**Q1:** Add one more bird category at the level of ‘laysun_albatross’, ‘black_footed_albatross’, and ‘rumpeter_swan’. You can choose a Swedish/European bird, or any other bird from the American continent.

Remember that your task is to implement an expert system, think about the current expert system, and decide if you can add a Swedish or European bird easily, or if the current implementation only allows adding American birds.

=> The current expert system uses logical rules to infer a bird species based on its family, order, and traits (like color, voice, feet, etc.). It is structured hierarchically as:

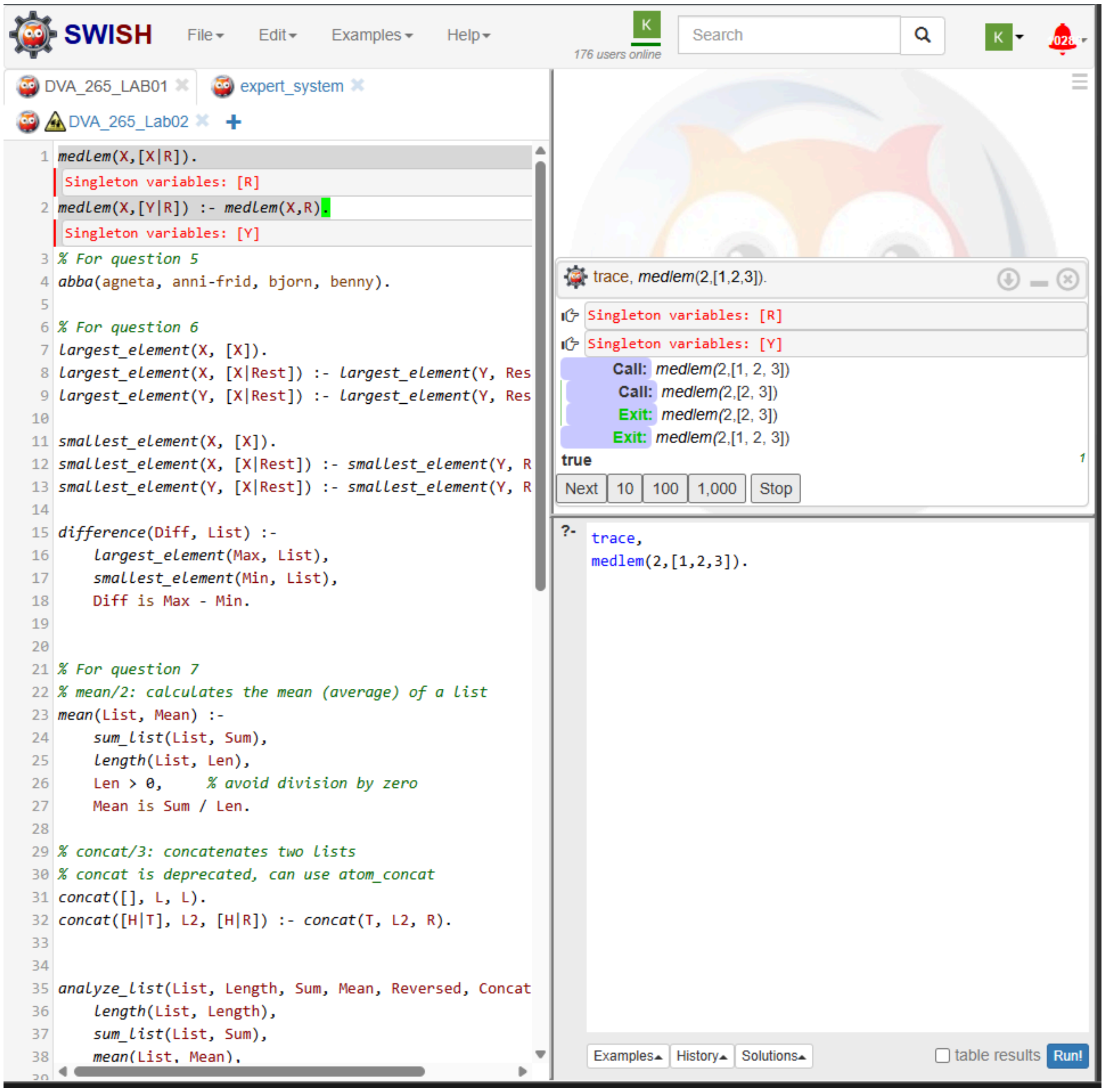
- `bird/1` (species-level)
- `family/1`
- `order/1`
- traits (attributes used in rules and user queries)

