Search

Adapted from slides by Peter Flach


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

Agenda = Goal ∪ 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),
  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

Agenda-based search

Intermission

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

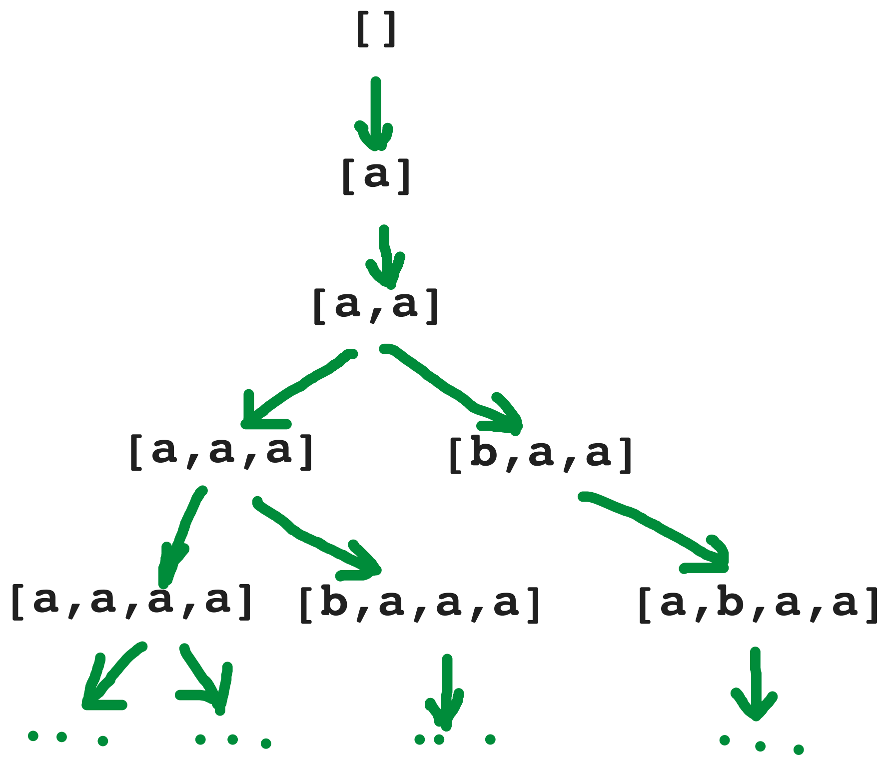
```
children(Node,Children):-
  findall(C,arc(Node,C),Children).
```

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

We will explain this in a moment

One way to represent `children`

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



An Example – Let's define "arc"

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

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

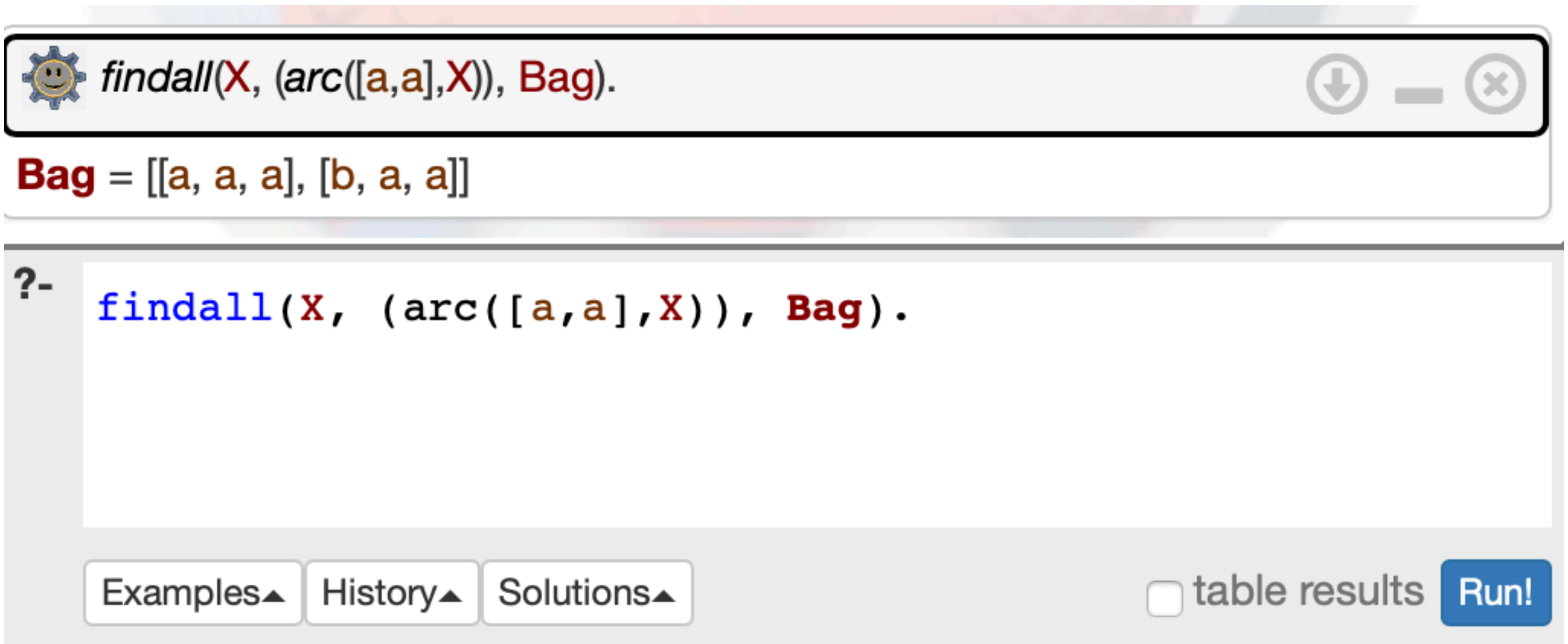
`findall: findall(Template, Goal, Bag)`

E.g.:

