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

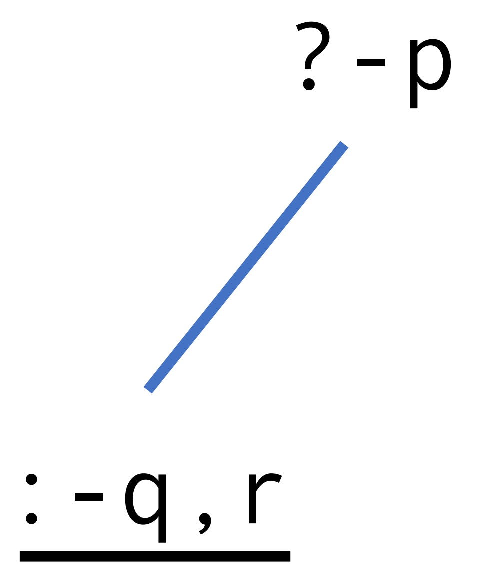
```
p:-q,r.
p:-not(q),s.
s.

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

```
?-p
```

not

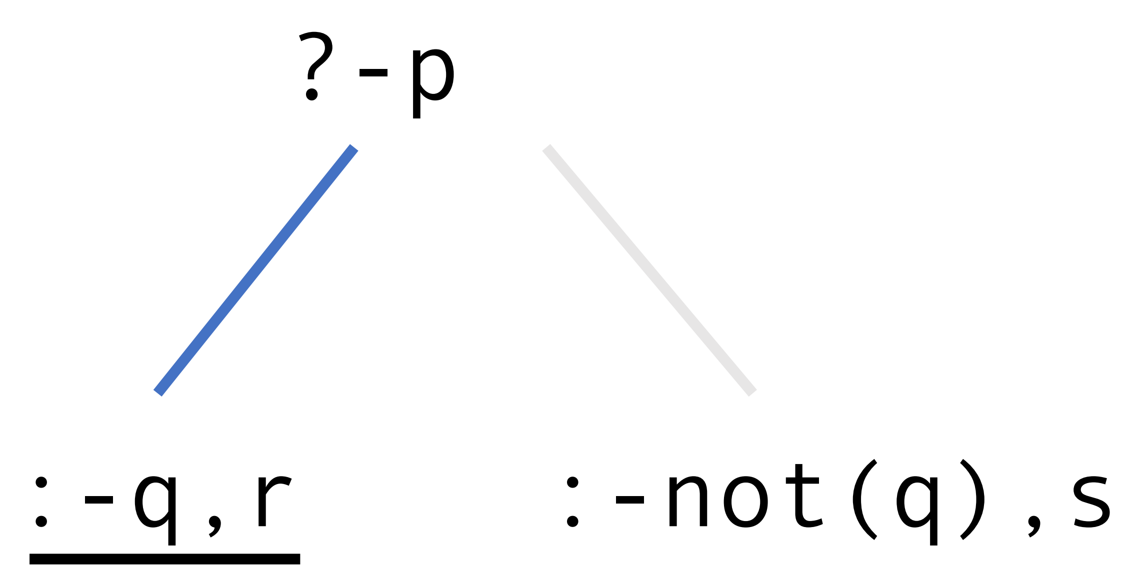
```
p:-q,r.
p:-not(q),s.
s.
```

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



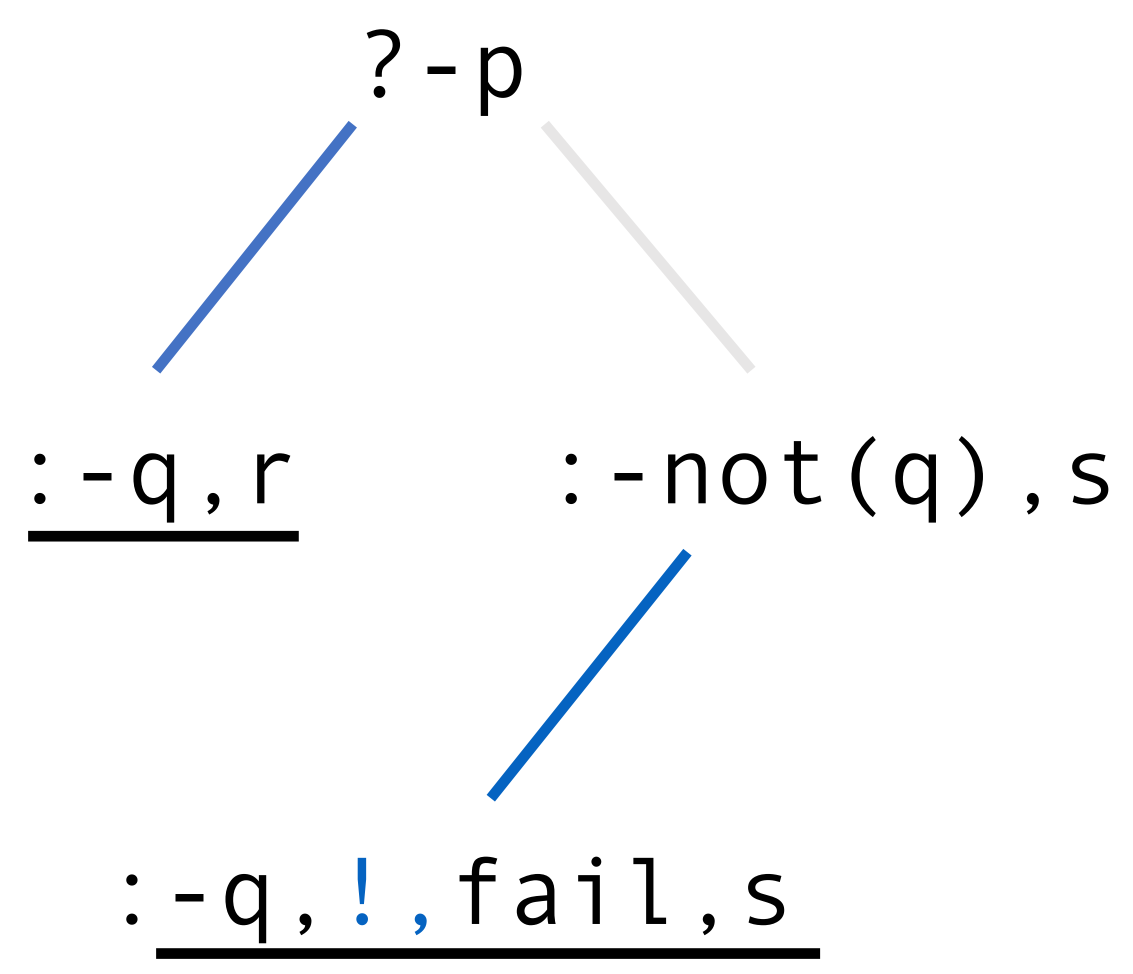
not

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

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



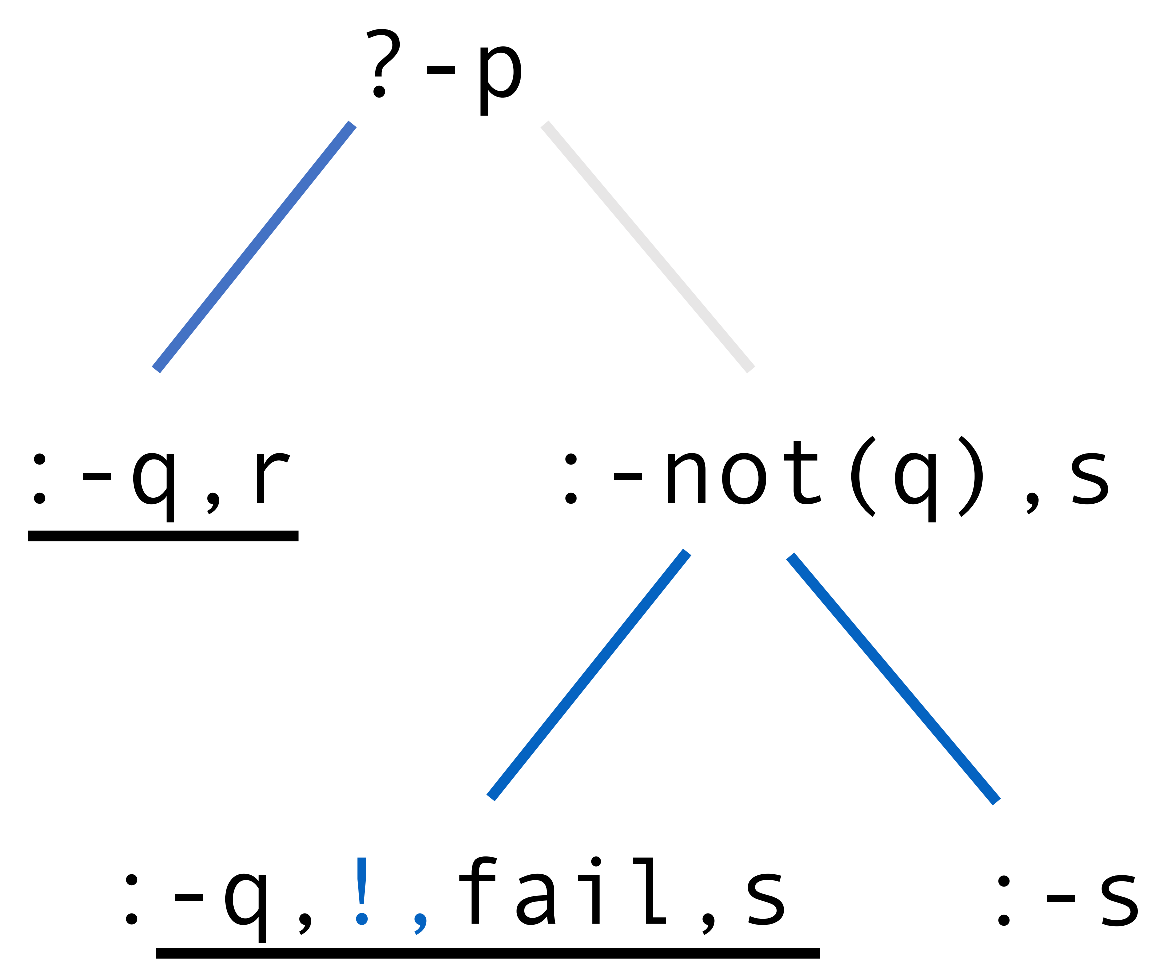
not

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

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



not

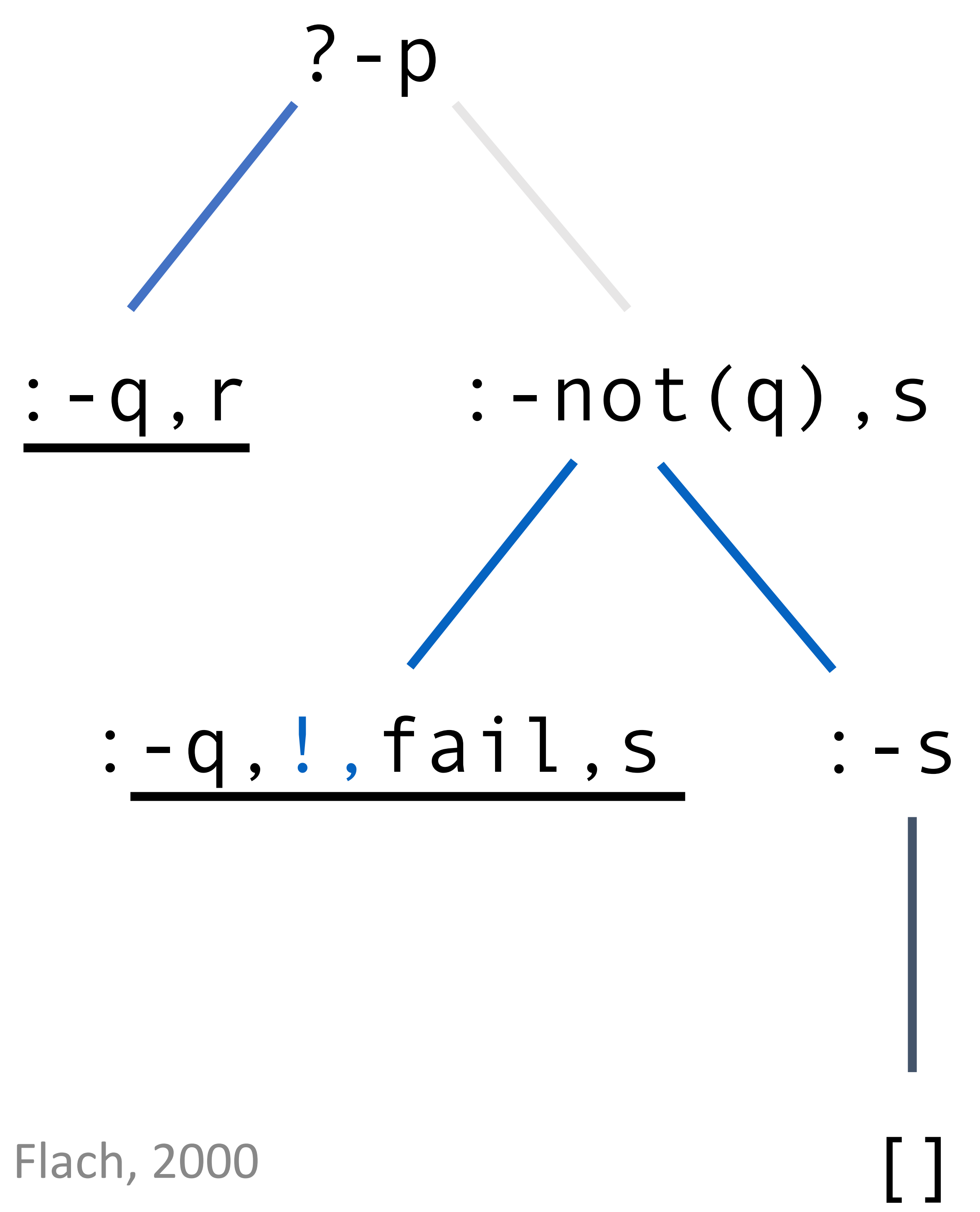
```
p:-q,r.
p:-not(q),s.
s.

