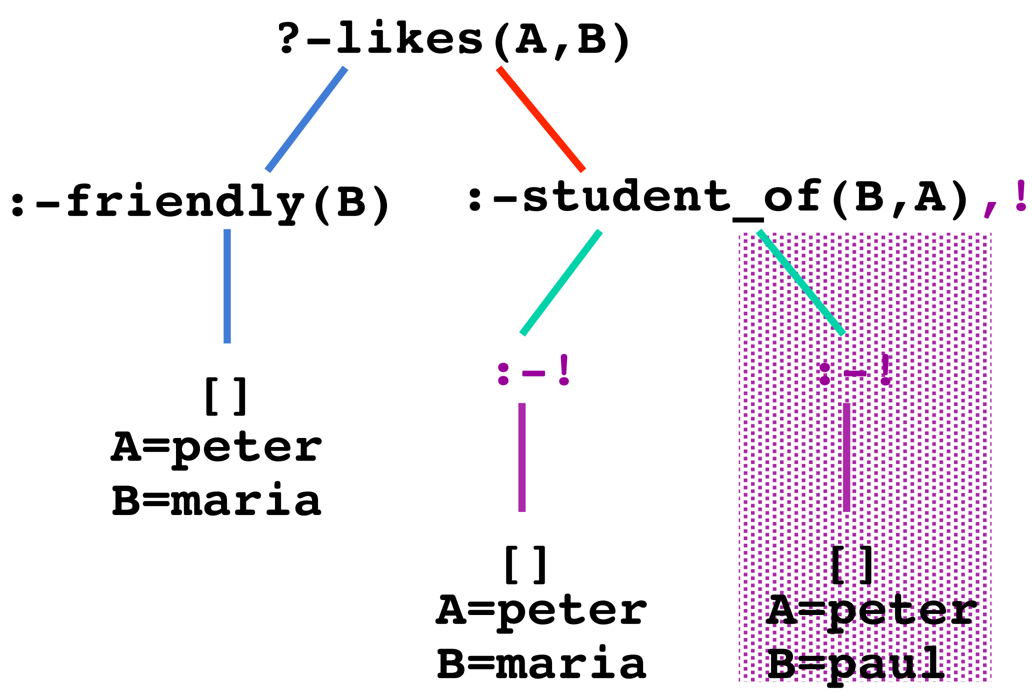
the cut is only green when the literal to its left is deterministic

Pruning the search by means of cut: *placement of cut*

```
likes(peter,Y):-friendly(Y).
likes(T,S):-student_of(S,T).
student_of(maria,peter).
student_of(paul,peter).
friendly(maria).
```




```
likes(peter,Y):-!,friendly(Y).
```




```
likes(T,S):-student_of(S,T),!.
```

Pruning the search by means of cut: *more dangers of cut*

```
max(M,N,M) :- M>=N.
max(M,N,N) :- M=<N.
```

clauses are not mutually exclusive
two ways to solve query `?-max(3,3,5)`

```
max(M,N,M) :- M>=N, !.
max(M,N,N).
```

could use red cut to prune second way

only correct when used in queries with uninstantiated third argument

Better to use >= and <

problem:
`?-max(5,3,3)`
succeeds

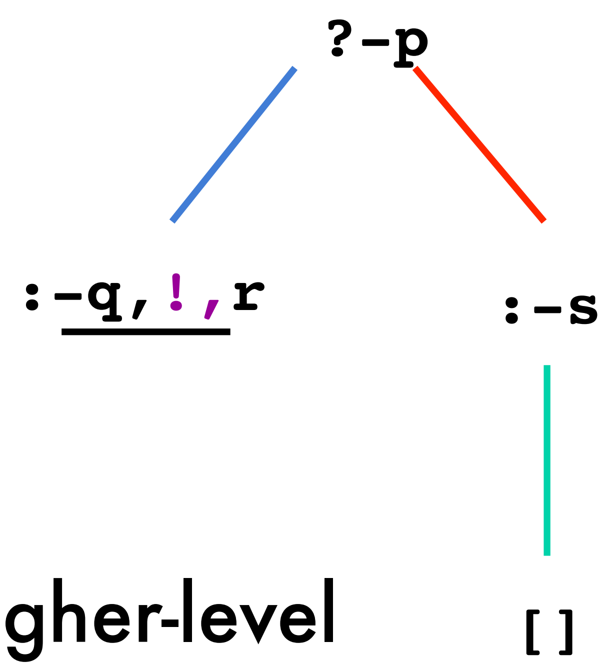
Negation as failure: *specific usage pattern of cut*