Here, I can add Swedish/European birds, but I’ll need to describe them in terms of the traits my system currently uses (`color`, `bill`, `wings`, `feet`, `voice`, etc.).

So I added the new bird category (mute_swan) -> same Swan family, but the voice trait is different.

**Q2: trace**

=> For Q1 in Lab1:



```
medlem(X,[X|R]).
Singleton variables: [R]
medlem(X,[Y|R]) :- medlem(X,R).
Singleton variables: [Y]
% For question 5
abba(agneta, anni-frid, bjorn, benny).

% For question 6
largest_element(X, [X]).
largest_element(X, [X|Rest]) :- largest_element(Y, Res
largest_element(Y, [X|Rest]) :- largest_element(Y, Res

smallest_element(X, [X]).
smallest_element(X, [X|Rest]) :- smallest_element(Y, R
smallest_element(Y, [X|Rest]) :- smallest_element(Y, R

difference(Diff, List) :-
    largest_element(Max, List),
    smallest_element(Min, List),
    Diff is Max - Min.


% For question 7
% mean/2: calculates the mean (average) of a list
mean(List, Mean) :-
    sum_list(List, Sum),
    length(List, Len),
    Len > 0,      % avoid division by zero
    Mean is Sum / Len.

% concat/3: concatenates two lists
% concat is deprecated, can use atom_concat
concat([], L, L).
concat([H|T], L2, [H|R]) :- concat(T, L2, R).


analyze_list(List, Length, Sum, Mean, Reversed, Concat
    length(List, Length),
    sum_list(List, Sum),
    mean(List, Mean),
```

```
trace, medlem(2,[1,2,3]).
Singleton variables: [R]
Singleton variables: [Y]
 Call: medlem(2,[1, 2, 3])
  Call: medlem(2,[2, 3])
  Exit: medlem(2,[2, 3])
 Exit: medlem(2,[1, 2, 3])
true
```

```
?- trace,
   medlem(2,[1,2,3]).
```

**Q3:**

1) **findall(Template, Goal, Bag):-**
   **findall(Template. Goal, Bag, [ ] )**

- `Template`: what to collect (e.g., a variable)
- `Goal`: the predicate to solve

- `Bag`: resulting list of all matching `Template` values
- Succeeds with an empty list if the Goal has no solutions.

2) **bagof(Template, Goal, List).**

   - Like findall, but groups answers by free variables
   - Fails if the goal has no solution

3) **setof(Template, Goal, List).**

   - Like bagof/3, but results are sorted and duplicate-free

**Q4:**
Unification is the fundamental operation in Prolog that attempts to make two terms equal by finding a suitable substitution (i.e., binding variables so they match).

The screenshot below shows a `trace` session in **SWI-Prolog**, where we are calling:

?- trace, bird(trumpeter_swan).

1) **Unifying the Query with a Rule Head**

We have:

```
bird(trumpeter_swan):-
        family(swan),
        voice(loud_trumpeting).
```

Prolog matches the query bird(trumpeter_swan) with this rule head. This is unification—Prolog sees that the query exactly matches the head of a rule, so it proceeds to evaluate the rule's body.

2) **Resolving Subgoal: `family(swan)`**
   Prolog tries to unify family(swan) with this rule:

```
family(swan) :-
        order(waterfowl),
        neck(long),
        color(white),
        flight(ponderous).
```

Since the goal and rule head match (family(swan)), unification succeeds again, and Prolog moves into resolving its body.