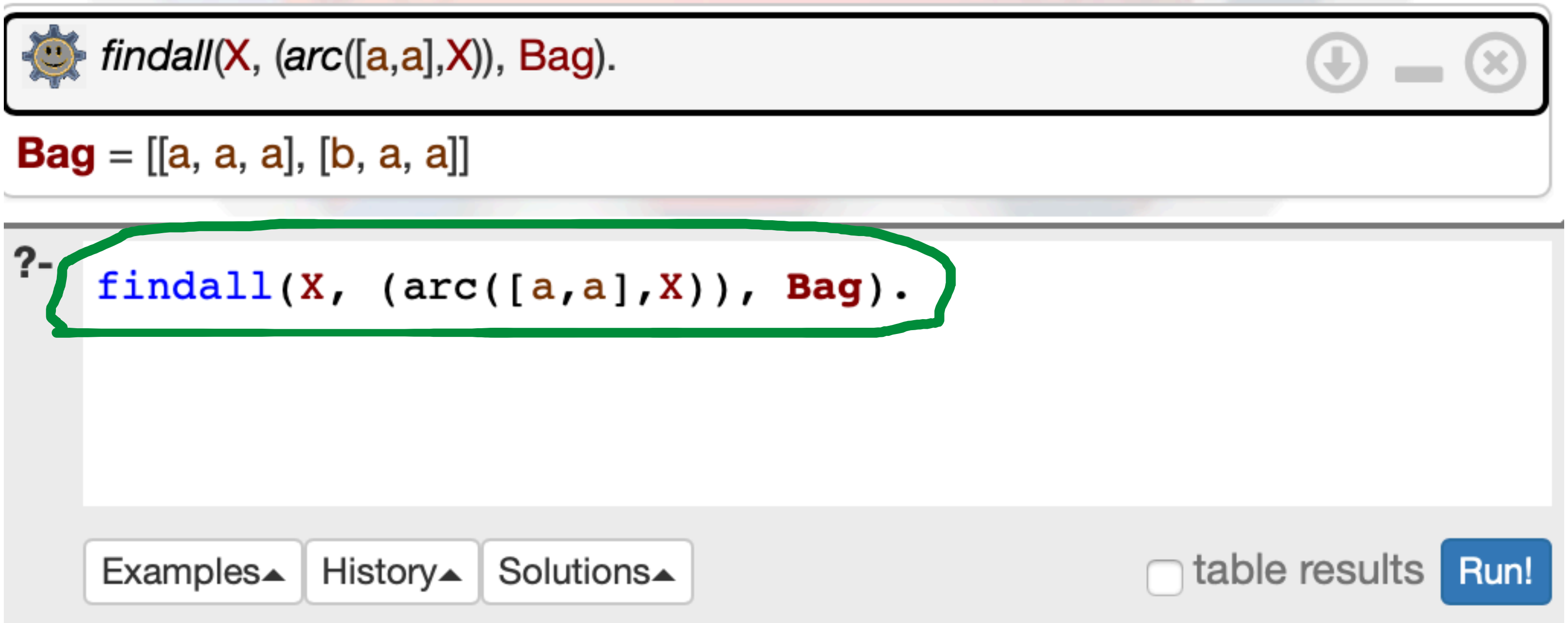
```
?- findall(X, (arc([a,a],X)), Bag).
```

One way to represent `children`

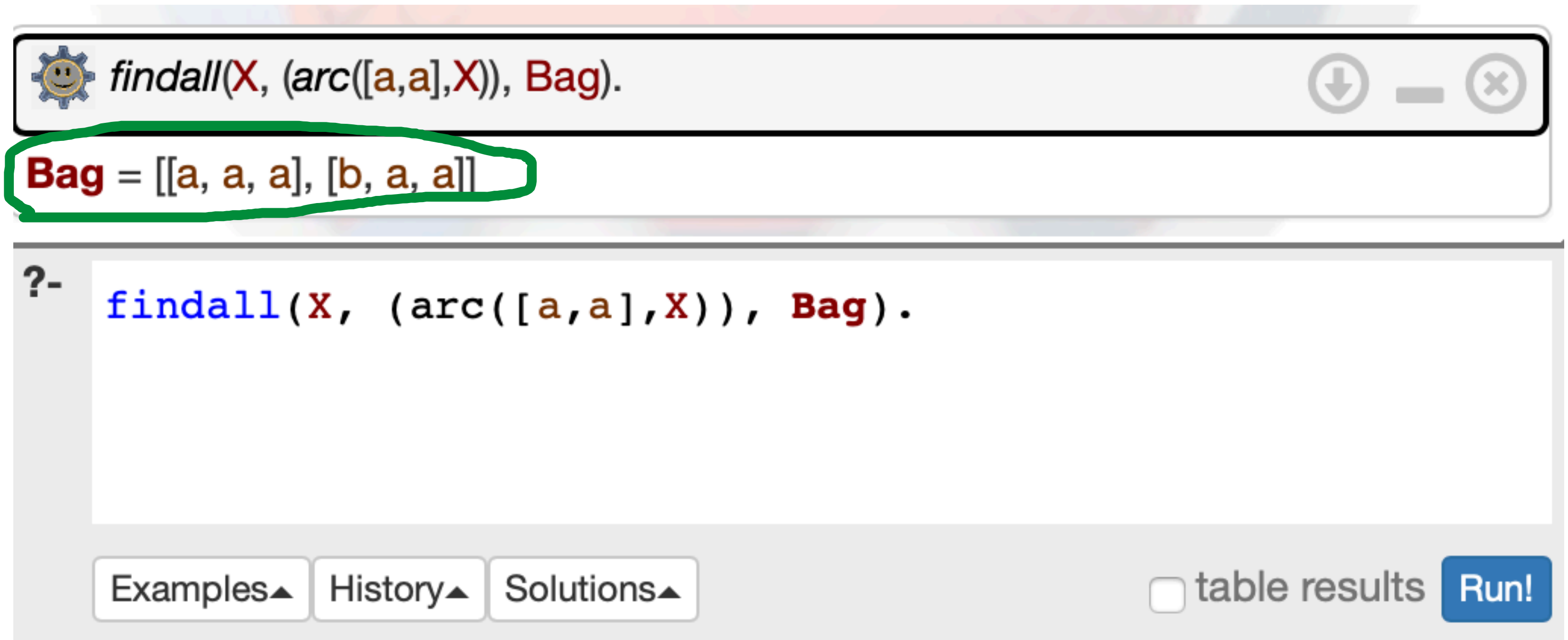
```
arc(L,[a|L]).
arc([a,a|T],[b,a,a|T]).
```



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




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



Back to 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).
```

AGENDA

Depth-first vs. breadth-first 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).
```

AGENDA

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

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



```
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

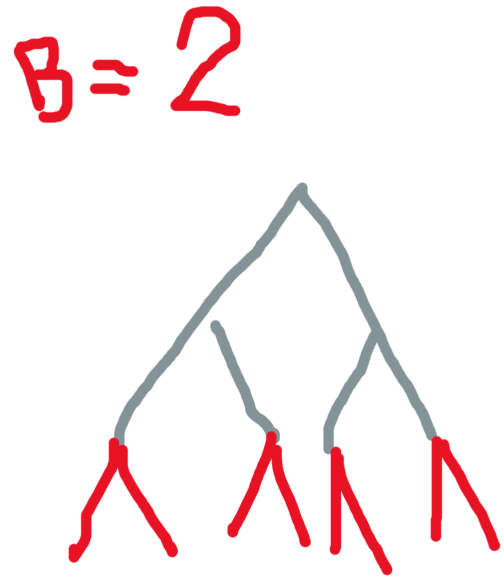
- agenda = stack (last-in first-out)
- incomplete: may get trapped in infinite branch
- no shortest-path property
- requires $O(B \times n)$ memory

Depth-first vs. breadth-first search

Depth-first vs. breadth-first search

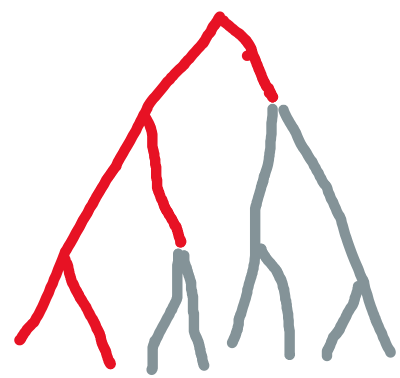
Breadth-first search

- 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 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).
```

THIS IS NEW

```
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).
```

THIS IS NEW

```
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).
```

Instead of `append`

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.