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



not

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

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



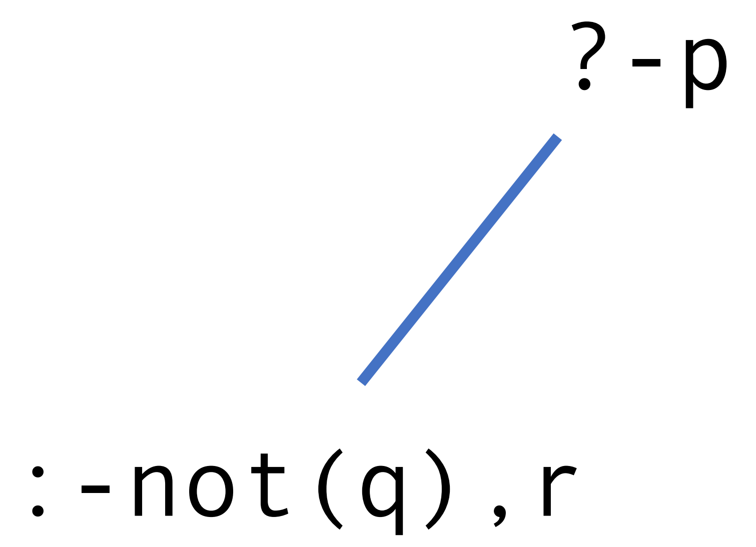
An example: How it works when `:-not(q)` fails

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

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

```
?-p
```

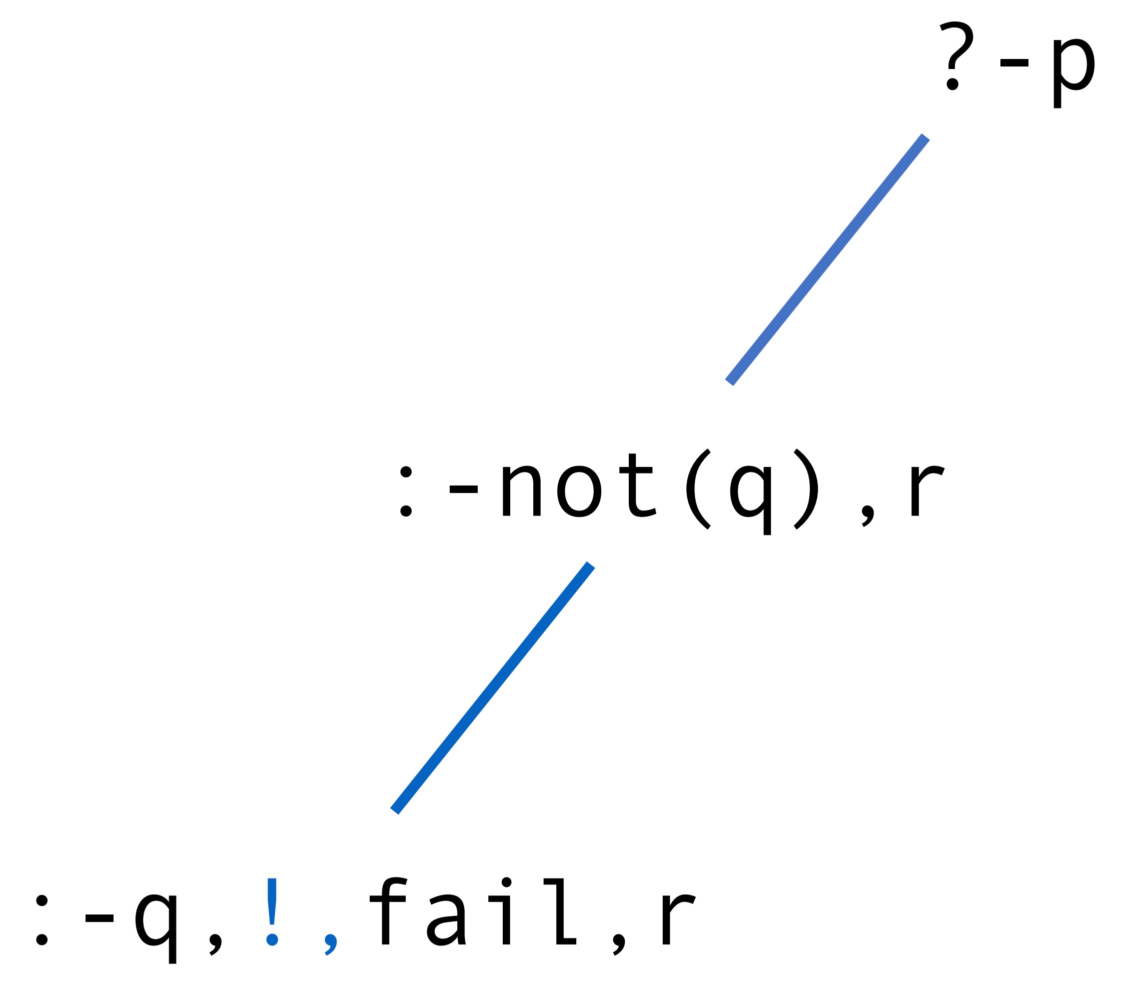
An example: How it works when `:-not(q)` fails

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

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



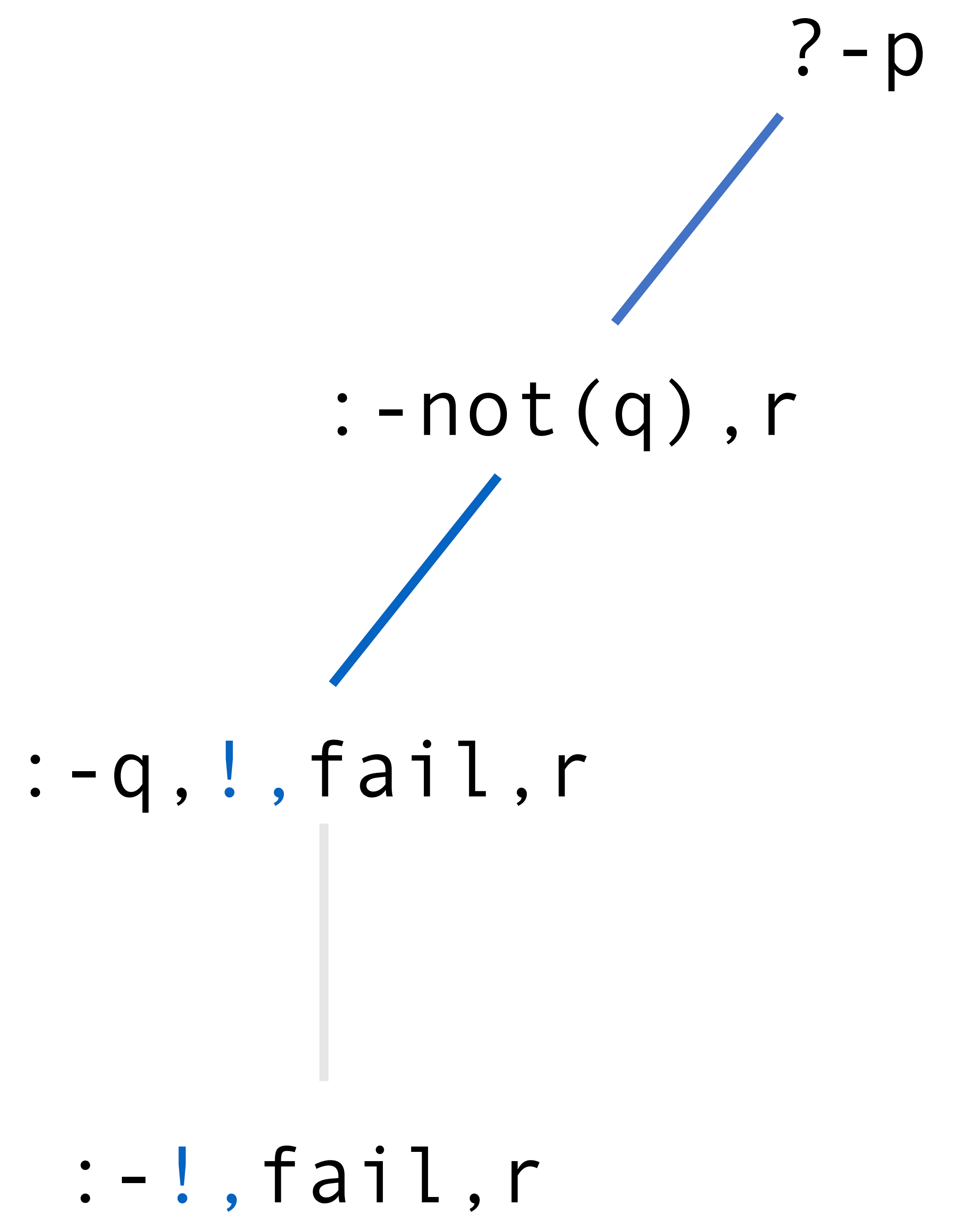
An example: How it works when `:-not(q)` fails

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

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



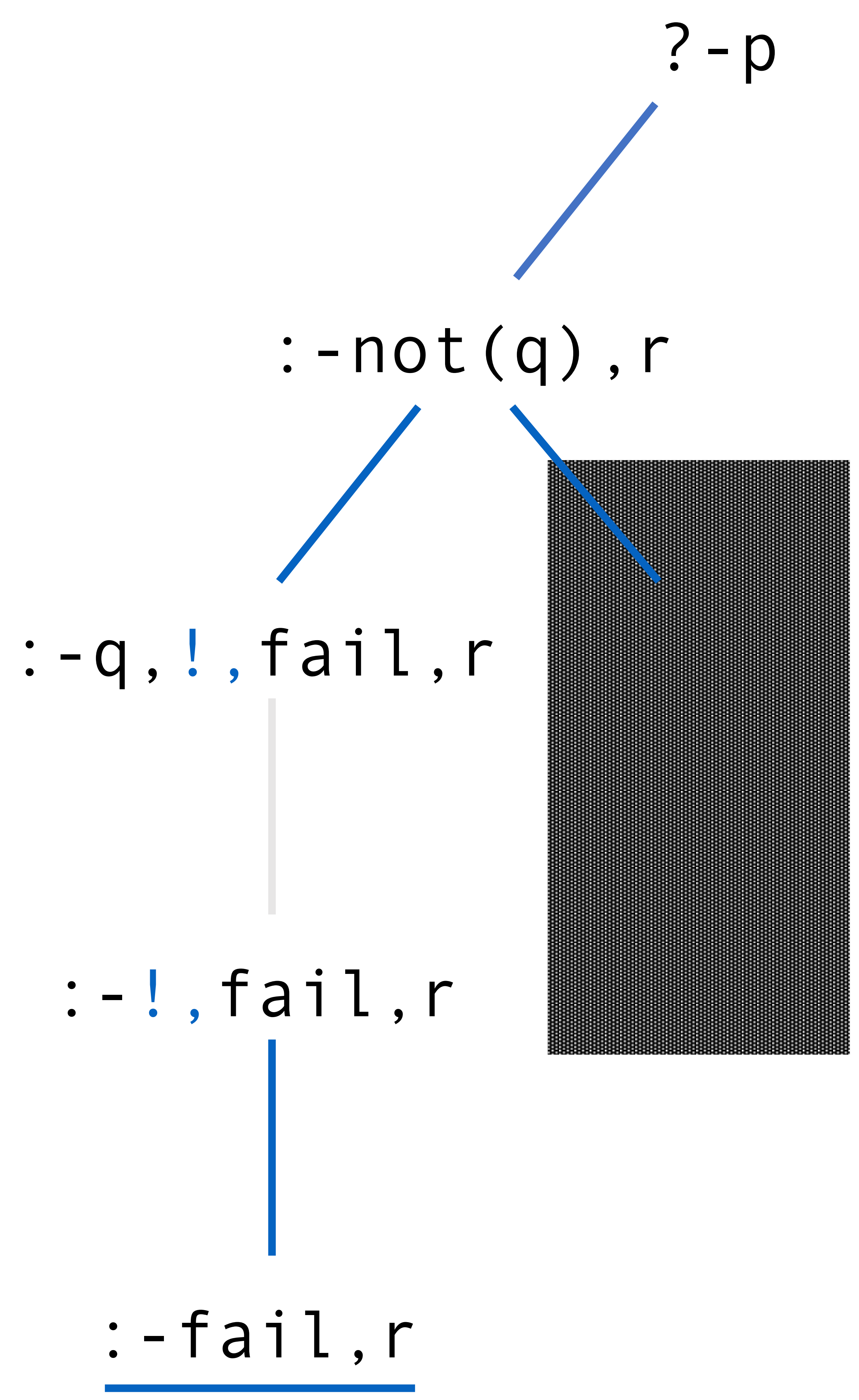
An example: How it works when `:-not(q)` fails

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

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



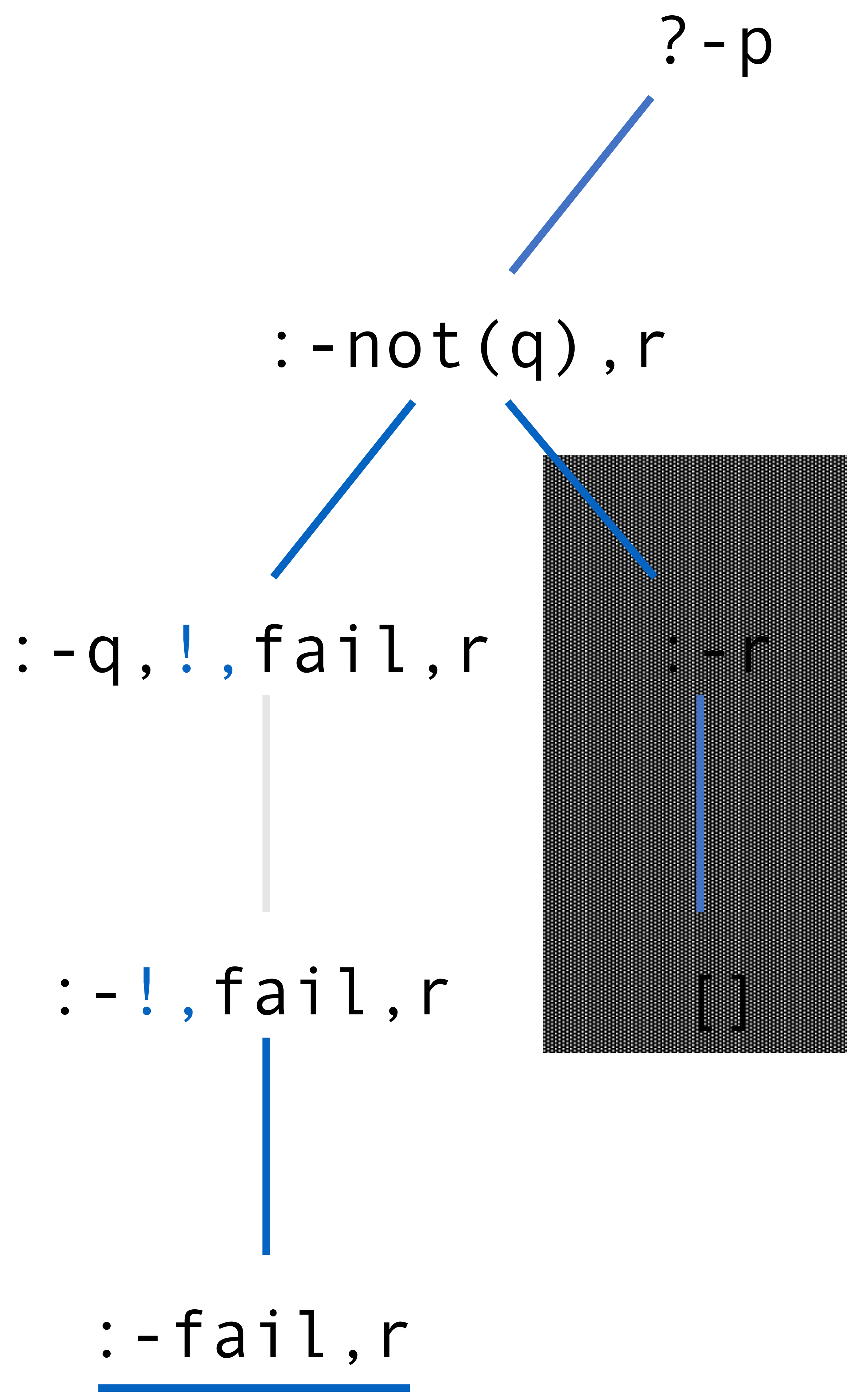
An example: How it works when :-not(q) fails

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

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



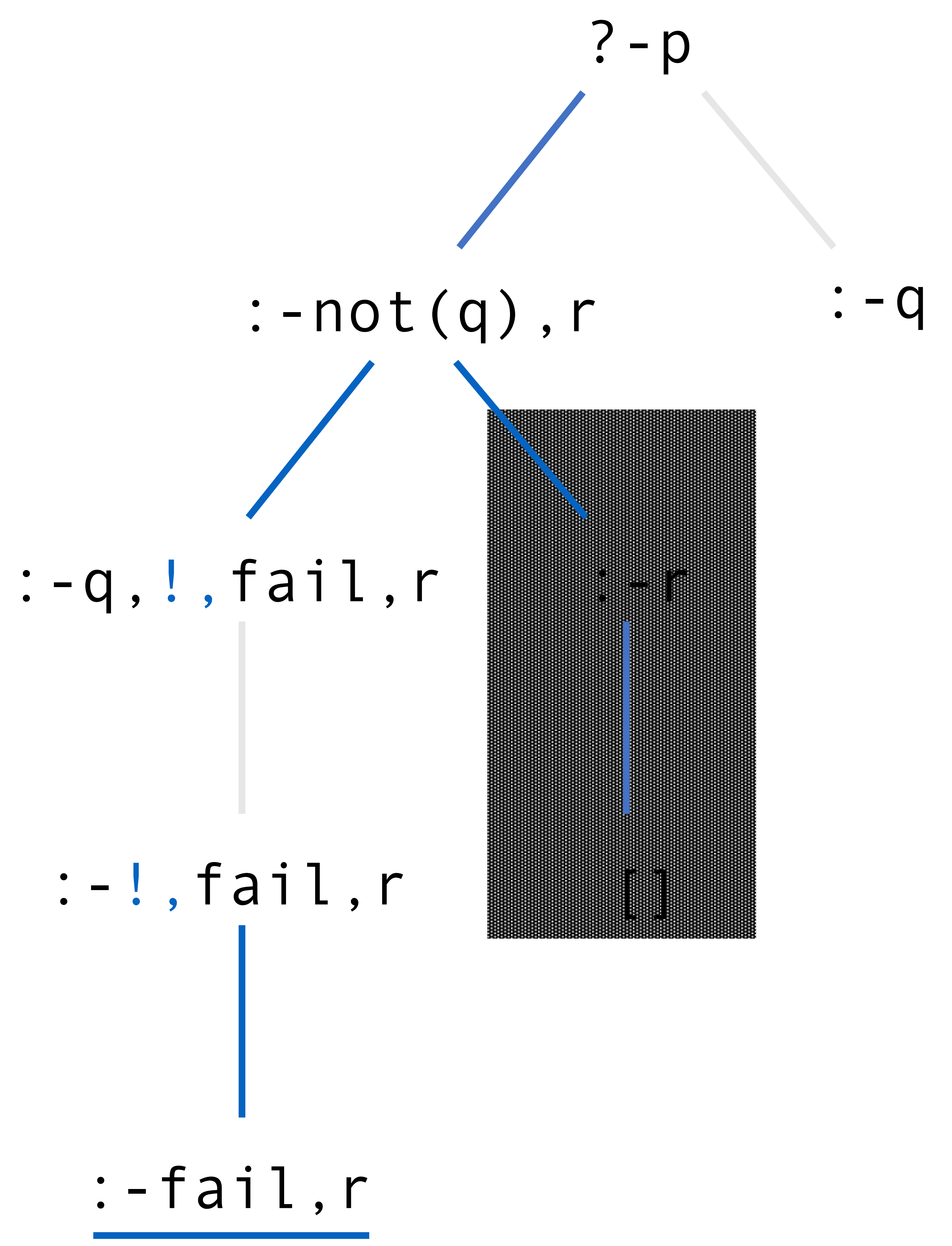
An example: How it works when `:-not(q)` fails

