POPL class 2 (2020-04-27)

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

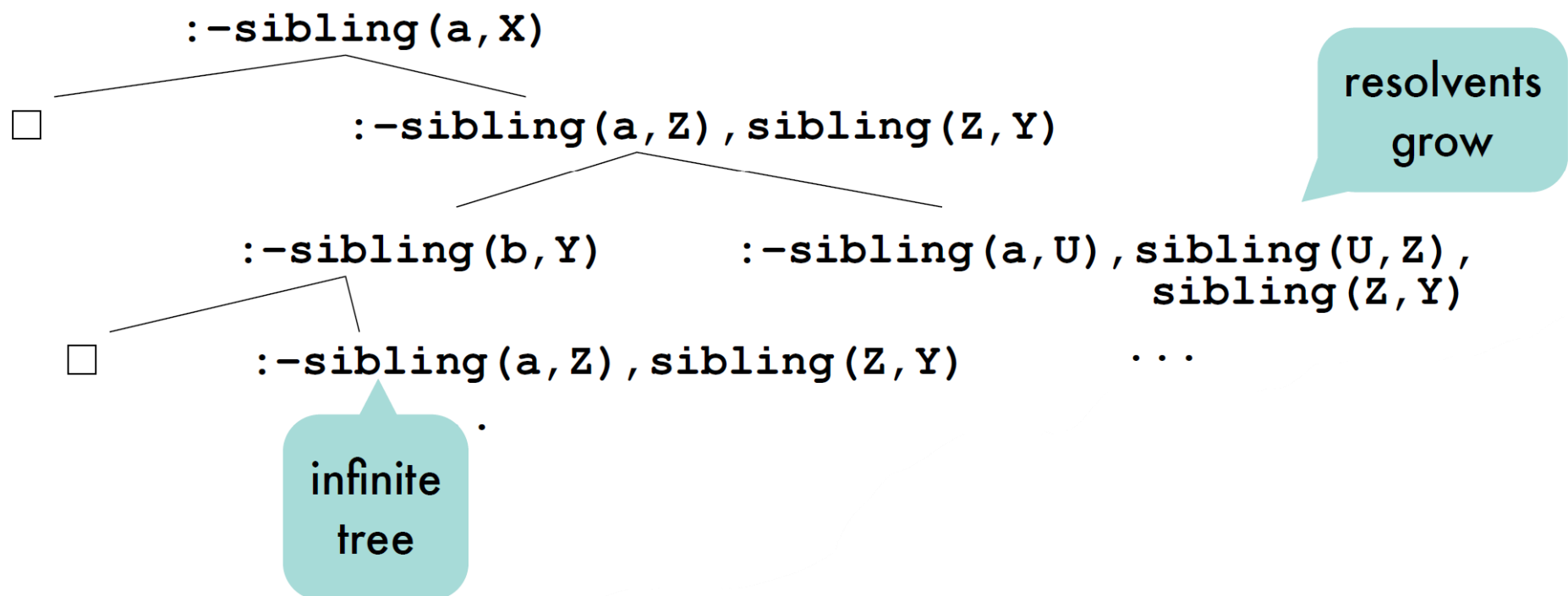
SLD

linear resolution

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

refers to the shape of the resulting proof trees

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

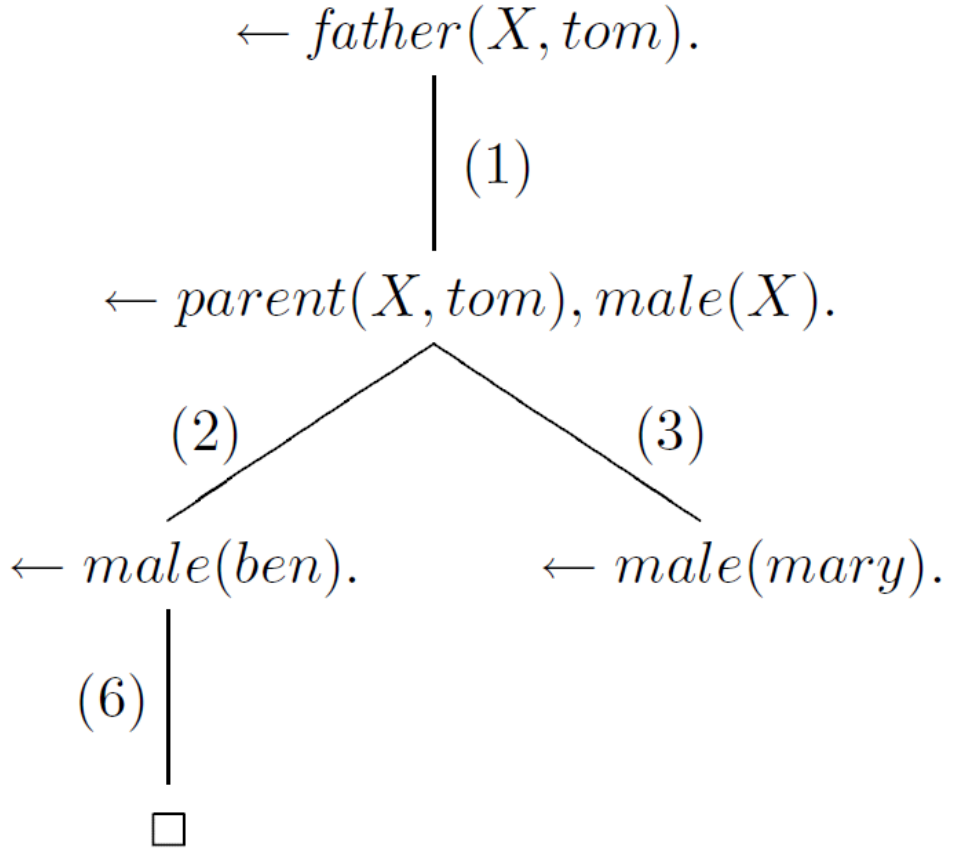


Cut : Pruning the SLD tree

- SLD-tree of a goal may have many failed branches
- programmer may want to prevent the interpreter from constructing failed branches by adding control information