Loop detection


```
% 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` (annotation pointing to `arc(Current,Child)`)

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 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).
```

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 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).
```

Backtracking search


```
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)

Iterative deepening

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

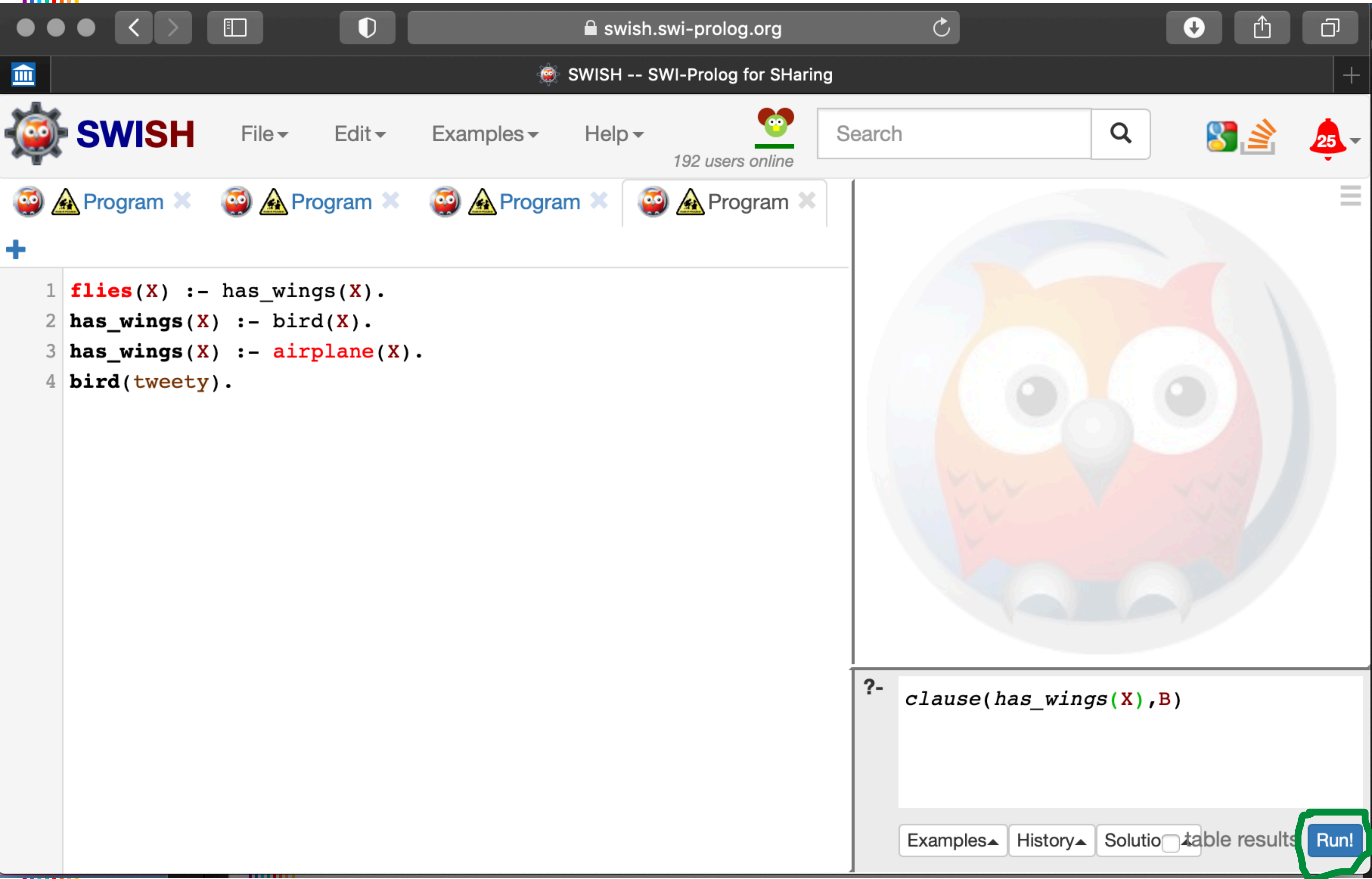
Agenda-based SLD-prover

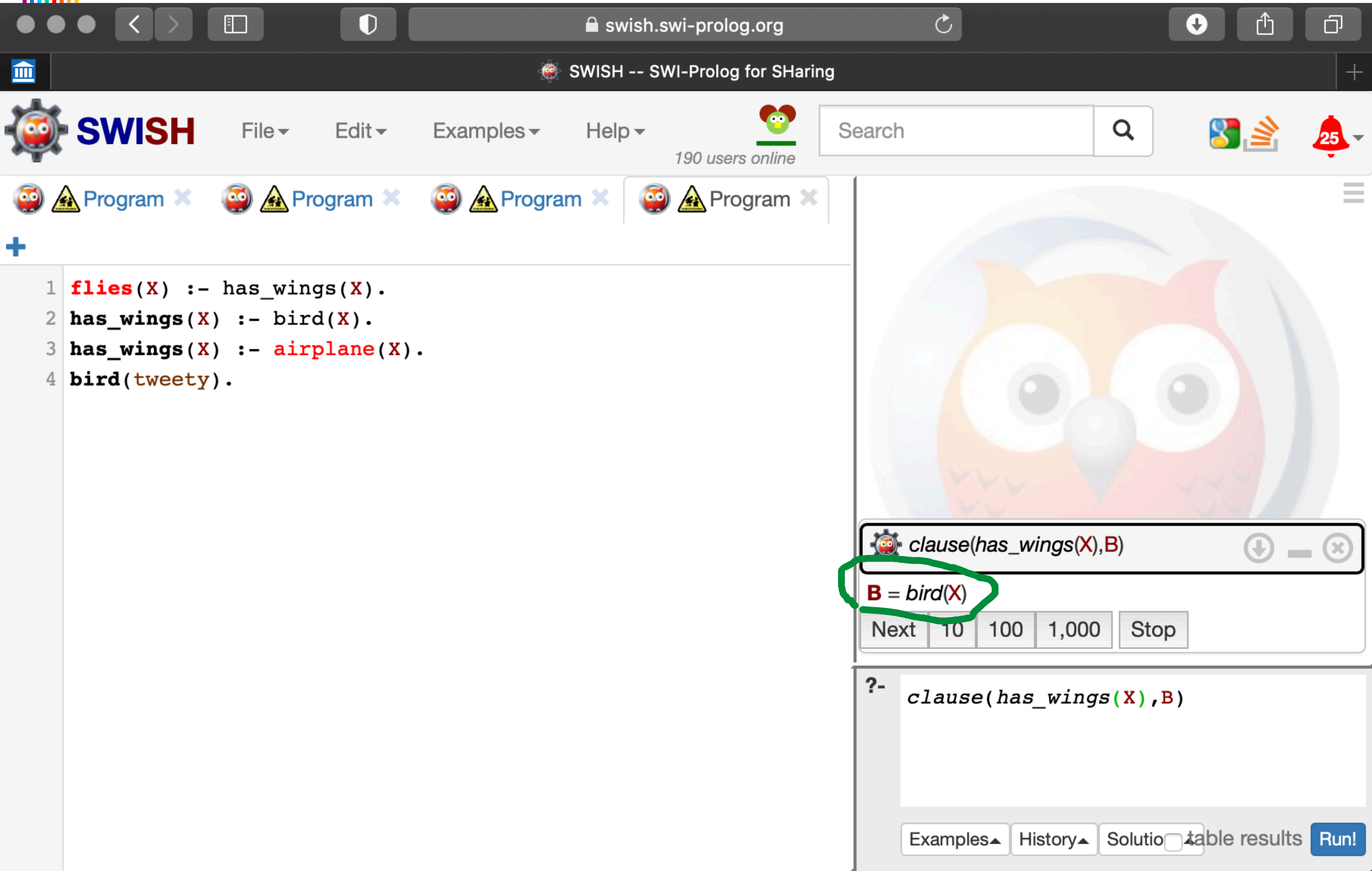

```
prove(true):-!.
prove((A,B)):-!,
    clause(A,C),
    conj_append(C,B,D),
    prove(D).
```

