University of BRISTOL

Artificial Intelligence: SWI Debugging Tips

Oliver Ray

bristol.ac.uk

Execution logging

Inserting print statements into your program code to log its progress can often be a quick and effective way to help track down errors as the program executes

The most convenient predicates for this are the built-ins **write/1**, **nl/0**, **writeln/1** and **format/2** (see the online manual for more information)

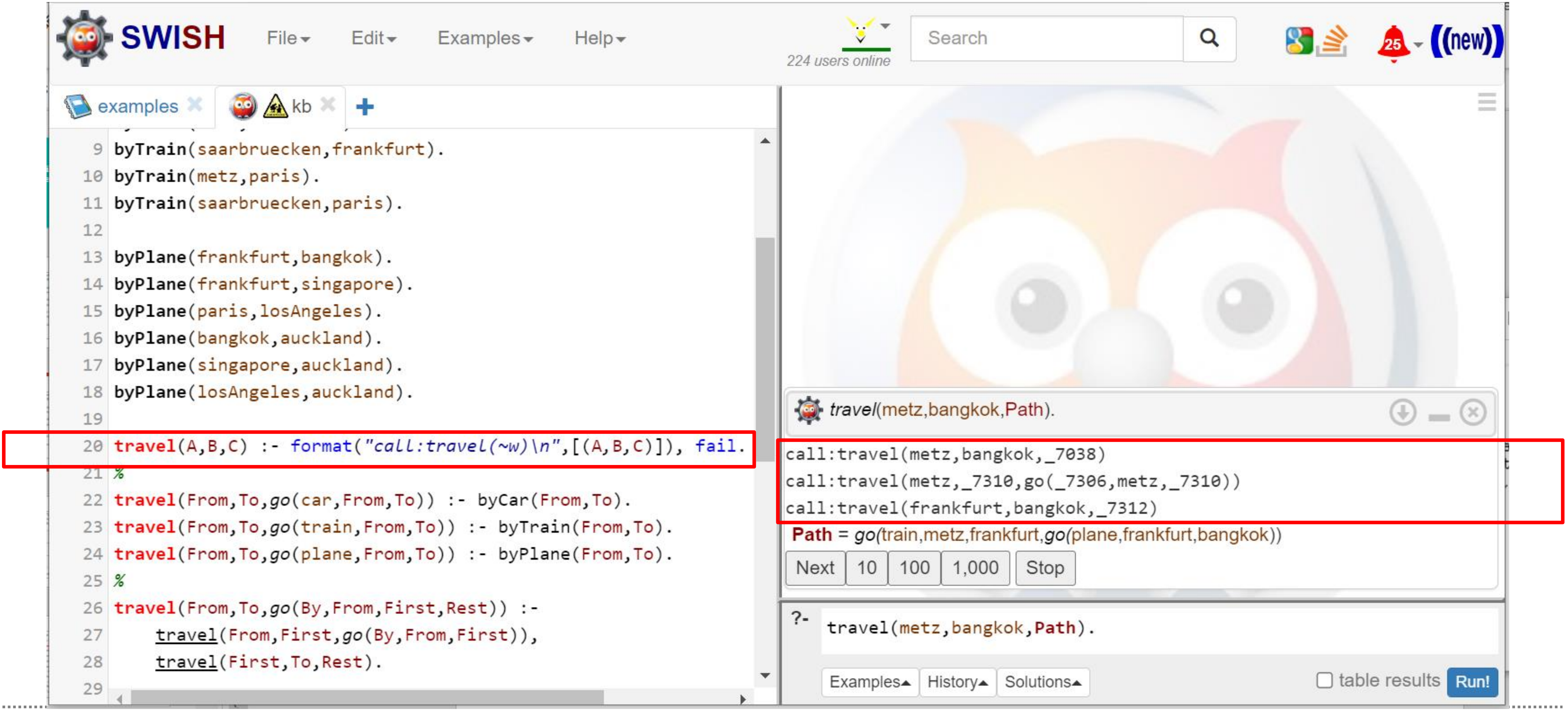
A useful trick (which exploits Prolog's standard depth-first search strategy) is to insert a dummy print statement as the FIRST definition for some target clause