- an infinite branch in the SLD-tree may prevent the interpreter from finding an existing correct answer.
- ‘Cut’ will control the search and avoid construction of some subtrees of the SLD-tree.

Cut

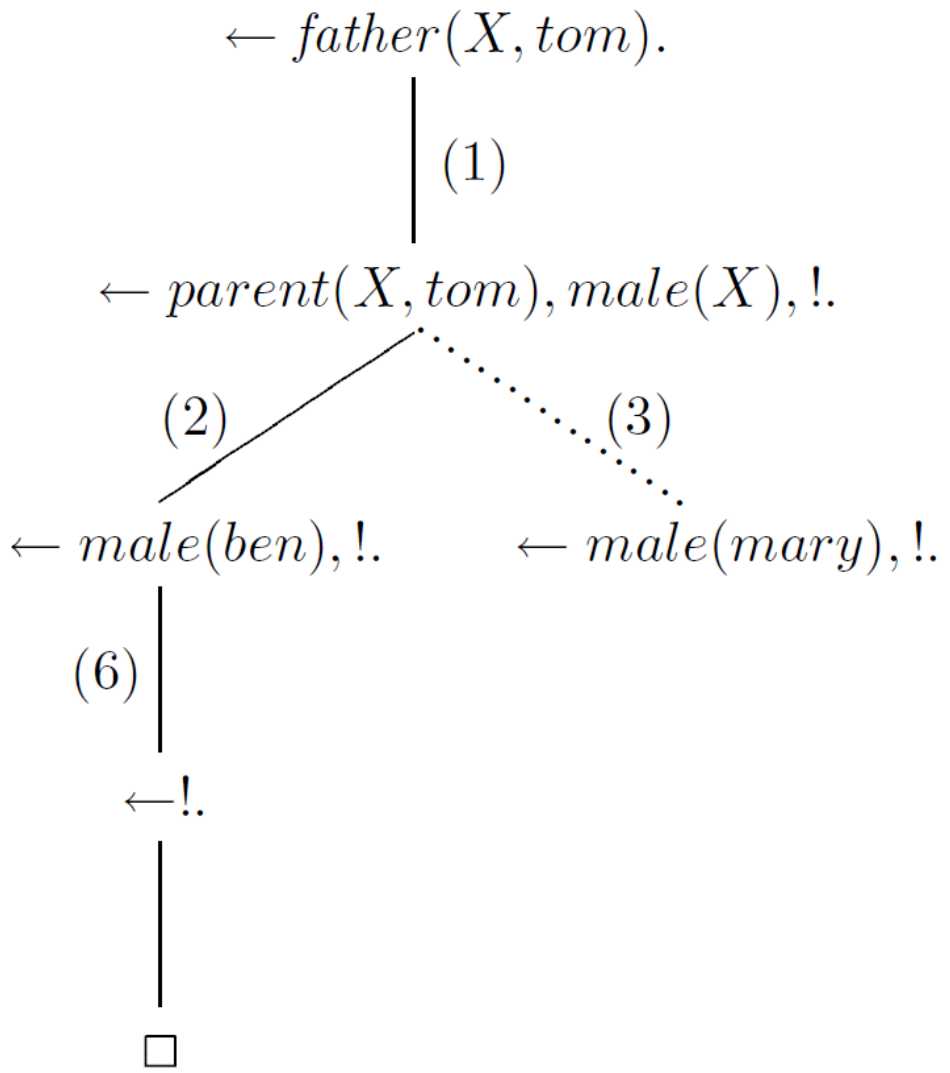
(1) $father(X, Y) \leftarrow parent(X, Y), male(X).$
(2) $parent(ben, tom).$
(3) $parent(mary, tom).$
(4) $parent(sam, ben).$
(5) $parent(alice, ben).$
(6) $male(ben).$
(7) $male(sam).$