cut is often used to ensure clauses are mutually exclusive

cf. previous example

```
p :- q,!,r.
p :- s.
```

only tried when q fails



such uses are equivalent to the higher-level

```
p :- q,r.
p :- not_q,s.
```

where

```
not_q:-q,!,fail.
not_q.
```

built-in predicate always false

Prolog's not/1 meta-predicate captures such uses:

```
not(Goal) :- Goal, ! fail.
not(Goal).
```

slight abuse of syntax equivalent to call(Goal)

not(Goal) is proved by failing to prove Goal

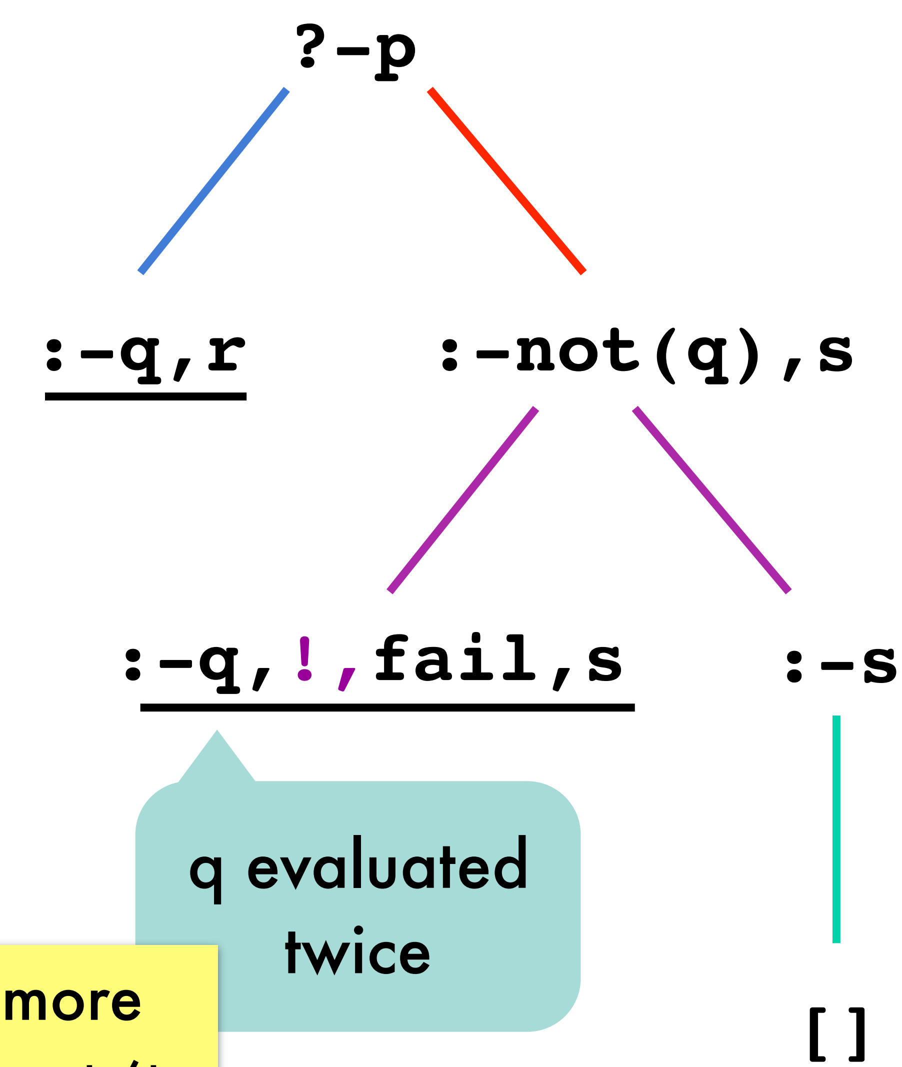
in modern Prologs: use \+ instead of not

Negation as failure:

SLD-tree where not(q) succeeds because q fails

```
p:-q,r.
p:-not(q),s.
s.

not(Goal):-Goal,!,fail.
not(Goal).
```

