Knowledge Representation & Reasoning: Prolog Tutorial

Merkouris Papamichail, CSD, UoC | ICS, FORTH · merkourisp@csd.uoc.gr

Based on Tutorial of Dr. Filippos Gouidis

Exercise 1: Socrates

We first load the file `socrates.lp`, with `prolog socrates.lp`.

```
% socrates.lp

man(socrates).
moral(X) :- man(X).
```

Now, we are able to ask questions:

```
man(X).
```

```
X = socrates
```

Prolog matches `X` with the *constant* `socrates`.

Another question:

```
moral(X).
```

```
X = socrates
```

Here the Prolog's inference is a bit more involved. Firstly, Prolog *cannot* find a *fact* (a rule without a body) to match our question. Thus, searches for a rule to match our question with its head. Namely, `moral(X) :- man(X).` . In order to *prove* the last formula, Prolog needs to prove the fact `man(X).` . The latter holds for `X = socrates` . This concludes the inference.

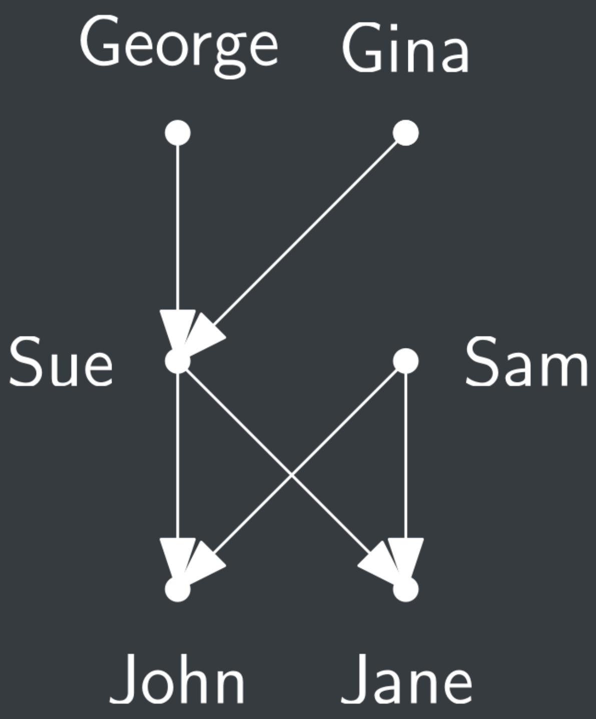
Exercise 2: Family Example

We load the file `family.lp` .

```
% family.lp

%% facts
child(john, sue).
child(jane, sue).
child(sue, george).
child(john, sam).
child(jane, sam).
child(sue, gina).

male(john).
male(sam).
male(george).

female(sue).
female(jane).
female(june).
female(gina).


%% rules
father(X, Y) :- child(Y, X), male(X).
mother(X, Y) :- child(Y, X), female(X).

parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).

grandfather(X, Y) :- father(X, Z), parent(Z, Y).
```

```
grandmother(X, Y) :- mother(X, Z), parent(Z, Y).
```

The `family.lp` describes the following family graph:



We ask some questions:

```
father(X, Y).
```

```
X = george,
Y = sue
```

```
% We use this instead of standard ";" in SWI-prolog's terminal.
% Just for this notebook.
% Please, do NOT use this in SWI-prolog's prompt.
% Use ";" instead!
jupyter:retry.
```

```
% Retrying goal: father(X,Y)
```

```
X = sam,
Y = john
```

```
% one more answer
jupyter:retry.
```

```
% Retrying goal: father(X,Y)
```

```
X = sam,
Y = jane
```

Oops.. there are no more "fathers". The reader is encouraged to verify the correctness of the above with the family graph. Let's ask another question.

```
grandmother(X, Y).
```

```
X = gina,
Y = john
```

Why does this work? Consider the rule `grandmother(X, Y) :- mother(X, Z), parent(Z, Y).`. This rule essentially translates to "X is a grand mother of Y, *if there is some* Z, such that X is mother of Z, *and* Z is a parent of Y". This is translated to the following formula:

$$(\forall X\ \forall Y\ \exists Z) \quad \mathtt{mother}(X, Z) \land \mathtt{parent}(Z, Y) \to \mathtt{grandmother}(X, Y)$$

The clear and rigorous *semantics* of Prolog is one of the greatest streanghts of the language! Above observe Prolog denotes the logical and operator $\land$ with `,`.

