

Negation, Search in Prolog

Adapted from slides provided by Peter Flach
for his book Simply Logical

Negation as Failure

not

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

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

not

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

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

This is syntactic sugar for `call(Goal)`

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

not

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

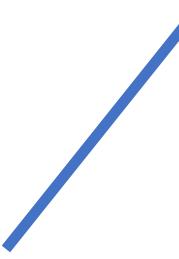
```
not(Goal):-Goal,! ,fail.  
not(Goal).
```

? - p

not

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

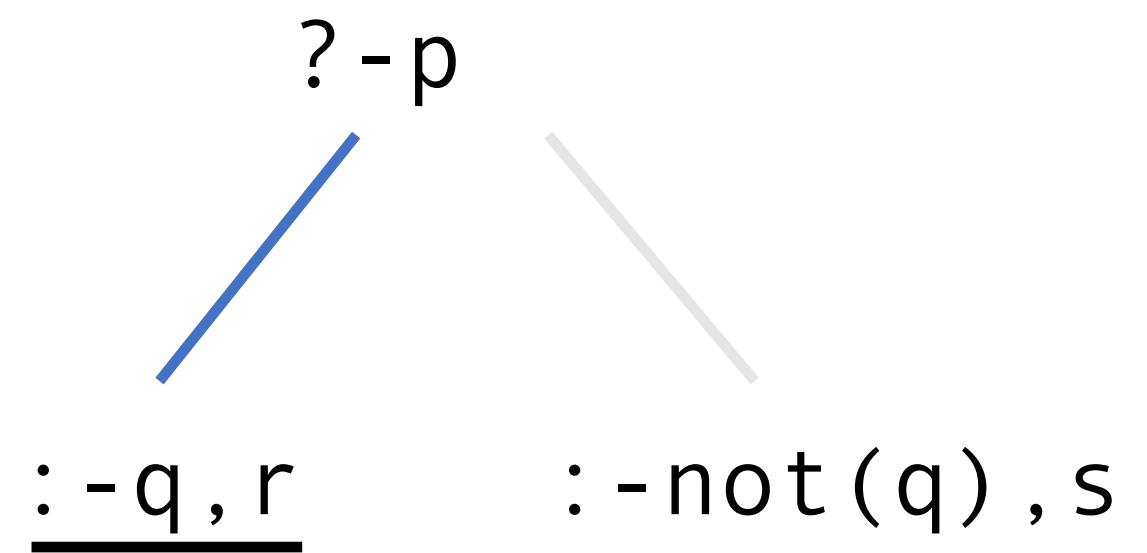
```
not(Goal):-Goal,! ,fail.  
not(Goal).
```

? - p

: - q , r

not

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

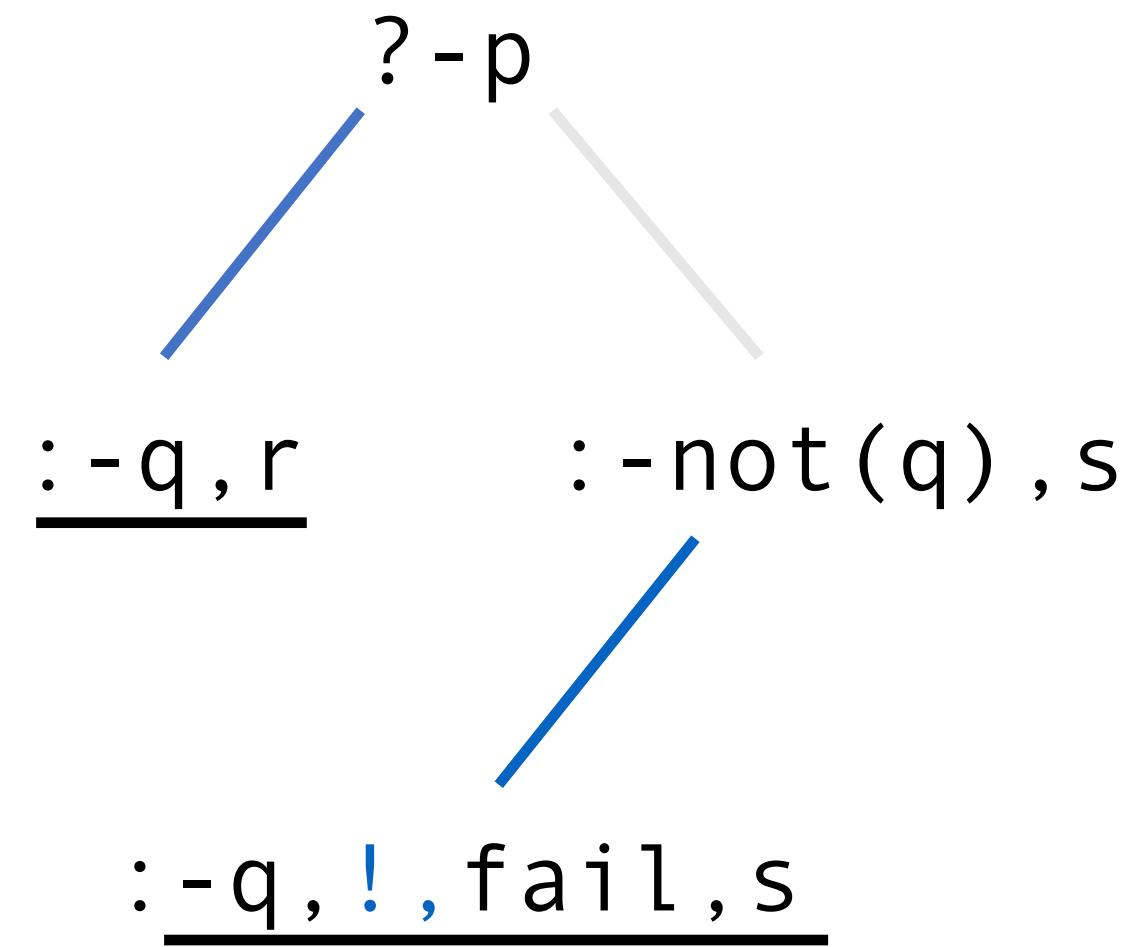
```
not(Goal):-Goal, !, fail.  
not(Goal).
```



not

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

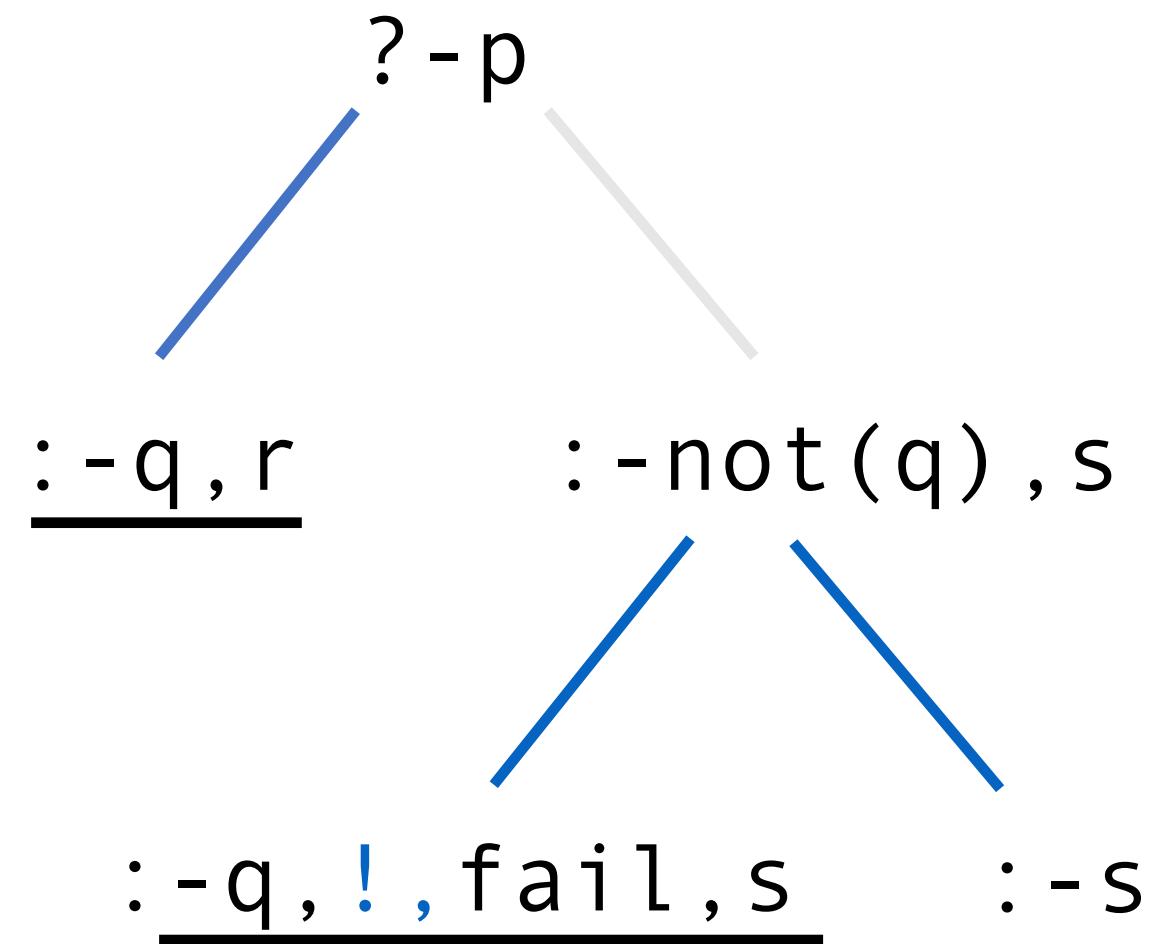
```
not(Goal):-Goal, !, fail.  
not(Goal).
```



not

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

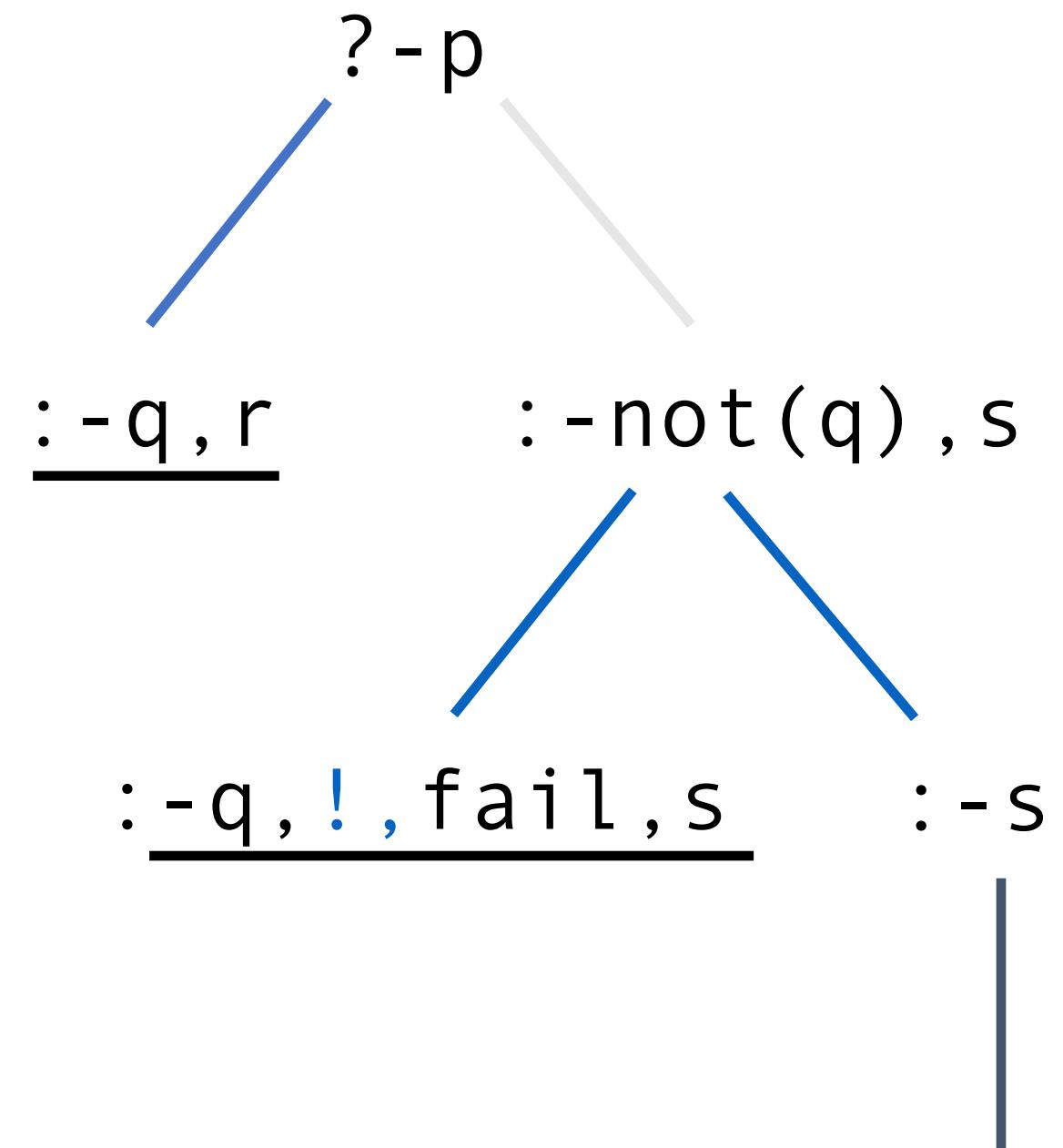
```
not(Goal):-Goal, !, fail.  
not(Goal).
```



not

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

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



An example: How it works when :- not (q) fails

```
p:-not(q),r.  
p:-q.  
q.  
r.  
  
not(Goal):-Goal,! ,fail.  
not(Goal).
```

?- p

An example: How it works when `: - not (q)` fails

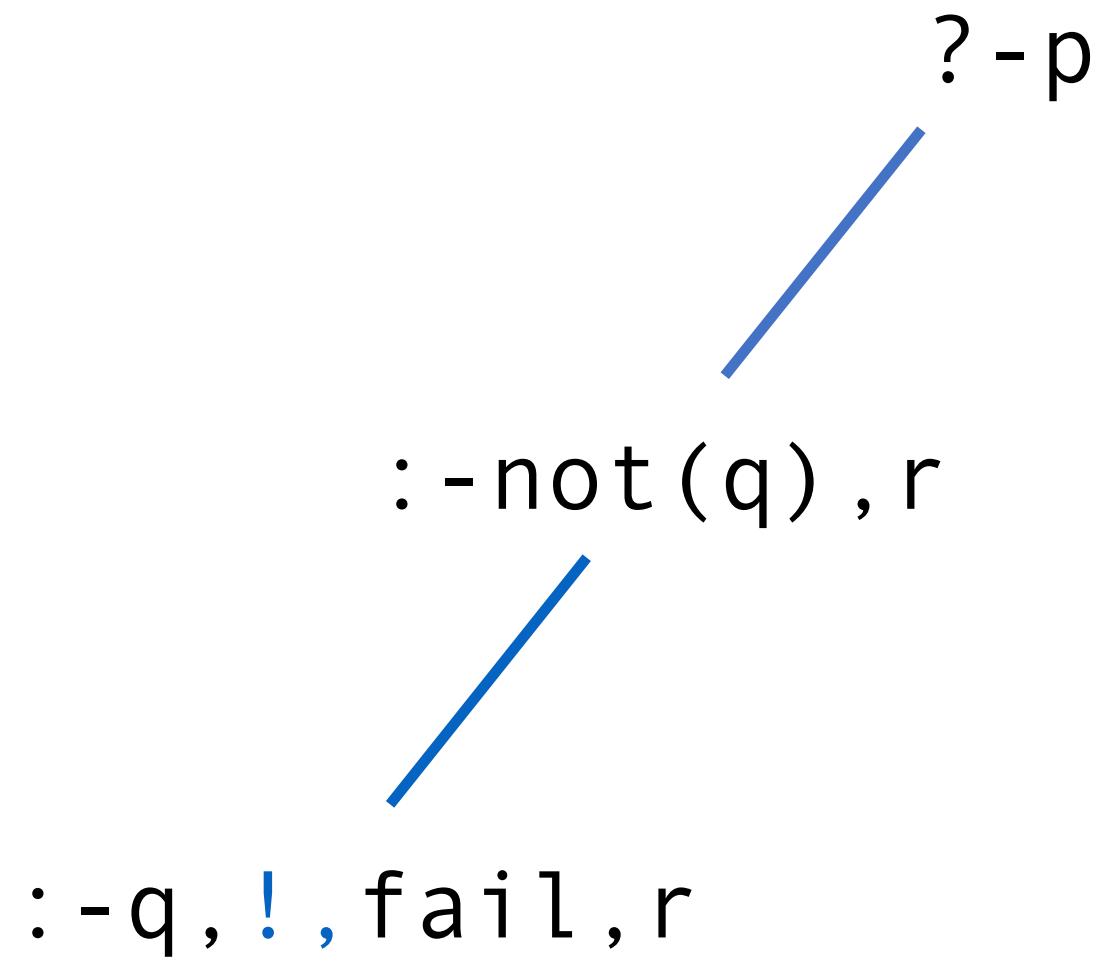
```
p:-not(q),r.  
p:-q.  
q.  
r.  
not(Goal):-Goal,! ,fail.  
not(Goal).
```