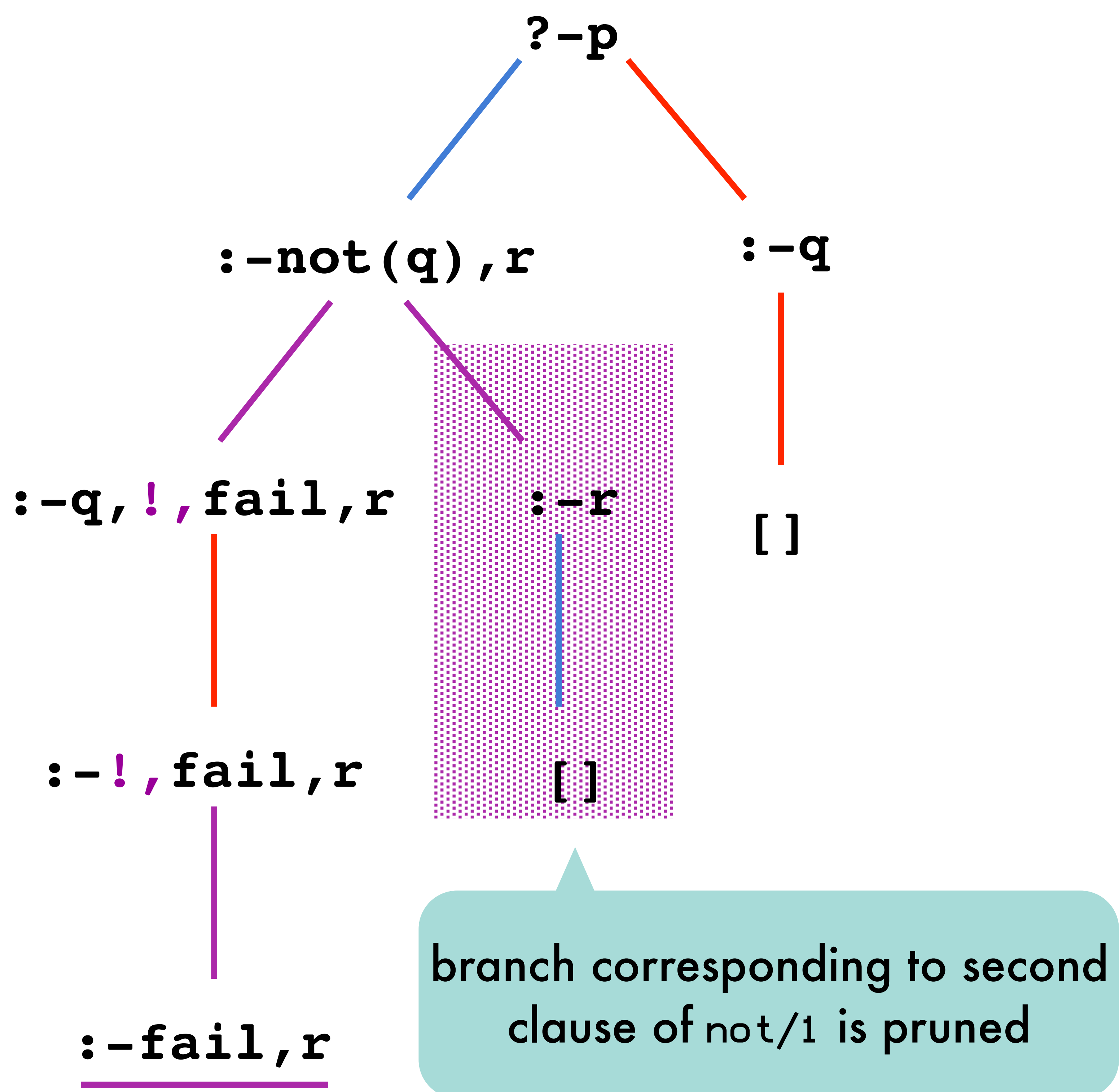


version with ! was more efficient, but uses of not/1 are easier to understand

Negation as failure:
SLD-tree where not(q) fails because q succeeds

```
p:-not(q),r.
p:-q.
q.
r.

not(Goal):-Goal,!,fail.
not(Goal).
```