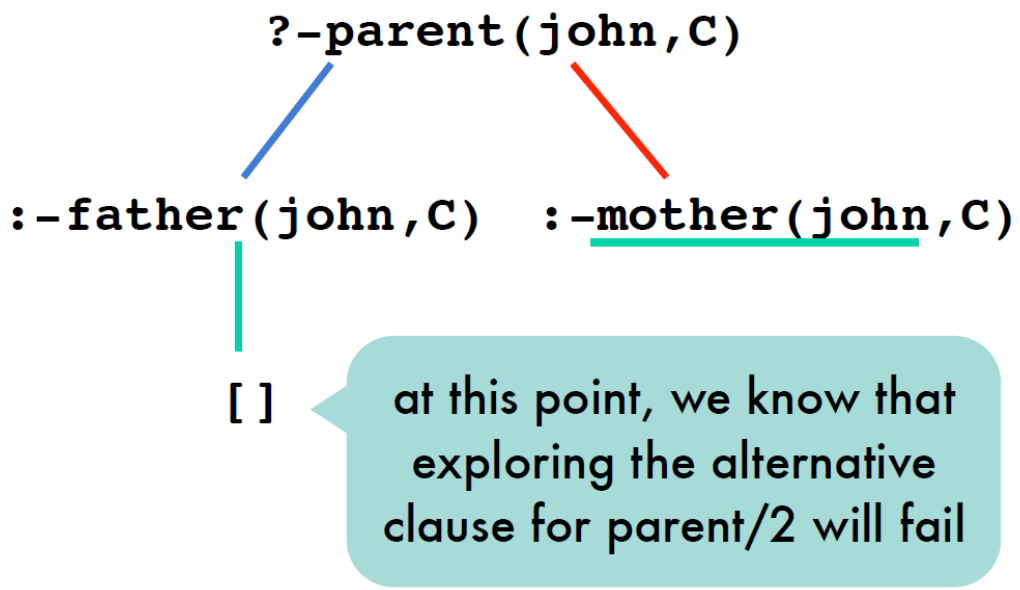
no person has more than one father, When a solution is found the search can be stopped

Cut

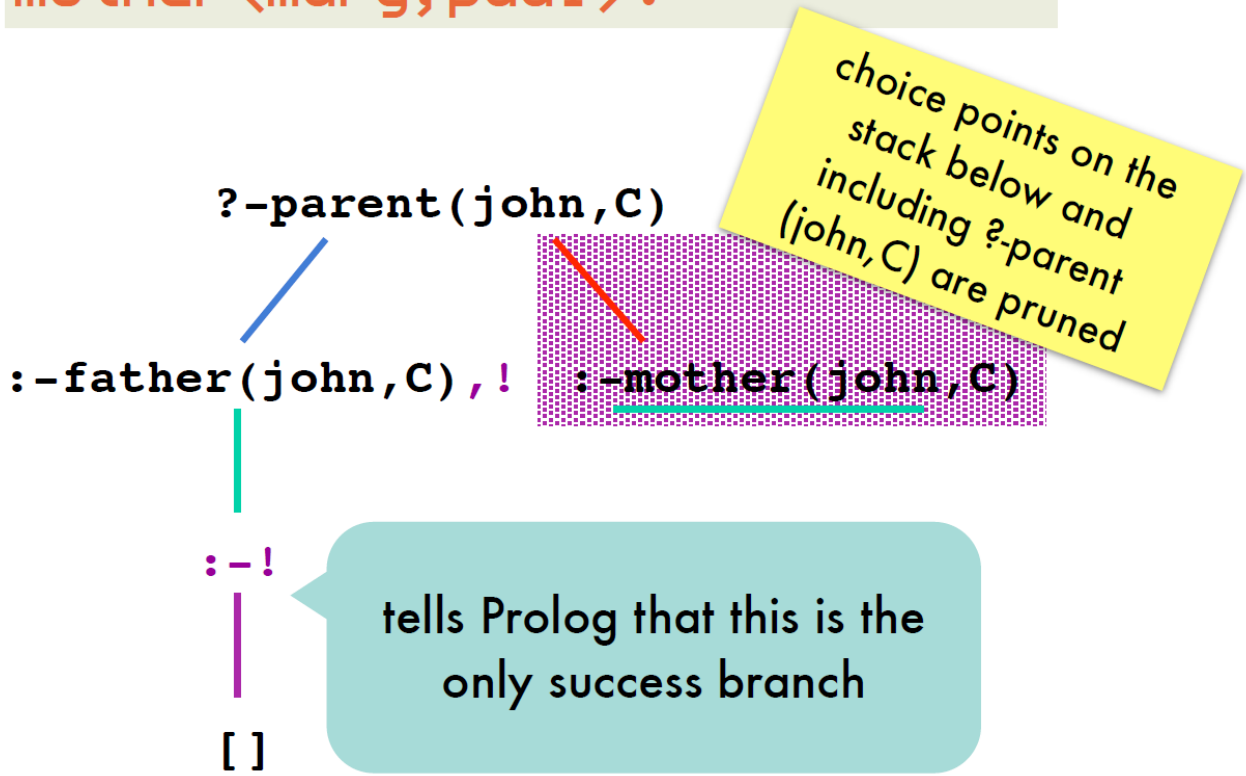
father (*X*; *Y*) ← *parent*(*X*; *Y*), *male*(*X*), !