?-p

: -not(q) , r

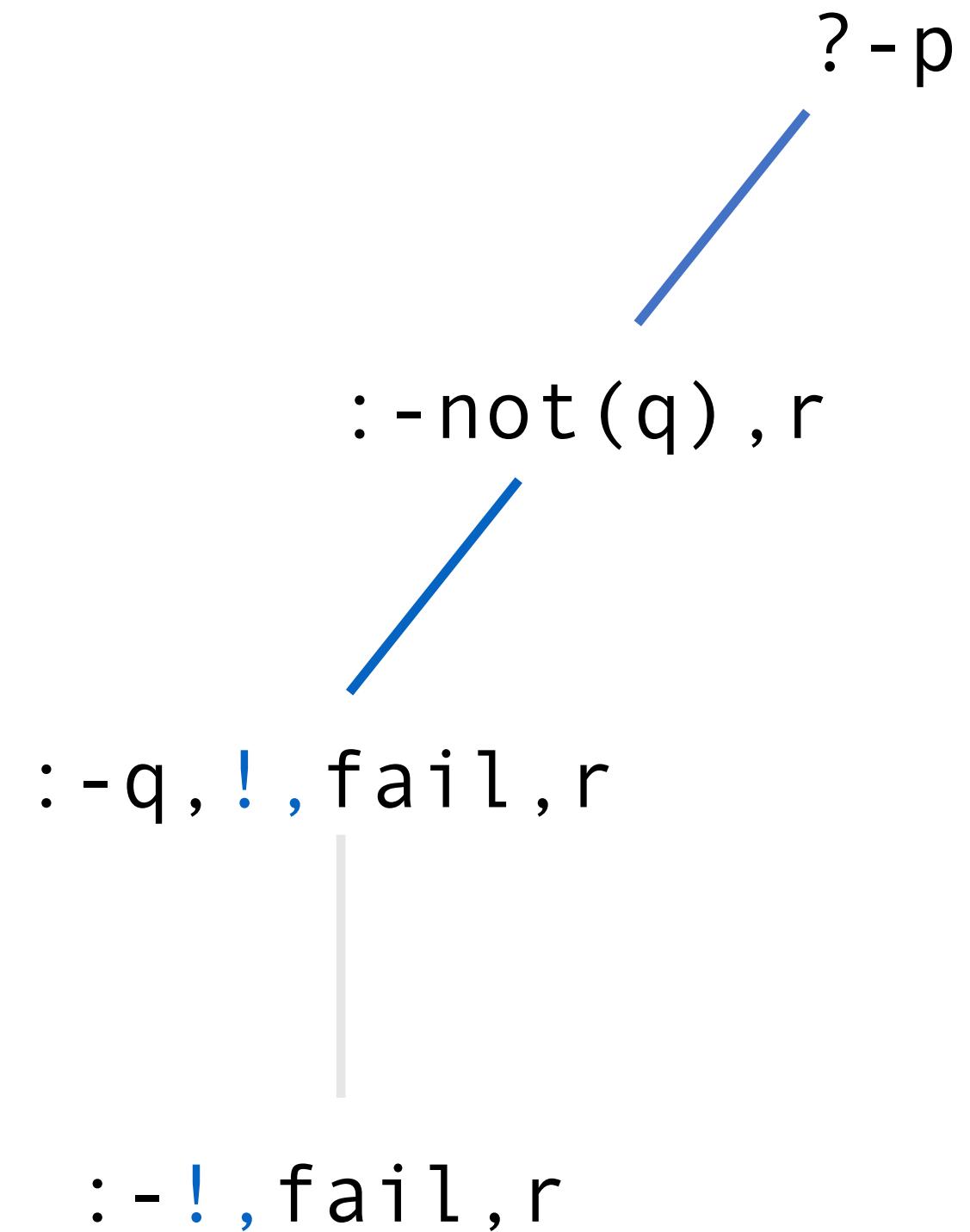
An example: How it works when `: - not (q)` fails

```
p:-not(q),r.  
p:-q.  
q.  
r.  
not(Goal):-Goal,! ,fail.  
not(Goal).
```



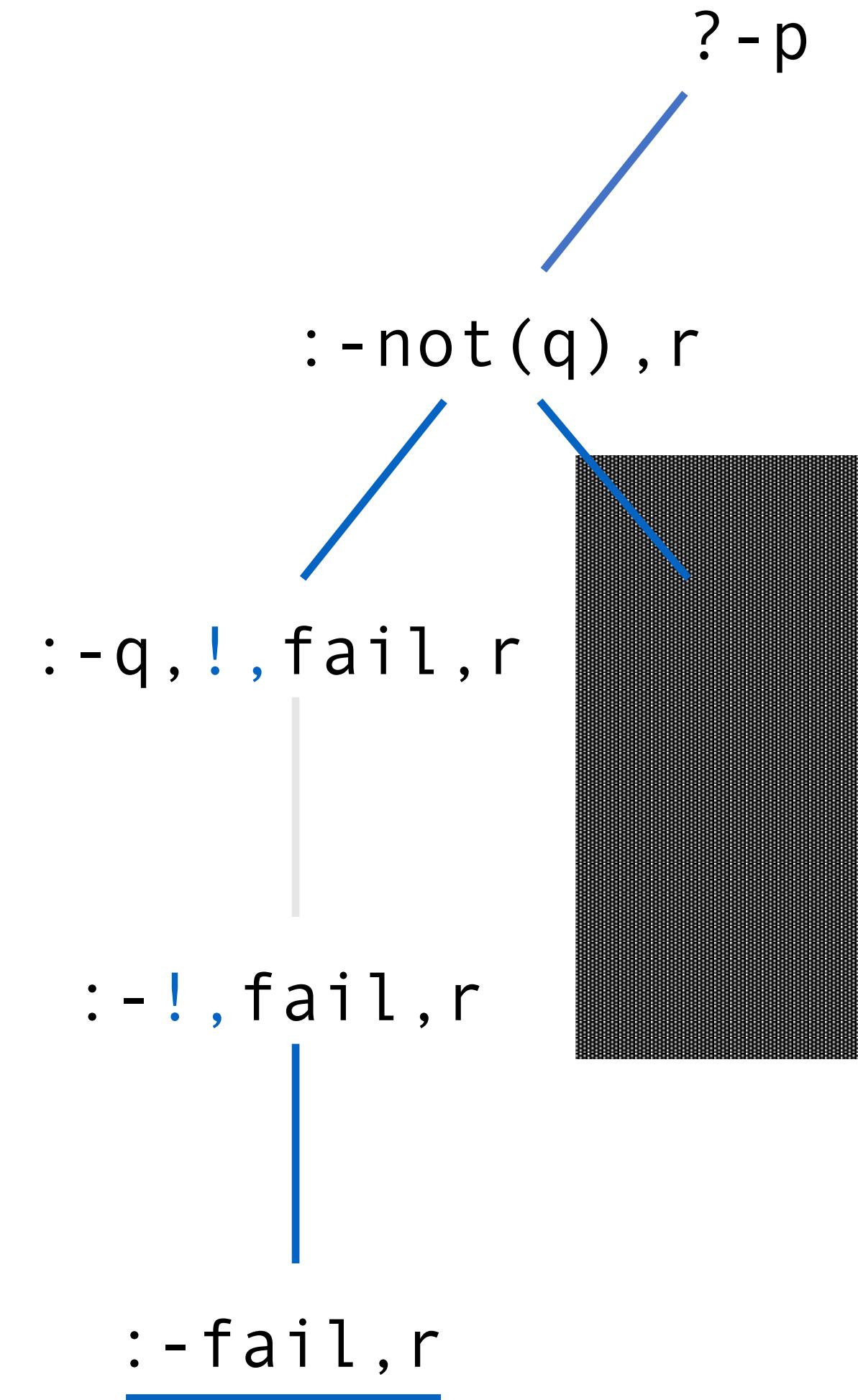
An example: How it works when `: - not (q)` fails

```
p:-not(q),r.  
p:-q.  
q.  
r.  
not(Goal):-Goal,! ,fail.  
not(Goal).
```



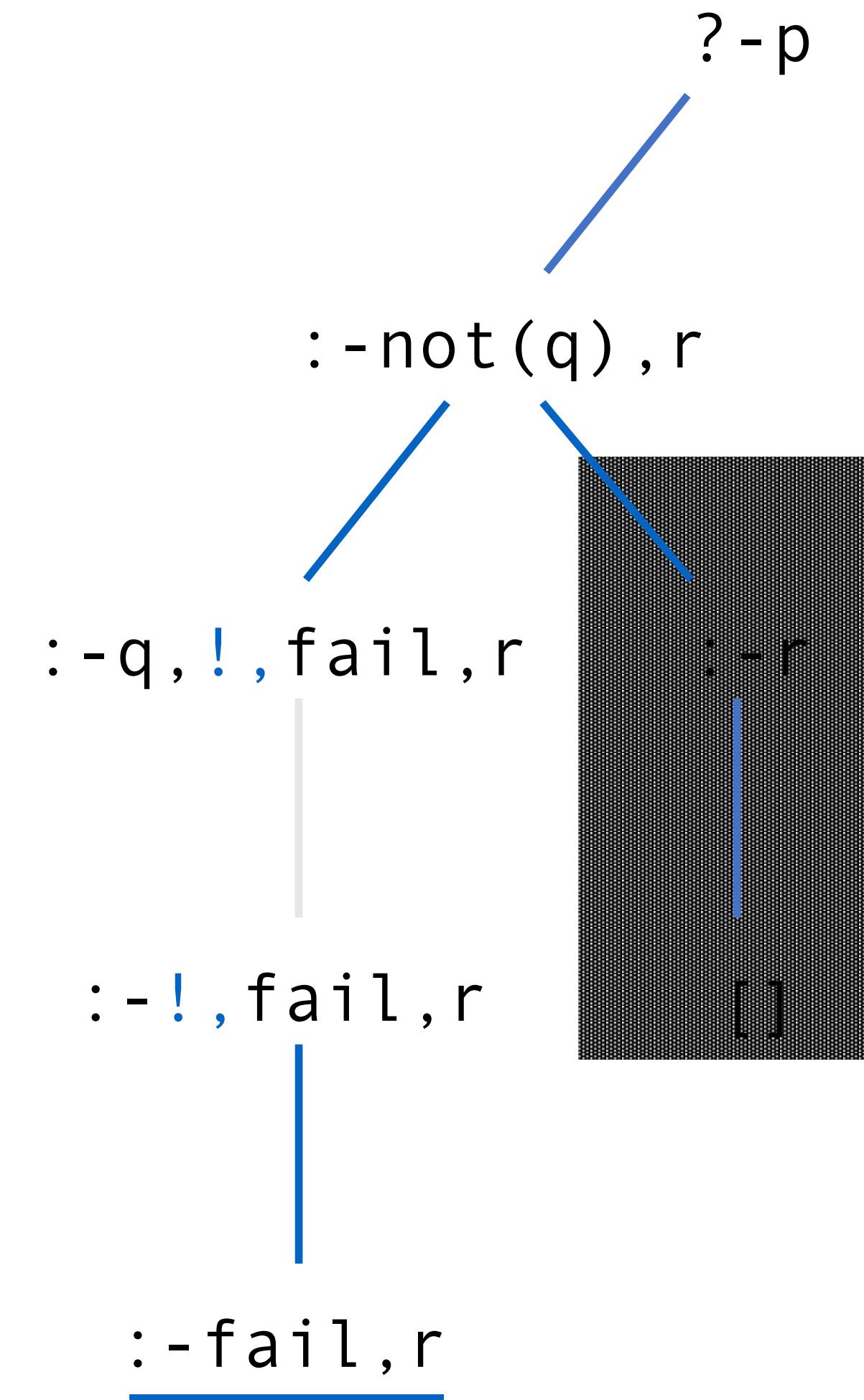
An example: How it works when `: - not (q)` fails

```
p:-not(q),r.  
p:-q.  
q.  
r.  
  
not(Goal):-Goal,! ,fail.  
not(Goal).
```



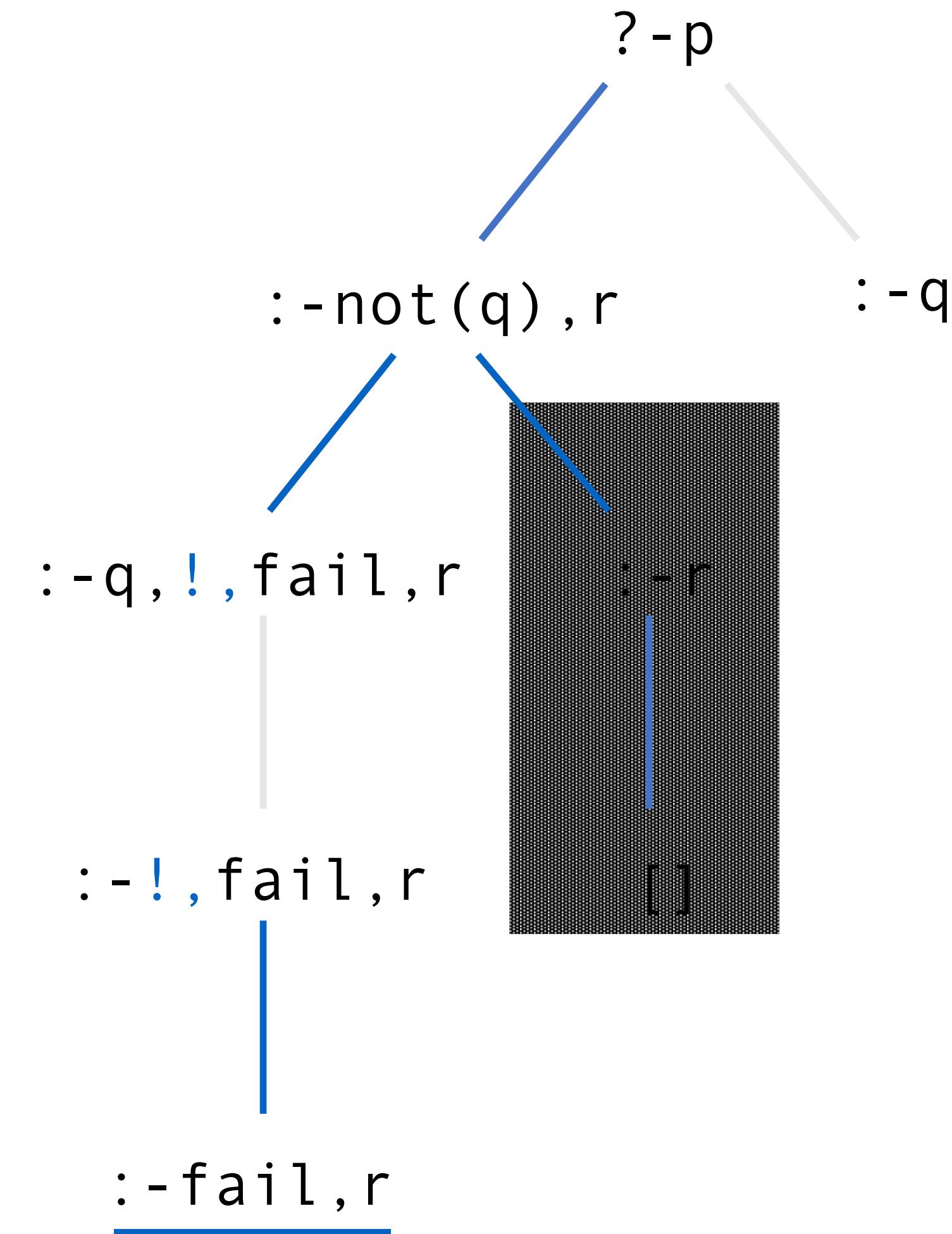
An example: How it works when `: - not (q)` fails

```
p:-not(q),r.  
p:-q.  
q.  
r.  
  
not(Goal):-Goal,! ,fail.  
not(Goal).
```