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

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



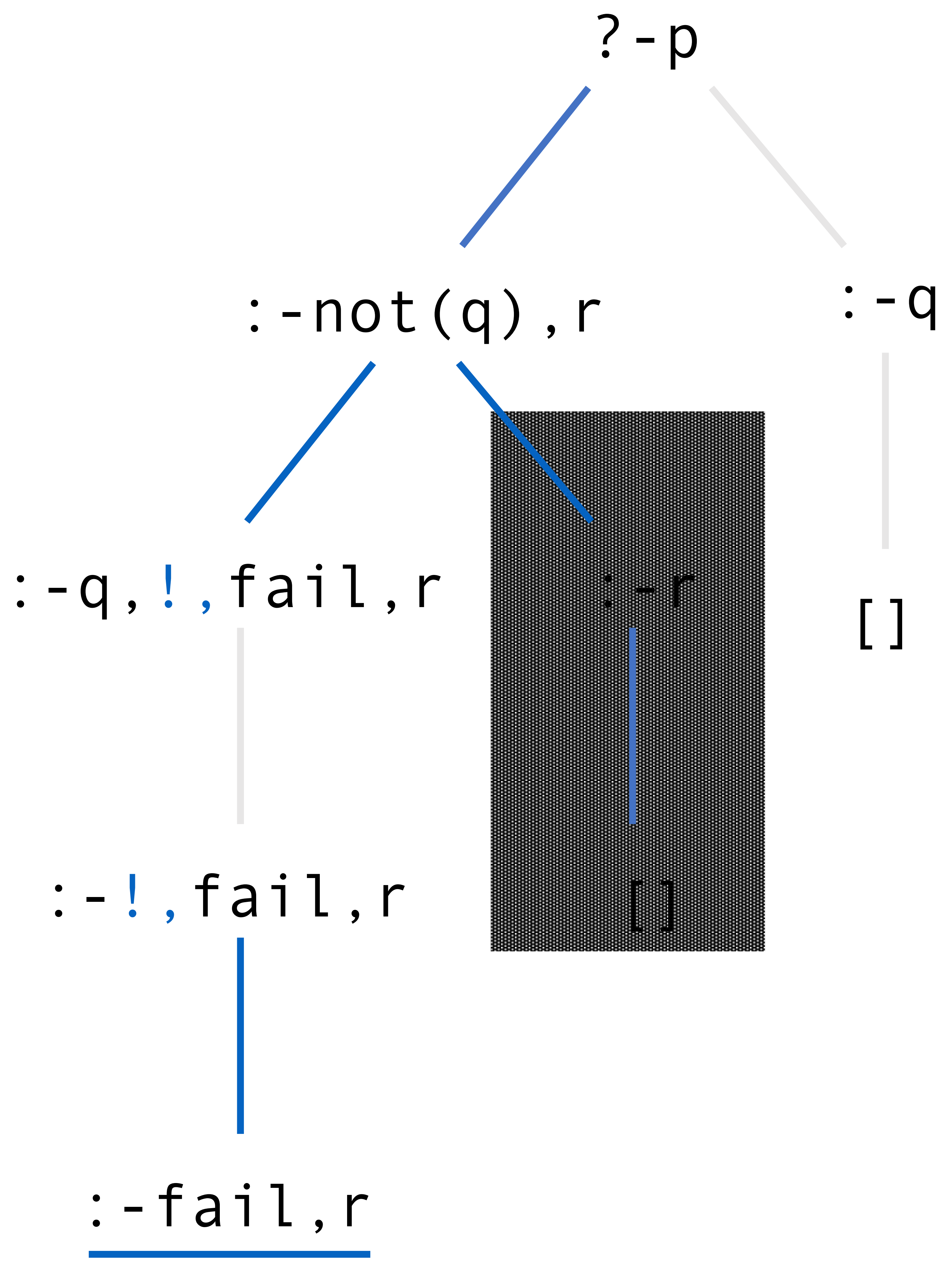
An example: How it works when `:-not(q)` fails

```
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)
```