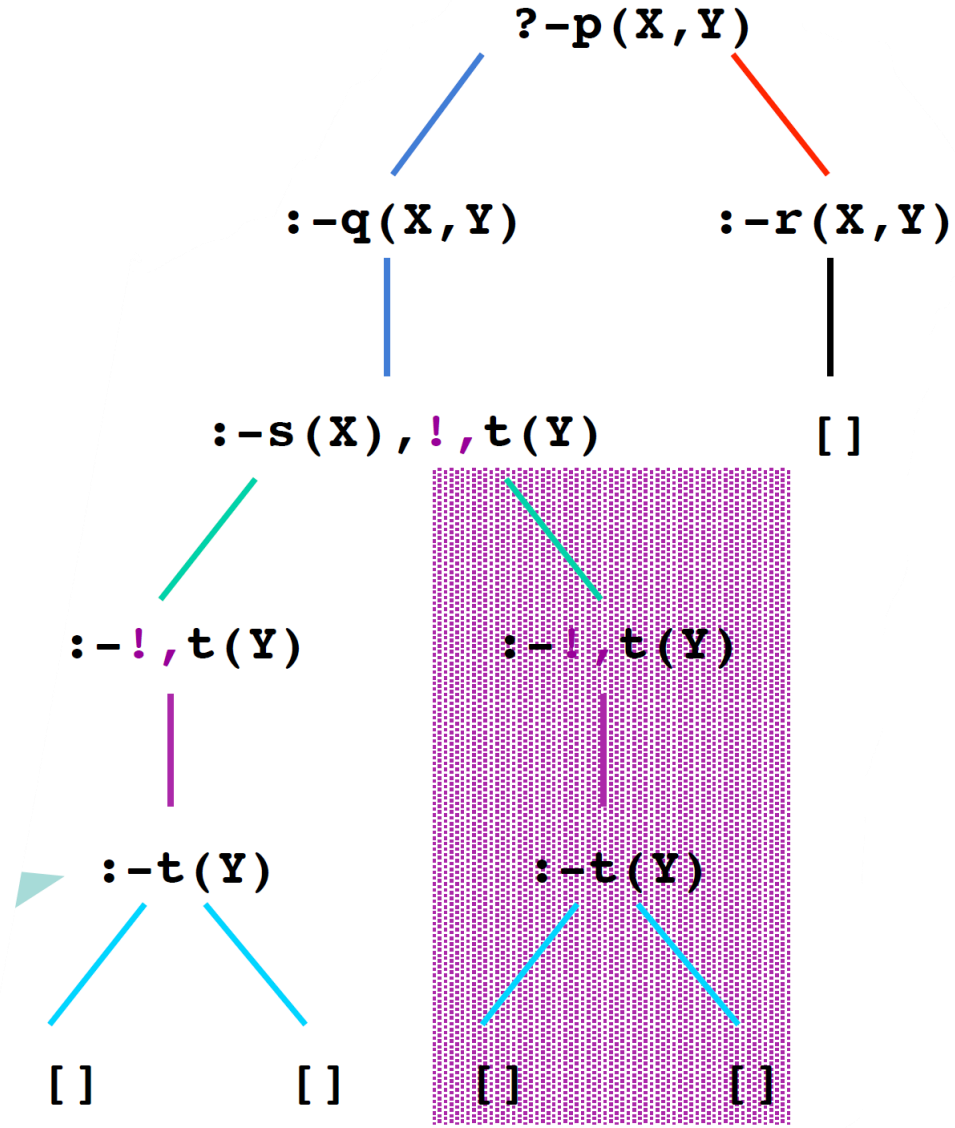
```
parent(X,Y):-father(X,Y).
parent(X,Y):-mother(X,Y).
father(john,paul).
mother(mary,paul).
```