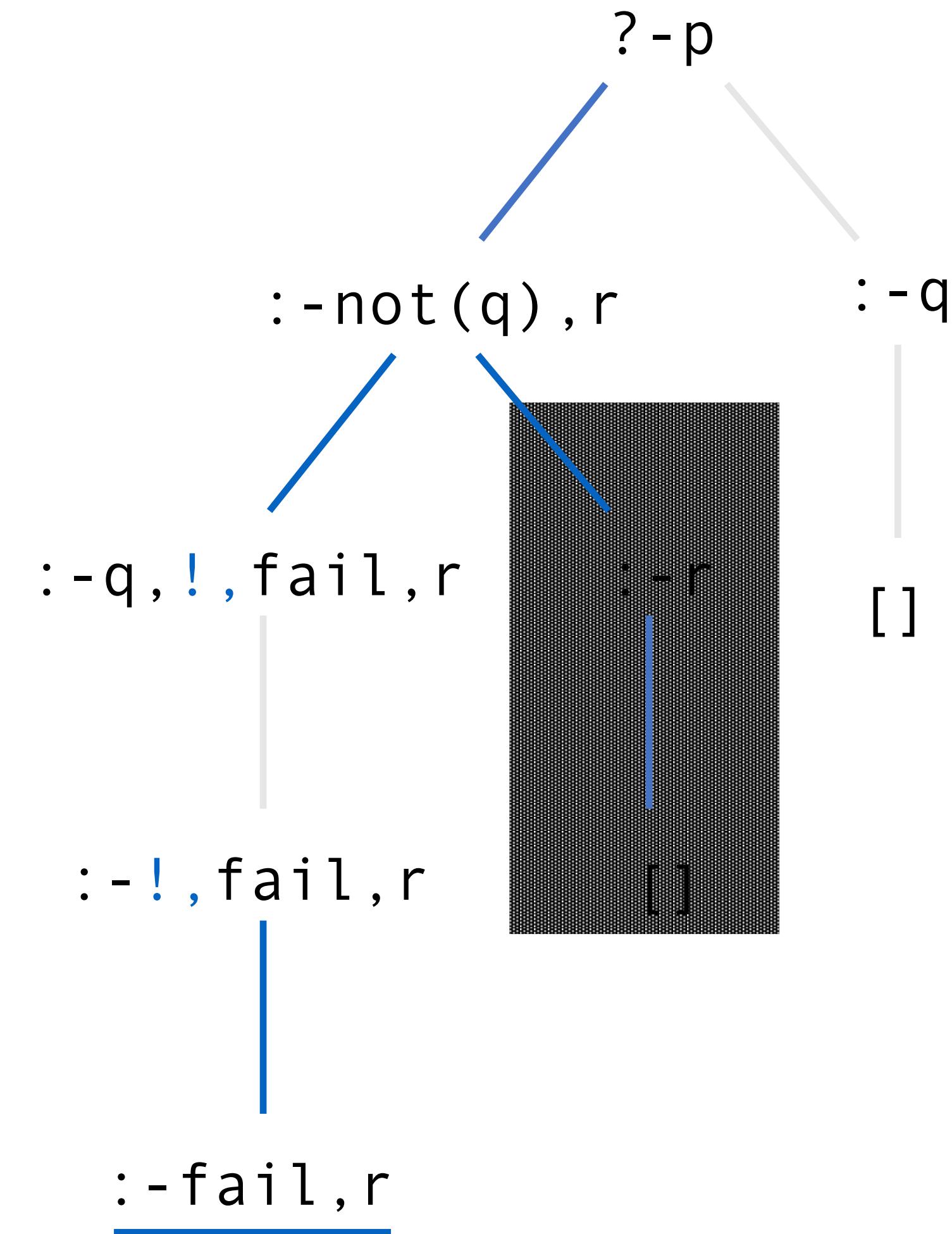
An example: How it works when `: - not (q)` fails

```
p:-not(q),r.  
p:-q.  
q.  
r.  
not(Goal):-Goal,! ,fail.  
not(Goal).
```



An example: How it works when `: - not (q)` fails

```
p:-not(q),r.  
p:-q.  
q.  
r.  
  
not(Goal):-Goal,! ,fail.  
not(Goal).
```