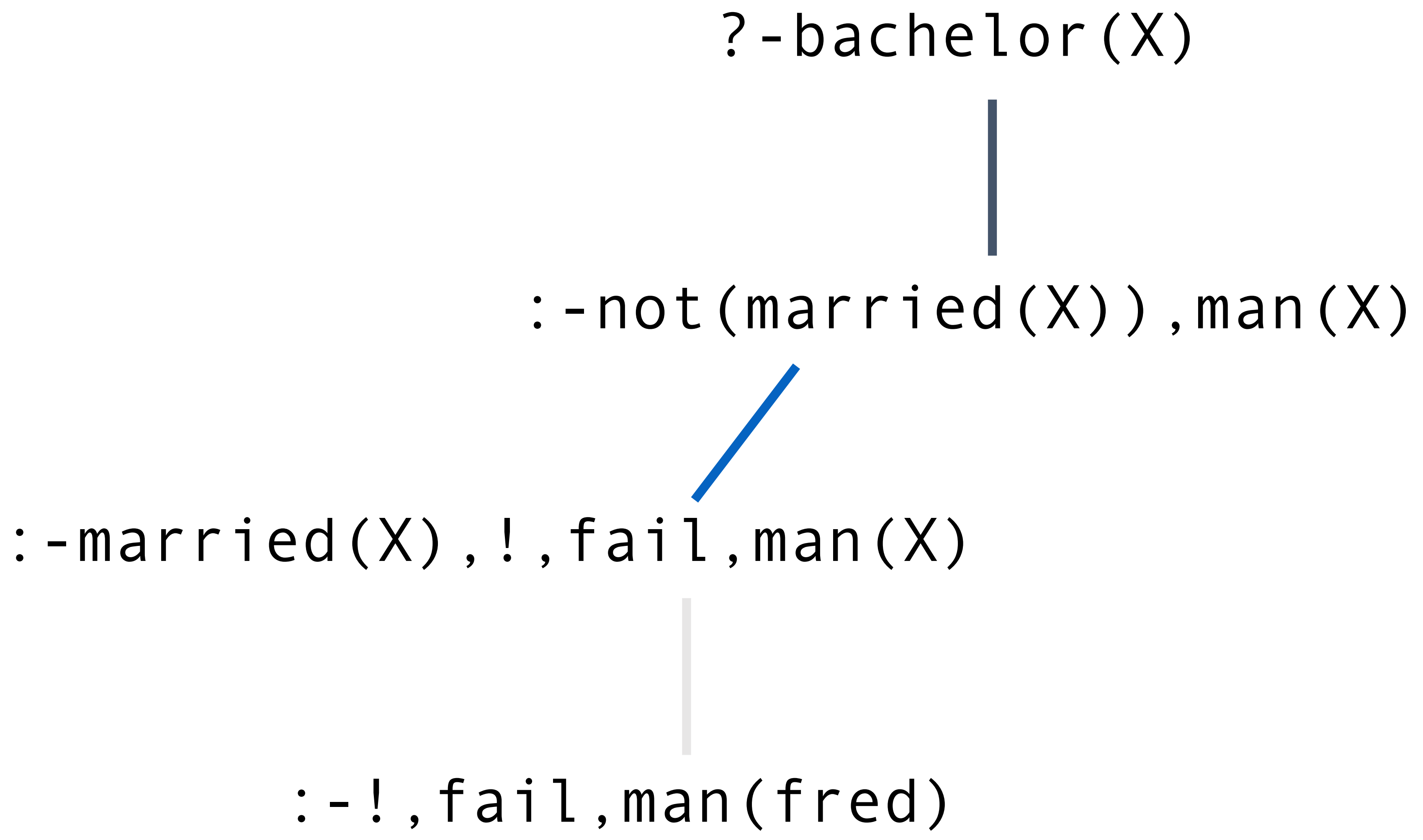
Prolog's `not` is unsound

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

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

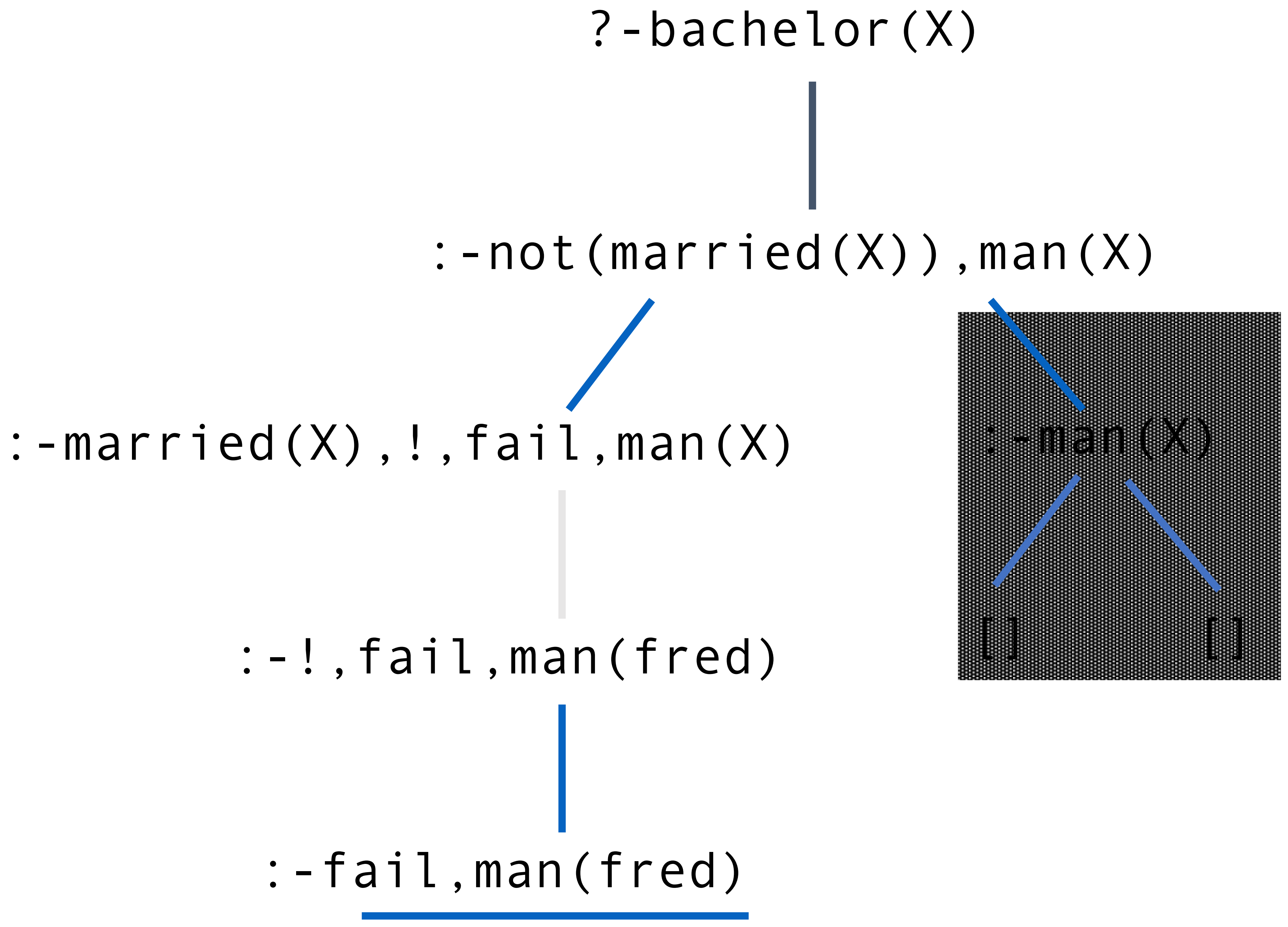
Prolog's `not` is unsound

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



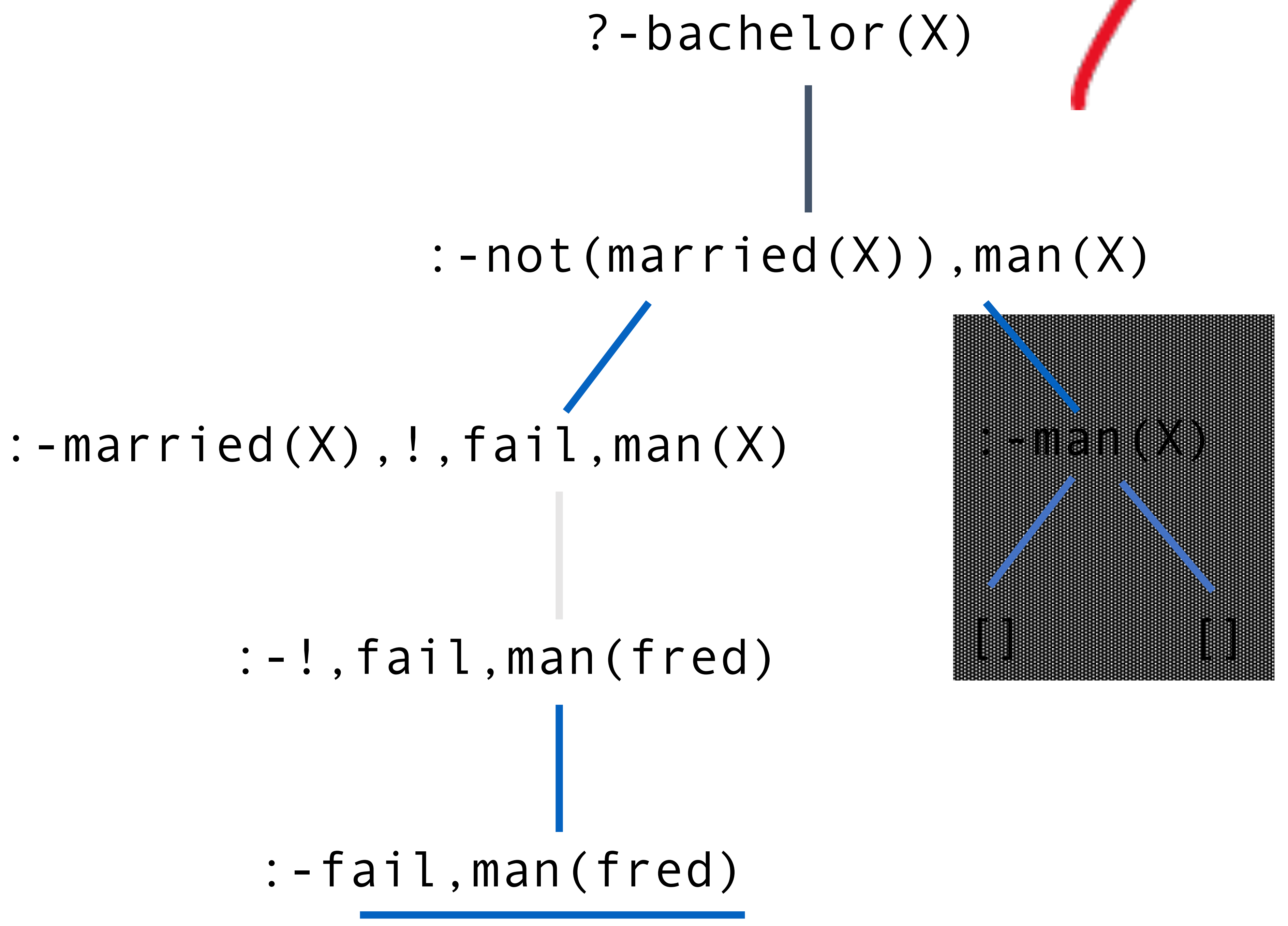
Prolog's `not` is unsound

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



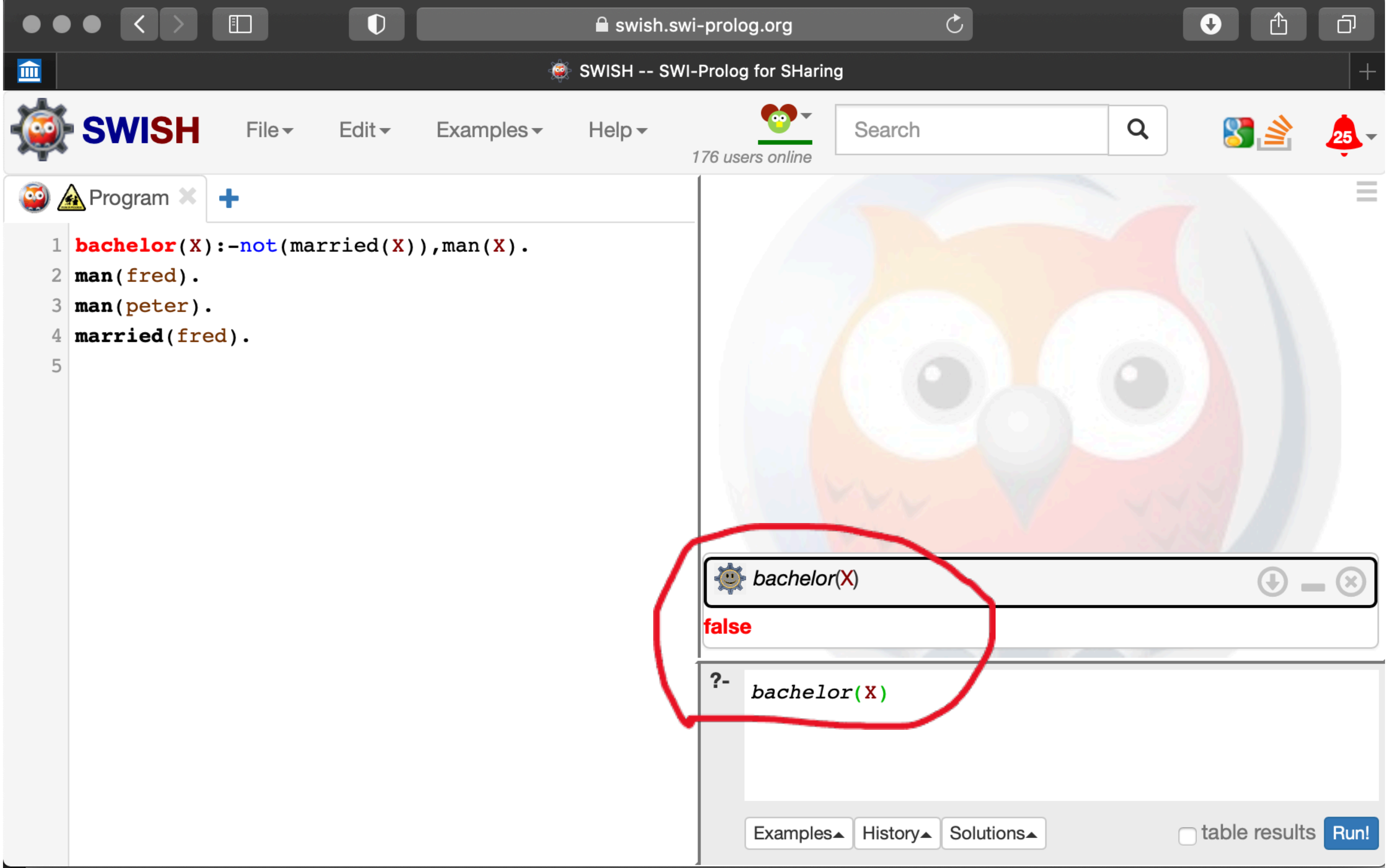
Prolog's `not` is unsound

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



But that's not "correct"!

`peter` is a bachelor and Prolog did not find the corresponding answer!


swish.swi-prolog.org
SWISH -- SWI-Prolog for SHaring
SWISH
File
Edit
Examples
Help
176 users online
Search
25
Program
bachelor(X):-not(married(X)),man(X).
man(fred).
man(peter).
married(fred).
bachelor(X)
false
?- bachelor(X)
Examples
History
Solutions
table results
Run!

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).
```

This will "ground" X.

```
?-bachelor(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.

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

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

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

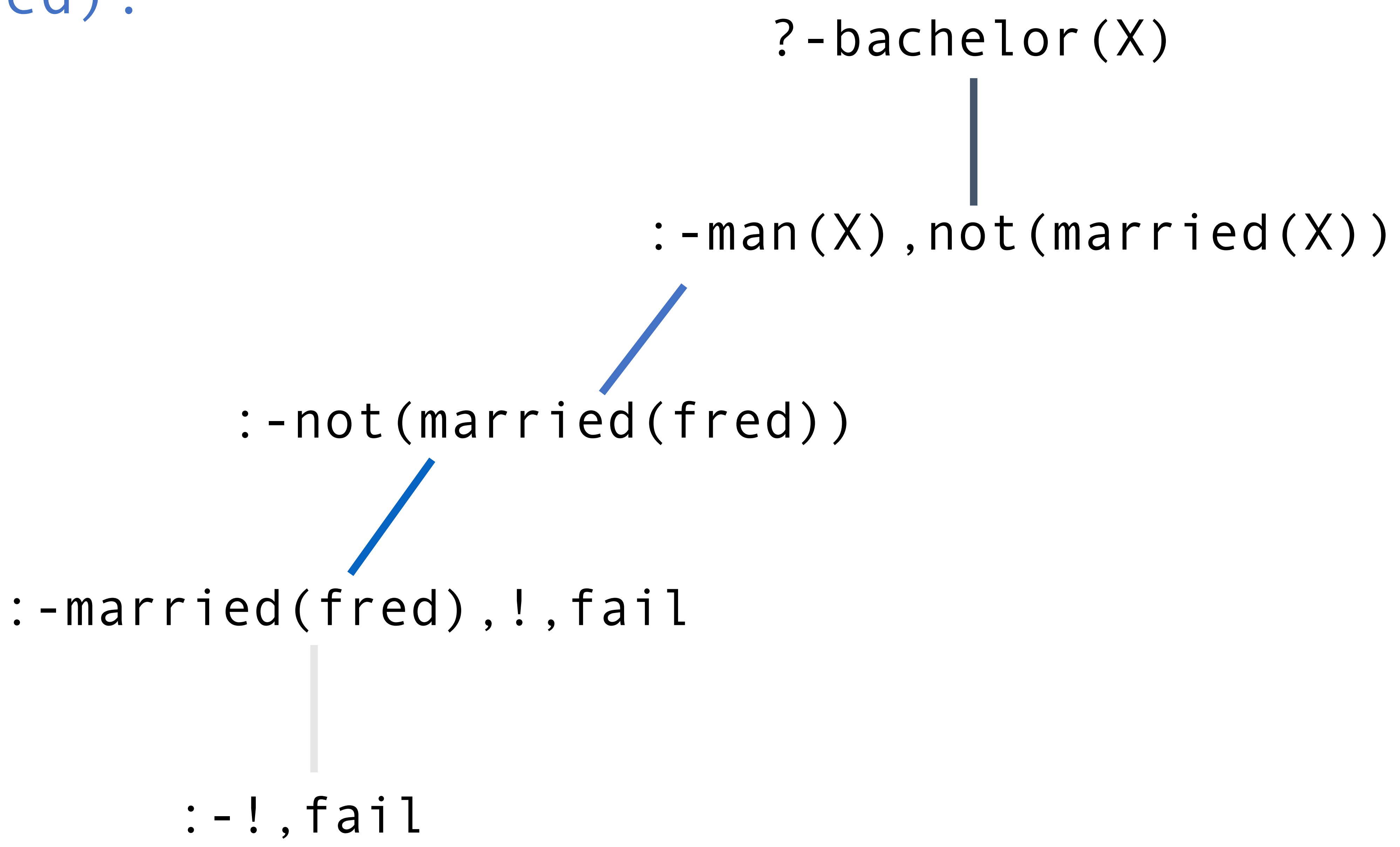
This will "ground" X.

```
?-bachelor(X)
:-man(X),not(married(X))
:-not(married(fred))
:-married(fred),!,fail
```

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

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

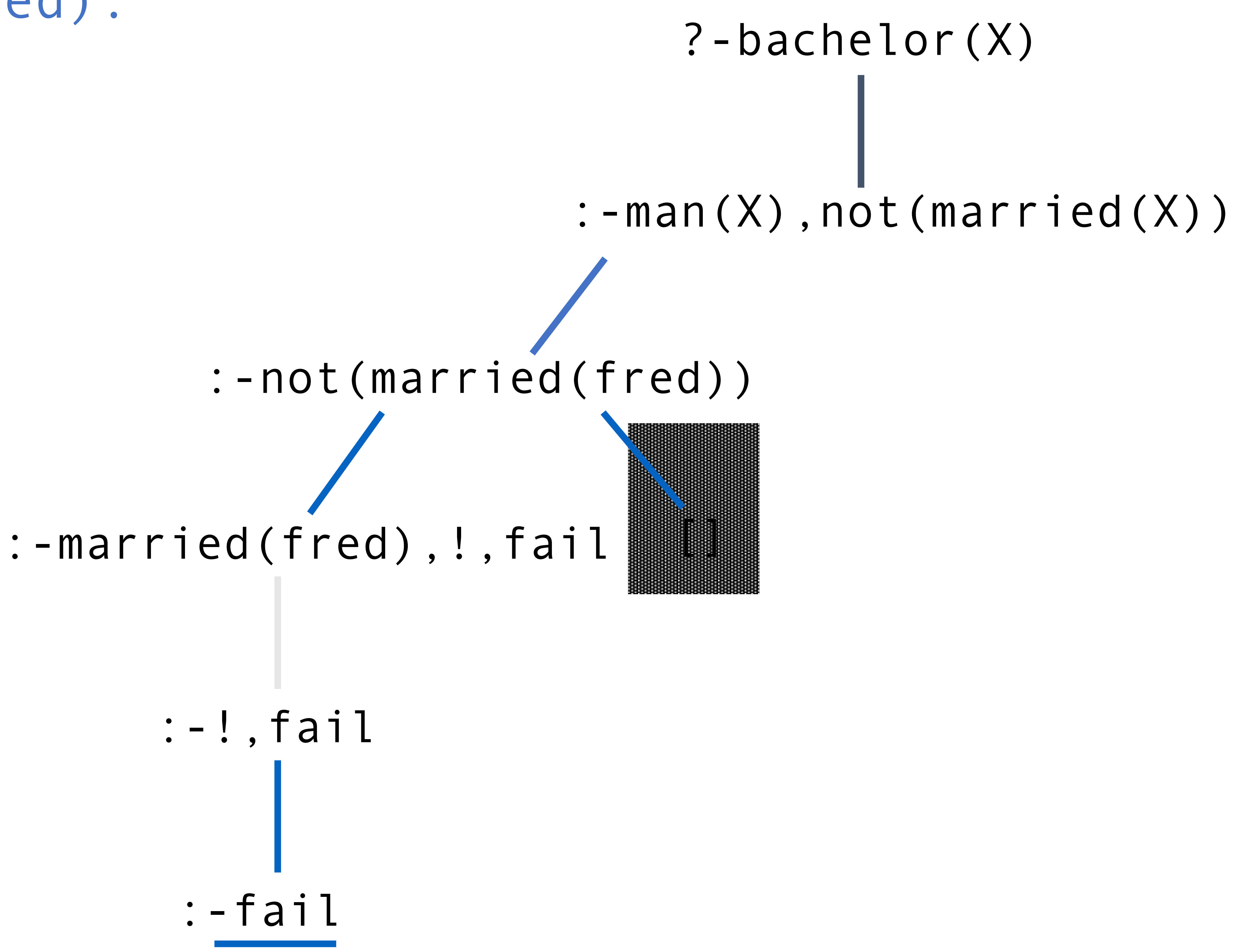
This will "ground" X.



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

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

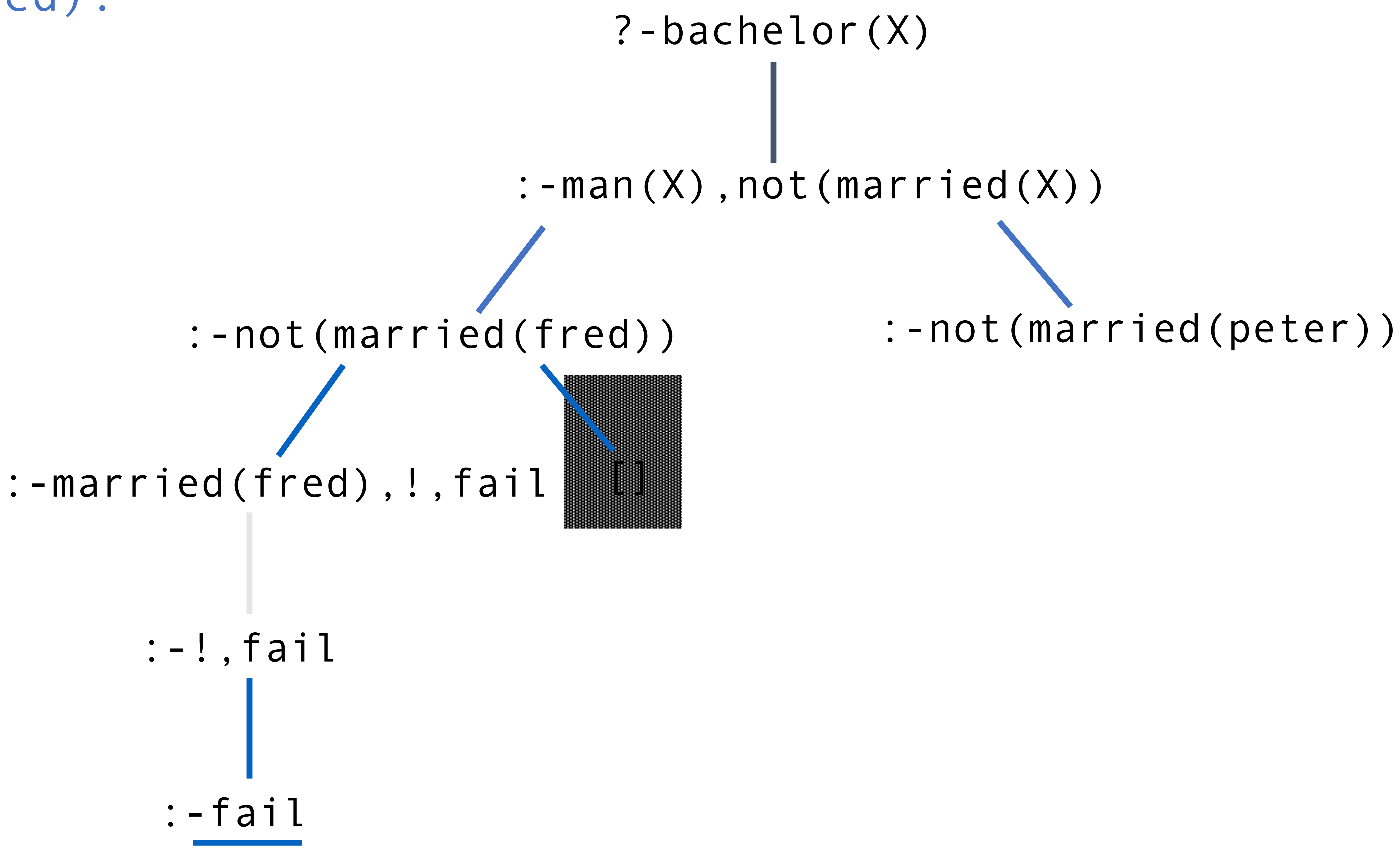
This will "ground" X.