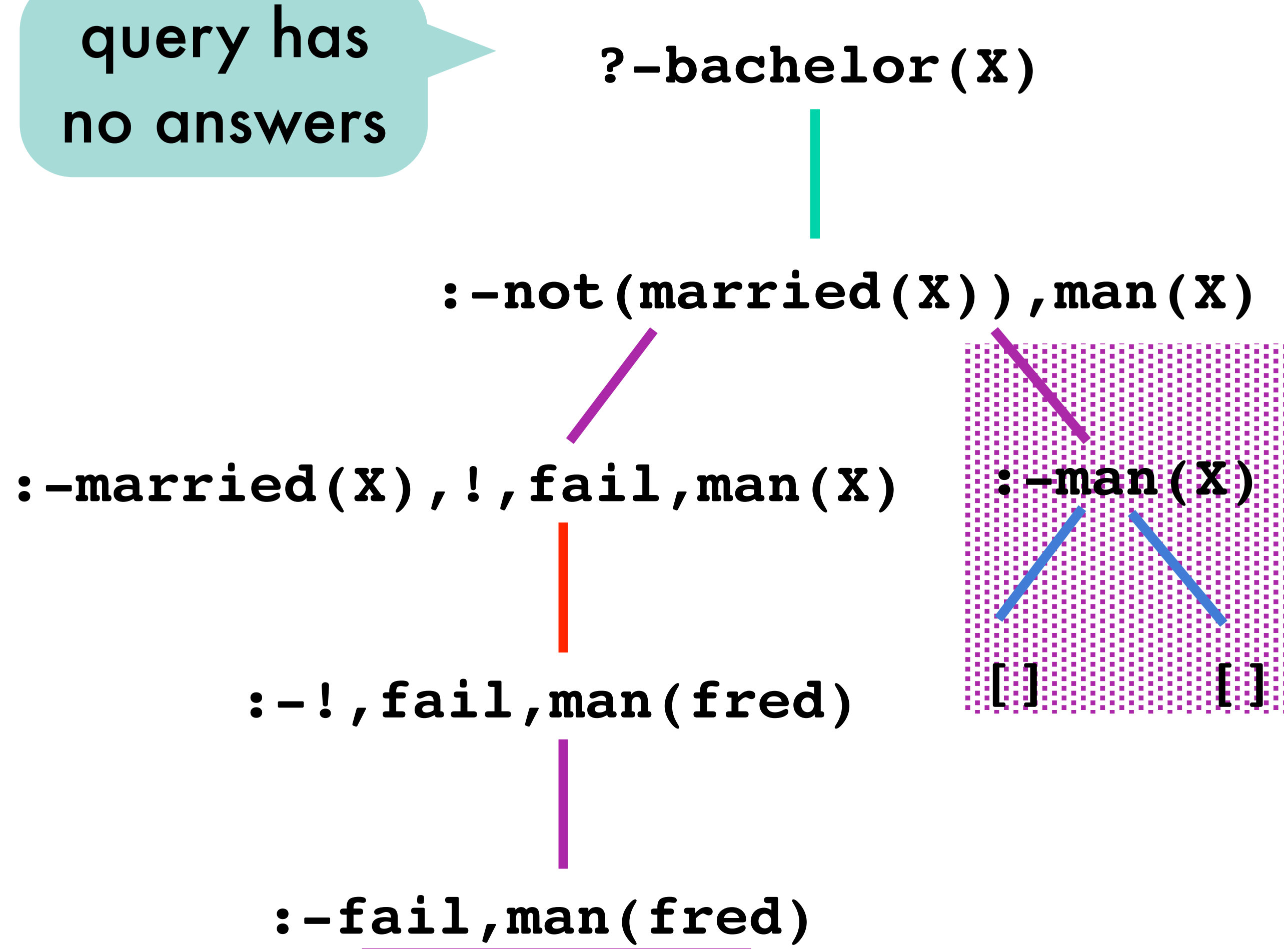


Negation as failure:
floundering occurs when argument is not ground

```
bachelor(X):-not(married(X)),
              man(X).
man(fred).
man(peter).
married(fred).
```

unintentionally interpreted as "X is a bachelor if nobody is married and X is man"

query has no answers




```
not(Goal):-Goal,!,fail.
not(Goal).
```

these are the bachelors we were looking for!