Prolog's `not` is unsound

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

```
?- bachelor(X)
```

Prolog's `not` is unsound

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

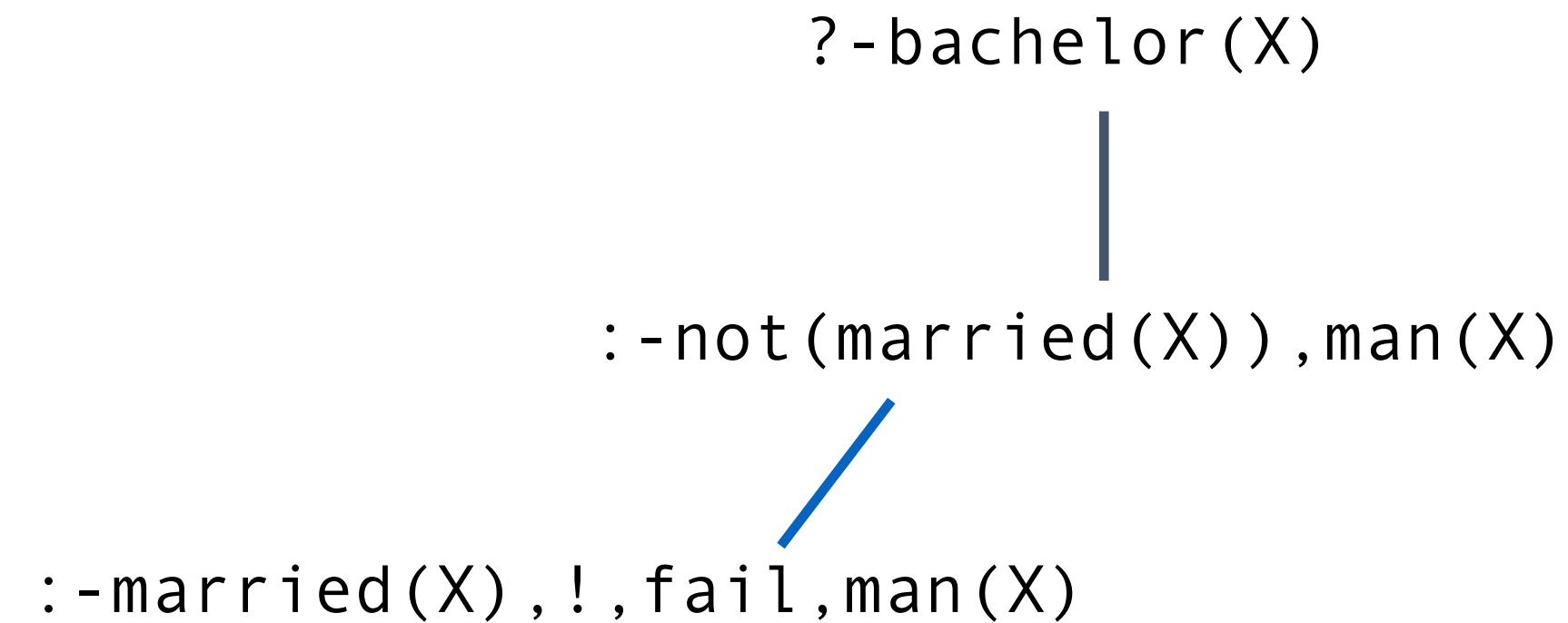
?- `bachelor(X)`



: - `not(married(X)), man(X)`

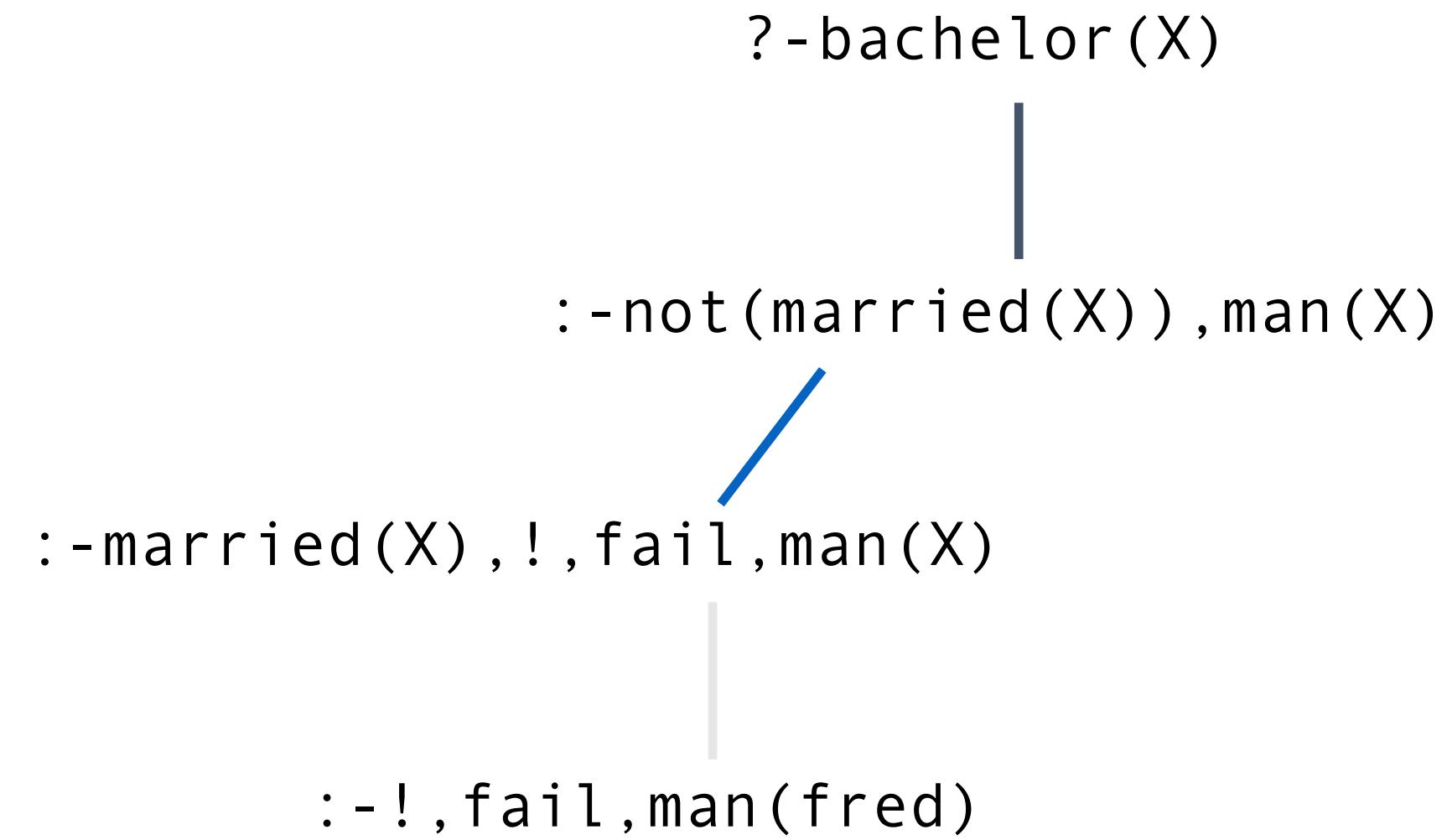
Prolog's `not` is unsound

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



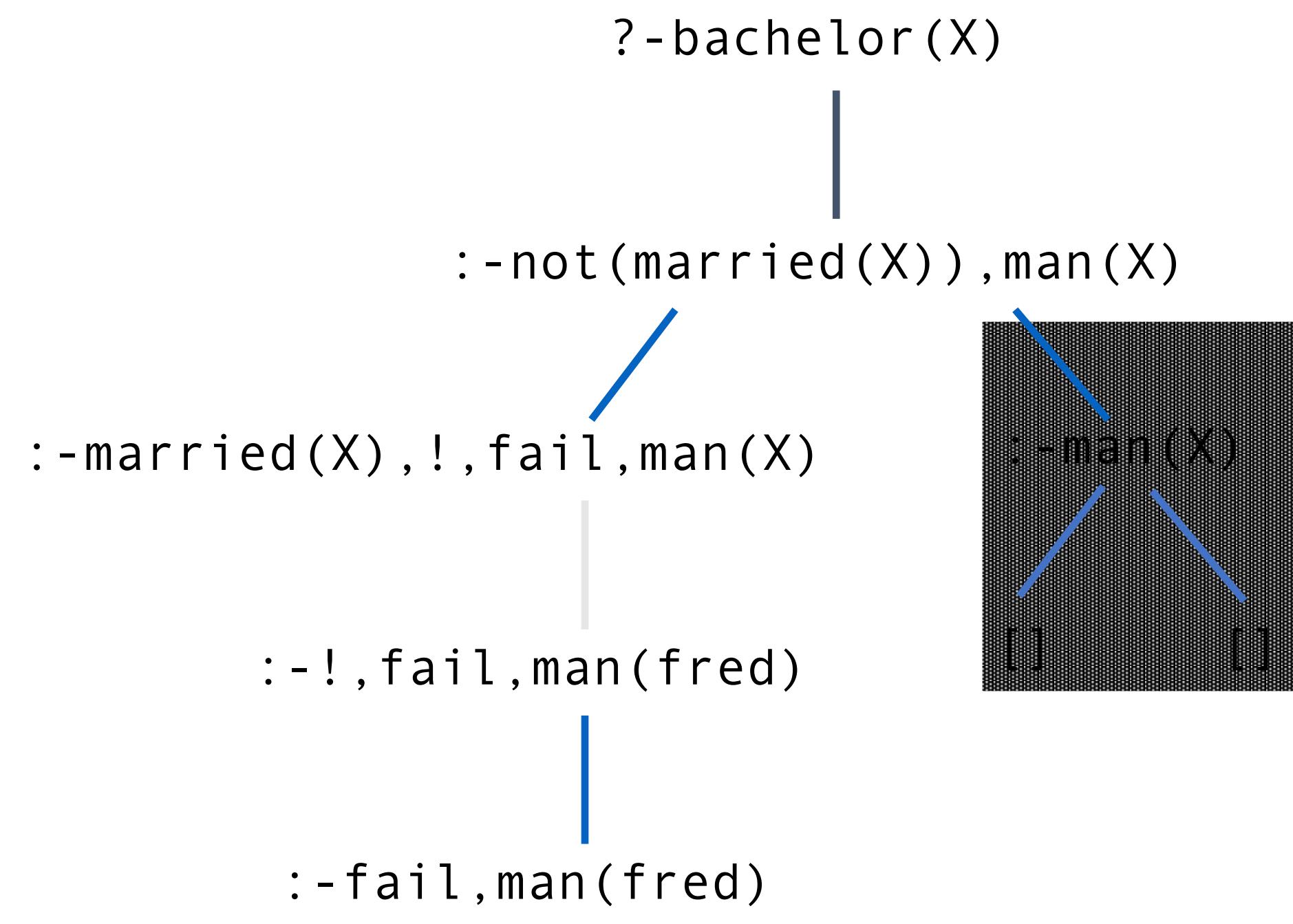
Prolog's `not` is unsound

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



Prolog's not is unsound

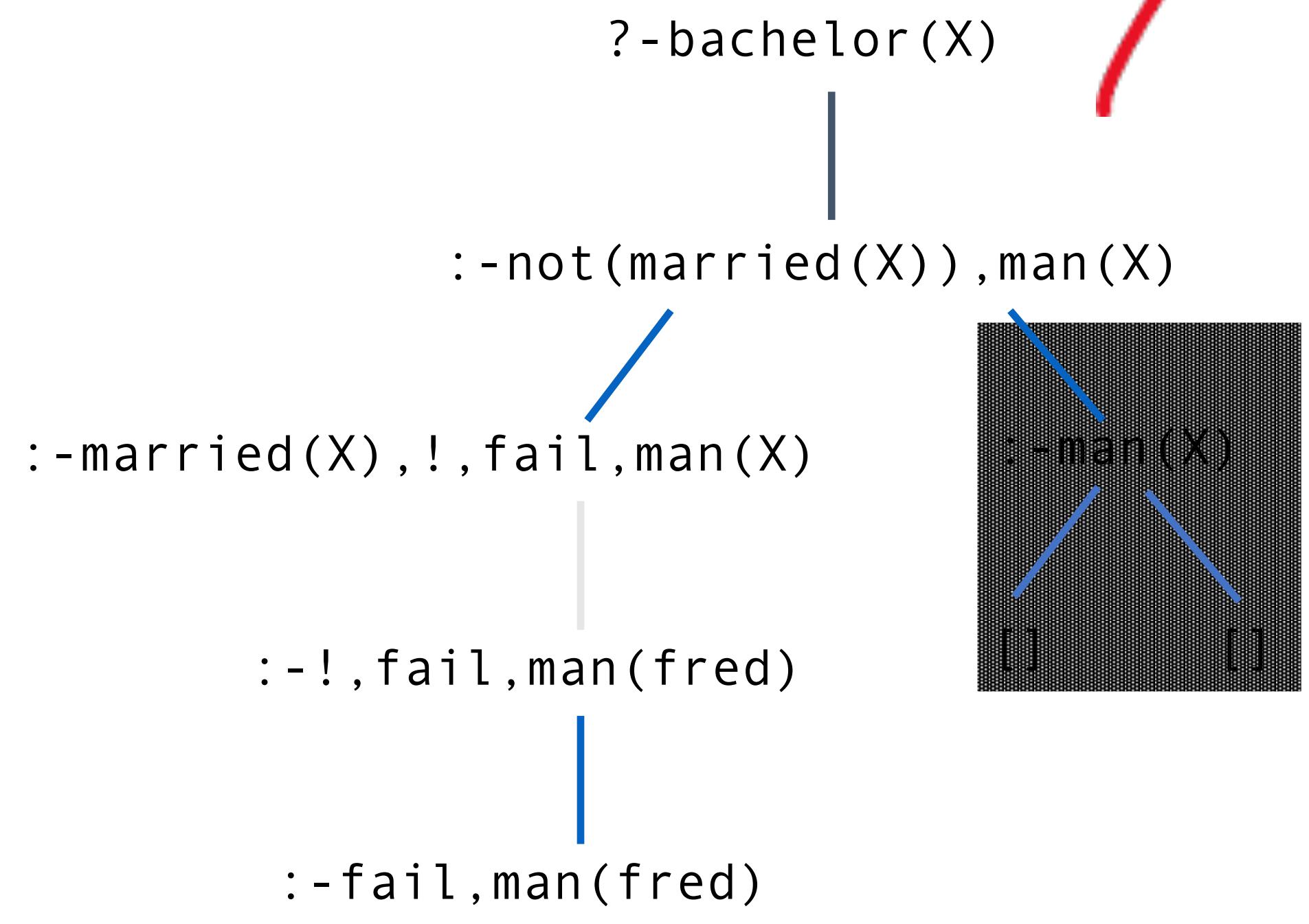
```
bachelor(X) :- not(married(X)), man(X).  
man(fred).  
man(peter).  
married(fred).
```

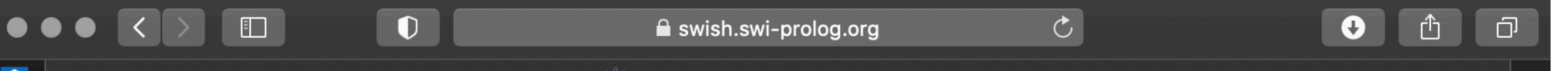


Prolog's `not` is unsound

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

But that's not "correct"!
`peter` is a bachelor and
Prolog did not find the
corresponding answer!





SWISH -- SWI-Prolog for SHaring

File ▾ Edit ▾ Examples ▾ Help ▾

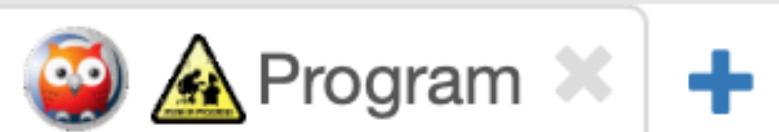
176 users online

Search

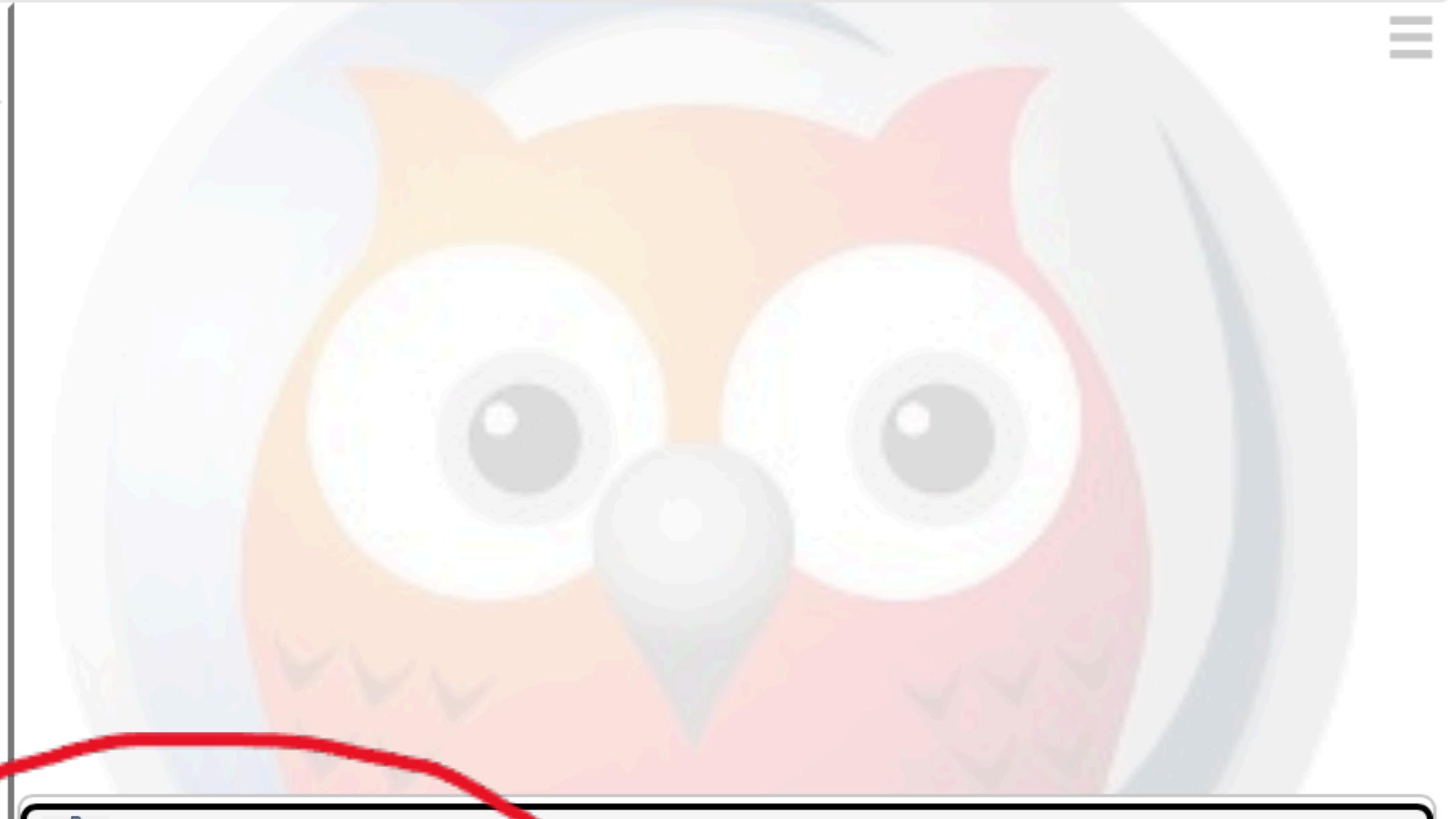
owl icon

Google Analytics icon

Bell icon with 25 notifications



```
1 bachelor(X) :- not(married(X)), man(X).  
2 man(fred).  
3 man(peter).  
4 married(fred).  
5
```



bachelor(X)

false

?- bachelor(X)

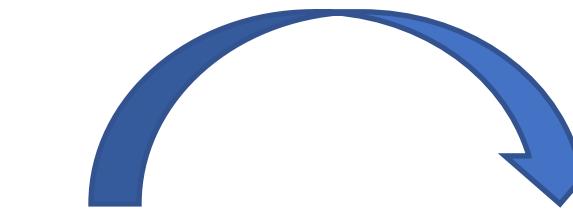
table results Run!

Examples ▾ History ▾ Solutions ▾

Prolog's `not` is unsound – Avoiding the Problem

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

Prolog's `not` is unsound – Avoiding the Problem



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

Prolog's `not` is unsound – Avoiding the Problem

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

This will "ground" X.

Prolog's not is unsound – Avoiding the Problem

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

?- bachelor(X)

This will "ground" X.



Prolog's `not` is unsound – Avoiding the Problem

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

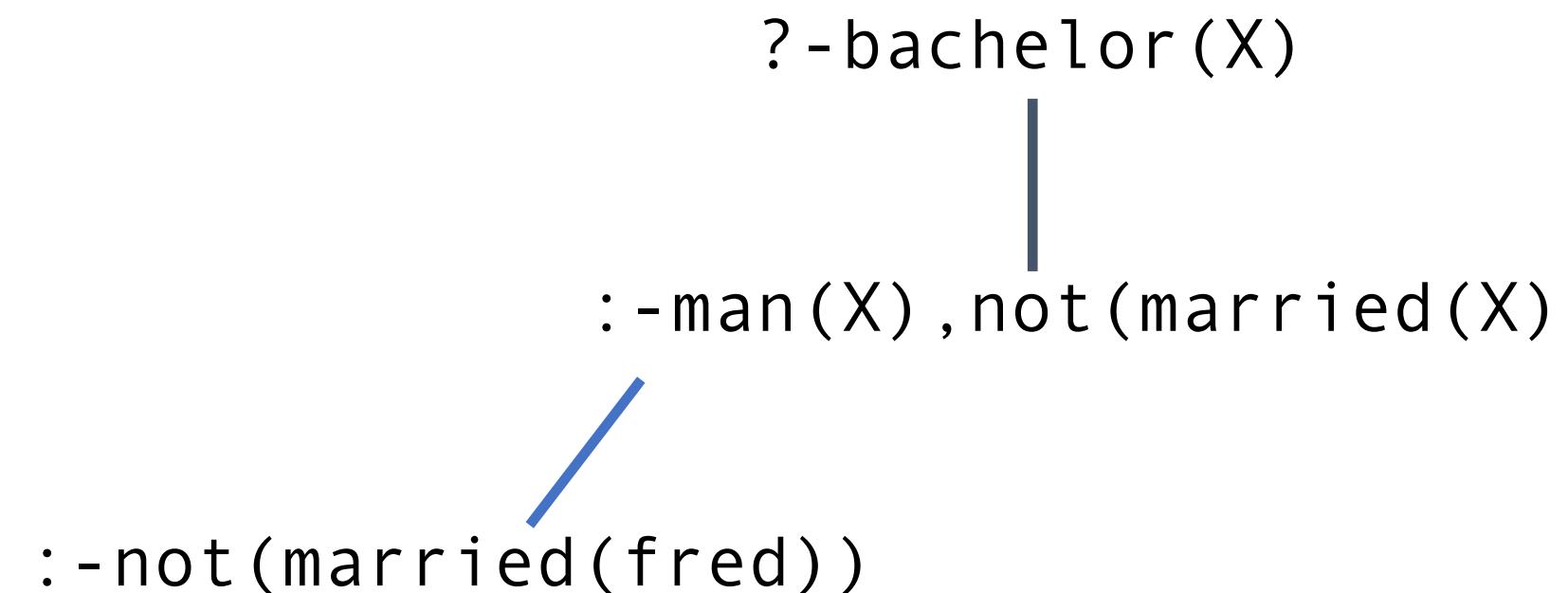
This will "ground" X.

```
?- bachelor(X)  
|  
:- man(X), not(married(X))
```

Prolog's `not` is unsound – Avoiding the Problem

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

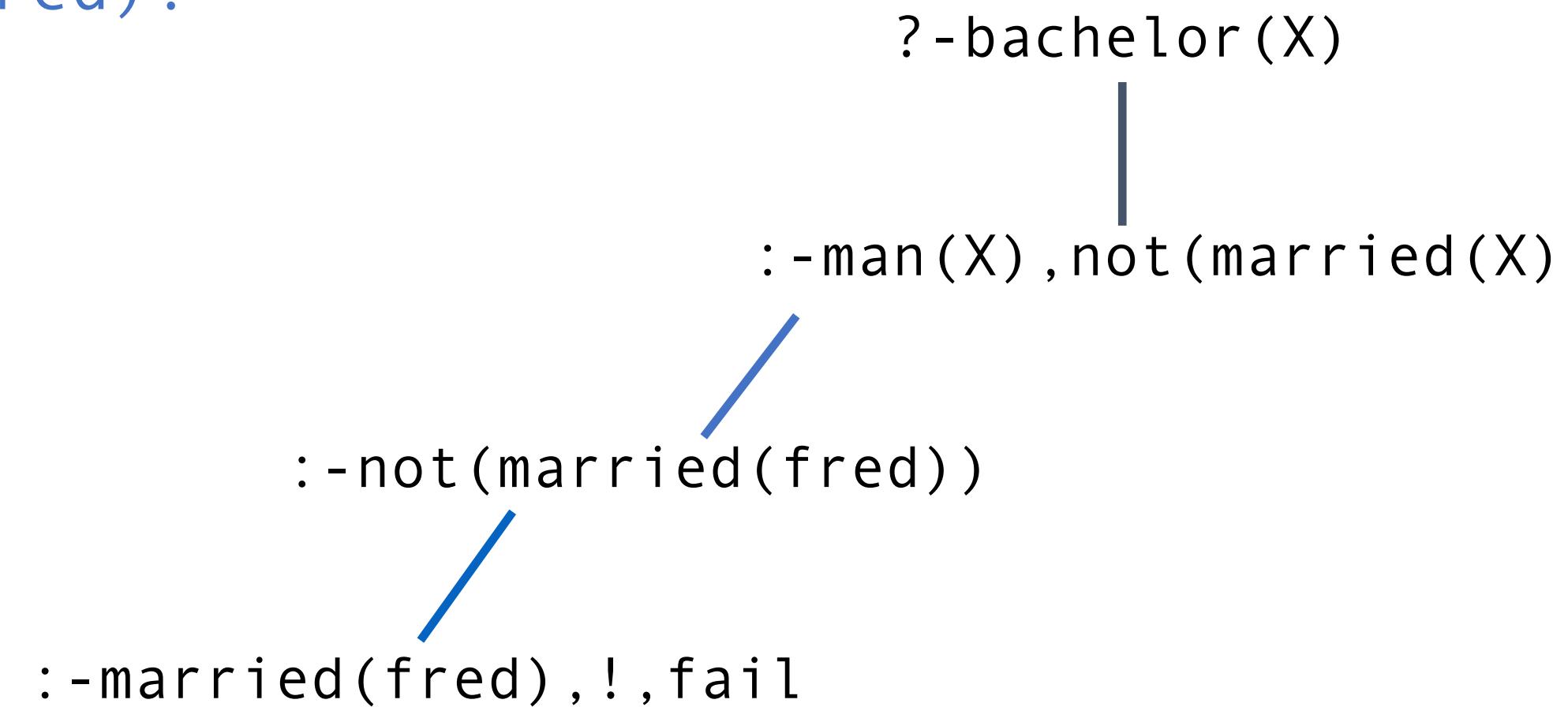
This will "ground" X.