Prolog’s not is unsound – Avoiding the Problem

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

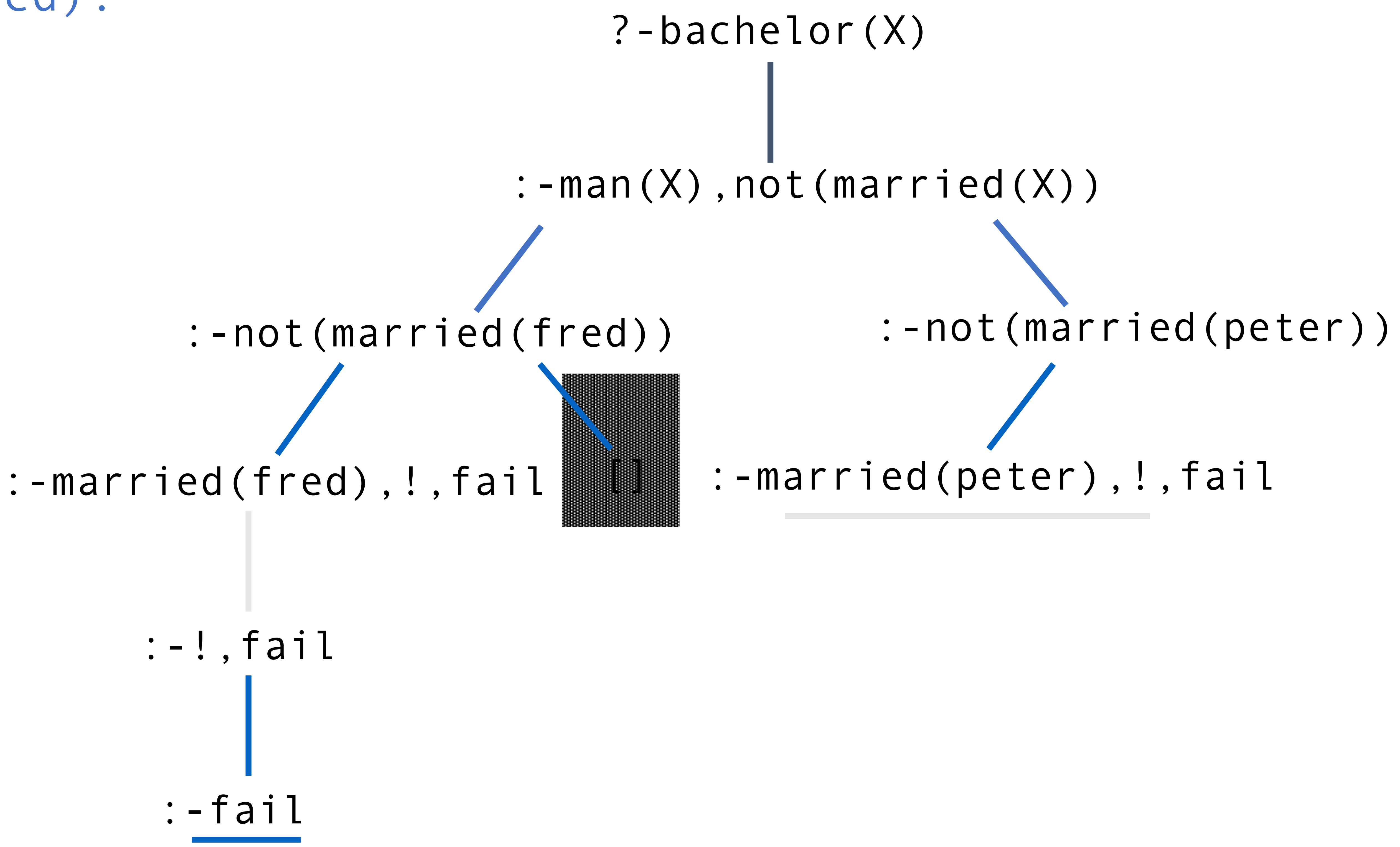
This will ”ground” X.



Prolog's not is unsound – Avoiding the Problem

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

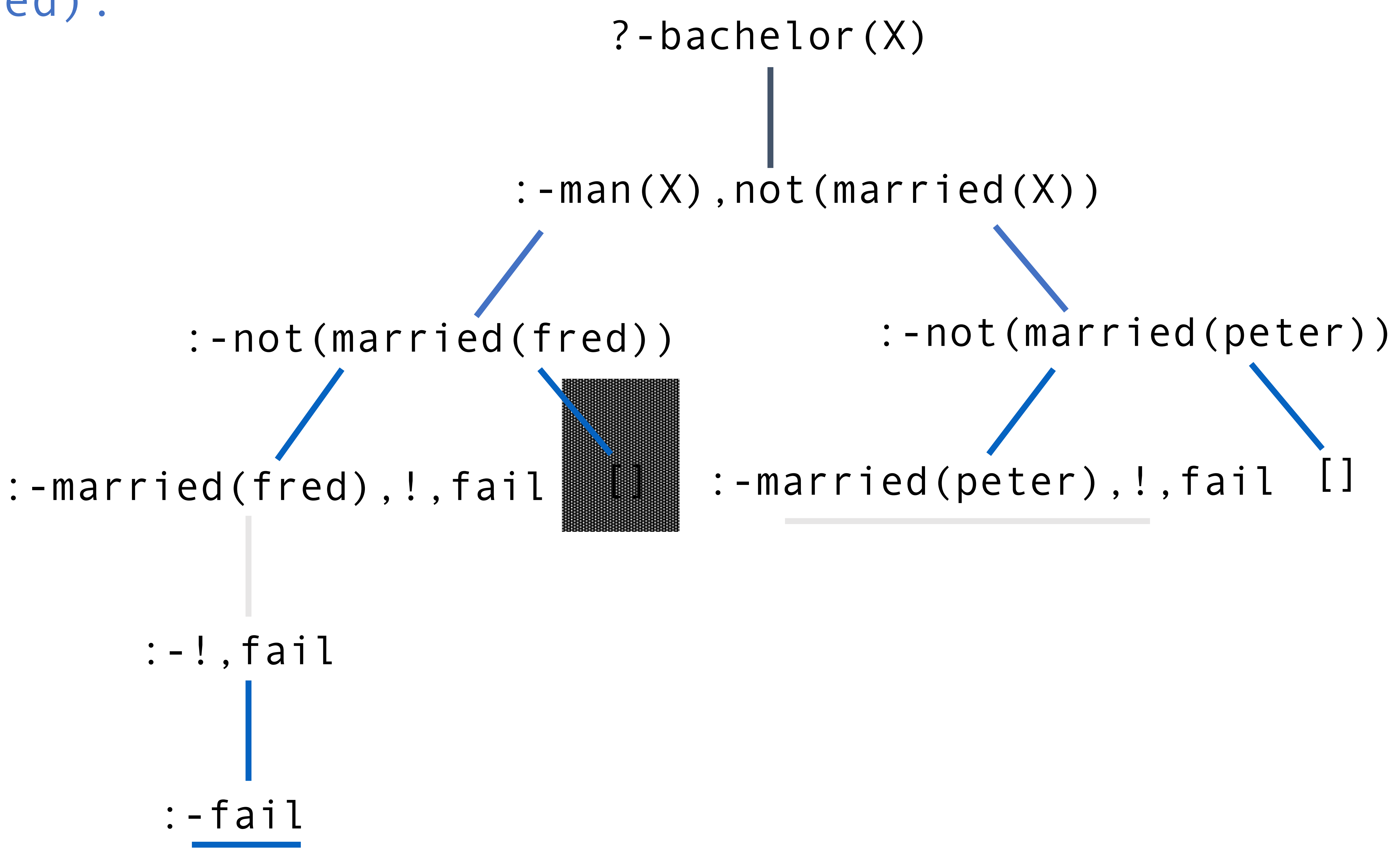
This will "ground" X.

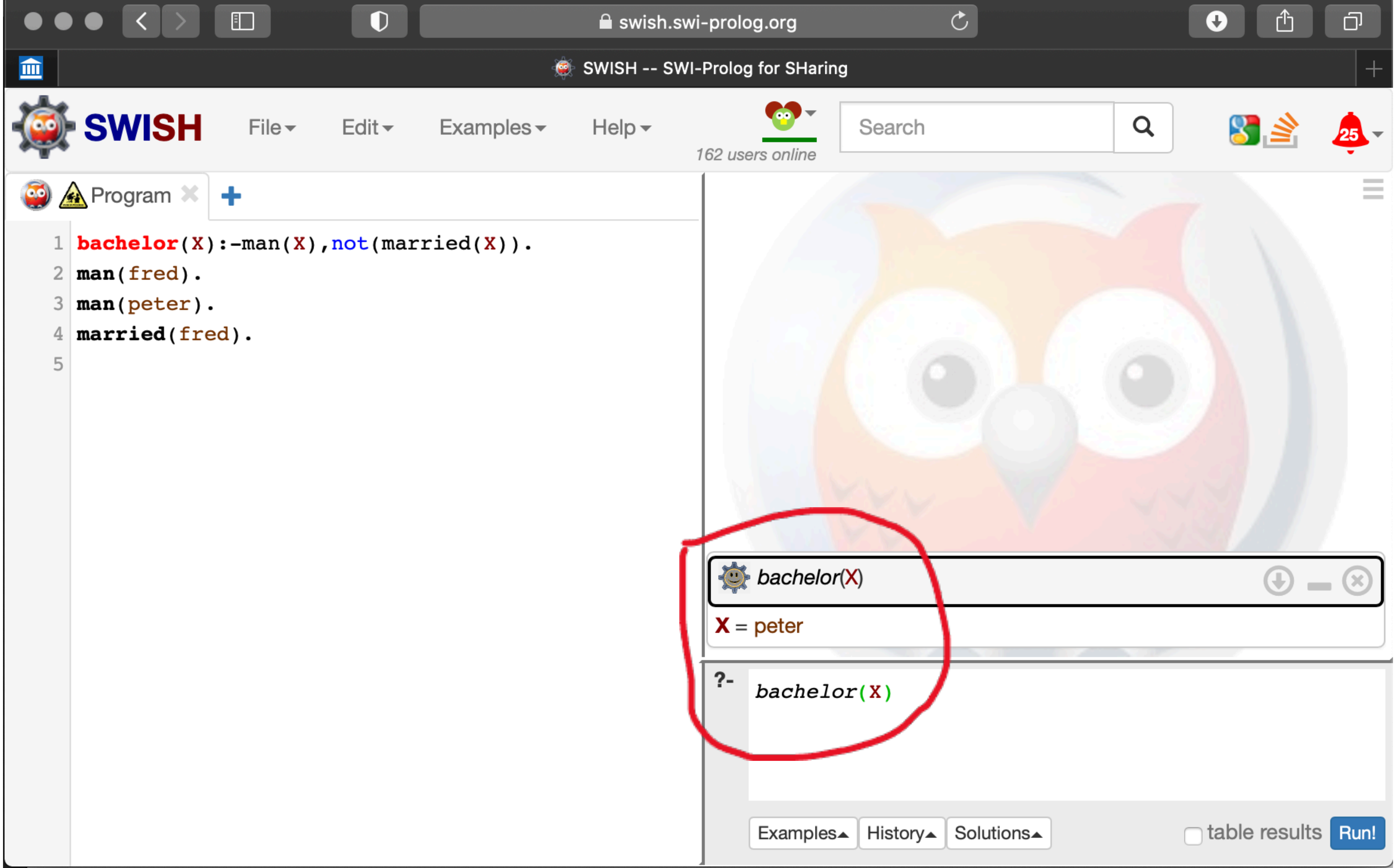


Prolog’s `not` is unsound – Avoiding the Problem

```
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
SWISH
File
Edit
Examples
Help
162 users online
Search
25
Program
bachelor(X):-man(X),not(married(X)).
man(fred).
man(peter).
married(fred).
bachelor(X)
X = peter
?-
bachelor(X)
Examples
History
Solutions
table results
Run!

Arithmetic in Prolog

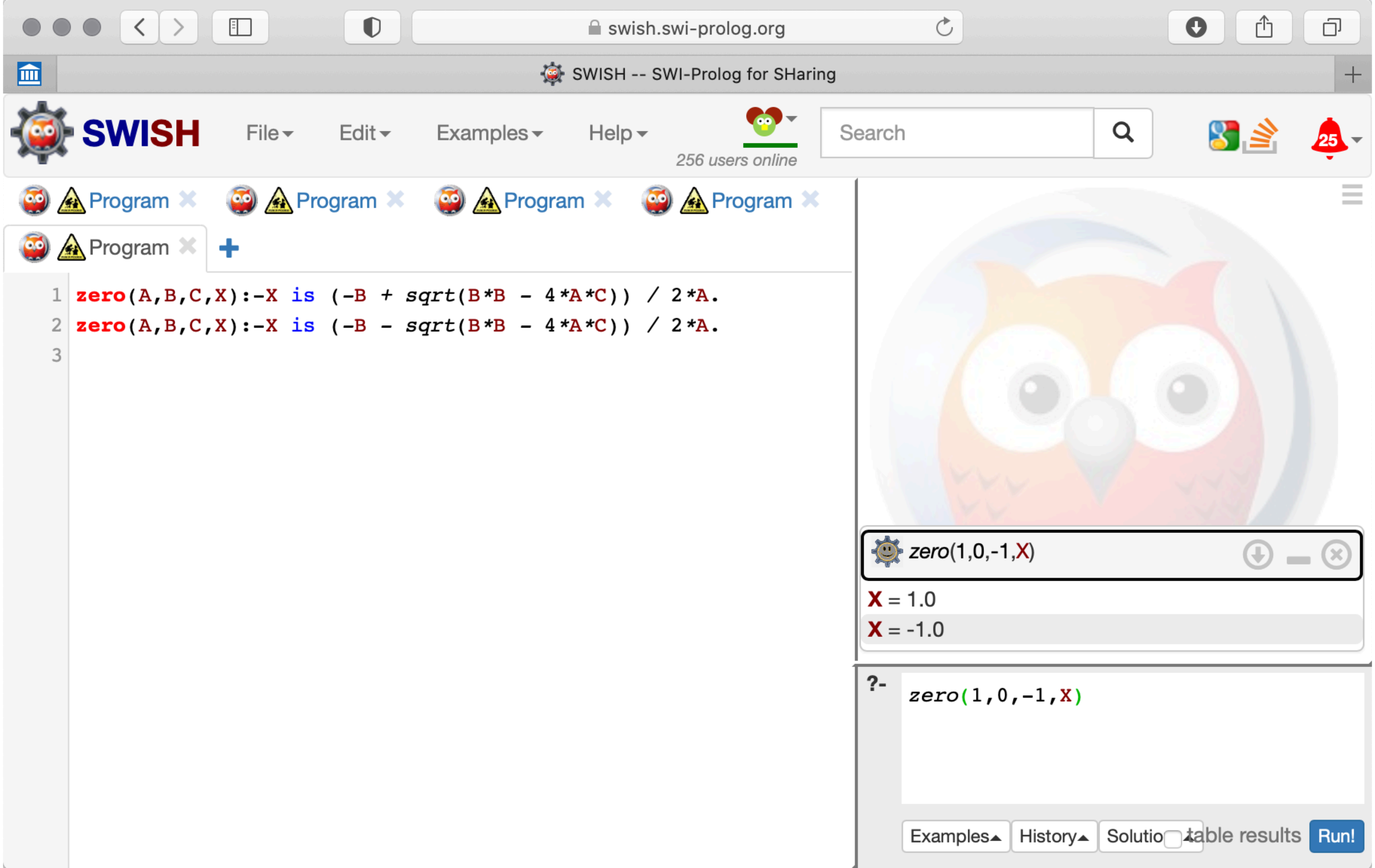
Prolog arithmetic vs.unification

```
?-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
SWISH
File
Edit
Examples
Help
256 users online
Search
25
Program
Program
Program
Program
Program
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.
zero(1,0,-1,X)
X = 1.0
X = -1.0
?-
zero(1,0,-1,X)
Examples
History
Solutio
table results
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).
```


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).
```

We have found a solution if Goal 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),
  goal(Goal).
search(Agenda,Goal):-
  next(Agenda,Current,Rest),
  children(Current,Children),
  add(Children,Rest,NewAgenda),
  search(NewAgenda,Goal).
```

We have found a solution if Goal is "next" on the agenda.

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),
  goal(Goal).
search(Agenda,Goal):-
  next(Agenda,Current,Rest),
  children(Current,Children),
  add(Children,Rest,NewAgenda),
  search(NewAgenda,Goal).
```

We have found a solution if Goal is "next" on the agenda.