This will print out the arguments of each call to that predicate and immediately fails in order to give control back to the actual predicate definition

For example, if you have a predicate p then you would insert a clause iike this:

```
p(A,B,…,Z) :- format("Call:p(~w)\n",[(A,B,…,Z)]), fail.
```

Example logging



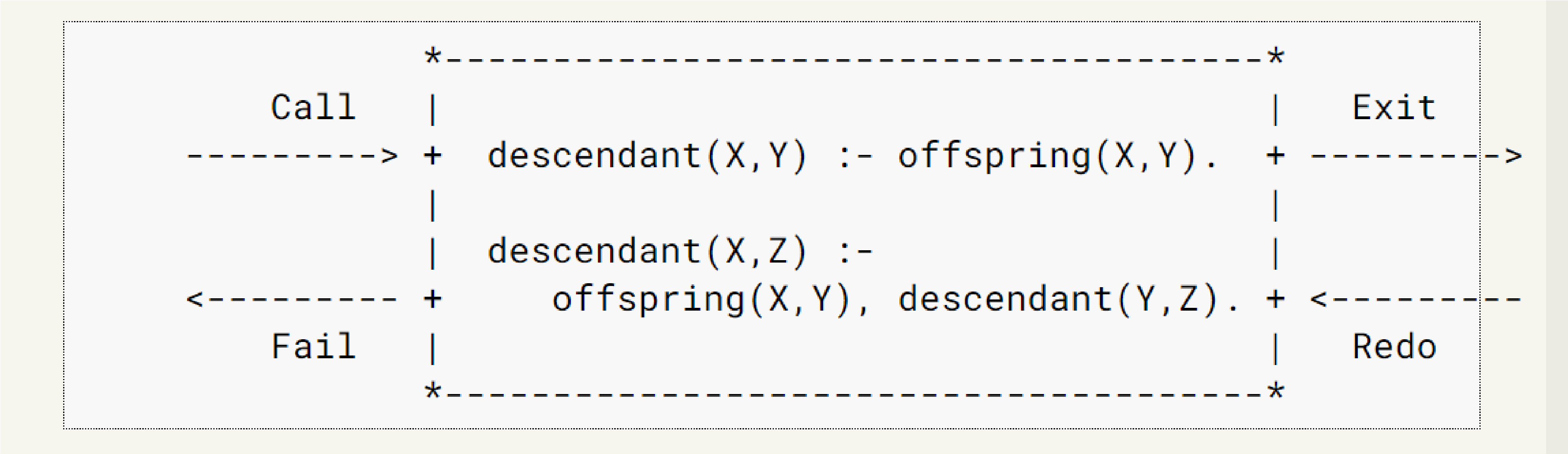
```
byTrain(saarbruecken,frankfurt).
byTrain(metz,paris).
byTrain(saarbruecken,paris).

byPlane(frankfurt,bangkok).
byPlane(frankfurt,singapore).
byPlane(paris,losAngeles).
byPlane(bangkok,auckland).
byPlane(singapore,auckland).
byPlane(losAngeles,auckland).

travel(A,B,C) :- format("call:travel(~w)\n",[(A,B,C)]), fail.
%
travel(From,To,go(car,From,To)) :- byCar(From,To).
travel(From,To,go(train,From,To)) :- byTrain(From,To).
travel(From,To,go(plane,From,To)) :- byPlane(From,To).
%
travel(From,To,go(By,From,First,Rest)) :-
    travel(From,First,go(By,From,First)),
    travel(First,To,Rest).
```

```
travel(metz,bangkok,Path).
call:travel(metz,bangkok,_7038)
call:travel(metz,_7310,go(_7306,metz,_7310))
call:travel(frankfurt,bangkok,_7312)
Path = go(train,metz,frankfurt,go(plane,frankfurt,bangkok))
```

```
?- travel(metz,bangkok,Path).
```