Prolog's `not` is unsound – Avoiding the Problem

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

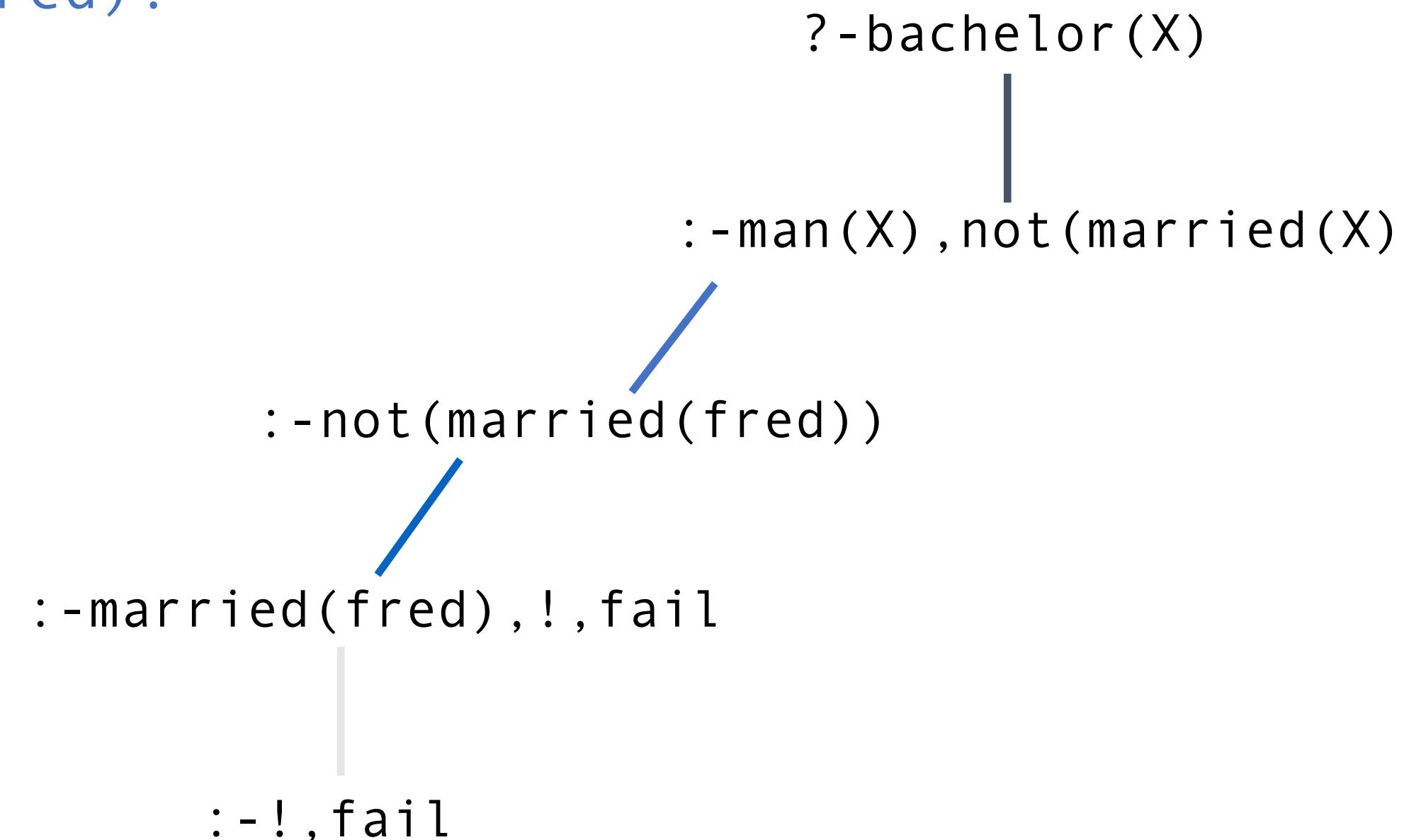
This will "ground" X.



Prolog's `not` is unsound – Avoiding the Problem

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

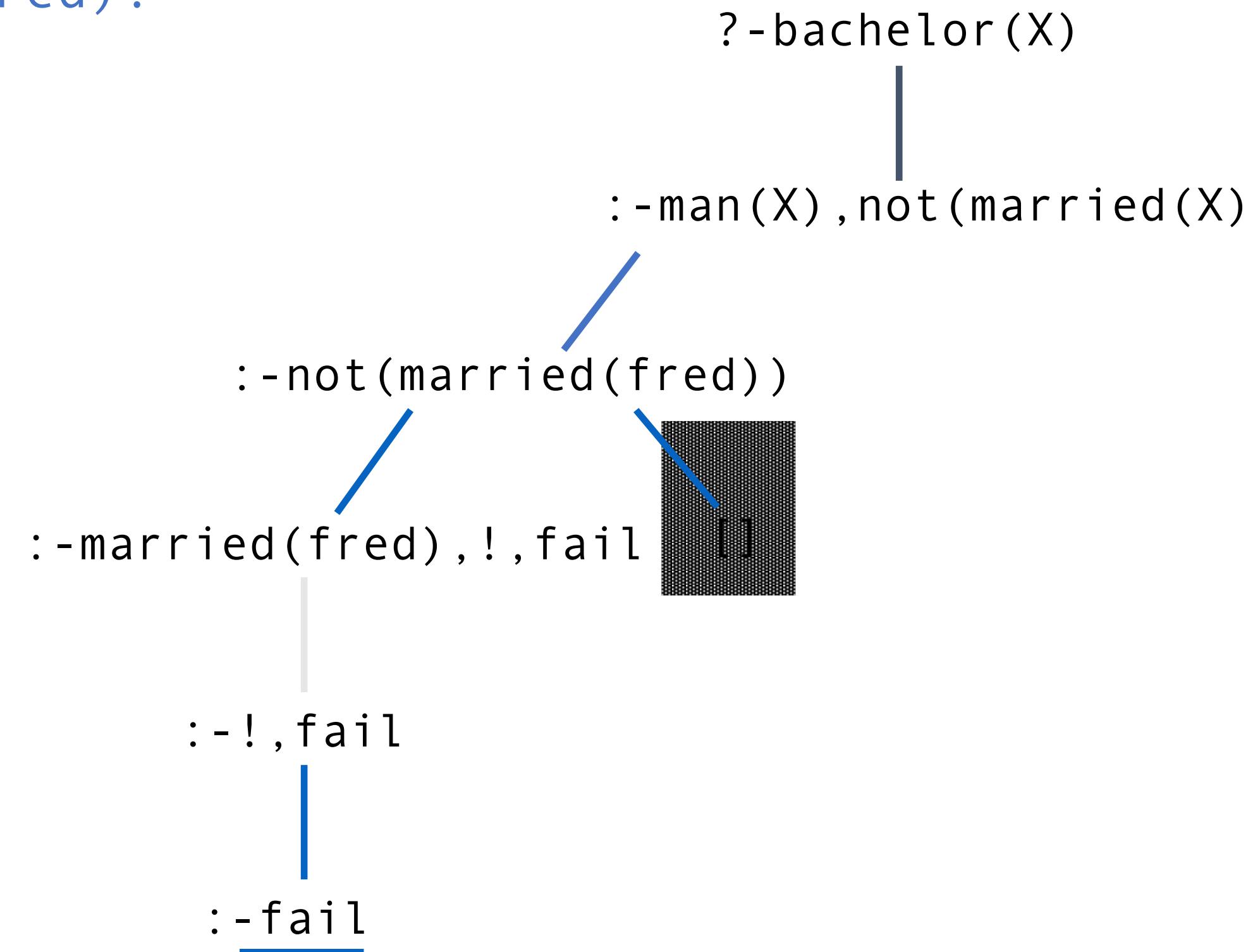
This will "ground" X.



Prolog's `not` is unsound – Avoiding the Problem

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

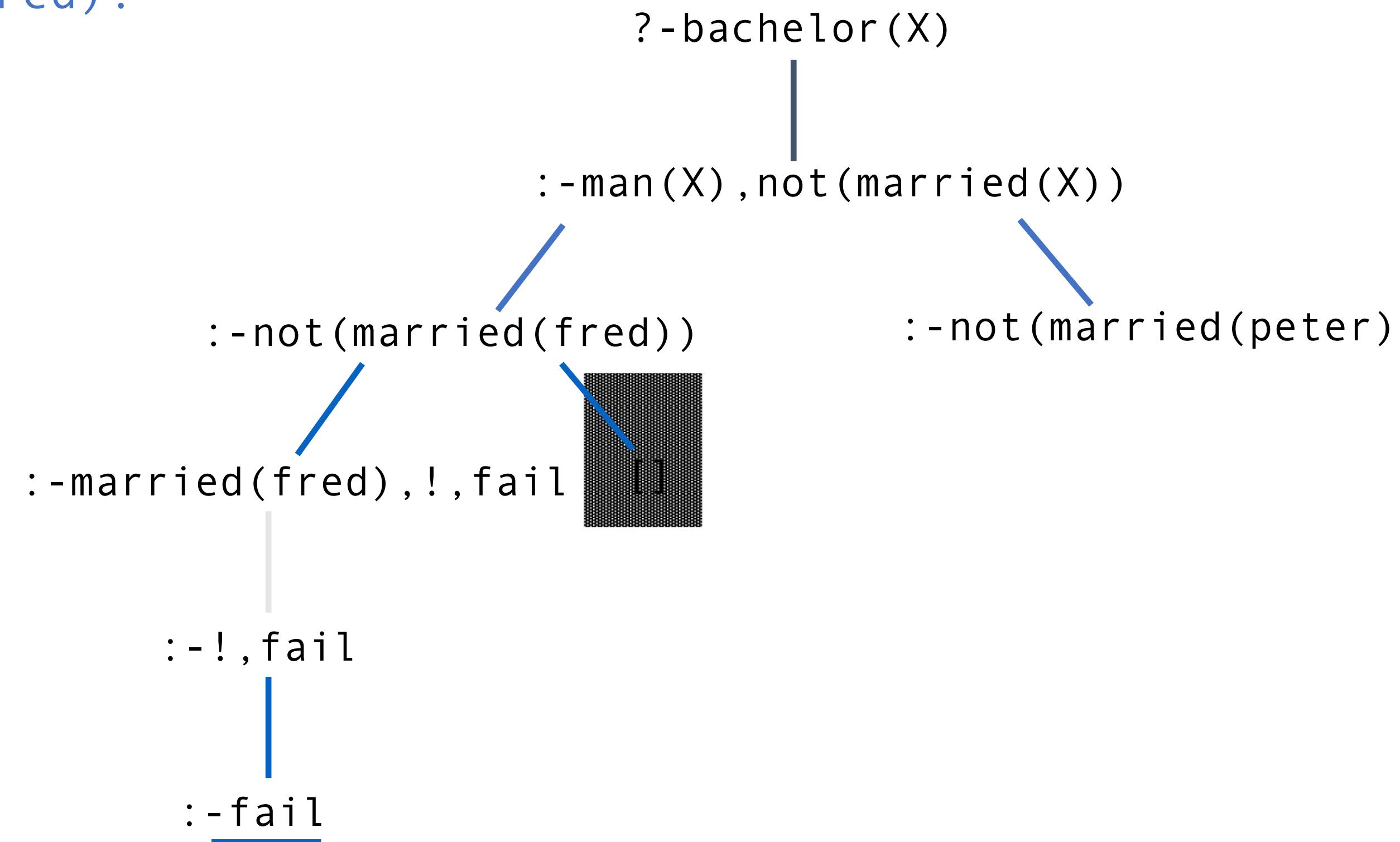
This will "ground" X.



Prolog's `not` is unsound – Avoiding the Problem

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

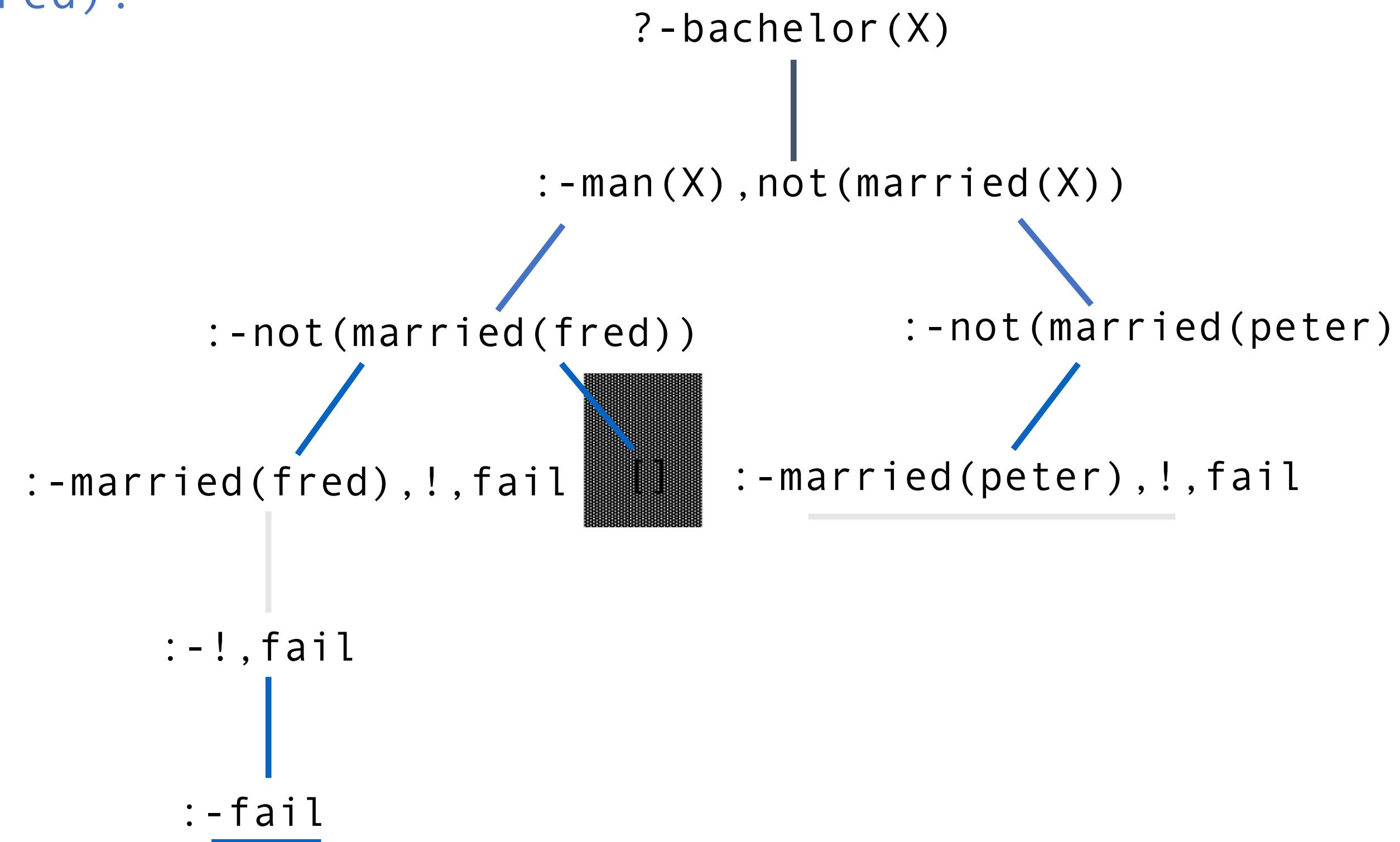
This will "ground" X.