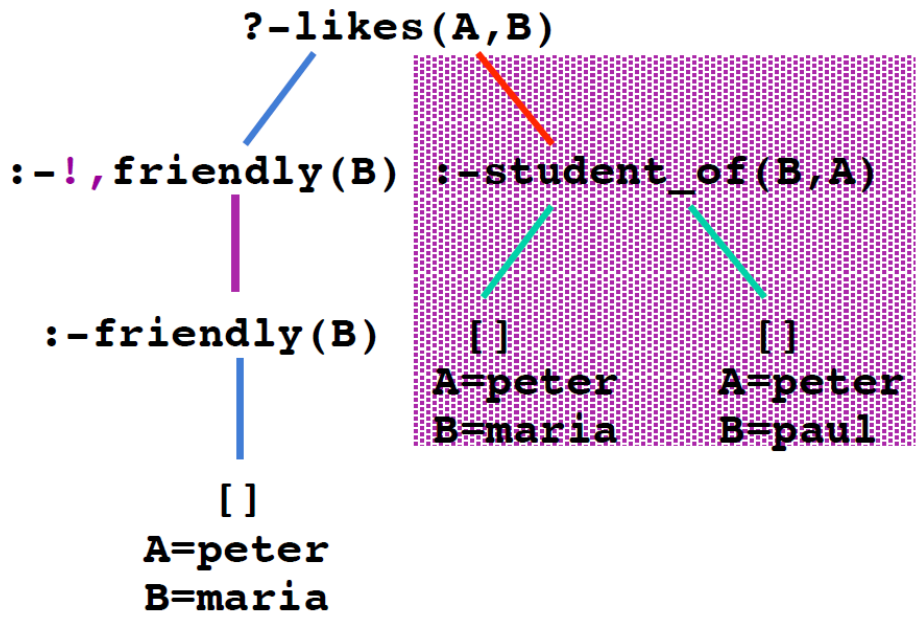

```
parent(X,Y):-father(X,Y),!.
parent(X,Y):-mother(X,Y).
father(john,paul).
mother(mary,paul).
```

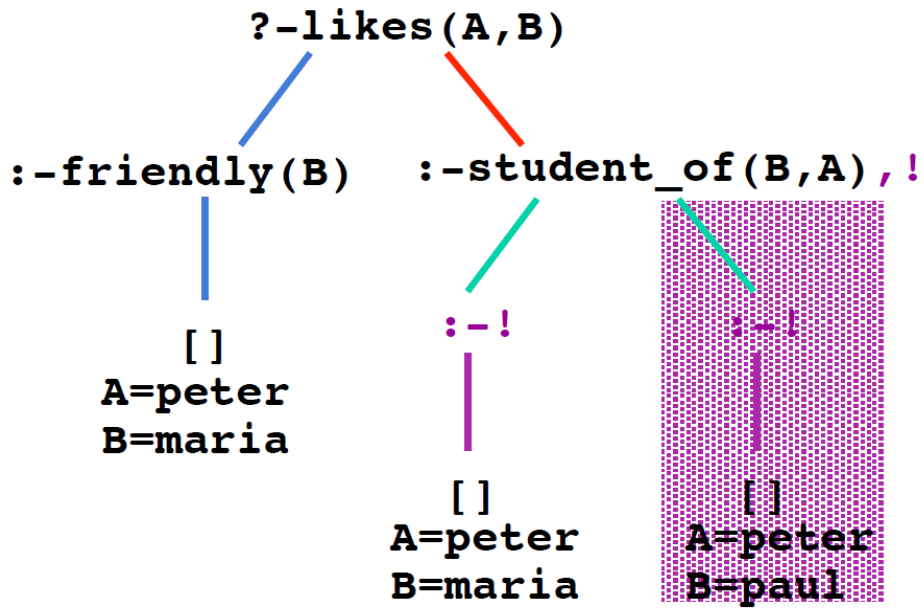


```
p(X,Y):-q(X,Y).
p(X,Y):-r(X,Y).
q(X,Y):-s(X),!,t(Y).
r(c,d).
s(a).
s(b).
t(a).
t(b).
```



```
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).
```