Procedure-Box Tracing

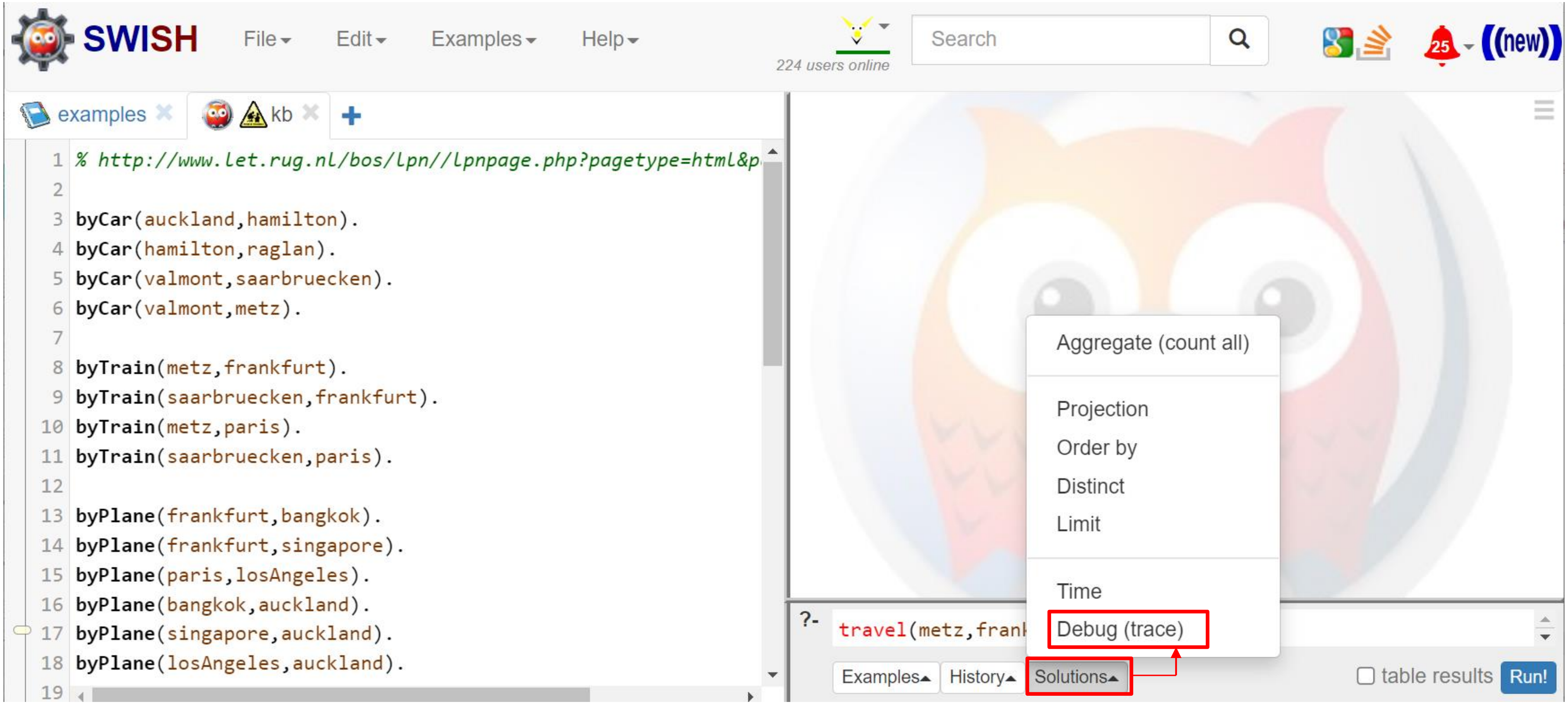
Most Prolog systems provide code "tracing" tools based on a "procedure-box" model with 4 (or 5 or 6) "ports" as described in http://gprolog.org/manual/html_node/gprolog012.html:



```
           *---------------------------------------*
    Call   |                                       |   Exit
---------> +  descendant(X,Y) :- offspring(X,Y).   + --------->
           |                                       |
           |  descendant(X,Z) :-                   |
<--------- +     offspring(X,Y), descendant(Y,Z).  + <---------
    Fail   |                                       |   Redo
           *---------------------------------------*
```