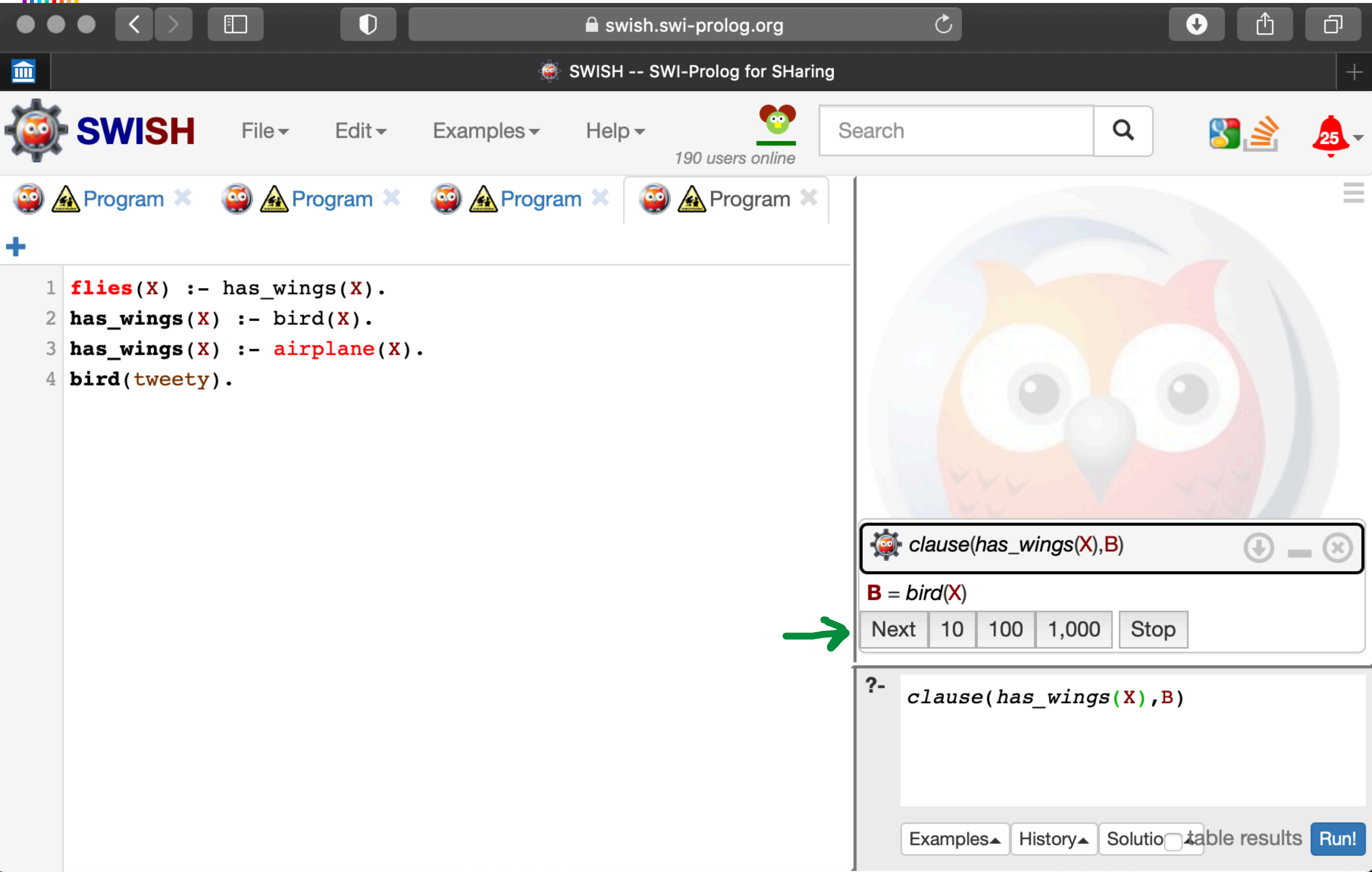
Built-in predicate.

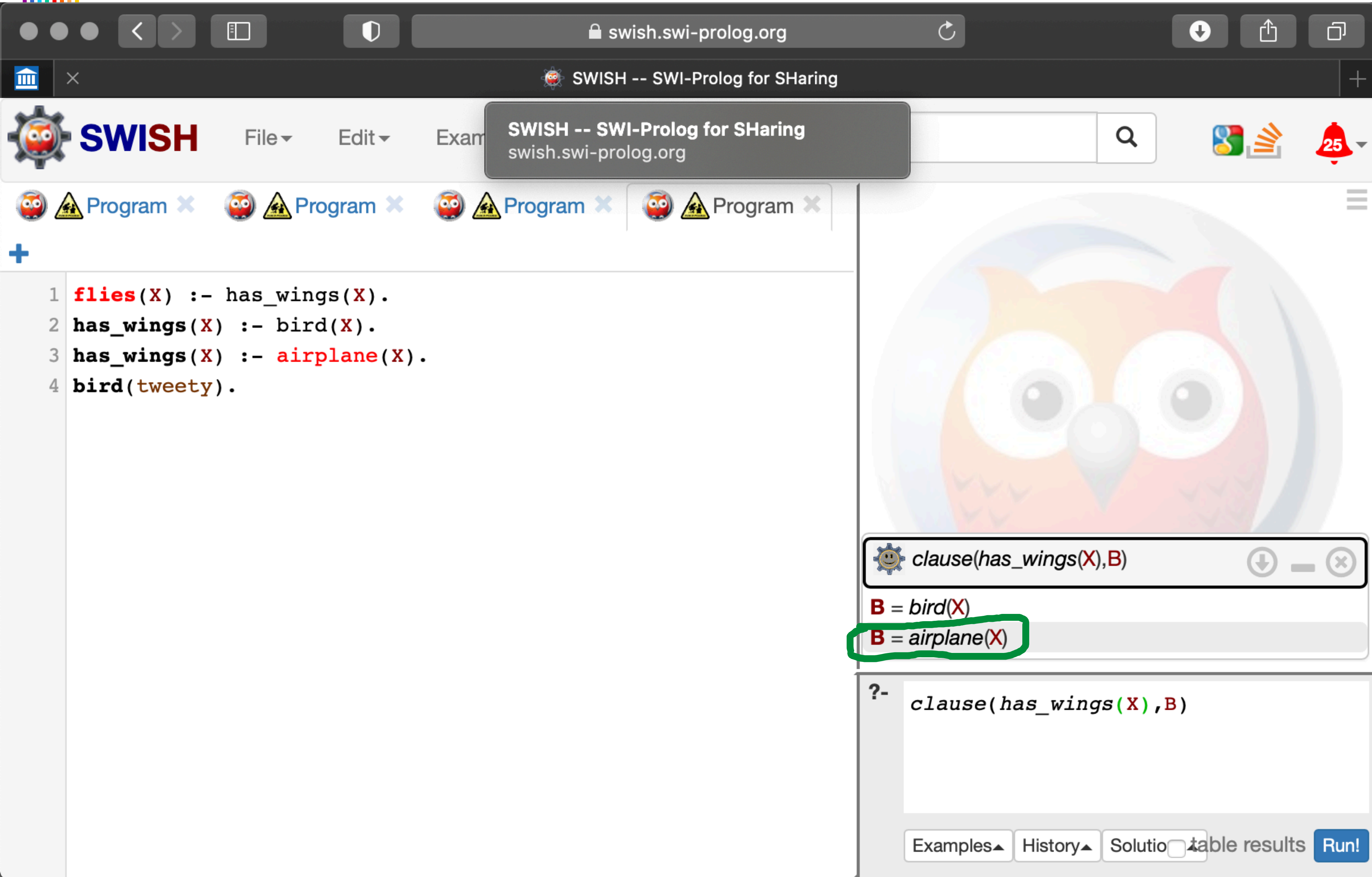
From SWI Prolog documentation: *True if* `A` *can be unified with a clause head and* `B` *with the corresponding clause body. Gives alternative clauses on backtracking. For facts,* `B` *is unified with the atom true.*

