COMP219: Artificial Intelligence
Lab Exercise 2 - To be carried out in Week 3

1. Continuing on from last week's exercises, add the ancestor relation discussed in Lecture 6 to the `familyTree.pl` program (available on the COMP219 website), save and reload. Predict what will be outcome for the following queries then try them.

   ```
   ancestor(ian,lucy).

   ancestor(pete,lucy).

   ancestor(pauline,lucy).

   ancestor(lucy,lucy).

   ancestor(X,lucy).
   ```

   Using the ancestor relation, find out who the ancestors of peter are. Using the family tree program with the new facts you added last week, find who phil is an ancestor of.

2. Prolog has the command `trace` as an inbuilt predicate that allows you to input queries and follow step by step how Prolog is evaluating the query in trying to satisfy the goal. Type

   ```
   trace, whatever-your-query-is
   ```

   on the command line to enter trace mode. Pressing return will give the new goal or whether the current goal has succeeded or failed.

   The trace for `ancestor(pete,lucy).` is given below.

   ```
   | ?- trace,  ancestor(pete,lucy).
           Call:   (8) ancestor(pete, lucy) ? creep
           Call:   (9) parent(pete, lucy) ? creep
           Fail:   (9) parent(pete, lucy) ? creep
           Redo:   (8) ancestor(pete, lucy) ? creep
           Call:   (9) parent(pete,_L196) ? creep
           Exit:   (9) parent(pete, ian) ? creep
           Call:   (9) ancestor(ian, lucy) ? creep
           Call:   (10) parent(ian, lucy) ? creep
           Exit:   (10) parent(ian, lucy) ? creep
           Exit:   (9) ancestor(ian, lucy) ? creep
           Exit:   (8) ancestor(pete, lucy) ? creep
   true.
   ```

Note that `trace` causes debugging mode to be turned on. To turn it off, type `nodebug.` when the querying has finished.

Trace the other queries in question 1.

3. Add the *silly* clause from Lecture 6, i.e.

   ```
   silly(X):-silly(X).
   ```

   to your Prolog program and try it out in queries. Use trace to see what is happening. If you get stuck in an infinite loop try typing control C.

4. Add the other predecessor relations discussed in Lecture 6 to the `familyTree.pl` program (i.e. predecessor, predecessor2). Try out each relation on the queries suggested in question 1. Do you get the same answers? Trace the queries to see what is happening.

5. Add a third version of predecessor, given below (and not in the lecture notes), to the `familyTree.pl` program

   ```
   predecessor3(X,Z):-
           parent(X,Z).
   predecessor3(X,Z):-
           predecessor3(Y,Z),
           parent(X,Y).
   ```

   Note how this version differs to the other versions. Try out the first four queries from question 1 using predecessor3. What answers do you get and why?

6. Using the `familyTree.pl` program, experiment with re-ordering some of the facts. For example, try the query `parent(ian,X).` and note down the outcomes. Then swap the order of some relevant facts; put `parent(ian,lucy).` above `parent(ian,peter).`. What do you notice? When the order is swapped, the declarative meaning is the same but the procedural meaning is different. Try out other queries and then retry having swapped some facts.

7. Look at the rules you added in lab exercise 1 for sibling, brother, aunt etc. Use these in further queries and work out how Prolog is finding the solutions. Trace the queries.

8. Download the program for the monkey and banana puzzle that we considered in Lecture 6 (see the COMP219 website for the program).
   Some code has been added to report what the monkey is doing. The predicate `write(X)` writes its argument to the screen and `nl` stands for `new line` and throws a line. You can run the program with

   ```
   canGet(state(door,onFloor,window,noBanana)).
   ```

   Use trace to see the execution broken down so that you understand how the program works. Why are the moves reported in reverse order?