In SWI, you can invoke the tracer using the SWISH “**Solutions**” menu; or by using the SWIPL commands “**trace**.” and “**notrace.**” to turn the tracer off and on, respectively

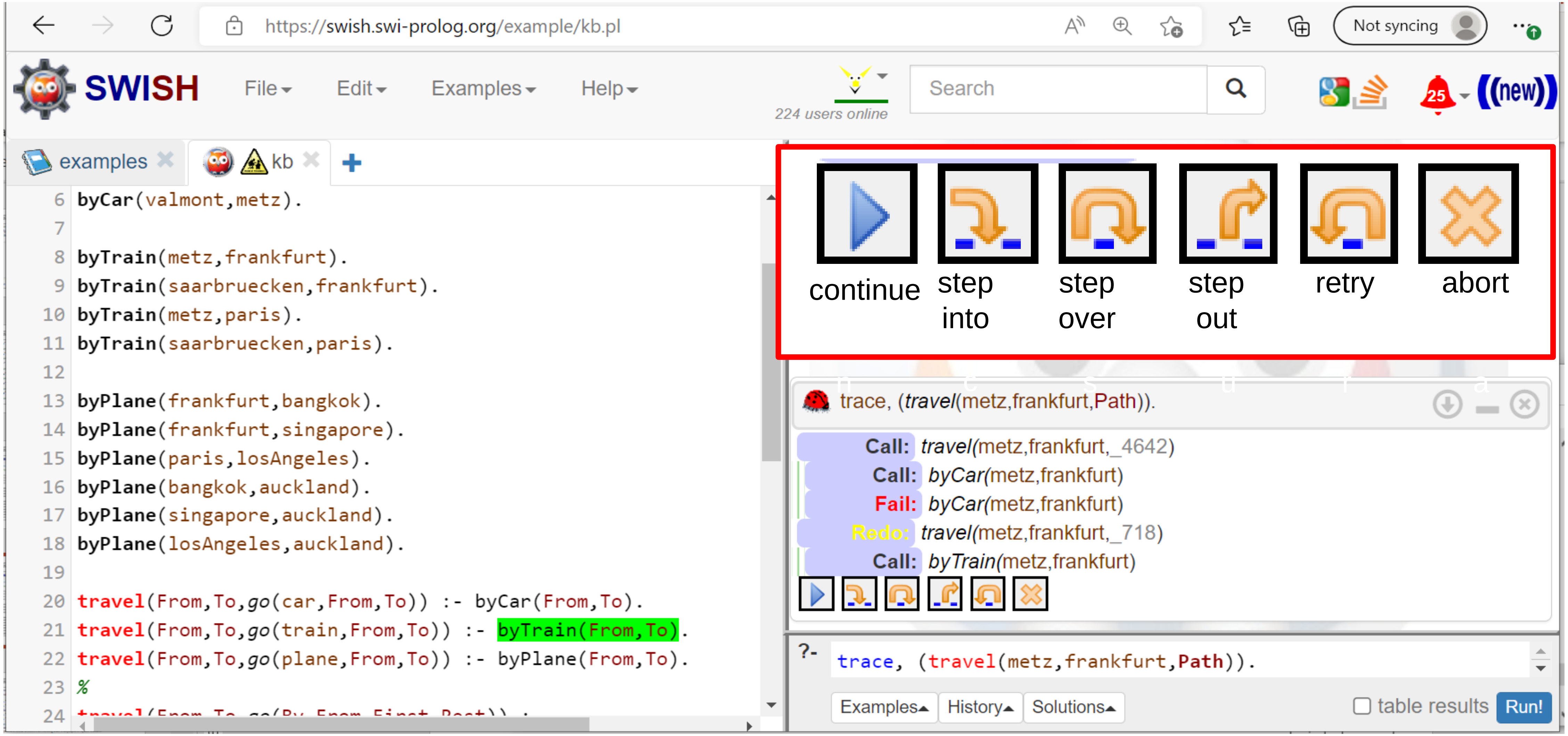
SWISH Tracing



```
% http://www.let.rug.nl/bos/lpn//lpnpage.php?pagetype=html&p

byCar(auckland,hamilton).
byCar(hamilton,raglan).
byCar(valmont,saarbruecken).
byCar(valmont,metz).

byTrain(metz,frankfurt).
byTrain(saarbruecken,frankfurt).
byTrain(metz,paris).
byTrain(saarbruecken,paris).

byPlane(frankfurt,bangkok).
byPlane(frankfurt,singapore).
byPlane(paris,losAngeles).
byPlane(bangkok,auckland).
byPlane(singapore,auckland).
byPlane(losAngeles,auckland).
```

```
?- travel(metz,frank
```