Negation as failure:
avoiding floundering

correct implementation of SLDNF-resolution:
`not(Goal)` fails only if `Goal` has a refutation with an **empty** answer substitution

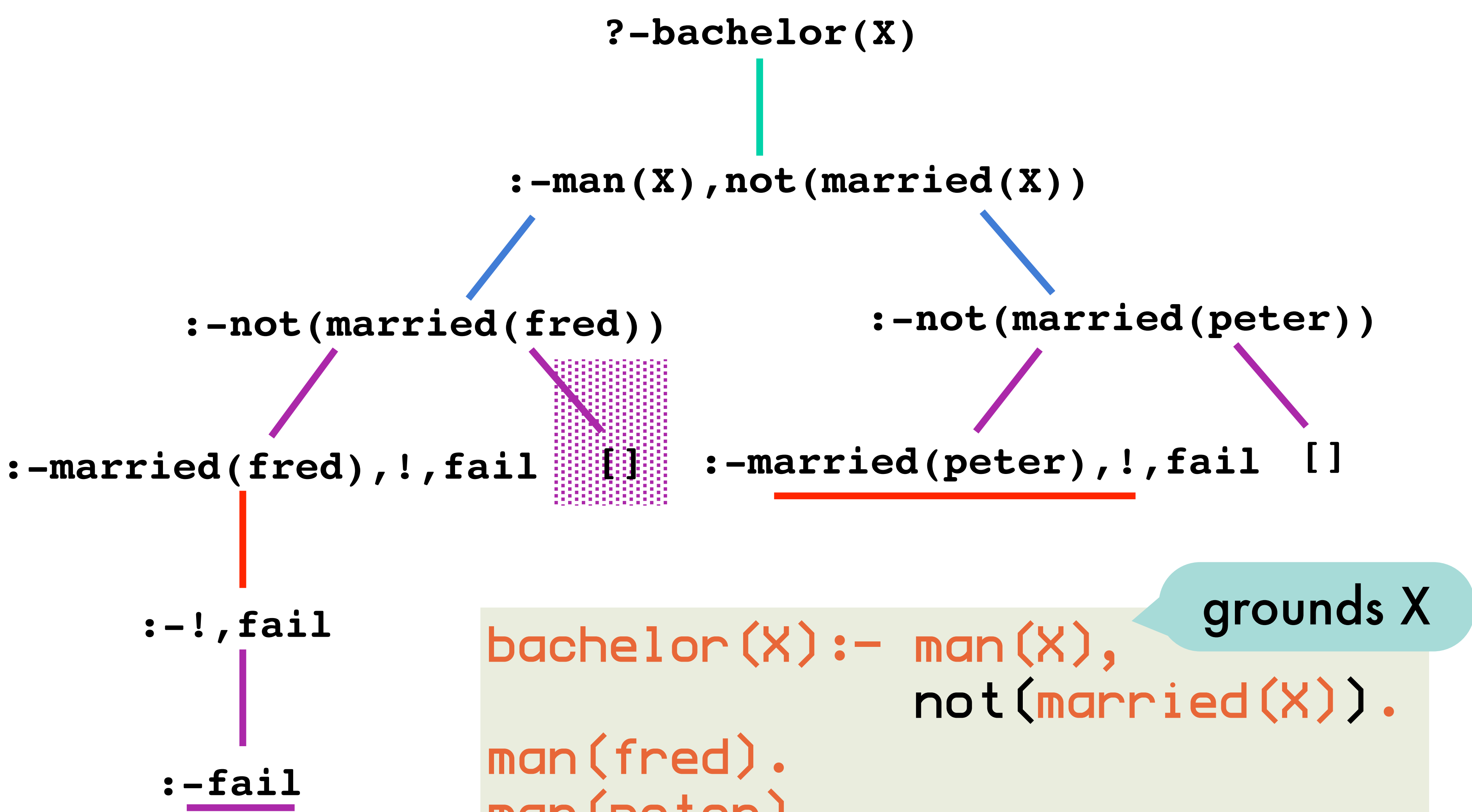
Prolog does not perform this check:
not(married(X)) failed because
married(X) succeeded with {X/fred}

work-around: if `Goal` is ground, only
empty answer substitutions are possible

grounds X

```
bachelor(X):- man(X),
              not(married(X)).
man(fred).
man(peter).
married(fred).
```

Negation as failure:
avoiding floundering




```
bachelor(X):- man(X),
              not(married(X)).
man(fred).
man(peter).
married(fred).
```