Prolog's `not` is unsound – Avoiding the Problem

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

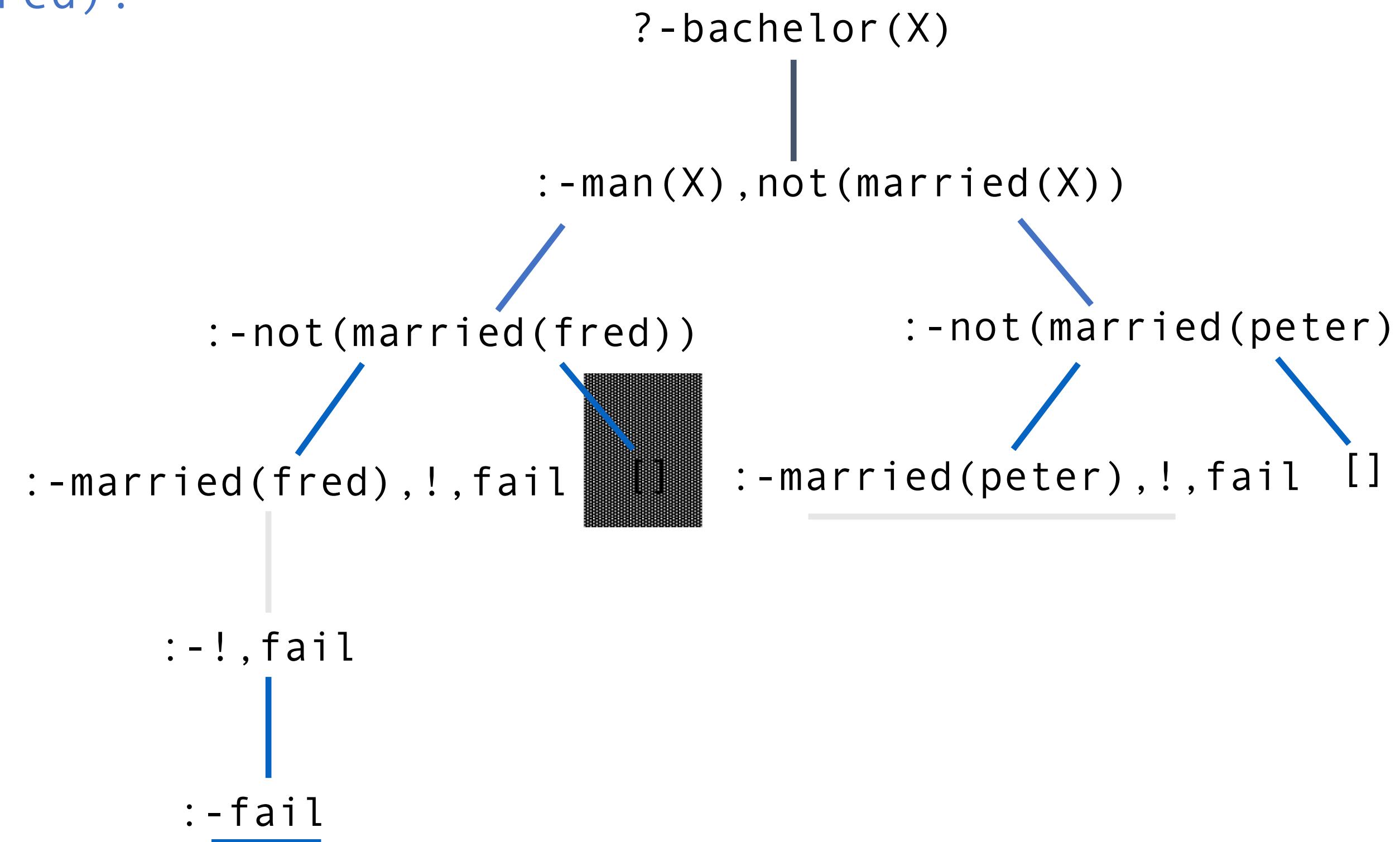
This will "ground" X.



Prolog's `not` is unsound – Avoiding the Problem

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

This will "ground" X.



swish.swi-prolog.org

SWISH -- SWI-Prolog for SHaring

File ▾ Edit ▾ Examples ▾ Help ▾

Search

162 users online

Program +

```
1 bachelor(X) :- man(X), not(married(X)).  
2 man(fred).  
3 man(peter).  
4 married(fred).
```

bachelor(X)

X = peter

?- bachelor(X)

Examples▲ History▲ Solutions▲

table results Run!

A red oval highlights the query window containing the output of the Prolog query.

Arithmetic in Prolog

Prolog arithmetic vs.unification

```
?- X is 5+7-3.  
     X = 9
```

```
?- 9 is 5+7-3.  
     Yes
```

```
?- 9 is X+7-3.  
     Error in arithmetic expression
```

```
?- X is 5*3+7/2.  
     X = 18.5
```

```
?- X = 5+7-3.  
     X = 5+7-3
```

```
?- 9 = 5+7-3.  
     No
```

```
?- 9 = X+7-3.  
     No
```

```
?- X = Y+7-3.  
     X = 947+7-3  
     Y = 947
```

Exercise 3.9

```
zero(A,B,C,X) :- X is (-B + sqrt(B*B - 4*A*C)) / 2*A.
```

```
zero(A,B,C,X) :- X is (-B - sqrt(B*B - 4*A*C)) / 2*A.
```

swish.swi-prolog.org

SWISH -- SWI-Prolog for SHaring

File ▾ Edit ▾ Examples ▾ Help ▾ 256 users online

Search

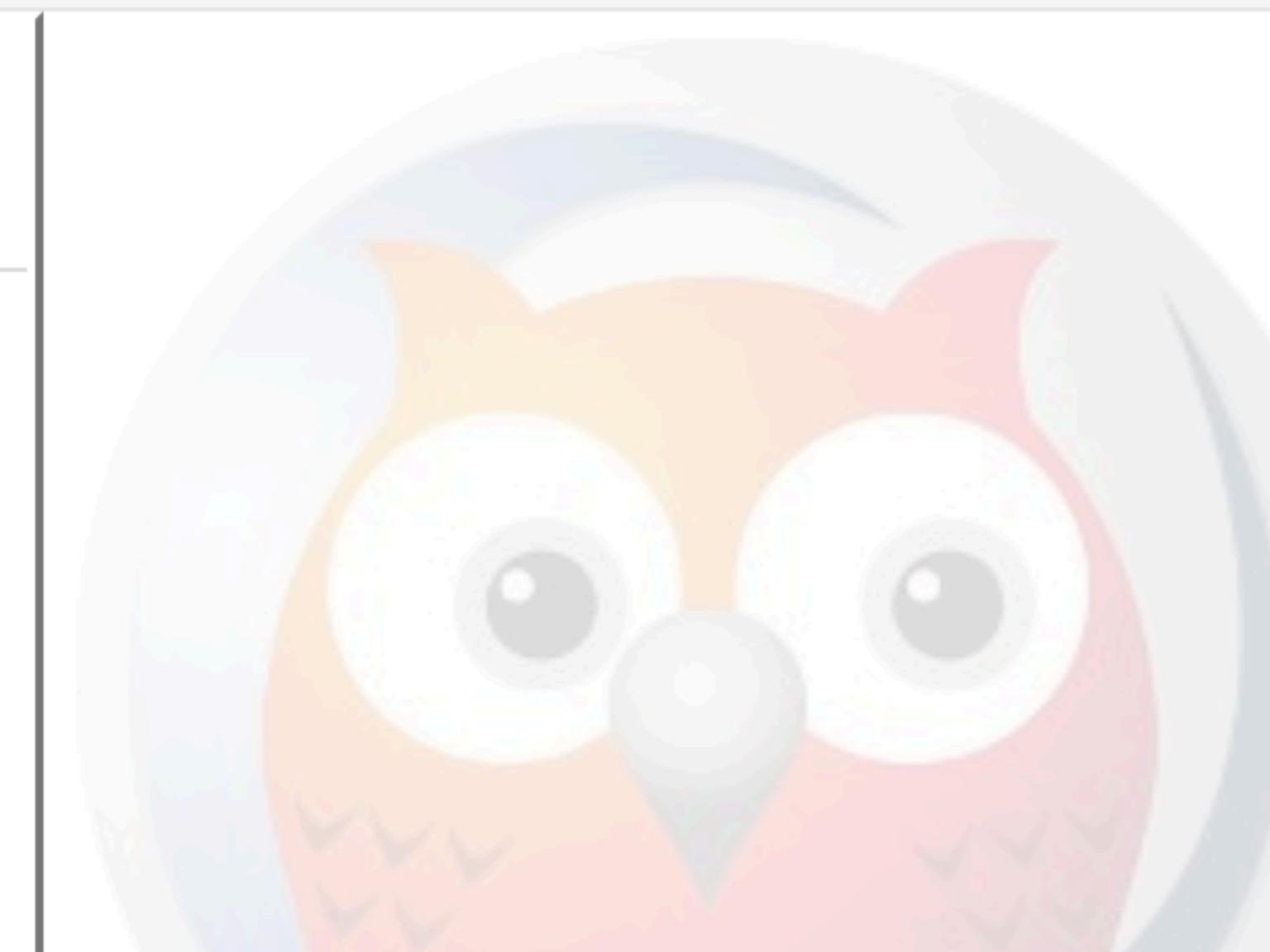
Program X Program X Program X Program X

Program X +

1 `zero(A,B,C,X):-X is (-B + sqrt(B*B - 4*A*C)) / 2*A.`

2 `zero(A,B,C,X):-X is (-B - sqrt(B*B - 4*A*C)) / 2*A.`

3



zero(1,0,-1,X)

X = 1.0
X = -1.0

?- zero(1,0,-1,X)

Examples ▾ History ▾ Solutions ▾ Run!

Search

Agenda-based search

```
% search(Agenda,Goal) :- Goal is a goal node, and a
%                                         descendant of one of the nodes
%                                         on the Agenda
search(Agenda,Goal):-
    next(Agenda,Goal,Rest),
    goal(Goal).
search(Agenda,Goal):-
    next(Agenda,Current,Rest),
    children(Current,Children),
    add(Children,Rest,NewAgenda),
    search(NewAgenda,Goal).
```

Agenda-based search

```
% search(Agenda,Goal) :- Goal is a goal node, and a  
%                               descendant of one of the nodes  
%                               on the Agenda  
search(Agenda,Goal) :-  
    next(Agenda,Goal,Rest), —— Agenda = Goal ∪ Rest  
    goal(Goal).  
search(Agenda,Goal) :-  
    next(Agenda,Current,Rest),  
    children(Current,Children),  
    add(Children,Rest,NewAgenda),  
    search(NewAgenda,Goal).
```

$$\text{Agenda} = \text{Goal} \cup \text{Rest}$$

Agenda-based search

```
% search(Agenda,Goal) :- Goal is a goal node, and a  
% descendant of one of the nodes  
% on the Agenda  
search(Agenda,Goal):-  
  next(Agenda,Goal,Rest), } We have found a solution  
  goal(Goal). if Goal is “next” on the agenda.  
search(Agenda,Goal):-  
  next(Agenda,Current,Rest),  
  children(Current,Children),  
  add(Children,Rest,NewAgenda),  
  search(NewAgenda,Goal).
```

Agenda-based search

```
% search(Agenda,Goal) :- Goal is a goal node, and a
%                               descendant of one of the nodes
%                               on the Agenda
search(Agenda,Goal) :-
    next(Agenda,Goal,Rest), } We have found a solution
    goal(Goal).           if Goal is “next” on the agenda.
search(Agenda,Goal) :-
    next(Agenda,Current,Rest),
    children(Current,Children),
    add(Children,Rest,NewAgenda),
    search(NewAgenda,Goal).
```

Otherwise, get the children
of the **Current** node that is next
on the agenda,

Agenda-based search

```
% search(Agenda,Goal) :- Goal is a goal node, and a
%                               descendant of one of the nodes
%                               on the Agenda
search(Agenda,Goal) :-
    next(Agenda,Goal,Rest), } We have found a solution
    goal(Goal).           if Goal is “next” on the agenda.
search(Agenda,Goal) :-
    next(Agenda,Current,Rest),
    children(Current,Children),
    add(Children,Rest,NewAgenda),
    search(NewAgenda,Goal).
```

Otherwise, get the children of the Current node that is next on the agenda, add the children to the rest of the agenda

Agenda-based search

```
% search(Agenda,Goal) :- Goal is a goal node, and a
%                               descendant of one of the nodes
%                               on the Agenda
search(Agenda,Goal) :-
    next(Agenda,Goal,Rest), } We have found a solution
    goal(Goal).           if Goal is “next” on the agenda.
search(Agenda,Goal) :-
    next(Agenda,Current,Rest),
    children(Current,Children),
    add(Children,Rest,NewAgenda),
    search(NewAgenda,Goal).
```

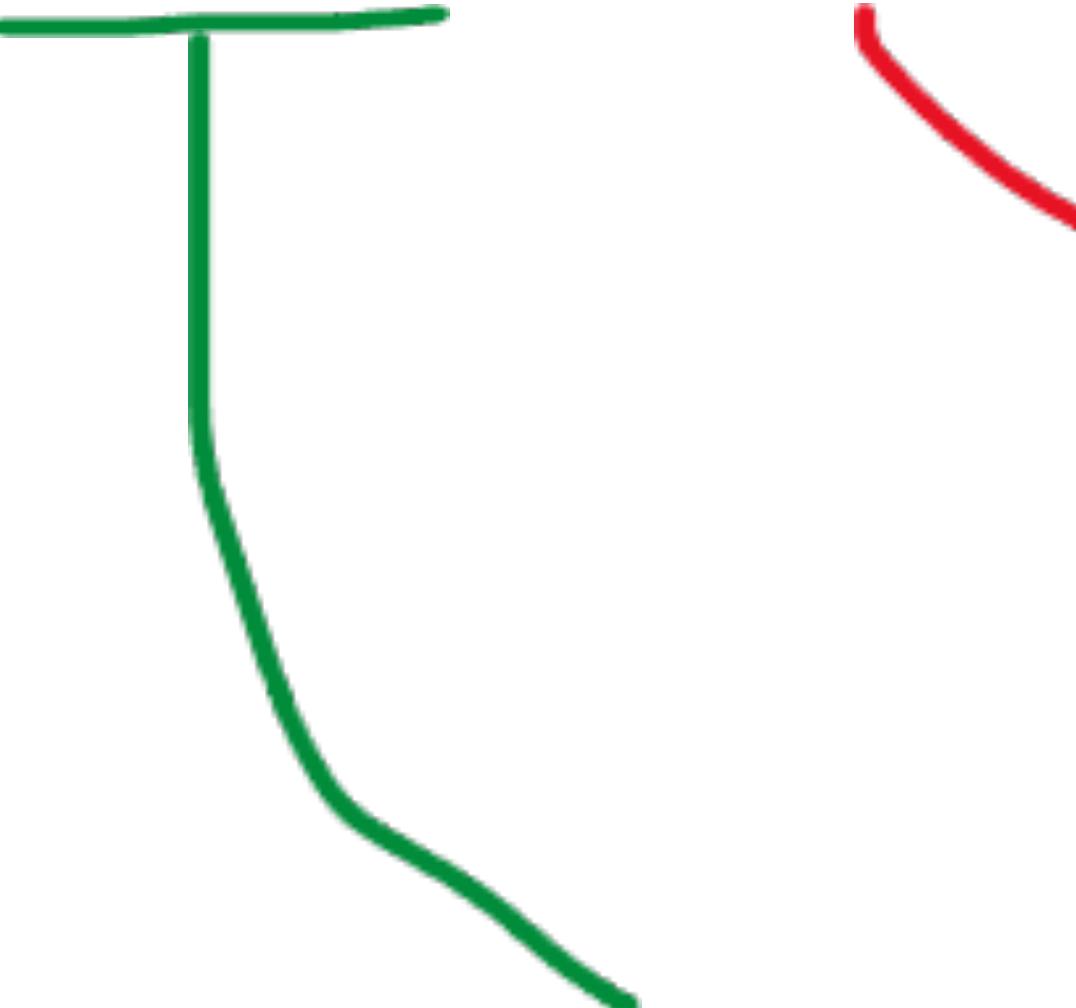
Otherwise, get the children of the Current node that is next on the agenda, add the children to the rest of the agenda and continue searching

Intermission

- Representing search space
- Higher-order predicates `findall`, `bagof`, `setof`

One way to represent children

```
children(Node, Children) :-  
    findall(C, arc(Node, C), Children).
```



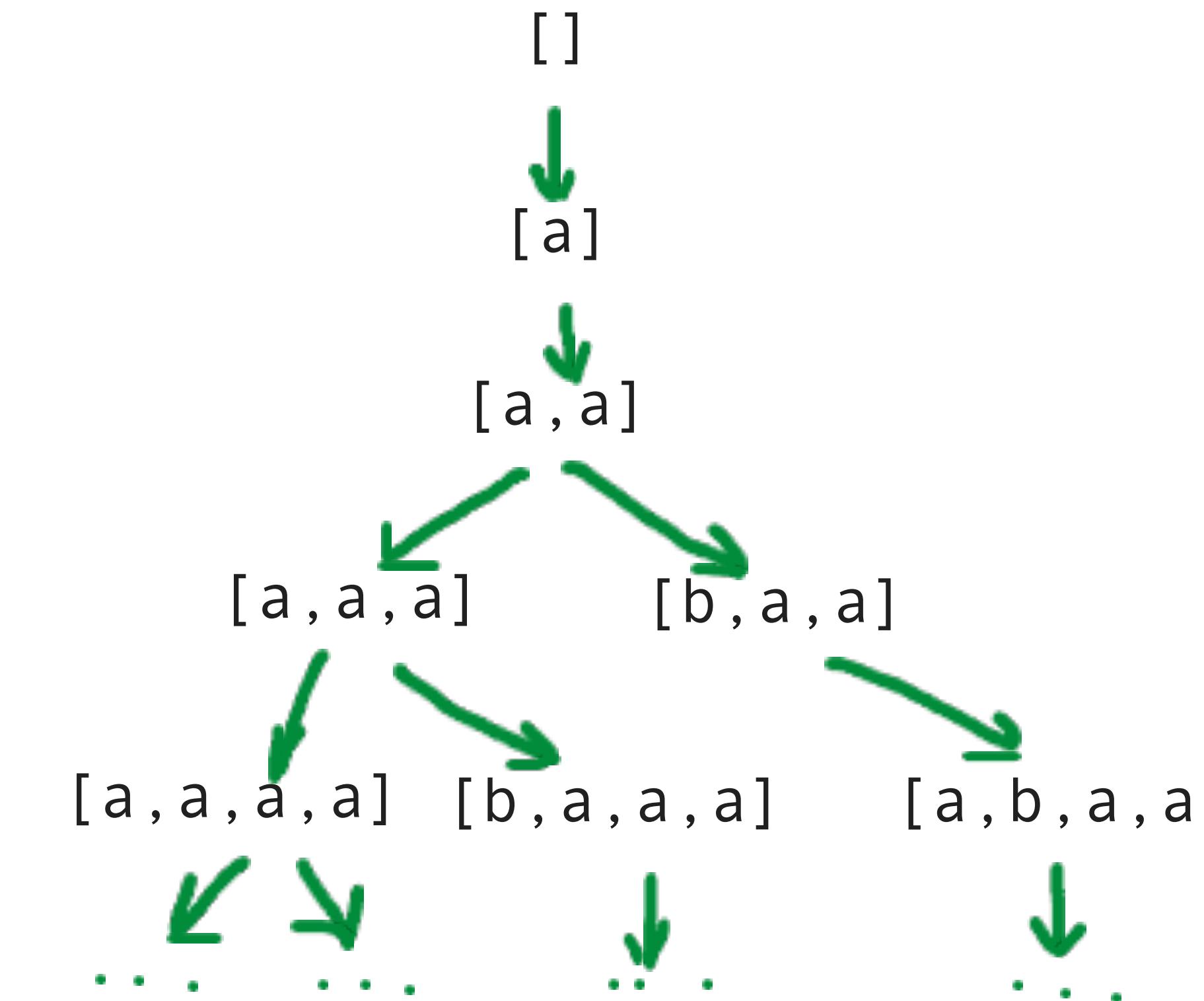
This is how we can represent
the state-space in Prolog.