Example Tracing



```
byCar(valmont,metz).

byTrain(metz,frankfurt).
byTrain(saarbruecken,frankfurt).
byTrain(metz,paris).
byTrain(saarbruecken,paris).

byPlane(frankfurt,bangkok).
byPlane(frankfurt,singapore).
byPlane(paris,losAngeles).
byPlane(bangkok,auckland).
byPlane(singapore,auckland).
byPlane(losAngeles,auckland).

travel(From,To,go(car,From,To)) :- byCar(From,To).
travel(From,To,go(train,From,To)) :- byTrain(From,To).
travel(From,To,go(plane,From,To)) :- byPlane(From,To).
%
travel(From,To,go(By,From,First_Rest)) :-
```

```
trace, (travel(metz,frankfurt,Path)).
 Call: travel(metz,frankfurt,_4642)
  Call: byCar(metz,frankfurt)
  Fail: byCar(metz,frankfurt)
 Redo: travel(metz,frankfurt,_718)
  Call: byTrain(metz,frankfurt)
```

```
?- trace, (travel(metz,frankfurt,Path)).
```

Basic SWIPL Tracing

- Hit "**c**" or <**enter**> or <**spacebar**> to **creep into** a call

  useful to see how this call unfolds step by step!

- Hit "**s**" to **skip over** a call

  useful to jump over an uninteresting call!

- Hit "**r**" to **retry** a call that just exited

  useful to replay a call you just skipped over!

- Hit "**u**" to **go up** out of this call to the exit of the parent call

  useful if you accidentally crept into a call that that you should have skipped over!

- Hit "**n**" to return to **no debug** mode and continue the computation normally

  useful if you realise debugging is no longer required!

- Hit "**a**" to **abort** the computation

  useful if you've now realised what the bug is!

Intermediate SWIPL Tracing

Once you're familiar with the basic tracing options, the following commands more advanced are also useful (only in SWIPL):

- Hit "**+**" to "set a spy point" on the current predicate
- Hit "**l**" to "leap" to the next spy point

  useful if you are interested in calls to specific predicates
- Hit "**-**" to "remove spy points" on the current predicate
- Hit "**b**" to invoke an interactive Prolog "break session"