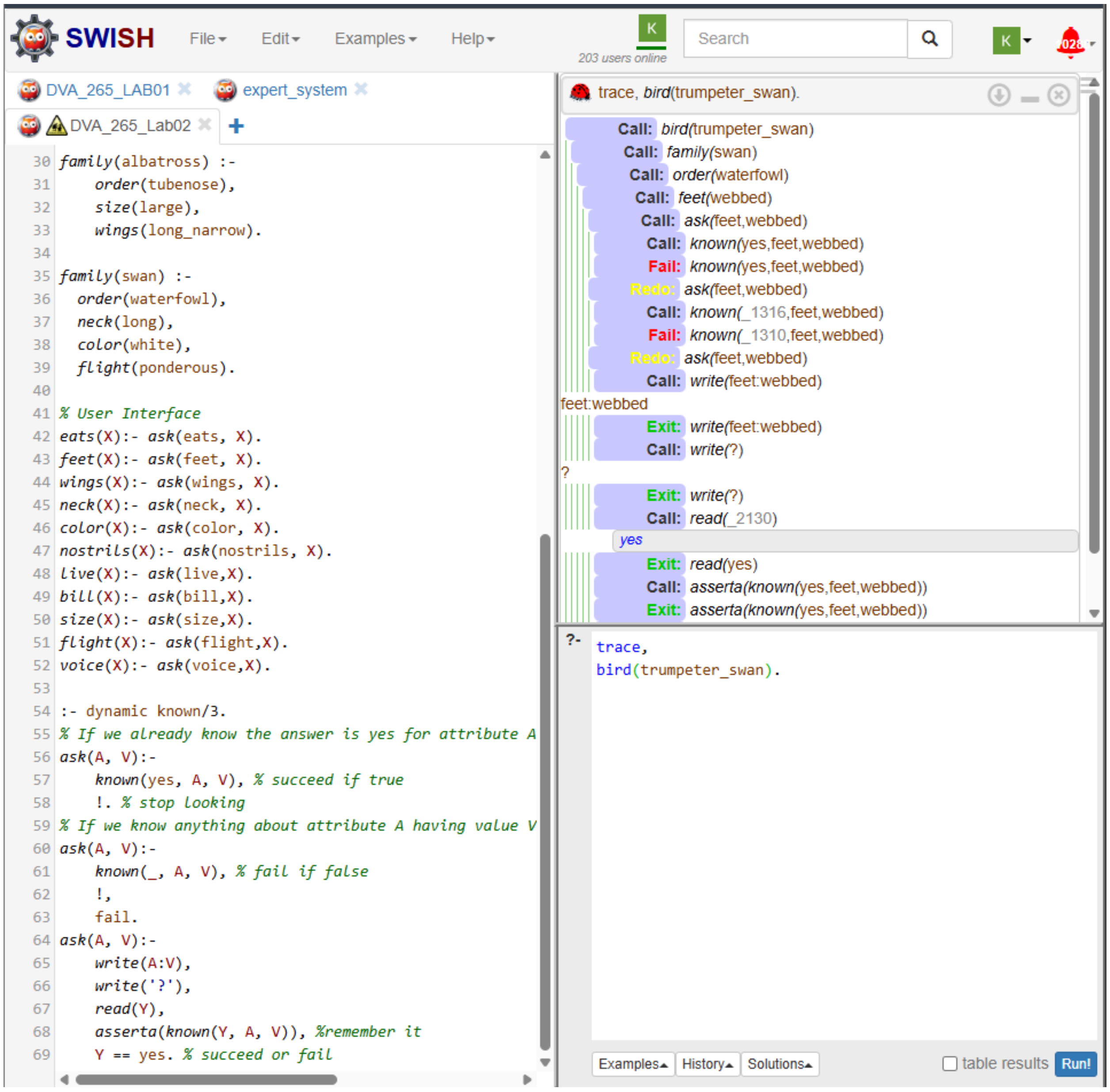
3) **Resolving Sub-subgoal: order(waterfowl)**
   This unifies with:
   ```
           order(waterfowl) :-
              feet(webbed),
              bill(flat).
   ```

   Again, unification matches the head, and Prolog checks each trait.