We will explain this in a moment

An Example – Let's define “arc”

$\text{arc}(\text{L}, [\text{a} | \text{L}]) .$

$\text{arc}([\text{a}, \text{a} | \text{T}], [\text{b}, \text{a}, \text{a} | \text{T}]) .$



One way to represent children

```
children(Node,Children) :-  
    findall(C,arc(Node,C),Children).
```



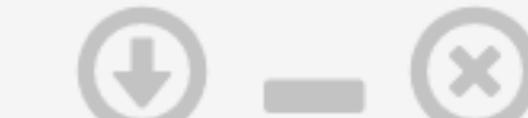
This is how we can represent
the state-space in Prolog.

findall: findall(Template, Goal, Bag)
E.g.:

```
?- findall(X, (arc([a,a],X)), Bag).
```

```
arc(L, [a|L]).  
arc([a,a|T], [b,a,a|T]).
```

 **findall(X, (arc([a,a],X)), Bag).**



Bag = [[a, a, a], [b, a, a]]

?- **findall(X, (arc([a,a],X)), Bag).**

Examples▲ History▲ Solutions▲

table results **Run!**

Back to Search

Depth-first vs. breadth-first search

```
search_df([Goal|Rest],Goal) :-  
    goal(Goal).  
search_df([Current|Rest],Goal) :-  
    children(Current,Children),  
    append(Children,Rest,NewAgenda),  
    search_df(NewAgenda,Goal).
```

```
search_bf([Goal|Rest],Goal) :-  
    goal(Goal).  
search_bf([Current|Rest],Goal) :-  
    children(Current,Children),  
    append(Rest,Children,NewAgenda),  
    search_bf(NewAgenda,Goal).
```

```
children(Node,Children) :-  
    findall(C,arc(Node,C),Children).
```

WE UNDERSTAND
THIS

Depth-first vs. breadth-first search

```

search_df([Goal|Rest],Goal) :-  

    goal(Goal).  

search_df([Current|Rest],Goal) :-  

    children(Current,Children),  

    append(Children,Rest,NewAgenda),  

    search_df(NewAgenda,Goal).  
  

search_bf([Goal|Rest],Goal) :-  

    goal(Goal).  

search_bf([Current|Rest],Goal) :-  

    children(Current,Children),  

    append(Rest,Children,NewAgenda),  

    search_bf(NewAgenda,Goal).  
  

children(Node,Children) :-  

    findall(C,arc(Node,C),Children).

```

**The predicate
next is implicitly
represented using
unification.
We could also
define it as:**

next([H|T],H,T).

Depth-first vs. breadth-first search

```
search_df([Goal|Rest],Goal) :-  
    goal(Goal).  
search_df([Current|Rest],Goal) :-  
    children(Current,Children),  
    append(Children,Rest,NewAgenda),  
    search_dt(NewAgenda,Goal).
```

```
search_bf([Goal|Rest],Goal) :-  
    goal(Goal).  
search_bf([Current|Rest],Goal) :-  
    children(Current,Children),  
    append(Rest,Children,NewAgenda),  
    search_bf(NewAgenda,Goal).
```

```
children(Node,Children) :-  
    findall(C,arc(Node,C),Children).
```

This is where
they differ.

Depth-first vs. breadth-first search

- Breadth-first search
- Depth-first search

Depth-first vs. breadth-first search

- Breadth-first search
 - agenda = queue (first-in first-out)
- Depth-first search
 - agenda = stack (last-in first-out)

Depth-first vs. breadth-first search

- Breadth-first search
 - agenda = queue (first-in first-out)
 - complete: guaranteed to find all solutions
- Depth-first search
 - agenda = stack (last-in first-out)
 - incomplete: may get trapped in infinite branch

Depth-first vs. breadth-first search

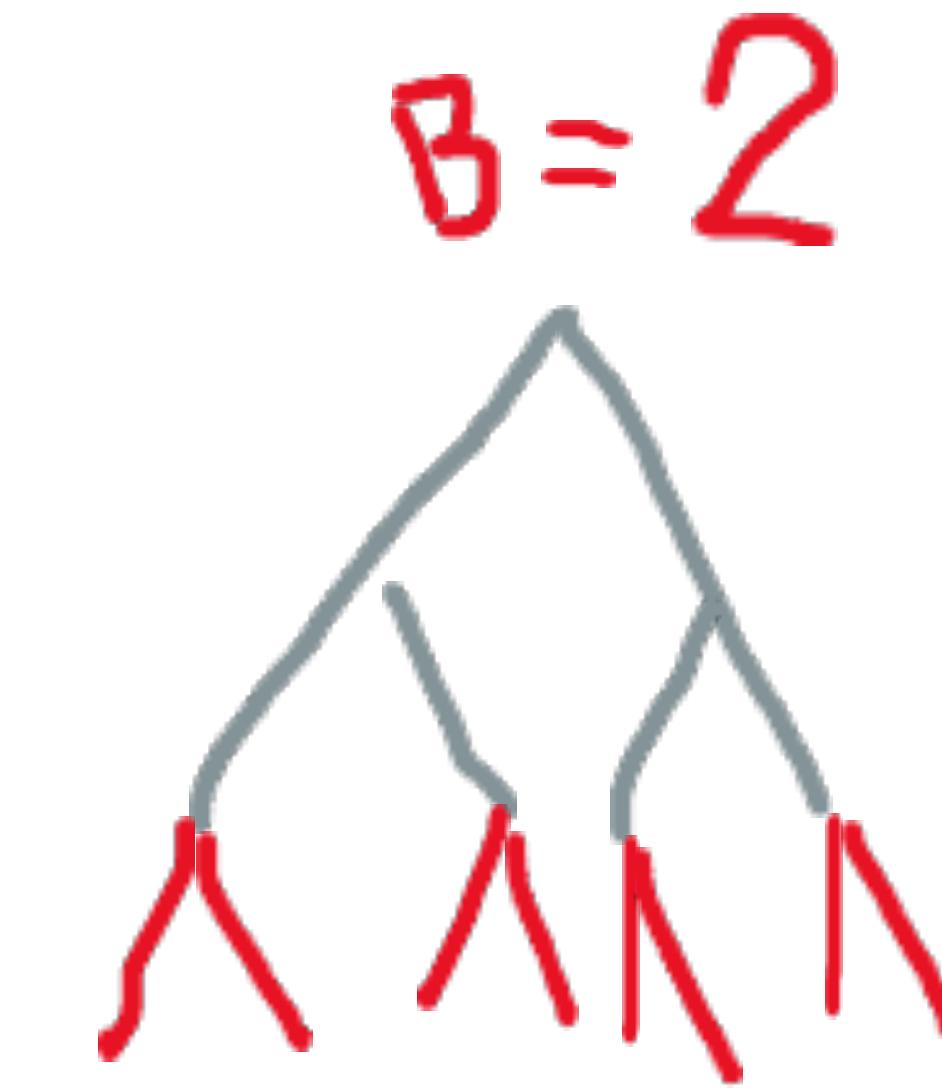
- Breadth-first search
 - agenda = queue (first-in first-out)
 - complete: guaranteed to find all solutions
 - first solution founds along shortest path
- Depth-first search
 - agenda = stack (last-in first-out)
 - incomplete: may get trapped in infinite branch
 - no shortest-path property

Depth-first vs. breadth-first search

- Breadth-first search
 - agenda = queue (first-in first-out)
 - complete: guaranteed to find all solutions
 - first solution founds along shortest path
 - requires $O(B^n)$ memory
- Depth-first search
 - agenda = stack (last-in first-out)
 - incomplete: may get trapped in infinite branch
 - no shortest-path property
 - requires $O(B \times n)$ memory

Depth-first vs. breadth-first search

- Breadth-first search
 - agenda = queue (first-in first-out)
 - complete: guaranteed to find all solutions
 - first solution founds along shortest path
 - requires $O(B^n)$ memory
- Depth-first search
 - agenda = stack (last-in first-out)
 - incomplete: may get trapped in infinite branch
 - no shortest-path property
 - requires $O(B \times n)$ memory



Loop detection

```
% depth-first search with loop detection
search_df_loop([Goal|Rest],Visited,Goal) :-
    goal(Goal).
search_df_loop([Current|Rest],Visited,Goal) :-
    children(Current,Children),
    add_df(Children,Rest,Visited,NewAgenda),
    search_df_loop(NewAgenda, [Current|Visited], Goal).

add_df([],Agenda,Visited,Agenda).
add_df([Child|Rest],OldAgenda,Visited,[Child|NewAgenda]) :-
    not element(Child,OldAgenda),
    not element(Child,Visited),
    add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda) :-
    element(Child,OldAgenda),
    add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda) :-
    element(Child,Visited),
    add_df(Rest,OldAgenda,Visited,NewAgenda).
```

Loop detection

```
% depth-first search with loop detection
search_df_loop([Goal|Rest],Visited,Goal) :-  
    goal(Goal).  
search_df_loop([Current|Rest],Visited,Goal) :-  
    children(Current,Children),  
    add_df(Children,Rest,Visited,NewAgenda),  
    search_df_loop(NewAgenda,[Current|Visited],Goal).  
  
add_df([],Agenda,Visited,Agenda). 1 Add empty list of children  
add_df([Child|Rest],OldAgenda,Visited,[Child|NewAgenda]) :-  
    not element(Child,OldAgenda),  
    not element(Child,Visited),  
    add_df(Rest,OldAgenda,Visited,NewAgenda).  
add_df([Child|Rest],OldAgenda,Visited,NewAgenda) :-  
    element(Child,OldAgenda),  
    add_df(Rest,OldAgenda,Visited,NewAgenda).  
add_df([Child|Rest],OldAgenda,Visited,NewAgenda) :-  
    element(Child,Visited),  
    add_df(Rest,OldAgenda,Visited,NewAgenda).
```

Loop detection

```
% depth-first search with loop detection
search_df_loop([Goal|Rest],Visited,Goal) :-  

    goal(Goal).
search_df_loop([Current|Rest],Visited,Goal) :-  

    children(Current,Children),  

    add_df(Children,Rest,Visited,NewAgenda),  

    search_df_loop(NewAgenda,[Current|Visited],Goal).

add_df([],Agenda,Visited,Agenda). 1 Add empty list of children
add_df([Child|Rest],OldAgenda,Visited,[Child|NewAgenda]) :-  

    not element(Child,OldAgenda), 2 Add nodes that have not been  

    not element(Child,Visited), visited and not on the agenda  

    add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda) :-  

    element(Child,OldAgenda),
    add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda) :-  

    element(Child,Visited),
    add_df(Rest,OldAgenda,Visited,NewAgenda).
```

Loop detection

```
% depth-first search with loop detection
search_df_loop([Goal|Rest],Visited,Goal) :-  

    goal(Goal).
search_df_loop([Current|Rest],Visited,Goal) :-  

    children(Current,Children),  

    add_df(Children,Rest,Visited,NewAgenda),  

    search_df_loop(NewAgenda,[Current|Visited],Goal).

add_df([],Agenda,Visited,Agenda). 1 Add empty list of children  

add_df([Child|Rest],OldAgenda,Visited,[Child|NewAgenda]) :-  

    not element(Child,OldAgenda), 2 Add nodes that have not been  

    not element(Child,Visited), 2. visited and not on the agenda  

    add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda) :-  

    element(Child,OldAgenda), 3. If already on agenda, ignore.  

    add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda) :-  

    element(Child,Visited),
    add_df(Rest,OldAgenda,Visited,NewAgenda).
```

Loop detection

```
% depth-first search with loop detection
search_df_loop([Goal|Rest],Visited,Goal) :-  

    goal(Goal).
search_df_loop([Current|Rest],Visited,Goal) :-  

    children(Current,Children),  

    add_df(Children,Rest,Visited,NewAgenda),  

    search_df_loop(NewAgenda,[Current|Visited],Goal).
```

```
add_df([],Agenda,Visited,Agenda). 1. Add empty list of children  

add_df([Child|Rest],OldAgenda,Visited,[Child|NewAgenda]) :-  

    not element(Child,OldAgenda), 2. Add nodes that have not been  

    not element(Child,Visited), 2. visited and not on the agenda  

    add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda) :-  

    element(Child,OldAgenda), 3. If already on agenda, ignore.  

    add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda) :-  

    element(Child,Visited), 4. If already visited, ignore.  

    add_df(Rest,OldAgenda,Visited,NewAgenda).
```

Backtracking search

```
% depth-first search by means of backtracking
search_bt(Goal,Goal):-
    goal(Goal).
search_bt(Current,Goal):-
    arc(Current,Child),
    search_bt(Child,Goal).
```

Backtracking search

```
% depth-first search by means of backtracking
search_bt(Goal,Goal):-
    goal(Goal).
search_bt(Current,Goal):-
    arc(Current,Child), _____ We used it to define children
    search_bt(Child,Goal).
```

Backtracking search

```
% depth-first search by means of backtracking
search_bt(Goal,Goal):-
    goal(Goal).
search_bt(Current,Goal):-
    arc(Current,Child),
    search_bt(Child,Goal).
```

```
% backtracking depth-first search with depth bound
search_d(D,Goal,Goal):-
    goal(Goal).
search_d(D,Current,Goal):-
    D>0, D1 is D-1,
    arc(Current,Child),
    search_d(D1,Child,Goal).
```

Iterative deepening

```
search_id(First,Goal):-
    search_id(1,First,Goal).      % start with depth 1

search_id(D,Current,Goal):-
    search_d(D,Current,Goal).

search_id(D,Current,Goal):-
    D1 is D+1,                  % increase depth
    search_id(D1,Current,Goal).
```

- combines advantages of breadth-first search (complete, shortest path) with those of depth-first search (memory-efficient)

Agenda-based SLD-prover

```
prove(true) :- !.
```

Agenda-based SLD-prover

Built-in predicate.

```
prove(true) :- !.  
prove(A,B) :- !,  
    clause(A,C),  
    conj_append(C,B,D),  
    prove(D).
```

From SWI Prolog documentation: *True if A can be unified with a clause head and B with the corresponding clause body. Gives alternative clauses on backtracking. For facts, B is unified with the atom true.*

swish.swi-prolog.org

SWISH -- SWI-Prolog for SHaring

File ▾ Edit ▾ Examples ▾ Help ▾ 192 users online

Search

Program Program Program Program

+ 1 **flies(X) :- has_wings(X).**
2 **has_wings(X) :- bird(X).**
3 **has_wings(X) :- airplane(X).**
4 **bird(tweety).**



?- clause(has_wings(**X**),**B**)

Examples ▾ History ▾ Solutions ▾ Run!

