Debugging with `trace`

Here it is explained how to use `trace` to trace the execution of a Prolog program. This is very useful to debug faulty programs, but also to gain an understanding of how correct programs work. In particular, one can see first-hand how backtraccking works.

Typing `trace` into Prolog will lead to

```
?- trace.

Yes
[trace]  ?-
```

Now enter your query as usual, for example,

```
[trace]  ?- ancestor(X,george).
```

which will produce output

```
    Call: (7) ancestor(_G391, george) ?
```

`Call` means that Prolog will try to solve the predicate (that is, the predicate that is displayed on the same line as `Call`). `X` has been replaced by `_G391` which Prolog uses as an internal name for the variable `X`. Using the enter or return key will allow you to follow through the whole computation, the output of which then looks like this:

```
[trace]  ?- ancestor(X,george).
   Call: (7) ancestor(_G391, george) ? creep
   Call: (8) parent(_G391, george) ? creep
   Exit: (8) parent(arthur, george) ? creep
   Exit: (7) ancestor(arthur, george) ? creep

X = arthur ;
   Redo: (8) parent(_G391, george) ? creep
   Exit: (8) parent(harriet, george) ? creep
   Exit: (7) ancestor(harriet, george) ? creep

X = harriet

Yes
```

`Exit` means that the computation of the predicate has been successful (that is, a solution has been found). `Redo` means that Prolog is going to backtrack on the predicate. There is also `Fail` meaning that the predicate failed (that is, no solution has been found).

Exercise 1. Run the example above on your computer and explain the output.

Let us now see how to use `trace` to debug a faulty program:

```
ancestor4(X,Y):-ancestor4(X,Z),parent(Z,Y).
ancestor4(X,Y):-parent(X,Y).
```

This program is faulty because it does not compute ancestors:

```
?- ancestor4(X,george).
ERROR: Out of local stack
```

To understand what happens, we run the program in trace mode:

```
[trace]  ?- ancestor4(X,george).
   Call: (7) ancestor4(_G397, george) ? creep
   Call: (8) ancestor4(_G397, _G452) ? creep
   Call: (9) ancestor4(_G397, _G452) ? creep
   Call: (10) ancestor4(_G397, _G452) ? creep
   Call: (11) ancestor4(_G397, _G452) ? creep
   Call: (12) ancestor4(_G397, _G452) ? creep
   Call: (13) ancestor4(_G397, _G452) ? creep
   Call: (14) ancestor4(_G397, _G452) ? creep
   Call: (15) ancestor4(_G397, _G452) ?
```

We see that each call of `ancestor4` results in a new call of `ancestor4`. This is due the fact that in the definition the recursion is on the start of the first rule (in contrast to the definition of `ancestor`).

Understanding this now, we might not want to proceed with a further call of `ancestor4`. Instead of using the enter or return, we type 's'. 's' stands for 'skip' and means that, instead of tracing through the computation of the current predicate, we skip the tracing and just compute the predicate 'in one step'. In our case, this leads of course to the error encountered already before. Finally, we type 'a' to abort the computation and return to the familiar Prolog prompt:

```
   Call: (15) ancestor4(_G397, _G452) ? skip
ERROR: Out of local stack
   Exception: (15) ancestor4(_G397, _G457) ? abort
Execution Aborted
?-
```

Exercise 2. Run the example above on your computer and explain the output. In particular, explain the numbers (7), (8), ... (15). Why are they increasing? Going back to the trace of `ancestor(X,george)`, why are the numbers there also decreasing (from (8) to (7))?