At every step, Prolog tries to match:

- A **goal** (like `feet(webbed)`)
- With a **rule head or known fact**

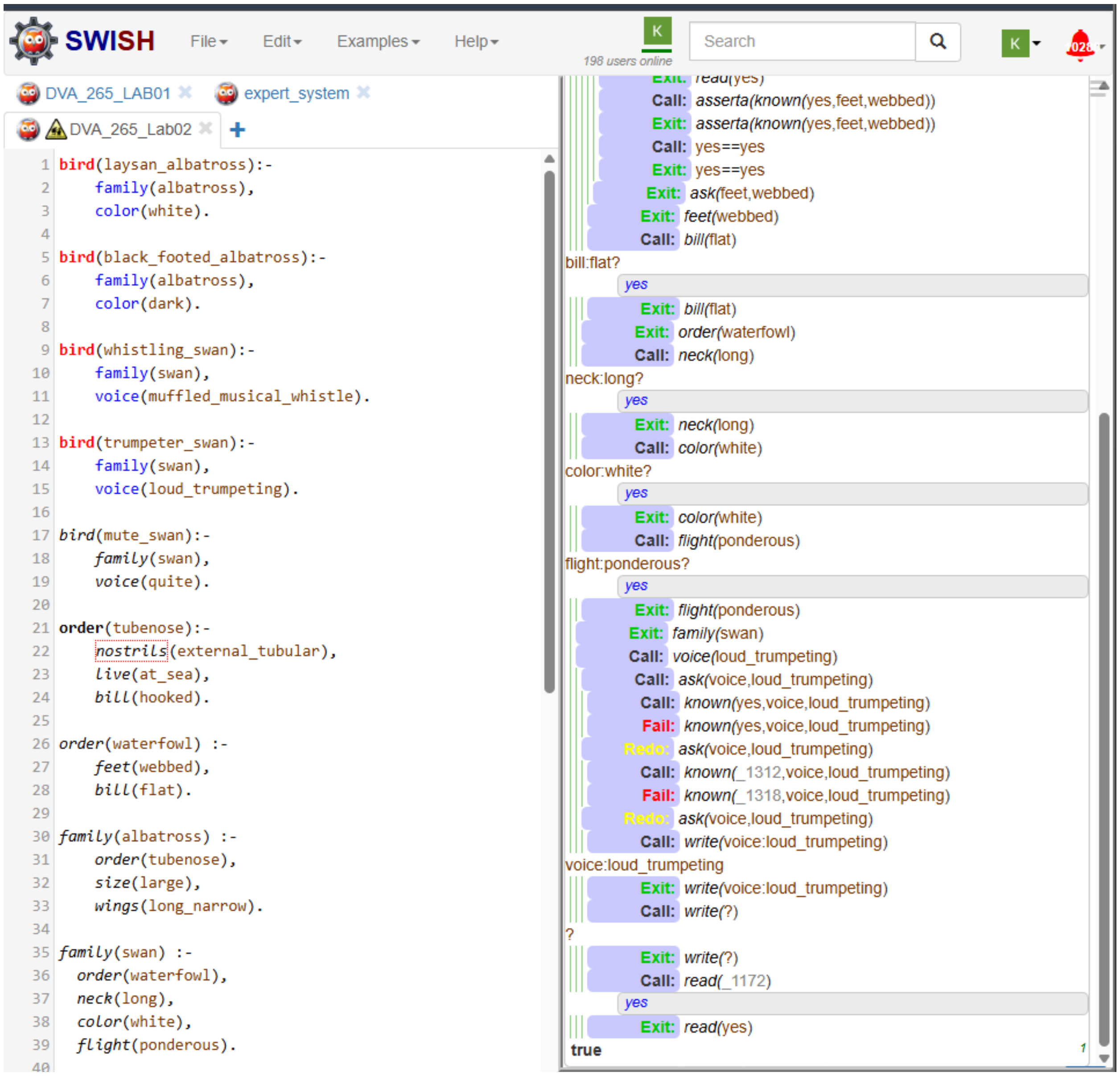
If it finds a match (or can bind variables to make a match), **unification succeeds**, and execution continues. If not, Prolog **backtracks** to try other options.



```
family(albatross) :-
    order(tubenose),
    size(large),
    wings(long_narrow).

family(swan) :-
  order(waterfowl),
  neck(long),
  color(white),
  flight(ponderous).

% User Interface
eats(X):- ask(eats, X).
feet(X):- ask(feet, X).
wings(X):- ask(wings, X).
neck(X):- ask(neck, X).
color(X):- ask(color, X).
nostrils(X):- ask(nostrils, X).
live(X):- ask(live,X).
bill(X):- ask(bill,X).
size(X):- ask(size,X).
flight(X):- ask(flight,X).
voice(X):- ask(voice,X).

:- dynamic known/3.
% If we already know the answer is yes for attribute A
ask(A, V):-
    known(yes, A, V), % succeed if true
    !. % stop looking
% If we know anything about attribute A having value V
ask(A, V):-
    known(_, A, V), % fail if false
    !,
    fail.
ask(A, V):-
    write(A:V),
    write('?'),
    read(Y),
    asserta(known(Y, A, V)), %remember it
    Y == yes. % succeed or fail
```

```
trace, bird(trumpeter_swan).

 Call: bird(trumpeter_swan)
  Call: family(swan)
   Call: order(waterfowl)
    Call: feet(webbed)
     Call: ask(feet,webbed)
      Call: known(yes,feet,webbed)
      Fail: known(yes,feet,webbed)
     Redo: ask(feet,webbed)
      Call: known(_1316,feet,webbed)
      Fail: known(_1310,feet,webbed)
     Redo: ask(feet,webbed)
      Call: write(feet:webbed)
feet:webbed
      Exit: write(feet:webbed)
      Call: write(?)
?
      Exit: write(?)
      Call: read(_2130)
         yes
      Exit: read(yes)
      Call: asserta(known(yes,feet,webbed))
      Exit: asserta(known(yes,feet,webbed))