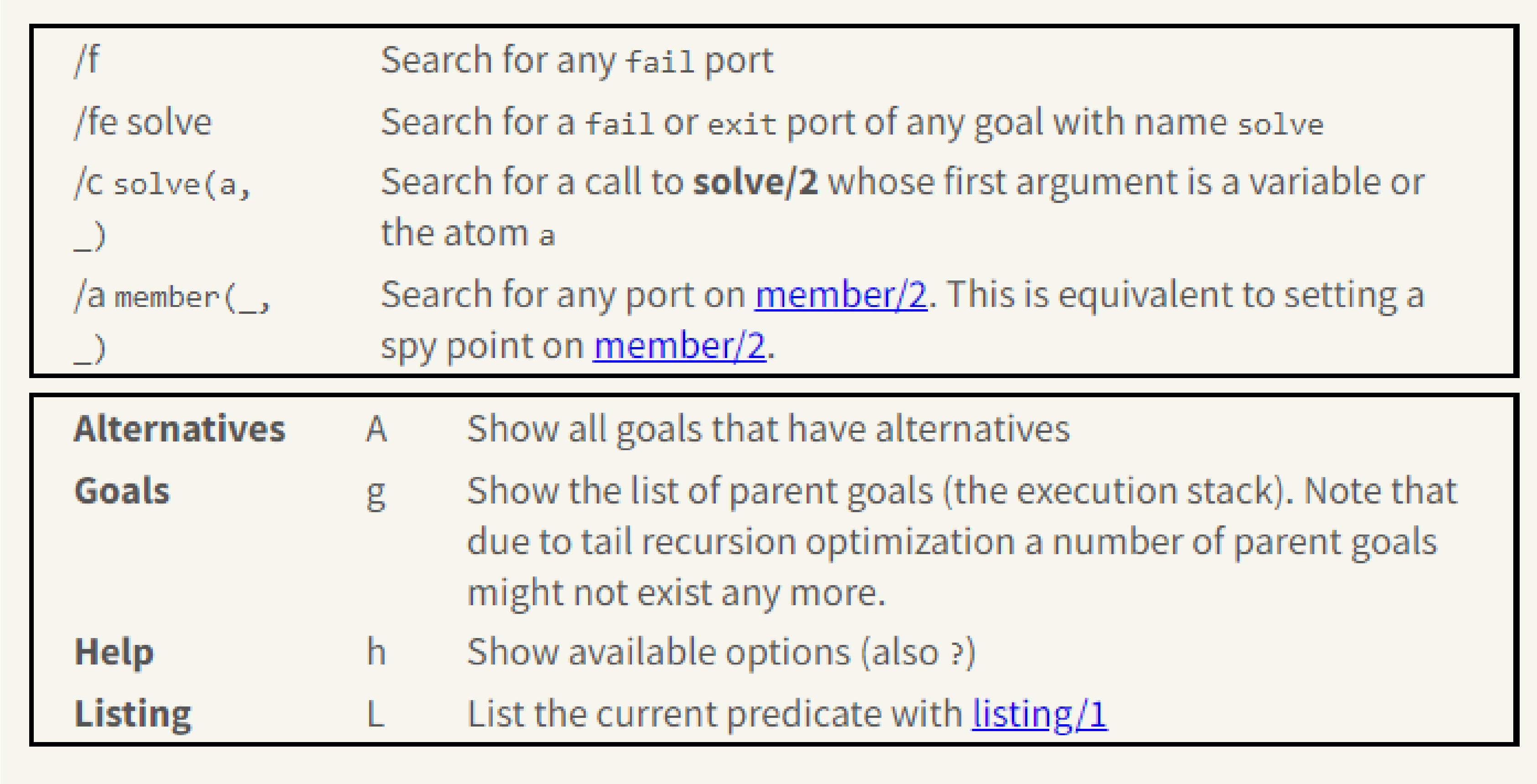
  where you can type commands in the current debug context

  you can turn off tracing in the break session using the "**n**" option

  if you nest break sessions then nesting level is displayed in square brackets

  use **<ctrl>-d** or type "**end_of_file.**" to exit and return to the parent session
- Hit "**?**" or "**h**" to show a “help screen” listing available commands

Further SWIPL Tracing



/f	Search for any `fail` port
/fe solve	Search for a `fail` or `exit` port of any goal with name `solve`
/c `solve(a, _)`	Search for a call to **solve/2** whose first argument is a variable or the atom `a`
/a `member(_, _)`	Search for any port on member/2. This is equivalent to setting a spy point on member/2.