Understanding Disjunction in Prolog

Finally, observe that we have *two* rules defining the predicate `parent`. There is no contradiction there. Indee, lets compute all the answers of the `parent` predicate.

```
parent(X, Y).
```

```
X = george,
Y = sue
```

```
jupyter:retry.
```

```
% Retrying goal: parent(X,Y)
```

```
X = sam,
Y = john
```

```
jupyter:retry.
```

```
% Retrying goal: parent(X,Y)
```

```
X = sam,
Y = jane
```

```
jupyter:retry.
```

```
% Retrying goal: parent(X,Y)
```

```
X = sue,
Y = john
```

```
jupyter:retry.
```

```
% Retrying goal: parent(X,Y)
```

```
X = sue,
Y = jane
```

```
jupyter:retry.
```

```
% Retrying goal: parent(X,Y)
```

```
X = gina,
Y = sue
```

```
jupyter:retry.
```

```
% Retrying goal: parent(X,Y)
```

```
false
```

Disjunction in Questions

For the sake of verifing our results, consider the following question:

```
father(X, Y); mother(X, Y)
```

```
X = george,
Y = sue
```

```
jupyter:retry.
```

```
% Retrying goal: father(X,Y);mother(X,Y)
```

```
X = sam,
Y = john
```

```
jupyter:retry.
```

```
% Retrying goal: father(X,Y);mother(X,Y)
```

```
X = sam,
Y = jane
```

```
jupyter:retry.
```

```
% Retrying goal: father(X,Y);mother(X,Y)
```

```
X = sue,
Y = john
```

```
jupyter:retry.
```

```
% Retrying goal: father(X,Y);mother(X,Y)
```

```
X = sue,
Y = jane
```

```
jupyter:retry.
```

```
% Retrying goal: father(X,Y);mother(X,Y)
```

```
X = gina,
Y = sue
```

```
jupyter:retry.
```

```
% Retrying goal: father(X,Y);mother(X,Y)
```

```
false
```

Observe that the results are the same! The question `father(X, Y); mother(X, Y)` *translates* to "print all X, Y, such that, *either* X is a father of Y, *or* X is a mother of Y". Thusly, Prolog denotes the logical or operator $\lor$ with `;` .

Disjunction in Rules

From all the above, we can rewrite the `family.lp` program as follows:

```
% family-or.lp

%% facts
child(john, sue).
child(jane, sue).
child(sue, george).
child(john, sam).
child(jane, sam).
child(sue, gina).

male(john).
male(sam).
male(george).

female(sue).
female(jane).
female(june).
female(gina).


%% rules
father(X, Y) :- child(Y, X), male(X).
mother(X, Y) :- child(Y, X), female(X).


%% Observe here!
parent(X, Y) :- father(X, Y) ; mother(X, Y). % <--


grandfather(X, Y) :- father(X, Z), parent(Z, Y).
grandmother(X, Y) :- mother(X, Z), parent(Z, Y).
```

The reader is encouraged to test the equivalence between `family.lp` and `family-or.lp`.

Exercise 2: Understanding Negation

We rewrite `family.lp`:

```
% family-not.lp

person(john).
person(sam).
person(george).
person(sue).
person(jane).
person(june).
person(gina).

child(john, sue).
child(jane, sue).
child(sue, george).
child(john, sam).
child(jane, sam).
child(sue, gina).

male(john).
male(sam).
male(george).



% Negation as Failure
female(X) :- person(X), \+ male(X). % a bit too binary..



father(X, Y) :- child(Y, X), male(X).
mother(X, Y) :- child(Y, X), female(X).

parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).

grandfather(X, Y) :- father(X, Z), parent(Z, Y).
grandmother(X, Y) :- mother(X, Z), parent(Z, Y).
```

We have now remove the explicit definition for `female`, adding a rule for its computation. The rule `female(X) :- person(X), \+ male(X).` essentially says that "X is a female if X is a person, and X is *not* a male". Let's follow Prolog's inference for the `female` predicate. We use the `trace` directive.