Agenda-based SLD-prover


swish.swi-prolog.org
SWISH -- SWI-Prolog for SHaring
SWISH
File
Edit
Examples
Help
192 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)
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
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
SWISH -- SWI-Prolog for SHaring
swish.swi-prolog.org
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)
B = airplane(X)
?- clause(has_wings(X),B)
Examples
History
Run!


```
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).
```

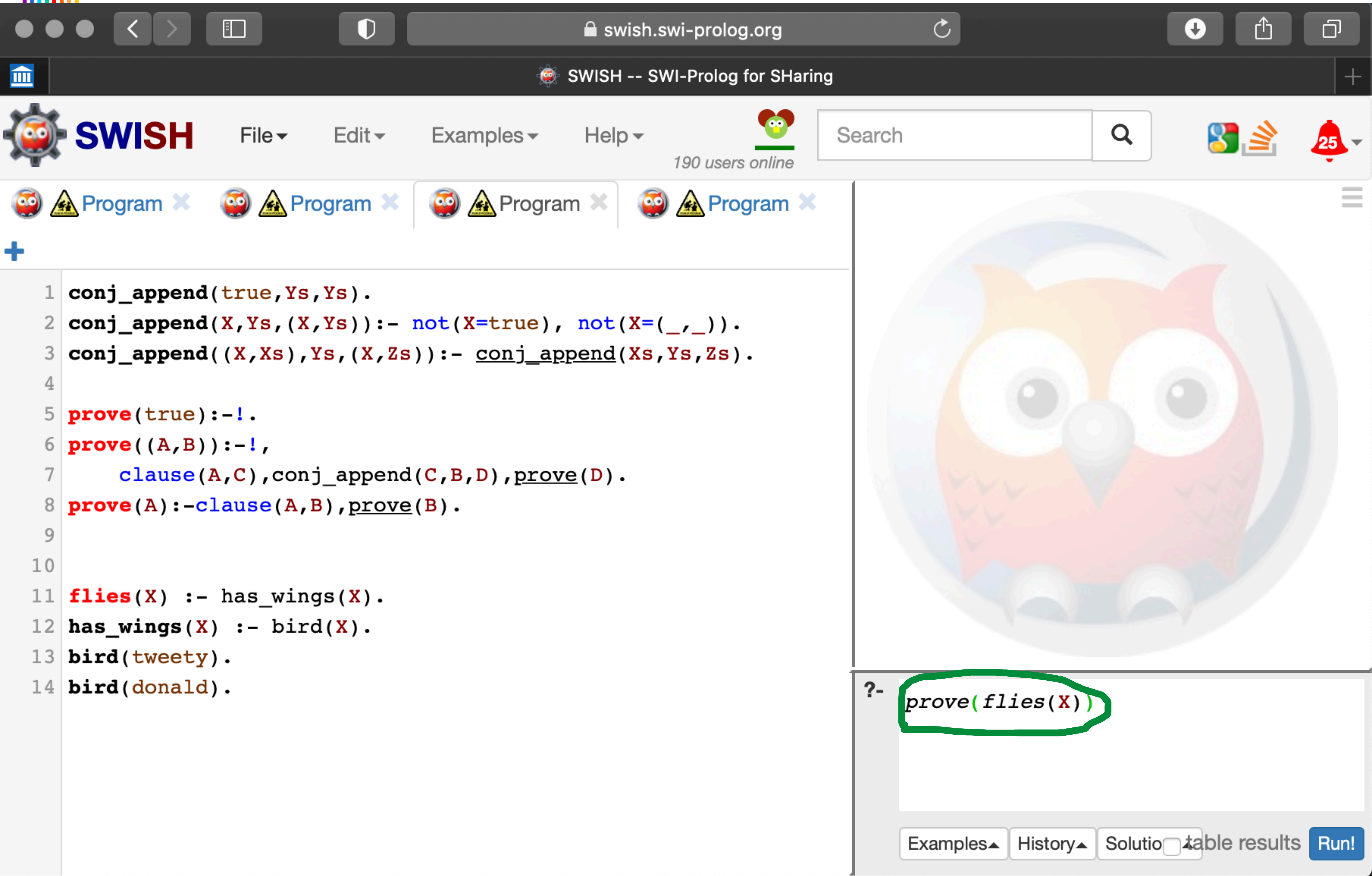
Agenda-based SLD-prover

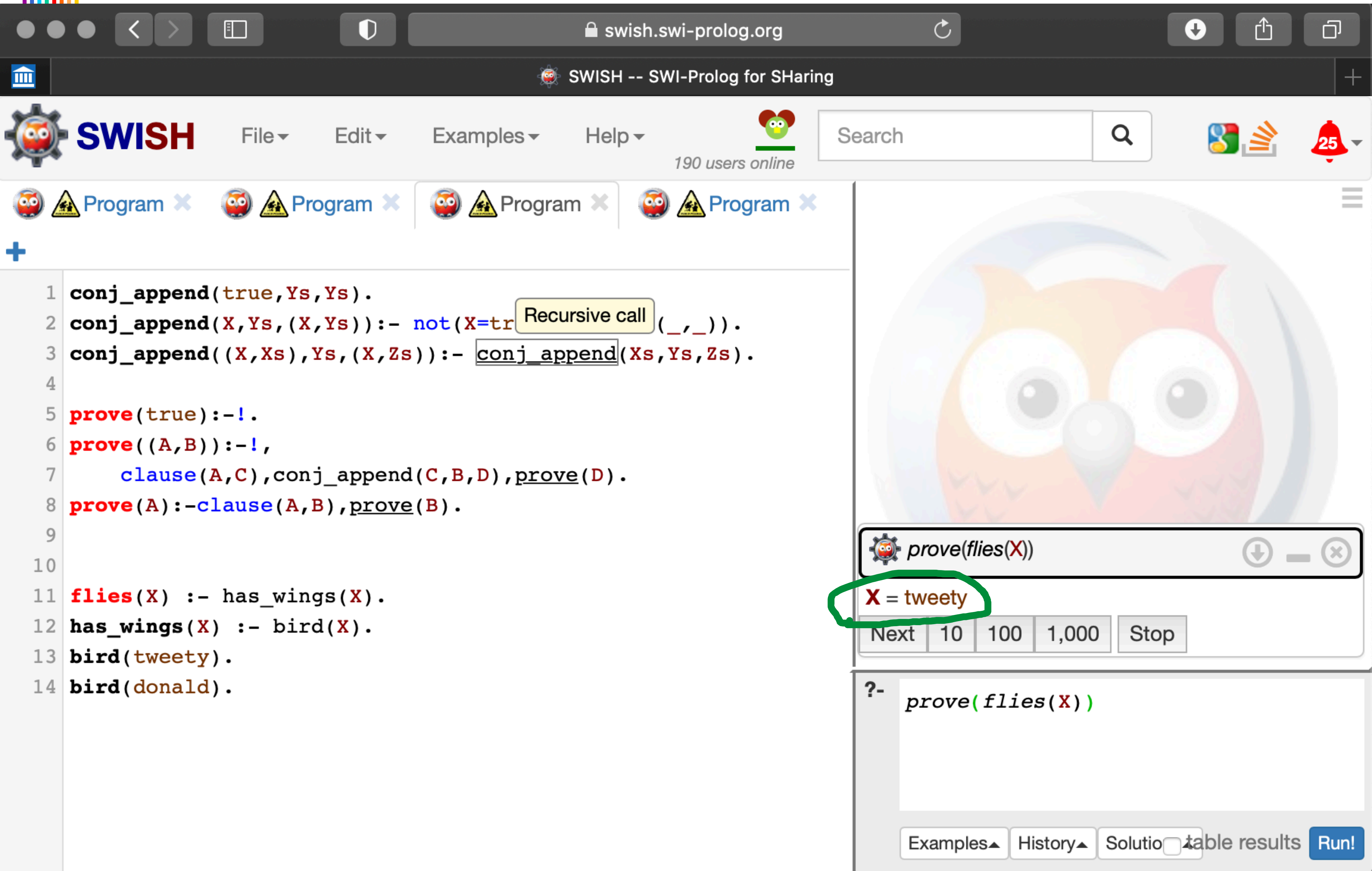

```
prove(true):-!.
prove((A,B)):-!,
    clause(A,C),
    conj_append(C,B,D),
    prove(D).
prove(A):-
    clause(A,B),
    prove(B).
```