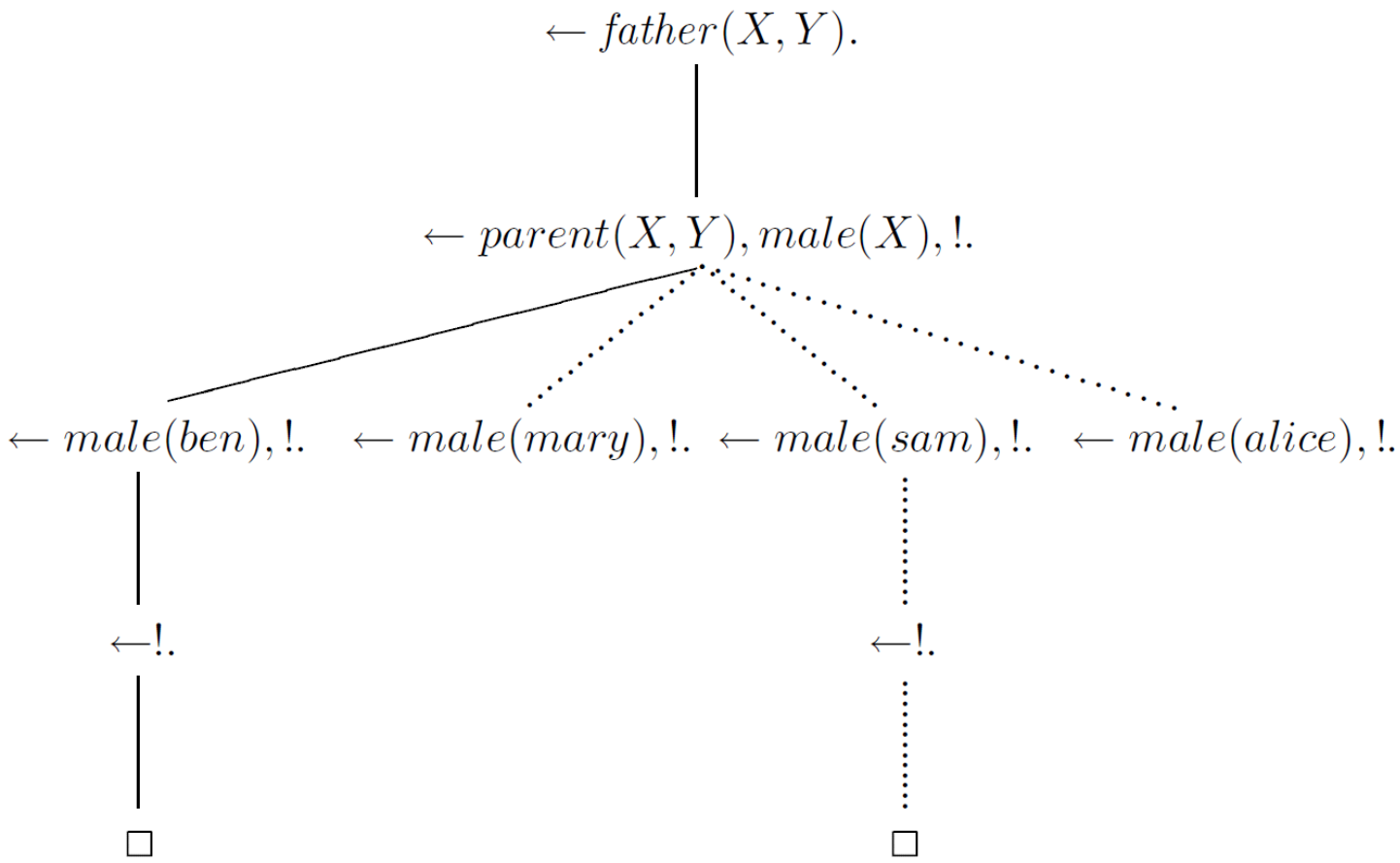
```
likes(T,S):-student_of(S,T),!.
```

Cut problems

- Cannot find more than one element

father (*X; Y*) ← *parent*(*X; Y*), *male*(*X*), !

(1) $father(X, Y) \leftarrow parent(X, Y), male(X).$
(2) $parent(ben, tom).$
(3) $parent(mary, tom).$
(4) $parent(sam, ben).$
(5) $parent(alice, ben).$
(6) $male(ben).$
(7) $male(sam).$



Cut problems

$(1)\quad proud(X) \leftarrow father(X,Y), newborn(Y).$

$(2)\quad father(X,Y) \leftarrow parent(X,Y), male(X).$
$(3)\quad parent(john, mary).$
$(4)\quad parent(john, chris).$
$(5)\quad male(john).$
$(6)\quad newborn(chris).$

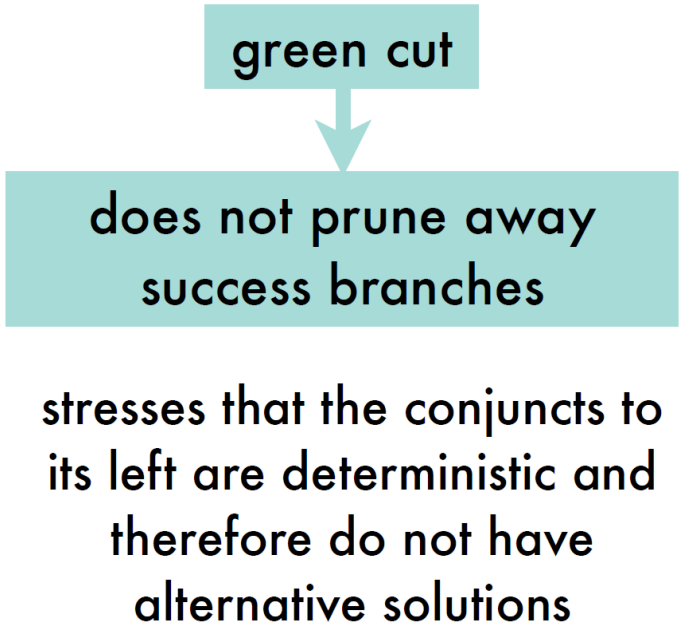
$(2')\quad father(X,Y) \leftarrow parent(X,Y), male(X), !.$