A screenshot of the SWISH web-based SWI-Prolog interface. The top navigation bar includes links for File, Edit, Examples, and Help, along with a search bar and user statistics (192 users online). Below the navigation is a toolbar with four tabs, each showing a program icon and the word "Program". The main workspace on the left contains a Prolog program with four rules: flies(X) :- has_wings(X), has_wings(X) :- bird(X), has_wings(X) :- airplane(X), and bird(tweety). A large, cartoonish owl logo is centered in the background. In the bottom right, there is a query window with the goal ?- clause(has_wings(**X**),**B**). At the very bottom, there are buttons for Examples, History, Solutions, and a highlighted "Run!" button.

swish.swi-prolog.org

SWISH -- SWI-Prolog for SHaring

File ▾ Edit ▾ Examples ▾ Help ▾ 190 users online

Search

Program X Program X Program X Program X

+

```
1 flies(X) :- has_wings(X).
2 has_wings(X) :- bird(X).
3 has_wings(X) :- airplane(X).
4 bird(tweety).
```



clause(has_wings(X),B)

B = bird(X)

Next | 10 | 100 | 1,000 | Stop

?- clause(has_wings(X),B)

Examples ▾ History ▾ Solutions ▾ Run!

swish.swi-prolog.org

SWISH -- SWI-Prolog for SHaring

File ▾ Edit ▾ Examples ▾ Help ▾ 190 users online

Search 

Program X Program X Program X Program X

+
1 **flies(X) :- has_wings(X).**
2 **has_wings(X) :- bird(X).**
3 **has_wings(X) :- airplane(X).**
4 **bird(tweety).**

 190 users online

clause(has_wings(X),B)
B = bird(X)
Next 10 100 1,000 Stop

?- clause(has_wings(X),B)

Examples ▾ History ▾ Solutions Table results Run!

A green arrow points from the 'B = bird(X)' result in the top panel to the 'clause(has_wings(X),B)' query in the bottom panel.

The screenshot shows the SWISH web interface for SWI-Prolog. The title bar reads "swish.swi-prolog.org" and "SWISH -- SWI-Prolog for SHaring". The main area displays a Prolog program:

```
1 flies(X) :- has_wings(X).  
2 has_wings(X) :- bird(X).  
3 has_wings(X) :- airplane(X).  
4 bird(tweety).
```

The interface includes tabs for "Program" and a search bar with a magnifying glass icon. On the right, there's a large cartoon owl logo. Below the program, the results of a query are shown:

```
clause(has_wings(X),B)  
B = bird(X)  
B = airplane(X)
```

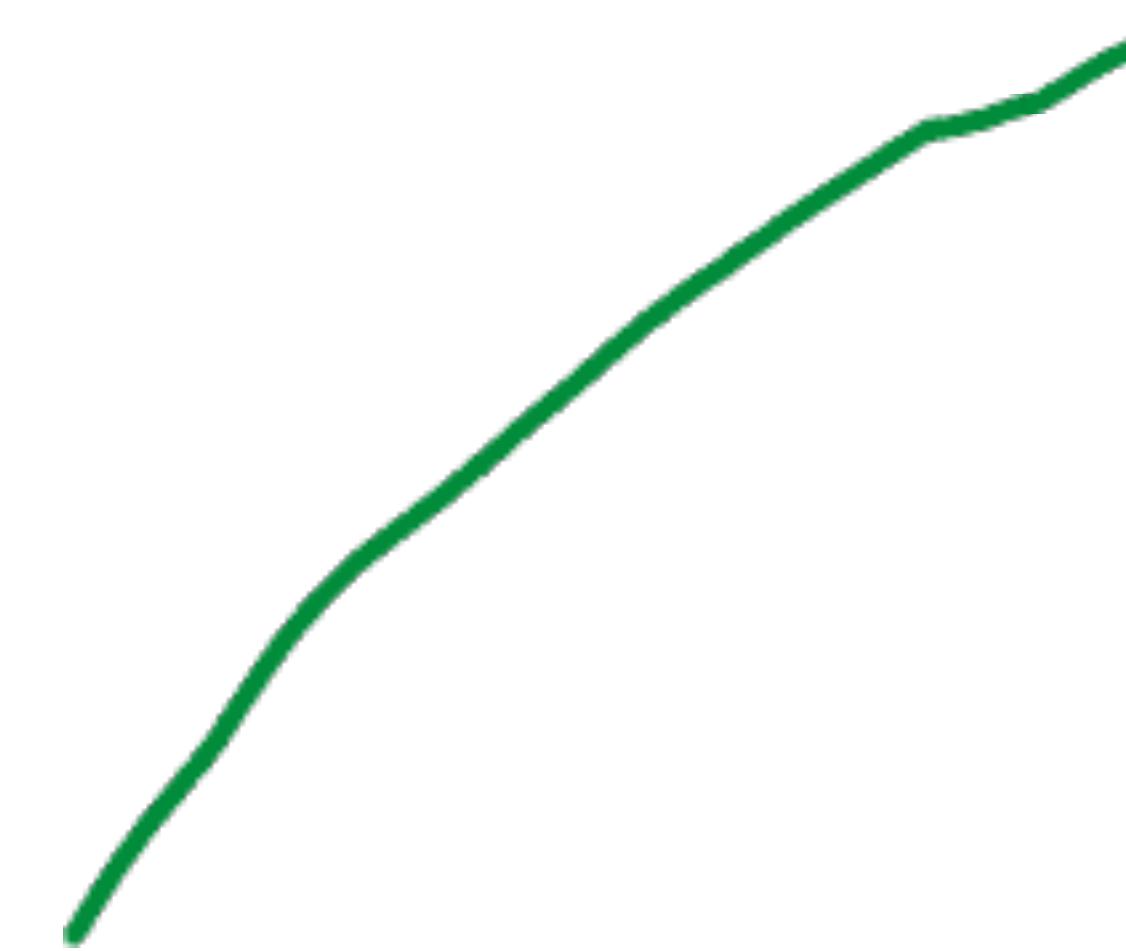
The third result, "B = airplane(X)", is highlighted with a green oval. At the bottom, a query prompt shows:

```
?- clause(has_wings(X),B)
```

At the very bottom, there are navigation links: "Examples", "History", "Solution", "table results", and a "Run!" button.

Agenda-based SLD-prover

```
prove(true) :- ! .  
prove(A,B) :- ! ,  
    clause(A,C) ,  
    conj_append(C,B,D) ,  
    prove(D) .
```



Auxiliary predicate `conj_append`:

```
conj_append(true, Ys, Ys) .  
conj_append(X, Ys, (X, Ys)) :- not(X=true), not(X=(_, _)) .  
conj_append((X, Xs), Ys, (X, Zs)) :- conj_append(Xs, Ys, Zs) .
```

Agenda-based SLD-prover

```
prove(true) :- ! .  
prove((A,B)) :- ! ,  
    clause(A,C) ,  
    conj_append(C,B,D) ,  
    prove(D) .  
prove(A) :-  
    clause(A,B) ,  
    prove(B) .
```

swish.swi-prolog.org

SWISH -- SWI-Prolog for SHaring

File ▾ Edit ▾ Examples ▾ Help ▾ 190 users online

Search

Program Program Program Program

+

```
1 conj_append(true, Ys, Ys).
2 conj_append(X, Ys, (X, Ys)):- not(X=true), not(X=(_, _)).
3 conj_append((X, Xs), Ys, (X, Zs)):- conj_append(Xs, Ys, Zs).
4
5 prove(true):-!.
6 prove((A,B)):-!,
7     clause(A,C), conj_append(C,B,D), prove(D).
8 prove(A):-clause(A,B), prove(B).
9
10
11 flies(X) :- has_wings(X).
12 has_wings(X) :- bird(X).
13 bird(tweety).
14 bird(donald).
```

?- prove(flies(X))

Examples ▾ History ▾ Solution ▾ Table results Run!



A screenshot of the SWISH web interface for SWI-Prolog. The interface includes a top bar with navigation icons, a search bar, and a user count of 190 users online. Below the bar, there are tabs for four different programs, with the third one currently selected. The main area contains a code editor with numbered lines of Prolog code. Lines 11 through 14, which define the `flies` predicate and its base cases, are highlighted with a green oval. To the right of the code editor is a large, stylized owl logo with three eyes. At the bottom right, there is a command-line interface showing the query `?- prove(flies(X))` and buttons for navigating between examples, history, solutions, and running the query.

swish.swi-prolog.org

SWISH -- SWI-Prolog for SHaring

File ▾ Edit ▾ Examples ▾ Help ▾ 190 users online

Search

Program Program Program Program

+

```
1 conj_append(true, Ys, Ys).
2 conj_append(X, Ys, (X, Ys)):- not(X=true), not(X=(_, _)).
3 conj_append((X, Xs), Ys, (X, Zs)):- conj_append(Xs, Ys, Zs).
4
5 prove(true):-!.
6 prove((A,B)):-!,
7     clause(A,C), conj_append(C,B,D), prove(D).
8 prove(A):-clause(A,B), prove(B).
9
10
11 flies(X) :- has_wings(X).
12 has_wings(X) :- bird(X).
13 bird(tweety).
14 bird(donald).
```



?- prove(flies(X))

Examples ▾ History ▾ Solution ▾ Table results Run!

swish.swi-prolog.org

SWISH -- SWI-Prolog for SHaring

File ▾ Edit ▾ Examples ▾ Help ▾ 190 users online

Search

Program Program Program Program

+

```
1 conj_append(true, Ys, Ys).
2 conj_append(X, Ys, (X, Ys)) :- not(X = true), !.          Recursive call
3 conj_append((X, Xs), Ys, (X, Zs)) :- conj_append(Xs, Ys, Zs).
4
5 prove(true) :- !.
6 prove((A, B)) :- !,
7   clause(A, C), conj_append(C, B, D), prove(D).
8 prove(A) :- clause(A, B), prove(B).
9
10
11 flies(X) :- has_wings(X).
12 has_wings(X) :- bird(X).
13 bird(tweety).
14 bird(donald).
```



prove(flies(X))

X = tweety

Next 10 100 1,000 Stop

?- prove(flies(X))

Examples ▾ History ▾ Solution ▾ Table results Run!

The screenshot shows the SWISH -- SWI-Prolog for SHaring web application. At the top, there's a browser header with a lock icon, the URL "swish.swi-prolog.org", and standard navigation buttons. Below the header is the SWISH logo and a status bar indicating "190 users online". The main interface has tabs for "Program" and a search bar. On the left, a code editor displays a Prolog program:

```
1 conj_append(true, Ys, Ys).
2 conj_append(X, Ys, (X, Ys)) :- not(X = true), !.          Recursive call
3 conj_append((X, Xs), Ys, (X, Zs)) :- conj_append(Xs, Ys, Zs).
4
5 prove(true) :- !.
6 prove((A, B)) :- !,
7   clause(A, C), conj_append(C, B, D), prove(D).
8 prove(A) :- clause(A, B), prove(B).

9
10
11 flies(X) :- has_wings(X).
12 has_wings(X) :- bird(X).
13 bird(tweety).
14 bird(donald).
```

A green arrow points from the "prove(flies(X))" query in the results panel to the "conj_append" rule in the code editor, highlighting the recursive call.

The results panel shows the query `prove(flies(X))` and its answer `X = tweety`. It includes a navigation bar with buttons for "Next", "10", "100", "1,000", and "Stop".

At the bottom, there are links for "Examples", "History", "Solution", "Table results", and a "Run!" button.

The screenshot shows the SWISH -- SWI-Prolog for SHaring web application. The interface includes a top bar with navigation icons, a search bar, and a user count of "189 users online". Below the bar, there are tabs for "Program" and a main code editor area.

Code Editor Area:

```
1 conj_append(true, Ys, Ys).
2 conj_append(X, Ys, (X, Ys)) :- not(X=true), not(X=(_, _)).
3 conj_append((X, Xs), Ys, (X, Zs)) :- conj_append(Xs, Ys, Zs).
4
5 prove(true) :- !.
6 prove((A, B)) :- !,
7     clause(A, C), conj_append(C, B, D), prove(D).
8 prove(A) :- clause(A, B), prove(B).
9
10
11 flies(X) :- has_wings(X).
12 has_wings(X) :- bird(X).
13 bird(tweety).
14 bird(donald).
```

Execution Results:

A large owl icon is displayed on the right side of the interface. Below it, the query `prove(flies(X))` is shown with results: `X = tweety` and `X = donald`. The result `X = donald` is highlighted with a green oval.

Bottom Panel:

```
?- prove(flies(X))
```

Buttons at the bottom include "Examples", "History", "Solution", "Table results", and "Run!".

Agenda-based SLD-prover

Agenda-based:

```

prove_df_a(Goal) :-  

    prove_df_a([Goal]).  
  

prove_df_a([true|Agenda]).  

prove_df_a([(A,B)|Agenda]) :- !,  

    findall(D,(clause(A,C),conj_append(C,B,D)),Children),  

    append(Children,Agenda,NewAgenda),  

    prove_df_a(NewAgenda).  

prove_df_a([A|Agenda]) :-  

    findall(B,clause(A,B),Children),  

    append(Children,Agenda,NewAgenda),  

    prove_df_a(NewAgenda).

```

Original:

```

prove(true) :- !.  

prove((A,B)) :- !,  

    clause(A,C),  

    conj_append(C,B,D),  

    prove(D).  

prove(A) :-  

    clause(A,B),  

    prove(B).

```

Agenda-based SLD-prover

We can turn it into a complete SLD prover using BFS.

```
prove_df_a(Goal) :-  
    prove_df_a([Goal]).  
  
prove_df_a([true|Agenda]).  
prove_df_a([(A,B)|Agenda]) :- !,  
    findall(D,(clause(A,C),conj_append(C,B,D)),Children),  
    append(Children,Agenda,NewAgenda),  
    prove_df_a(NewAgenda).  
prove_df_a([A|Agenda]) :-  
    findall(B,clause(A,B),Children),  
    append(Children,Agenda,NewAgenda),  
    prove_df_a(NewAgenda).
```

Agenda-based SLD-prover

We can turn it into a complete SLD prover using BFS.

```
prove_df_a(Goal) :-  
  prove_df_a([Goal]).  
  
prove_df_a([true|Agenda]).  
prove_df_a([(A,B)|Agenda]) :- !,  
  findall(D,(clause(A,C),conj_append(C,B,D)),Children),  
  append(Children,Agenda,NewAgenda),  
  prove_df_a(NewAgenda).  
prove_df_a([A|Agenda]) :-  
  findall(B,clause(A,B),Children),  
  append(Children,Agenda,NewAgenda),  
  prove_df_a(NewAgenda).
```

Agenda-based SLD-prover

```
prove_bf_a(Goal):-  
  prove_bf_a([Goal]).  
  
prove_bf_a([true|Agenda]).  
prove_bf_a([(A,B)|Agenda]):- !,  
  findall(D,(clause(A,C),conj_append(C,B,D)),Children),  
  append(Agenda, Children, NewAgenda),  
  prove_bf_a(NewAgenda).  
prove_bf_a([A|Agenda]):-  
  findall(B,clause(A,B),Children),  
  append(Agenda, Children, NewAgenda),  
  prove_bf_a(NewAgenda).
```

We can turn it into a complete SLD prover using BFS.

We would need a few more modifications to also obtain the answer substitutions
(you can read about it in Peter Flach's book which is recommended for this course).

Refutation prover for clausal logic

(Not shown here)

```

refute((false:-true)) .
refute((A,C)) :-  

    cl(Cl) ,  

    resolve(A,Cl,R) ,  

    refute(R) .

```

```

% refute_bf(Clause) <- Clause is refuted by clauses
% defined by cl/1
% (breadth-first search strategy)
refute_bf_a(Clause) :-
    refute_bf_a([a(Clause,Clause)],Clause).

refute_bf_a([a((false:-true),Clause)|Rest],Clause).
refute_bf_a([a(A,C)|Rest],Clause) :-
    findall(a(R,C),(cl(Cl),resolve(A,Cl,R)),Children),
    append(Rest,Children,NewAgenda), % breadth-first
    refute_bf_a(NewAgenda,Clause).

```