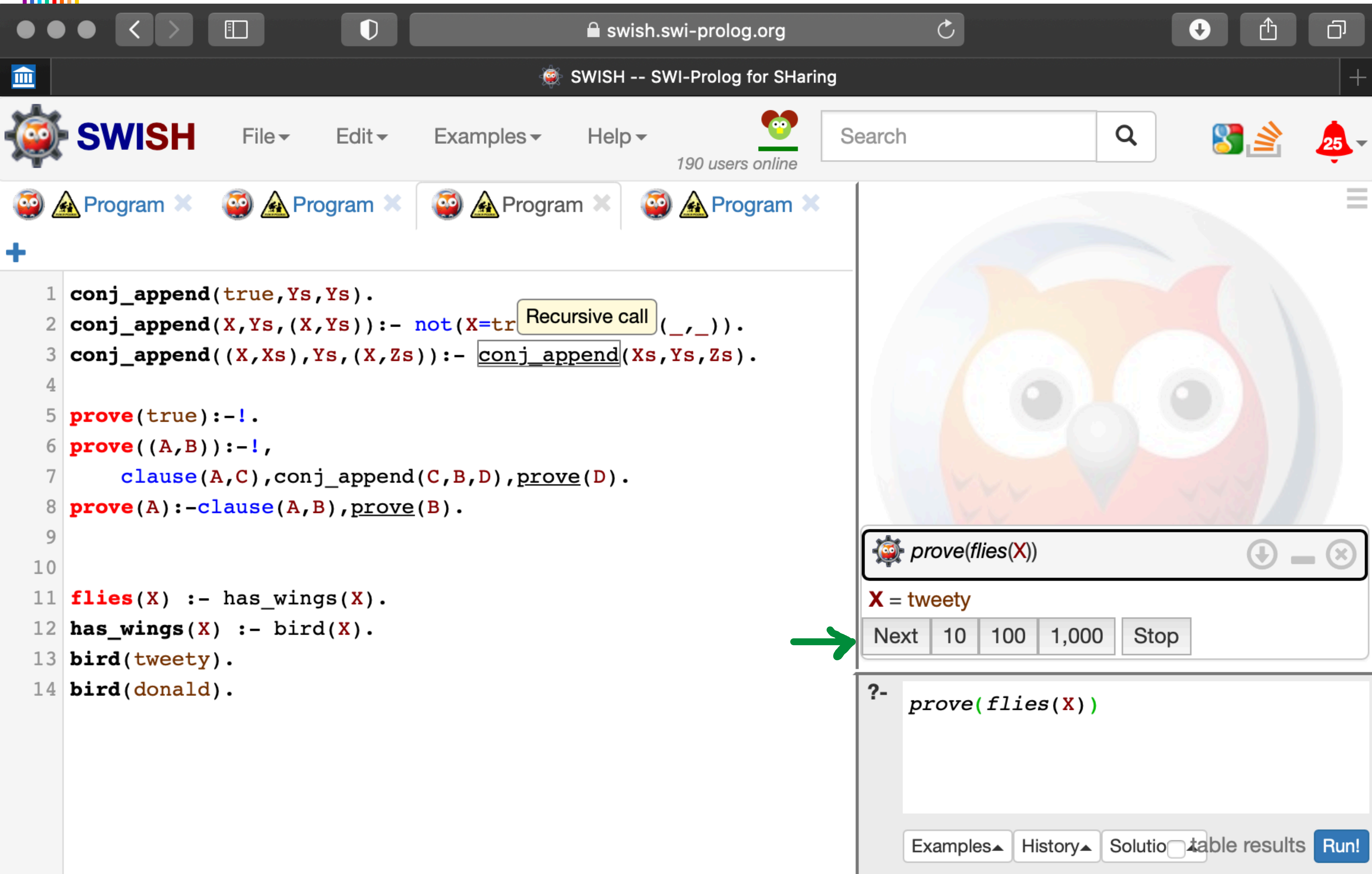
Agenda-based SLD-prover

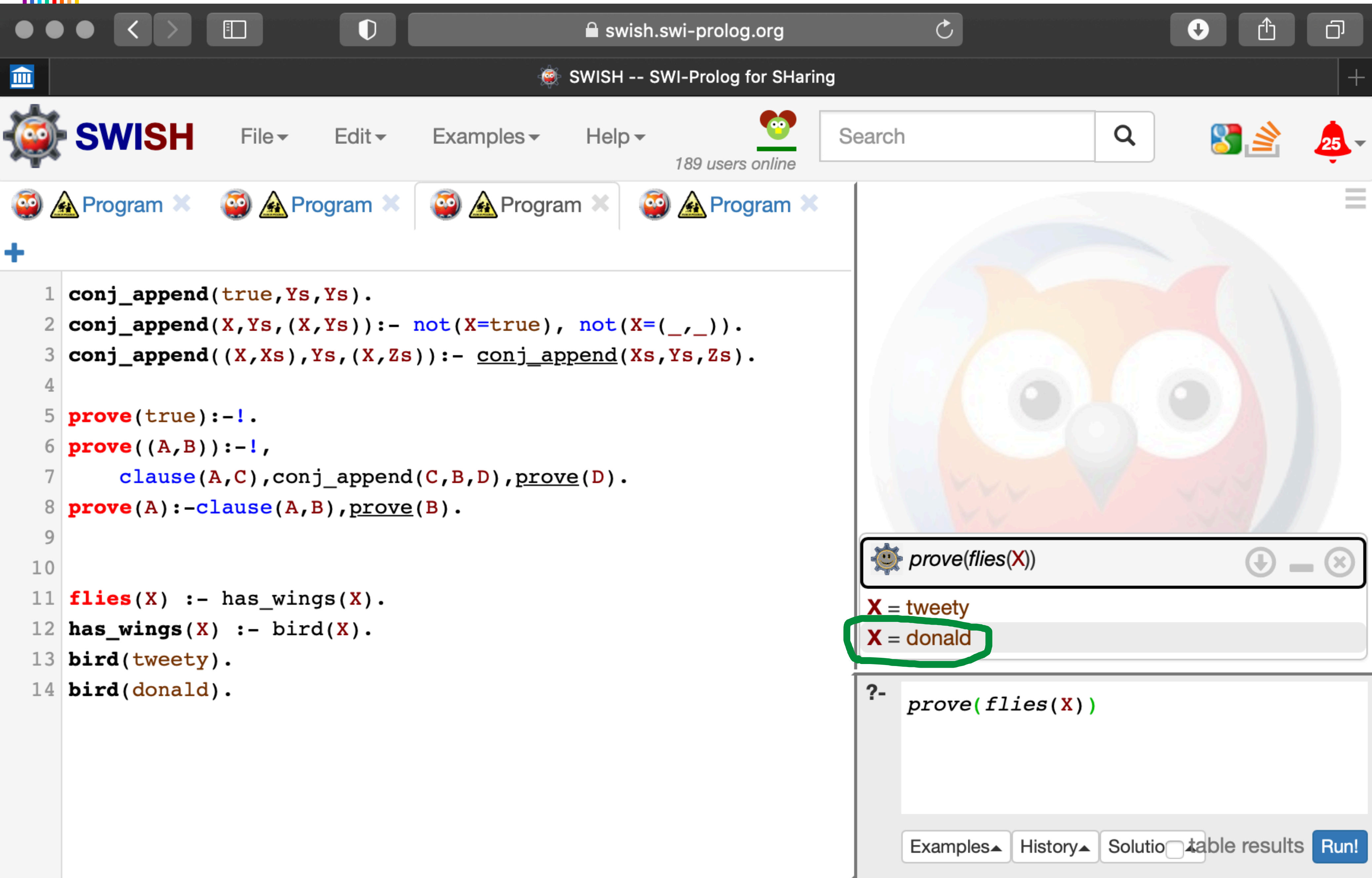
```prolog
conj_append(true,Ys,Ys).
conj_append(X,Ys,(X,Ys)):- not(X=true), not(X=(_,_)).
conj_append((X,Xs),Ys,(X,Zs)):- conj_append(Xs,Ys,Zs).

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).
```

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


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
Solutions
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
25
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:

```
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

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

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

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

```
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).
```

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

Agenda-based SLD-prover


```
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).
```

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

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).

Agenda-based SLD-prover


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

(Not shown here) — resolve(A,Cl,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).
```

Refutation prover for clausal logic

Informed Search

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

% add_bstf(A,B,C) <- C contains the elements of A and B
%                    (B and C sorted according to eval/2)
add_bstf([],Agenda,Agenda).
add_bstf([Child|Children],OldAgenda,NewAgenda):-