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),
  goal(Goal).
search(Agenda,Goal):-
  next(Agenda,Current,Rest),
  children(Current,Children),
  add(Children,Rest,NewAgenda),
  search(NewAgenda,Goal).
```

We have found a solution if Goal is “next” on the agenda.

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`

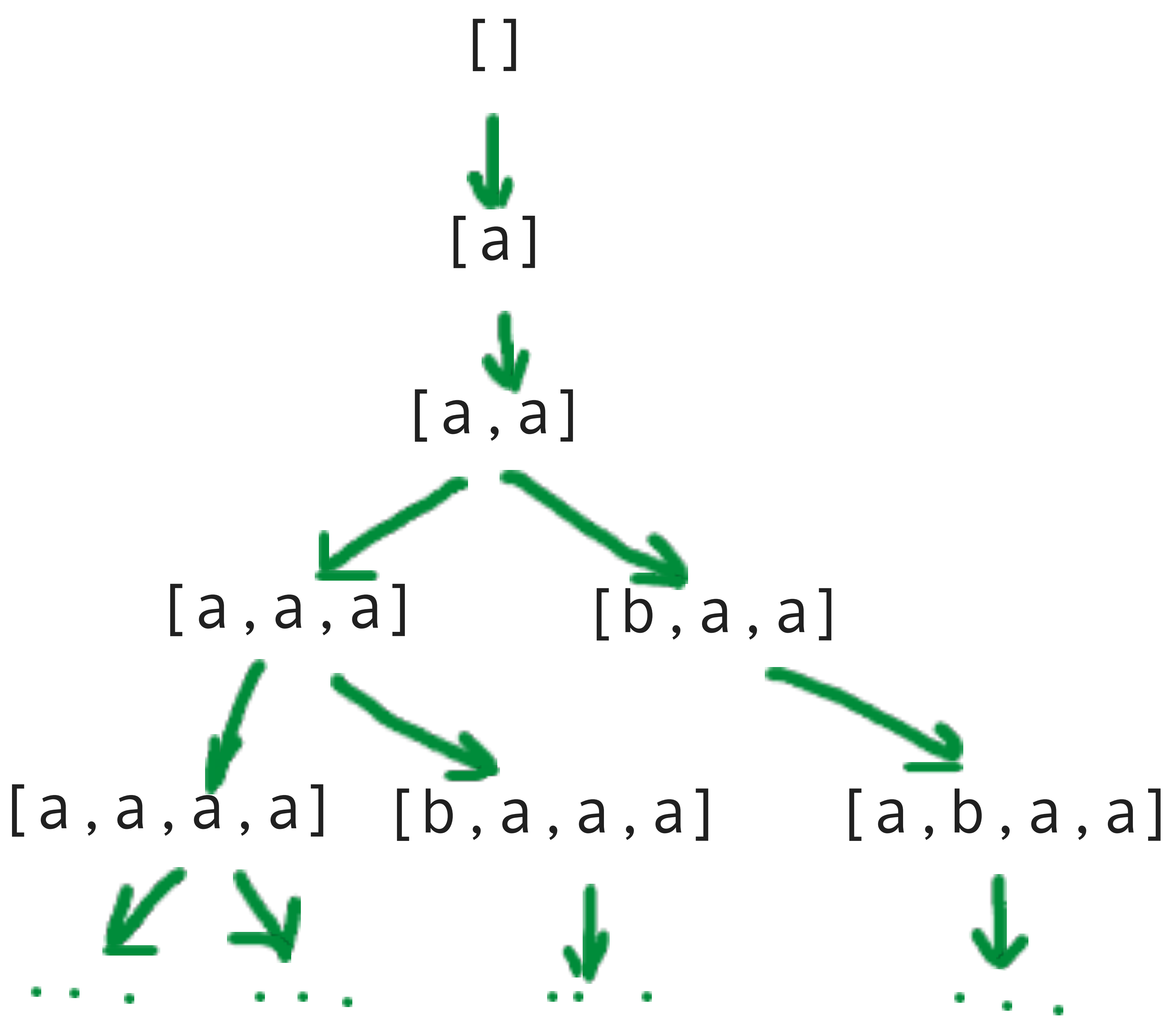
```
children(Node,Children):-
  findall(C,arc(Node,C),Children).
```

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

We will explain this in a moment

An Example – Let’s define “`arc`”

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



One way to represent `children`

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

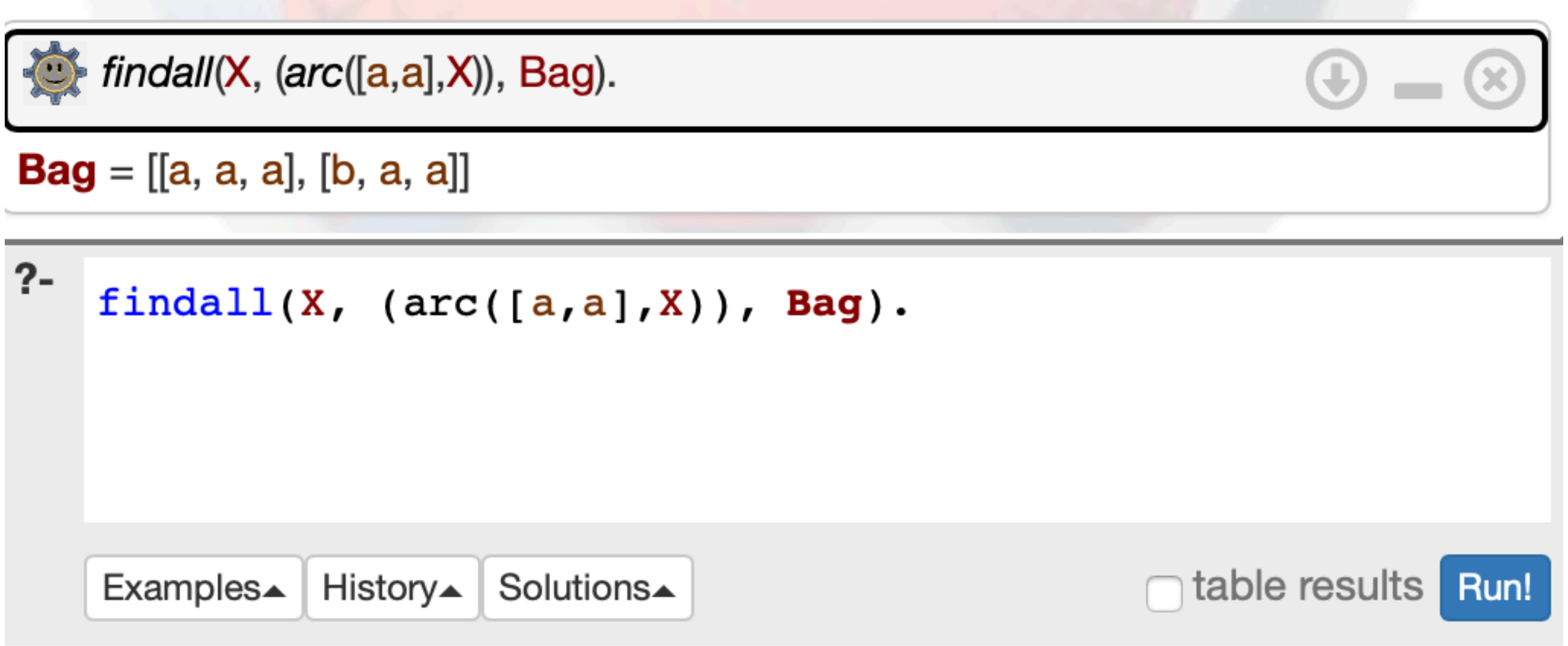
This is how we can represent the state-space in Prolog.

```
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]).
```



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_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).
```

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).
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).
```

1 **Add empty list of children**

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).
```

1 **Add empty list of children**

2. **Add nodes that have not been visited and not on the agenda**

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).
```

1 **Add empty list of children**

2. **Add nodes that have not been visited and not on the agenda**

3. **If already on agenda, ignore.**

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).
```

1. **Add empty list of children**
2. **Add nodes that have not been visited and not on the agenda**
3. **If already on agenda, ignore.**
4. **If already visited, ignore.**