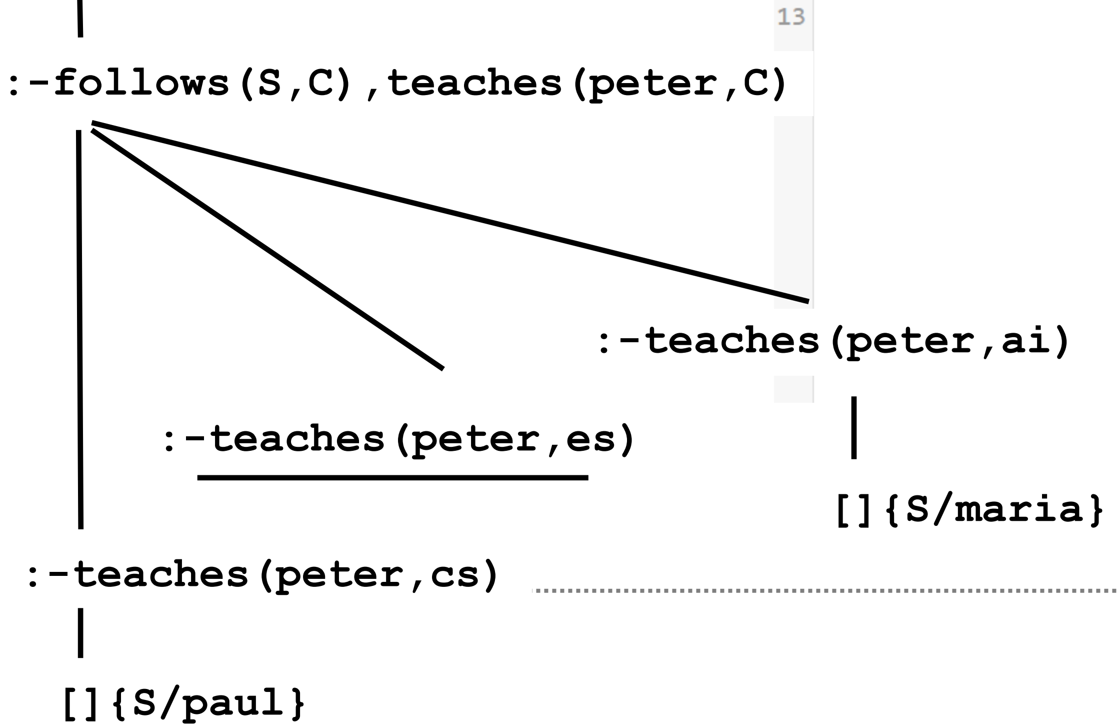
Alternatives	A	Show all goals that have alternatives
Goals	g	Show the list of parent goals (the execution stack). Note that due to tail recursion optimization a number of parent goals might not exist any more.
Help	h	Show available options (also `?`)
Listing	L	List the current predicate with listing/1

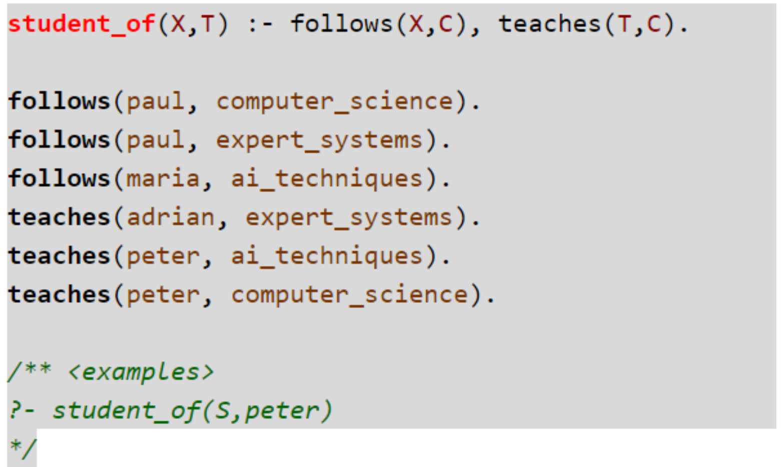
sometimes crash???