  add_one(Child,OldAgenda,TmpAgenda),
  add_bstf(Children,TmpAgenda,NewAgenda).

% add_one(S,A,B) <- B is A with S inserted acc. to eval/2
add_one(Child,OldAgenda,NewAgenda):-
  eval(Child,Value),
  add_one(Value,Child,OldAgenda,NewAgenda).
```

Best-first search

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

% add_bstf(A,B,C) <- C contains the elements of A and B
%                     (B and C sorted according to eval/2)
add_bstf([],Agenda,Agenda).
add_bstf([Child|Children],OldAgenda,NewAgenda):-
  add_one(Child,OldAgenda,TmpAgenda),
  add_bstf(Children,TmpAgenda,NewAgenda).

% add_one(S,A,B) <- B is A with S inserted acc. to eval/2
add_one(Child,OldAgenda,NewAgenda):-
  eval(Child,Value),
  add_one(Value,Child,OldAgenda,NewAgenda).
```

Best-first search

- An ***A algorithm*** is a best-first search algorithm that aims at minimising the **total cost** along a path from start to goal.

$$f(n) = g(n) + h(n)$$

- $f(n)$: estimate of total cost along path through n
- $g(n)$: actual cost to reach n
- $h(n)$: estimate of cost to reach goal from n

A algorithm

- A heuristic is (globally) ***optimistic*** or ***admissible*** if the estimated cost of **reaching a goal** is always less than the actual cost.

$$h(n) \leq h^*(n)$$

estimate of cost to reach goal from n

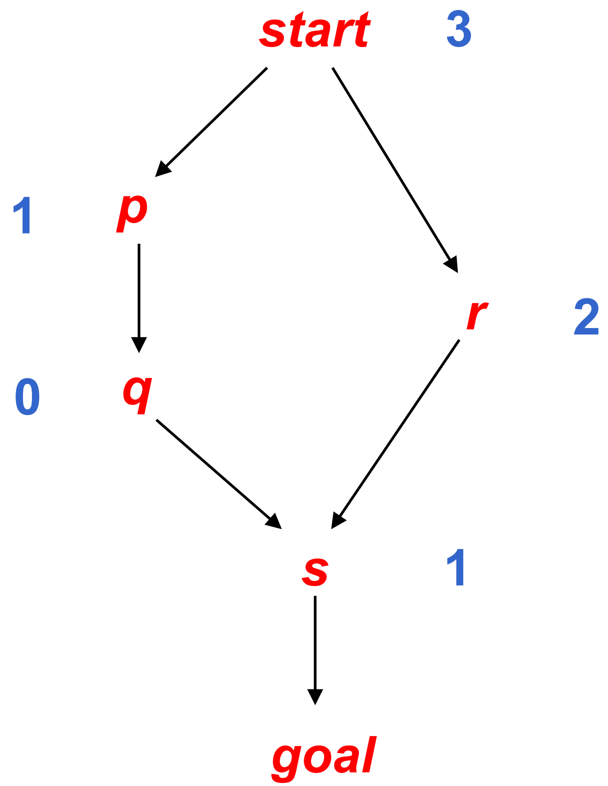
actual (unknown) cost to reach goal from n

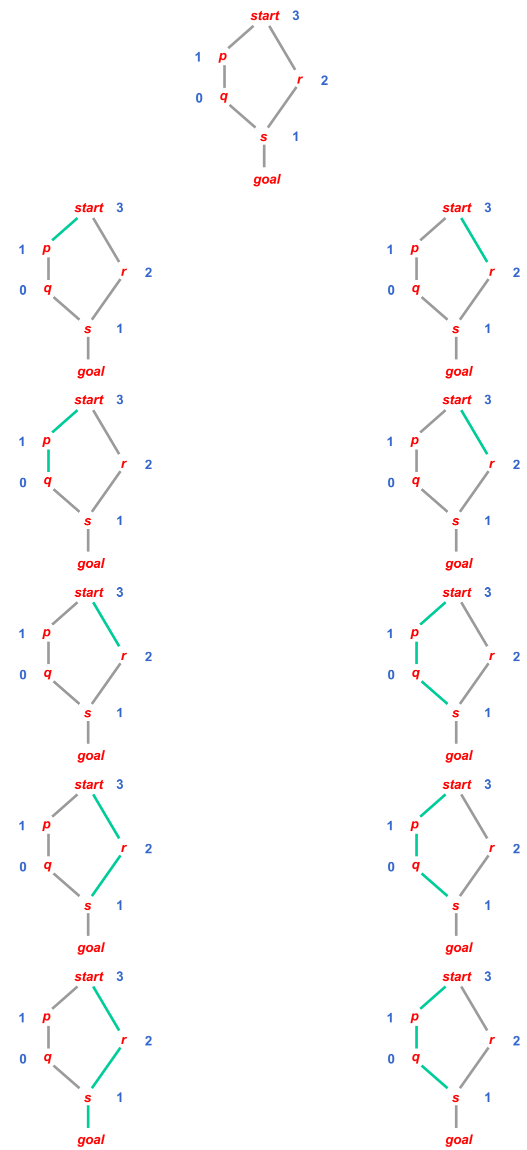
- A heuristic is ***monotonic*** (locally optimistic) if the estimated cost of **reaching any node** is always less than the actual cost.