```

```
?- trace,
   bird(trumpeter_swan).
```



```prolog
bird(laysan_albatross):-
    family(albatross),
    color(white).

bird(black_footed_albatross):-
    family(albatross),
    color(dark).

bird(whistling_swan):-
    family(swan),
    voice(muffled_musical_whistle).

bird(trumpeter_swan):-
    family(swan),
    voice(loud_trumpeting).

bird(mute_swan):-
    family(swan),
    voice(quite).

order(tubenose):-
    nostrils(external_tubular),
    live(at_sea),
    bill(hooked).

order(waterfowl) :-
    feet(webbed),
    bill(flat).

family(albatross) :-
    order(tubenose),
    size(large),
    wings(long_narrow).

family(swan) :-
  order(waterfowl),
  neck(long),
  color(white),
  flight(ponderous).
```

```
        Exit: read(yes)
        Call: asserta(known(yes,feet,webbed))
        Exit: asserta(known(yes,feet,webbed))
        Call: yes==yes
        Exit: yes==yes
       Exit: ask(feet,webbed)
      Exit: feet(webbed)
      Call: bill(flat)
bill:flat?
 yes
      Exit: bill(flat)
     Exit: order(waterfowl)
     Call: neck(long)
neck:long?
 yes
     Exit: neck(long)
     Call: color(white)
color:white?
 yes
     Exit: color(white)
     Call: flight(ponderous)
flight:ponderous?
 yes
     Exit: flight(ponderous)
    Exit: family(swan)
    Call: voice(loud_trumpeting)
     Call: ask(voice,loud_trumpeting)
      Call: known(yes,voice,loud_trumpeting)
      Fail: known(yes,voice,loud_trumpeting)
     Redo: ask(voice,loud_trumpeting)
      Call: known(_1312,voice,loud_trumpeting)
      Fail: known(_1318,voice,loud_trumpeting)
     Redo: ask(voice,loud_trumpeting)
      Call: write(voice:loud_trumpeting)
voice:loud_trumpeting
      Exit: write(voice:loud_trumpeting)
      Call: write(?)
?
      Exit: write(?)
      Call: read(_1172)
 yes
      Exit: read(yes)
true
```