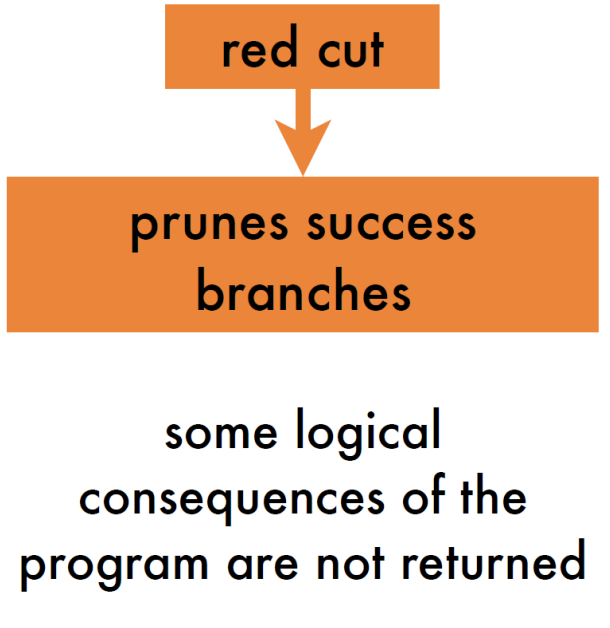
proud(*john*) is “yes" since, as described, John is the father of Chris who is newborn.

answer to the goal *proud*(*john*) is “no".

Cuts

- two principal uses of cut : cut failing branches and to prune succeeding branches.
- Green cuts : Cutting of failing branches is harmless, asit does not alter the answers
- Red cuts : cutting succeeding branches *is* considered harmful