$$h(n_1) - h(n_2) \leq h^*(n_1) - h^*(n_2)$$

Global and local optimism

Non-monotonic heuristic





```
[start-3]

[p-2,r-3]

[q-2,r-3]

[r-3,s-4]

[s-3,s-4]

[goal-3,s-4]
```

```
search_beam(Agenda,Goal):-
  search_beam(1,Agenda,[],Goal).

search_beam(D,[],NextLayer,Goal):-
  D1 is D+1,
  search_beam(D1,NextLayer,[],Goal).
search_beam(D,[Goal|Rest],NextLayer,Goal):-
  goal(Goal).
search_beam(D,[Current|Rest],NextLayer,Goal):-
  children(Current,Children),
  add_beam(D,Children,NextLayer,NewNextLayer),
  search_beam(D,Rest,NewNextLayer,Goal).
```

- Here, the number of children to be added to the beam is made dependent on the depth **D** of the node
  - in order to keep depth as a 'global' variable, search is layer-by-layer

Beam search


```
search_hc(Goal,Goal):-
 goal(Goal).
search_hc(Current,Goal):-
 children(Current,Children),
 select_best(Children,Best),
 search_hc(Best,Goal).
```

```
% hill_climbing as a variant of best-first search
search_hc([Goal|_],Goal):-
   goal(Goal).
search_hc([Current|_],Goal):-
   children(Current,Children),
   add_bstf(Children,[],NewAgenda),
   search_hc(NewAgenda,Goal).
```

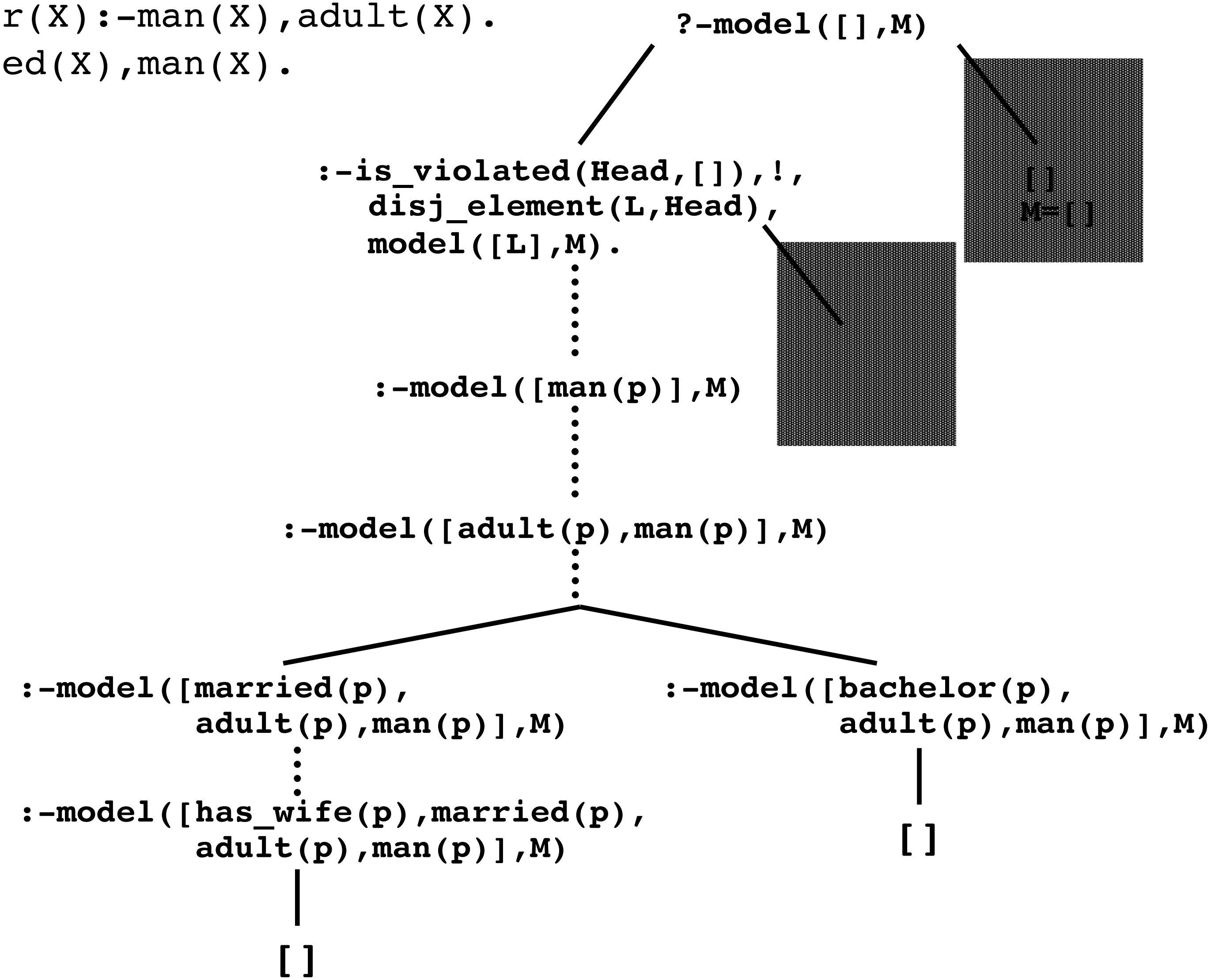
Hill-climbing

If time permits…

```
% model(M) <- M is a model of the clauses defined by cl/1
model(M):-
  model([],M).

model(M0,M):-
  is_violated(Head,M0),!,    % instance of violated clause
  disj_element(L,Head),      % L: ground literal from head
  model([L|M0],M).           % add L to the model
model(M,M).                  % no more violated clauses

is_violated(H,M):-
  cl((H:-B)),
  satisfied_body(B,M),       % grounds the variables
  not satisfied_head(H,M).
```

Forward chaining

```
married(X);bachelor(X):-man(X),adult(X).
has_wife(X):-married(X),man(X).
man(paul).
adult(paul).
```



Forward chaining: example


```
% model_d(D,M) <- M is a submodel of the clauses
%                 defined by cl/1
model_d(D,M):-
  model_d(D,[],M).

model_d(0,M,M).
model_d(D,M0,M):-
  D>0,D1 is D-1,
  findall(H,is_violated(H,M0),Heads),
  satisfy_clauses(Heads,M0,M1),
  model_d(D1,M1,M).

satisfy_clauses([],M,M).
satisfy_clauses([H|Hs],M0,M):-
  disj_element(L,H),
  satisfy_clauses(Hs,[L|M0],M).
```

Forward chaining with depth-bound