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),
  search_bt(Child,Goal).
```

We used it to define `children`

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

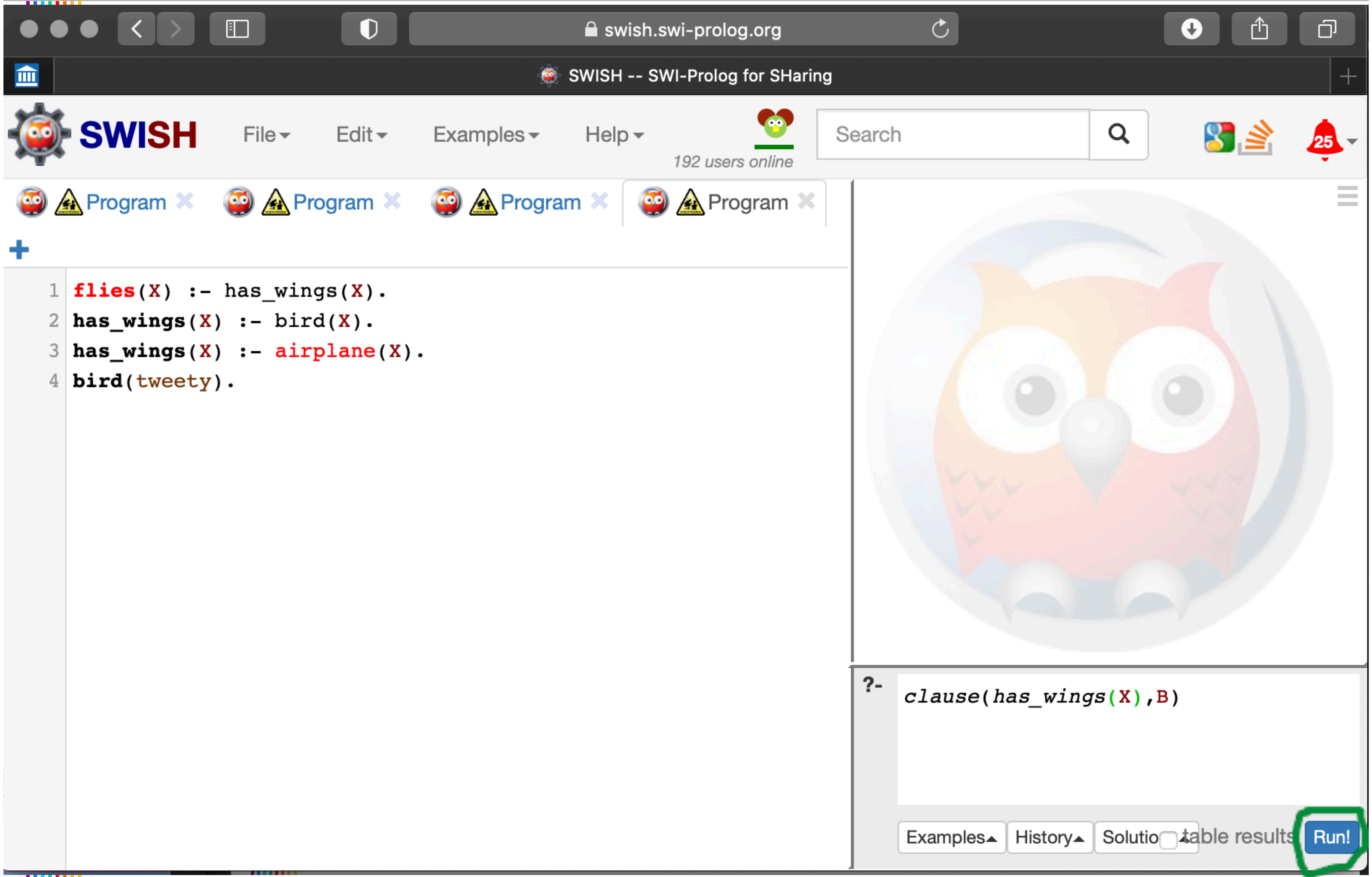
```
prove(true):-!.
```

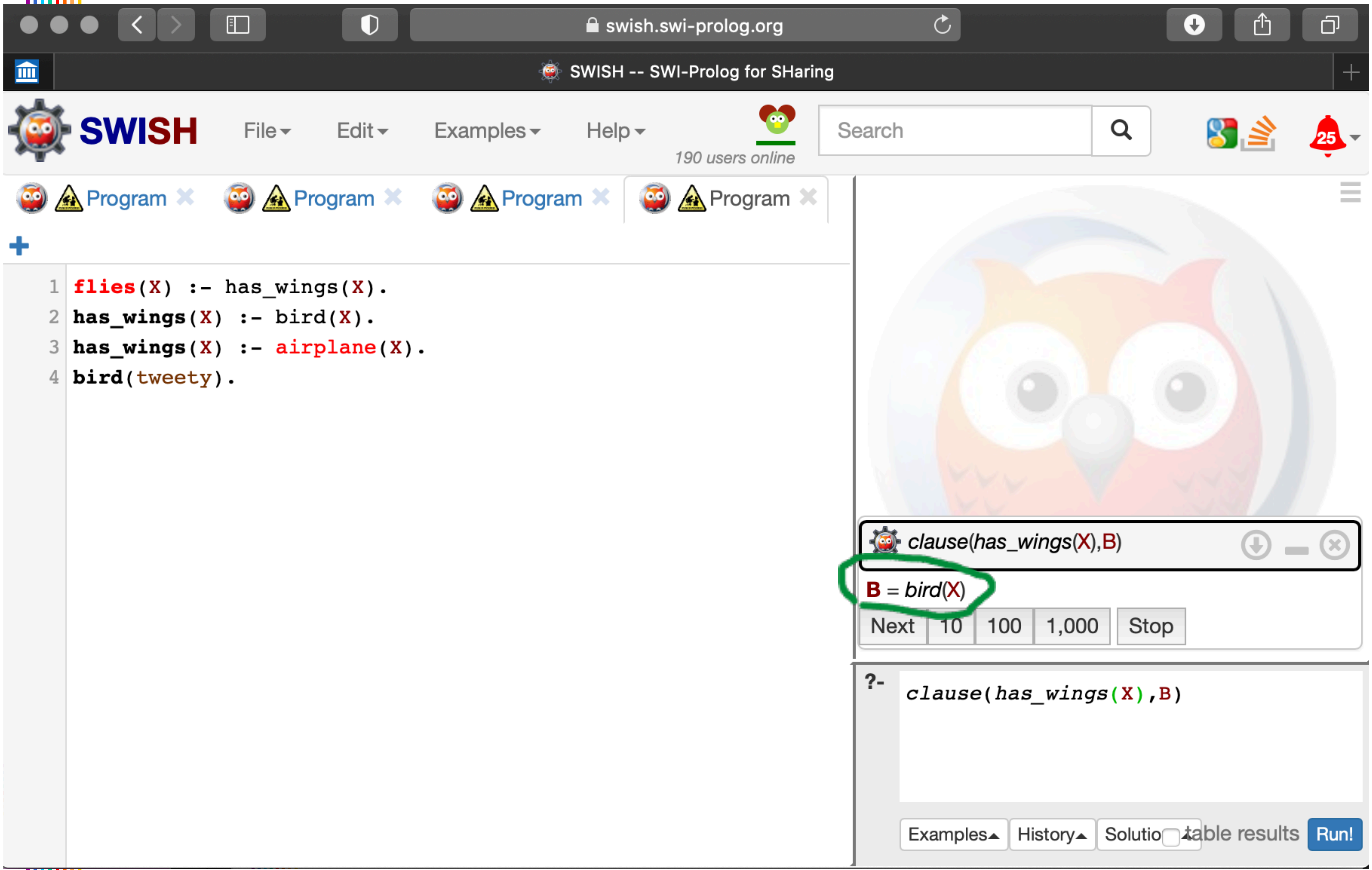

Agenda-based SLD-prover

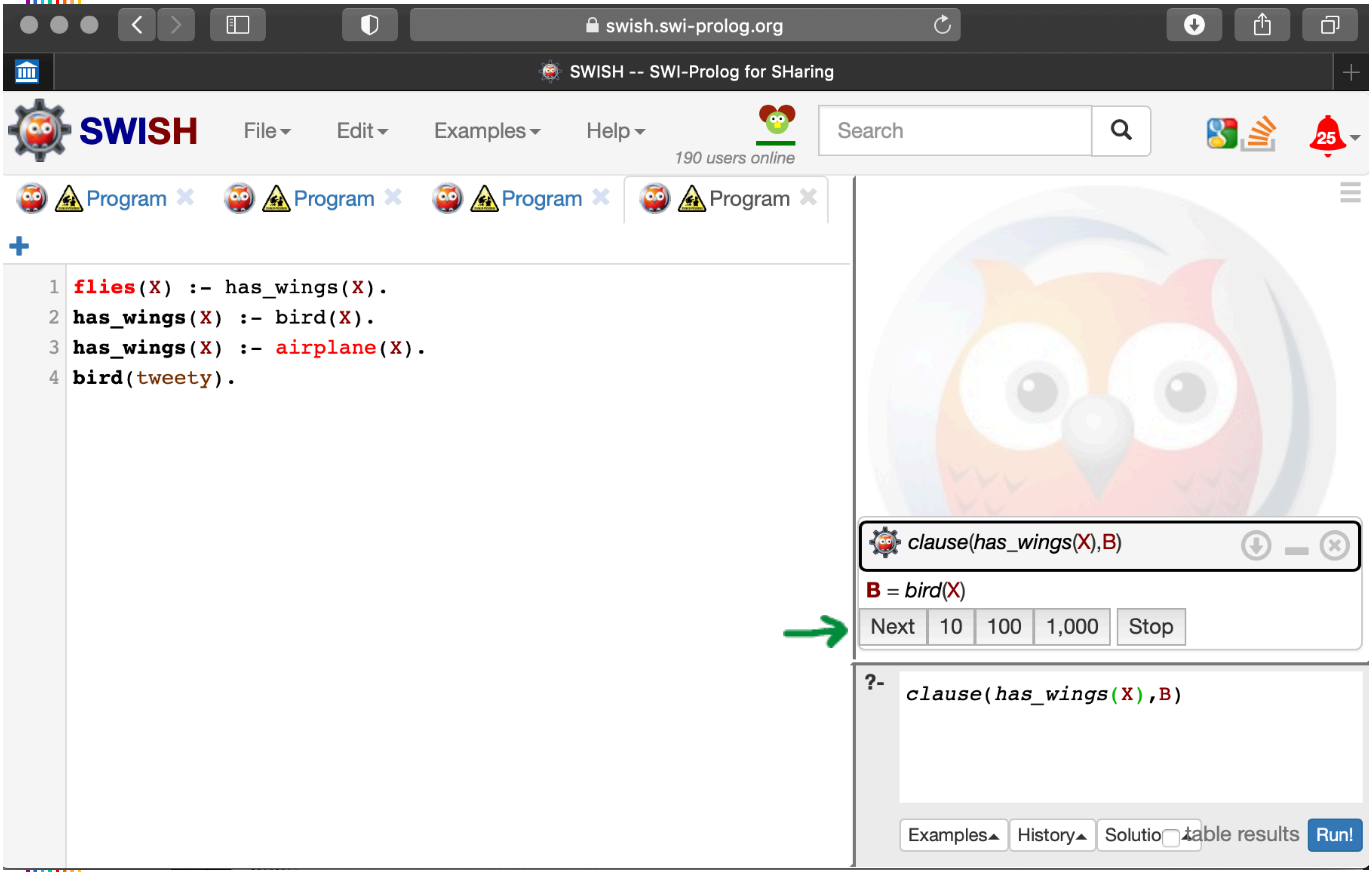
```
prove(true):-!.
prove((A,B)):-!,
    clause(A,C),
    conj_append(C,B,D),
    prove(D).
```

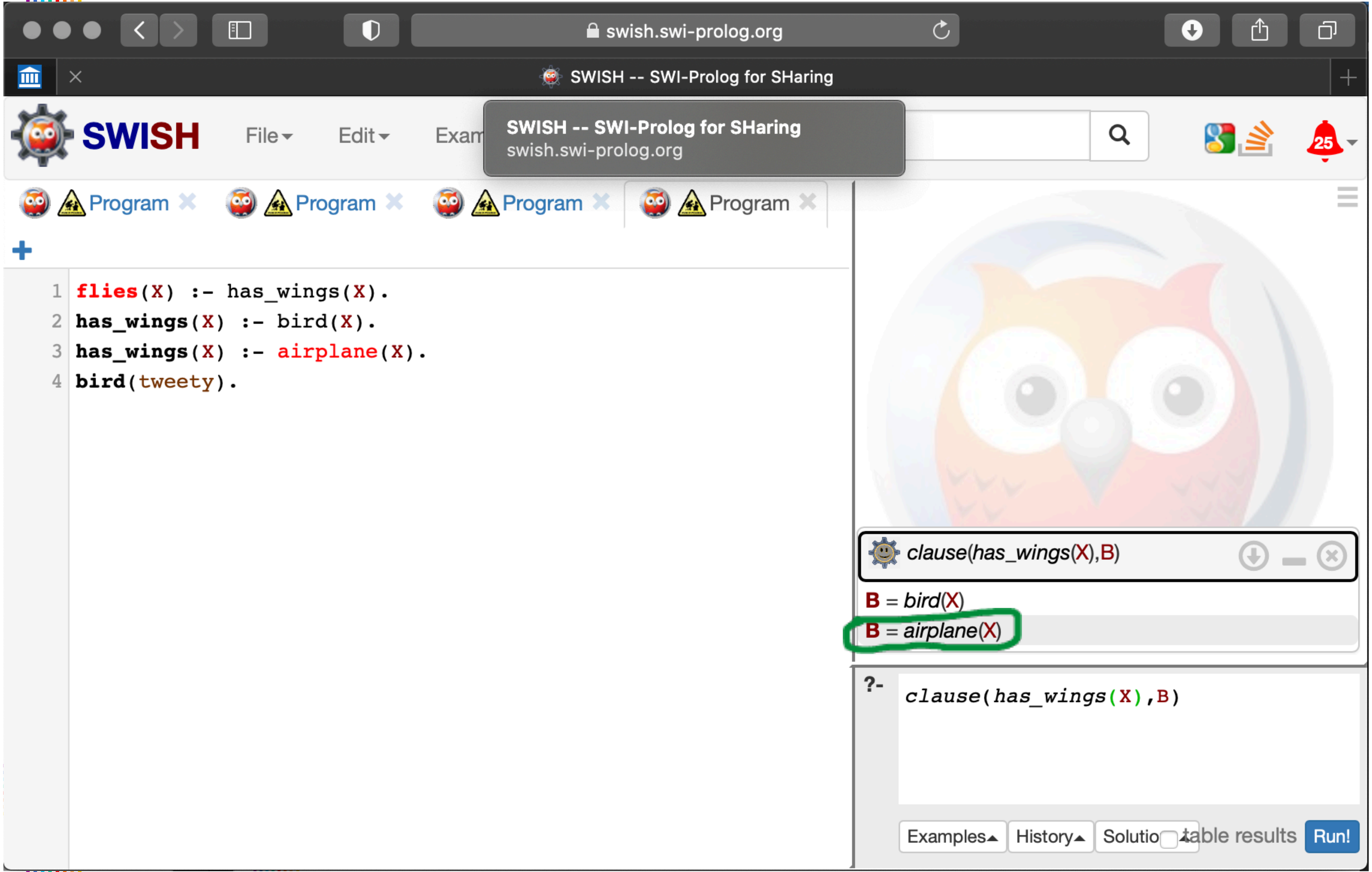
Built-in predicate.

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
SWISH
File
Edit
Examples
Help
192 users online
Search
Program
Program
Program
Program
flies(X) :- has_wings(X).
has_wings(X) :- bird(X).
has_wings(X) :- airplane(X).
bird(tweety).
?-
clause(has_wings(X),B)
Examples
History
Solutio
table results
Run!


swish.swi-prolog.org
SWISH -- SWI-Prolog for SHaring
SWISH
File
Edit
Examples
Help
190 users online
Search
25
Program
Program
Program
Program
flies(X) :- has_wings(X).
has_wings(X) :- bird(X).
has_wings(X) :- airplane(X).
bird(tweety).
clause(has_wings(X),B)
B = bird(X)
Next
10
100
1,000
Stop
?-
clause(has_wings(X),B)
Examples
History
Solutio
table results
Run!


swish.swi-prolog.org
SWISH -- SWI-Prolog for SHaring
SWISH
File
Edit
Examples
Help
190 users online
Search
Program
Program
Program
Program
flies(X) :- has_wings(X).
has_wings(X) :- bird(X).
has_wings(X) :- airplane(X).
bird(tweety).
clause(has_wings(X),B)
B = bird(X)
Next
10
100
1,000
Stop
?-
clause(has_wings(X),B)
Examples
History
Solutio
table results
Run!


swish.swi-prolog.org
SWISH -- SWI-Prolog for SHaring
SWISH
File
Edit
Exam
SWISH -- SWI-Prolog for SHaring
swish.swi-prolog.org
25
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)
B = bird(X)
B = airplane(X)
?-
clause(has_wings(X),B)
Examples
History
Solutio
table results
Run!

Agenda-based SLD-prover

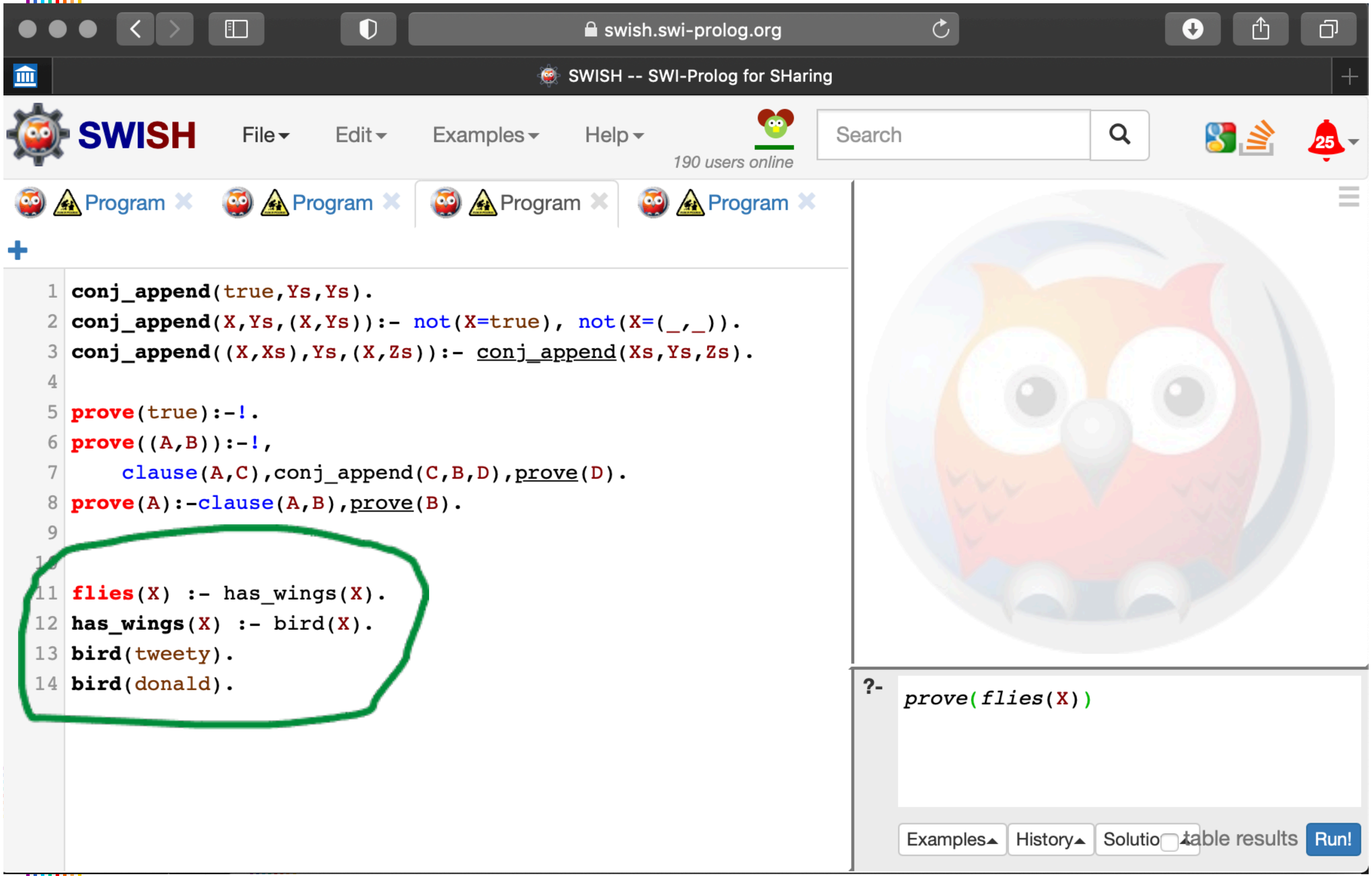
```
prove(true):-!.
prove((A,B)):-!,
    clause(A,C),
    conj_append(C,B,D),
    prove(D).
```

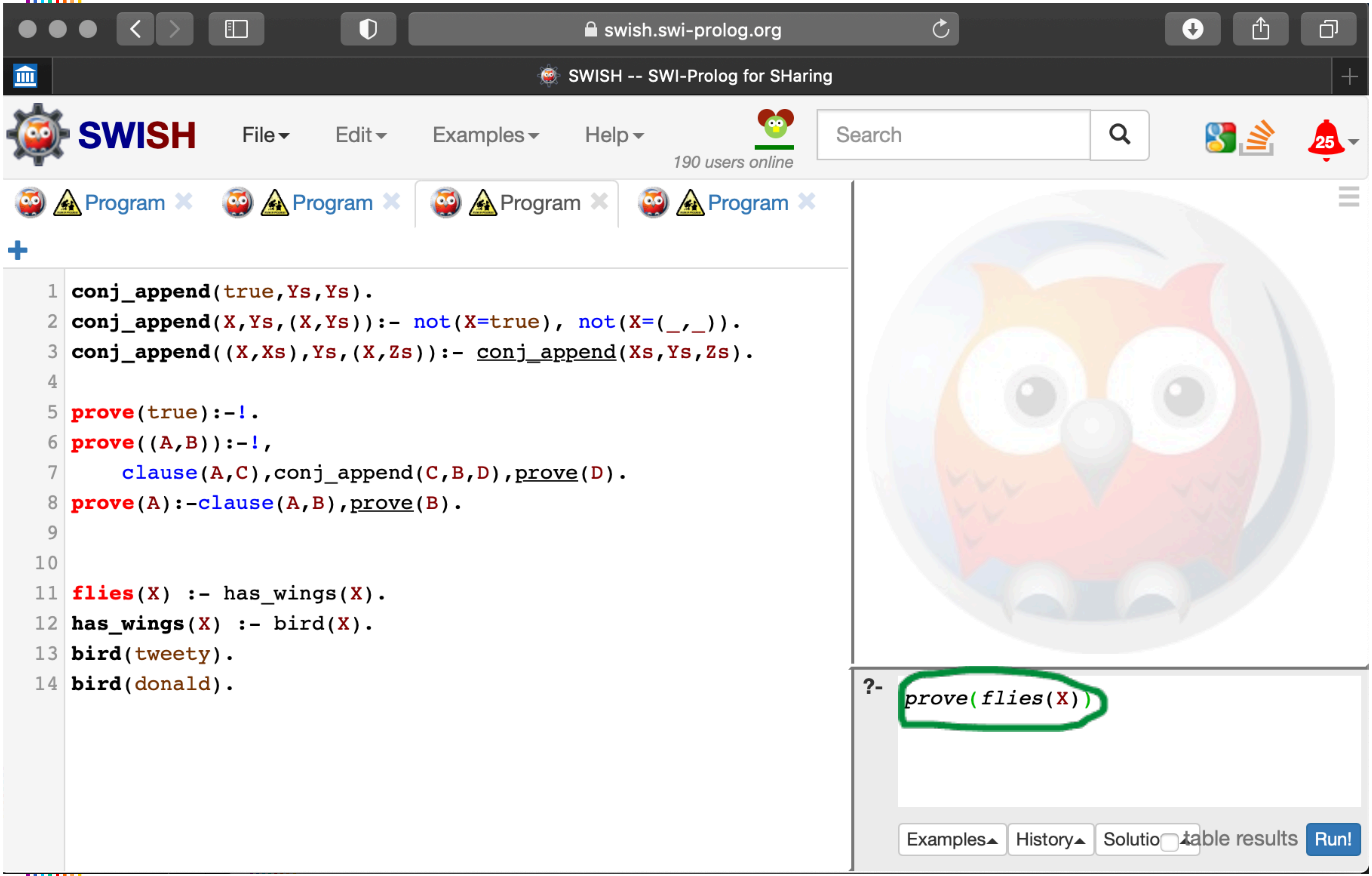
Auxiliary predicate `conj_append`:

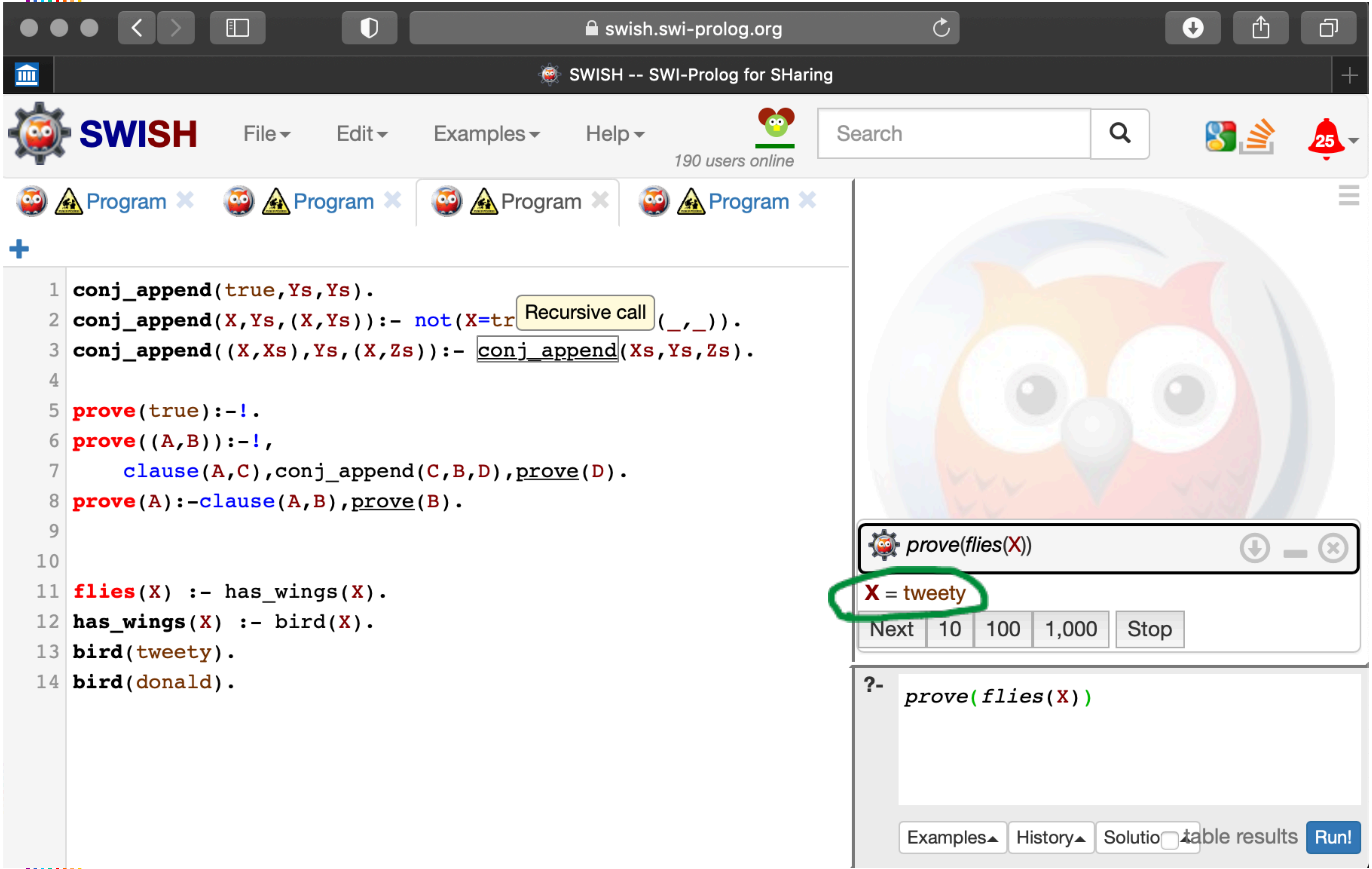
```
conj_append(true,Ys,Ys).
conj_append(X,Ys,(X,Ys)):-not(X=true), not(X=(_,_)).
conj_append((X,Xs),Ys,(X,Zs)):- conj_append(Xs,Ys,Zs).
```

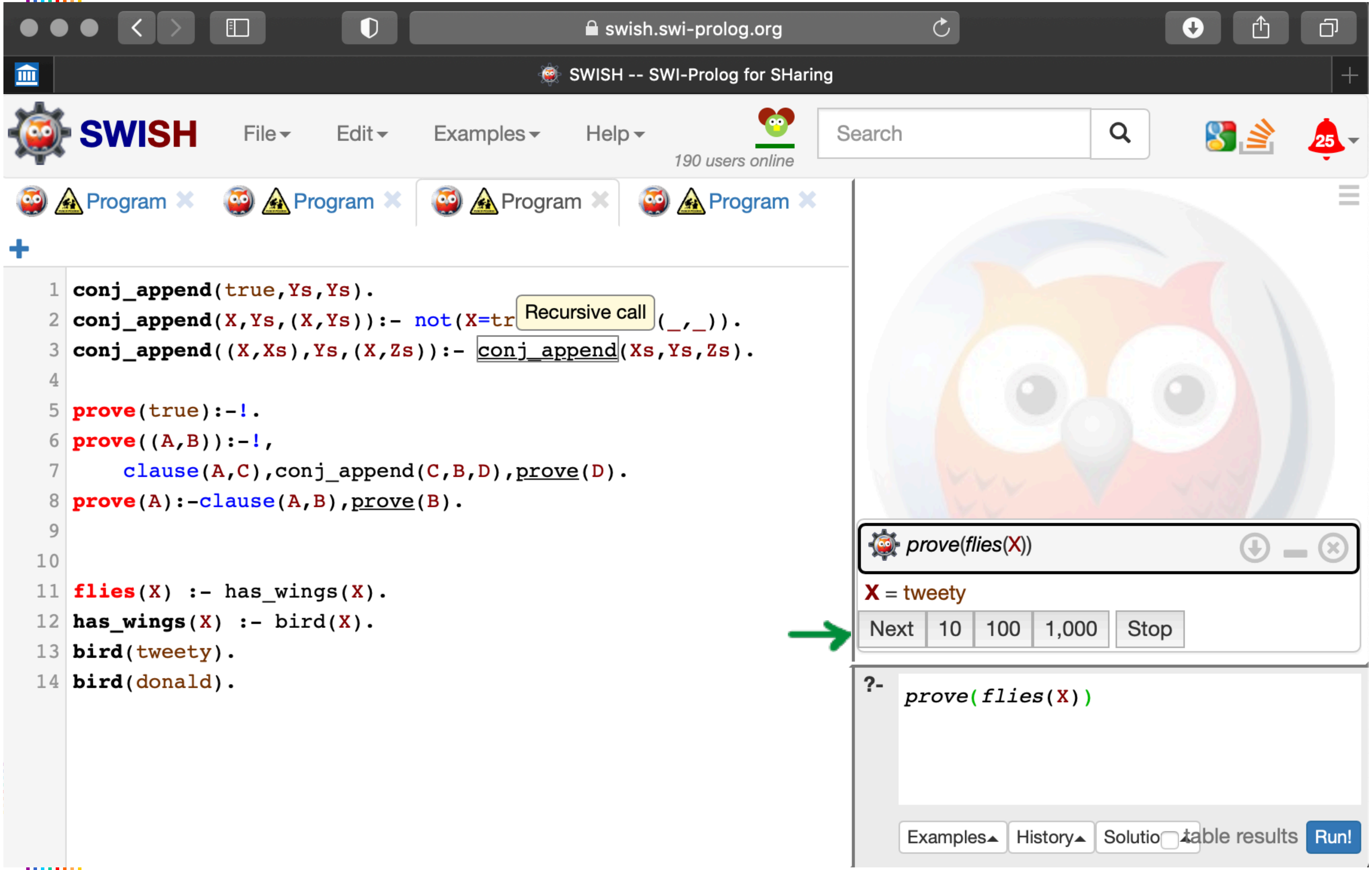

Agenda-based SLD-prover

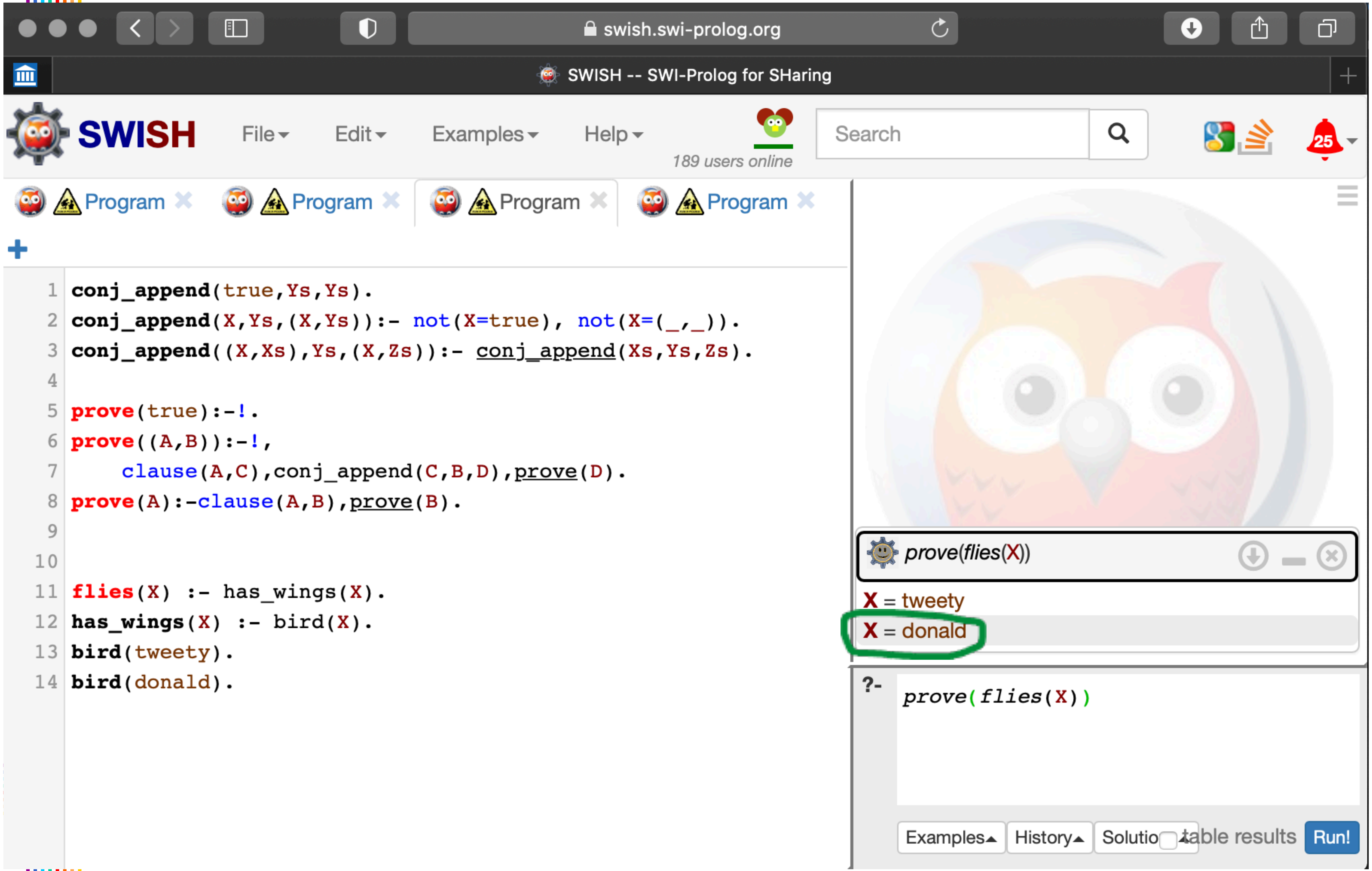
```
prove(true):-!.
prove((A,B)):-!,
  clause(A,C),
  conj_append(C,B,D),
  prove(D).
prove(A):-
  clause(A,B),
  prove(B).
```


swish.swi-prolog.org
SWISH -- SWI-Prolog for SHaring
SWISH
File
Edit
Examples
Help
190 users online
Search
Program
Program
Program
Program