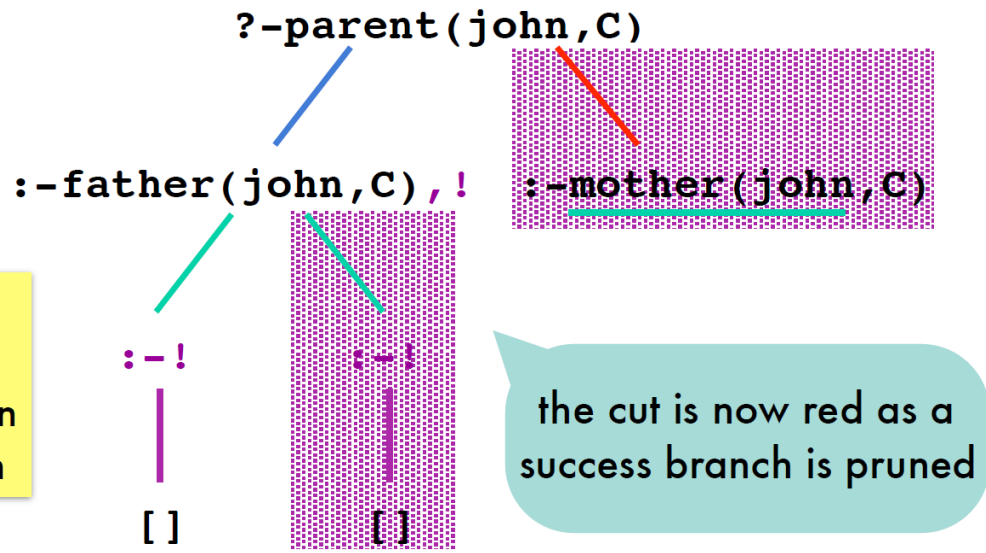


Pruning the search by means of cut: *red cuts*

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

{C/~~peter~~}

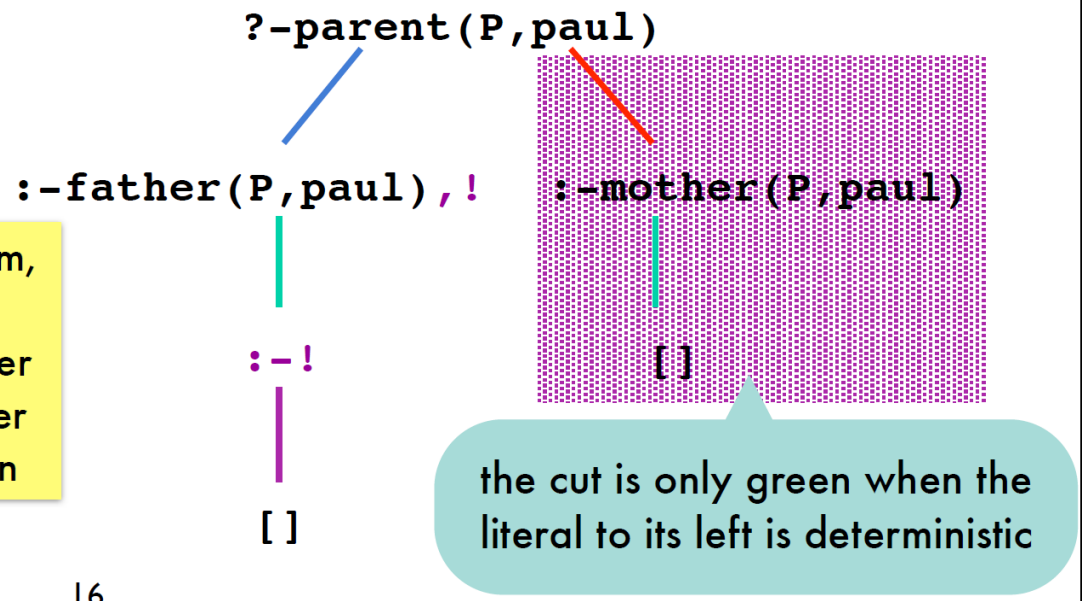
same query, but John has multiple children in this program




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

{P/~~mary~~}

same program, but query quantifies over parents rather than children



Green cut

$$proud(X) \leftarrow father(X, Y), newborn(Y).$$
$$\vdots$$
$$father(john, sue).$$
$$father(john, mary).$$
$$\vdots$$
$$newborn(sue).$$
$$newborn(mary).$$

$$\leftarrow proud(X)$$

How to avoid getting the same answer twice or more.

Red cuts

$$min(X, Y, X) \leftarrow X < Y, !.$$
$$min(X, Y, Y).$$

$$min(X, Y, X) \leftarrow X < Y, !.$$
$$min(X, Y, Y) \leftarrow X \geq Y.$$

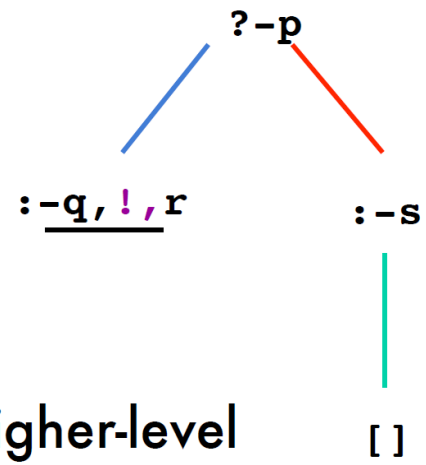
Negation as failure:
specific usage pattern of cut

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:

in modern Prologs: use \+ instead of not

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

not(Goal) is proved by failing to prove Goal

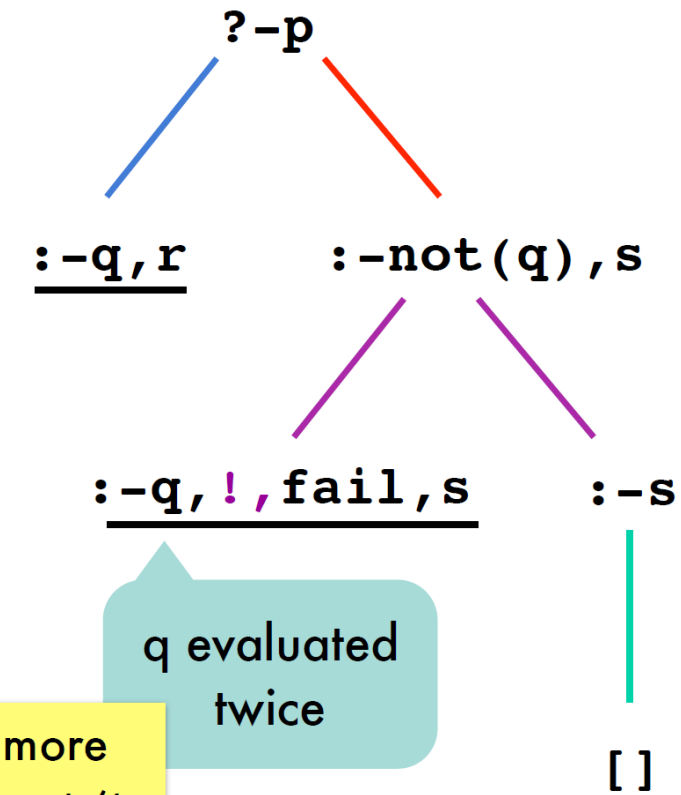
slight abuse of syntax equivalent to call(Goal)

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