For further information and even more options, please see the online manual: https://www.swi-prolog.org/pldoc/man?section=debugoverview

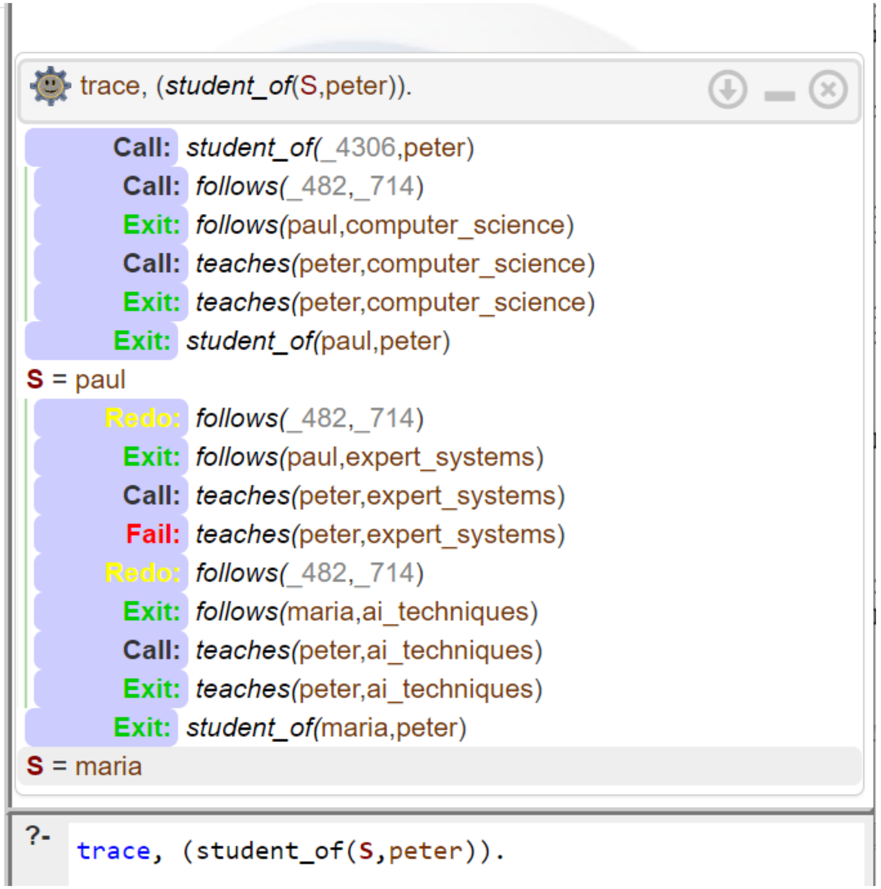
Search (SLD) Tree

?-student_of(S,peter)





```
student_of(X,T) :- follows(X,C), teaches(T,C).

follows(paul, computer_science).
follows(paul, expert_systems).
follows(maria, ai_techniques).
teaches(adrian, expert_systems).
teaches(peter, ai_techniques).
teaches(peter, computer_science).

/** <examples>
?- student_of(S,peter)
*/
```



```
trace, (student_of(S,peter)).
    Call: student_of(_4306,peter)
     Call: follows(_482,_714)
     Exit: follows(paul,computer_science)
     Call: teaches(peter,computer_science)
     Exit: teaches(peter,computer_science)
    Exit: student_of(paul,peter)
S = paul
     Redo: follows(_482,_714)
     Exit: follows(paul,expert_systems)
     Call: teaches(peter,expert_systems)
     Fail: teaches(peter,expert_systems)
     Redo: follows(_482,_714)
     Exit: follows(maria,ai_techniques)
     Call: teaches(peter,ai_techniques)
     Exit: teaches(peter,ai_techniques)
    Exit: student_of(maria,peter)
S = maria

?- trace, (student_of(S,peter)).
```

Declarative Debugging

Logging and tracing have a very procedural flavour; but there are also some very useful techniques in an area known as declarative debugging

Here’s a nice video intro to a couple of these techniques – called **generalisation** and **failure slicing**: https://www.youtube.com/watch?v=4IWruicMd4c

A textual description is available here: https://www.metalevel.at/prolog/debugging

Note that some of the claims in this video only apply to pure logic programs without cut or negation and where constraints are used instead of built-in arithmetic predicates - but the distinctions are not that important as you can still usefully use these techniques in the coursework or exam preparation

Thank you