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).
prove(true):-!.
prove((A,B)):-!,
clause(A,C),conj_append(C,B,D),prove(D).
prove(A):-clause(A,B),prove(B).
flies(X) :- has_wings(X).
has_wings(X) :- bird(X).
bird(tweety).
bird(donald).
?-
prove(flies(X))
Examples
History
Solutio
table results
Run!


swish.swi-prolog.org
SWISH -- SWI-Prolog for SHaring
SWISH
File
Edit
Examples
Help
190 users online
Search
25
Program
Program
Program
Program
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).
prove(true):-!.
prove((A,B)):-!,
clause(A,C),conj_append(C,B,D),prove(D).
prove(A):-clause(A,B),prove(B).
flies(X) :- has_wings(X).
has_wings(X) :- bird(X).
bird(tweety).
bird(donald).
?- prove(flies(X))
Examples
History
Solutio
table results
Run!


swish.swi-prolog.org
SWISH -- SWI-Prolog for SHaring
SWISH
File
Edit
Examples
Help
190 users online
Search
Program
Program
Program
Program
conj_append(true,Ys,Ys).
conj_append(X,Ys,(X,Ys)):- not(X=tr
Recursive call
(_,_)).
conj_append((X,Xs),Ys,(X,Zs)):- conj_append(Xs,Ys,Zs).
prove(true):-!.
prove((A,B)):-!,
clause(A,C),conj_append(C,B,D),prove(D).
prove(A):-clause(A,B),prove(B).
flies(X) :- has_wings(X).
has_wings(X) :- bird(X).
bird(tweety).
bird(donald).
prove(flies(X))
X = tweety
Next
10
100
1,000
Stop
?-
prove(flies(X))
Examples
History
Solutio
table results
Run!


swish.swi-prolog.org
SWISH -- SWI-Prolog for SHaring
SWISH
File
Edit
Examples
Help
190 users online
Search
Program
Program
Program
Program
conj_append(true,Ys,Ys).
conj_append(X,Ys,(X,Ys)):- not(X=tr(_,_)).
Recursive call
conj_append((X,Xs),Ys,(X,Zs)):- conj_append(Xs,Ys,Zs).
prove(true):-!.
prove((A,B)):-!,
clause(A,C),conj_append(C,B,D),prove(D).
prove(A):-clause(A,B),prove(B).
flies(X) :- has_wings(X).
has_wings(X) :- bird(X).
bird(tweety).
bird(donald).
prove(flies(X))
X = tweety
Next
10
100
1,000
Stop
?-
prove(flies(X))
Examples
History
Solutio
table results
Run!


swish.swi-prolog.org
SWISH -- SWI-Prolog for SHaring
SWISH
File
Edit
Examples
Help
189 users online
Search
Program
Program
Program
Program
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).
prove(true):-!.
prove((A,B)):-!,
clause(A,C),conj_append(C,B,D),prove(D).
prove(A):-clause(A,B),prove(B).
flies(X) :- has_wings(X).
has_wings(X) :- bird(X).
bird(tweety).
bird(donald).
prove(flies(X))
X = tweety
X = donald
?-
prove(flies(X))
Examples
History
Solutio
table results
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

```
refute((false:-true)).
refute((A,C)):-
    cl(Cl),
    resolve(A,Cl,R),
    refute(R).
```

(Not shown here)

```
% 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).
```