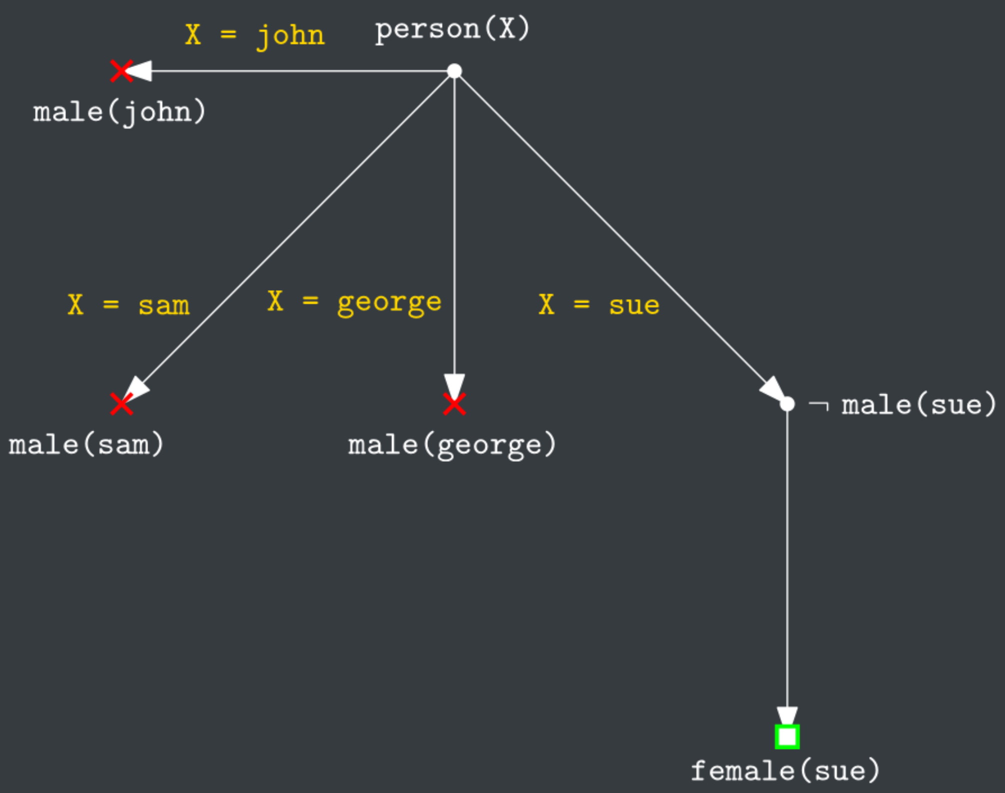
```
jupyter:trace(female(X)).
```

```
   Call: (117) female(_62812)
   Call: (118) person(_62812)
   Exit: (118) person(john)
   Call: (118) male(john)
   Exit: (118) male(john)
   Redo: (118) person(_62812)
   Exit: (118) person(sam)
   Call: (118) male(sam)
   Exit: (118) male(sam)
   Redo: (118) person(_62812)
   Exit: (118) person(george)
```

```
      Call: (118) male(george)
      Exit: (118) male(george)
      Redo: (118) person(_62812)
      Exit: (118) person(sue)
      Call: (118) male(sue)
      Fail: (118) male(sue)
      Redo: (117) female(sue)
      Exit: (117) female(sue)
```

```
X = sue
```

Observe the inference listed above. Bellow we present the inference tree consrtucted internally by prolog.



Negation as Failure

We should be very careful when using negation in prolog! The semantics of negation in Logic Programming differ from negation in First Order Logic. In particular, the Prolog negation `\+ p(X)` holds *iff* for every `X`, the inference mechanism cannot prove `p(X)`. Mathematically, this is equivalent to "if I can show that there is no proof for `p(X)`, then I have a proof for `\+ p(X)`. In classical mathematical logic, the latter does not hold. However, if one is careful in her programming, negation as failure is an extremely useful tool for writing an elegant program.

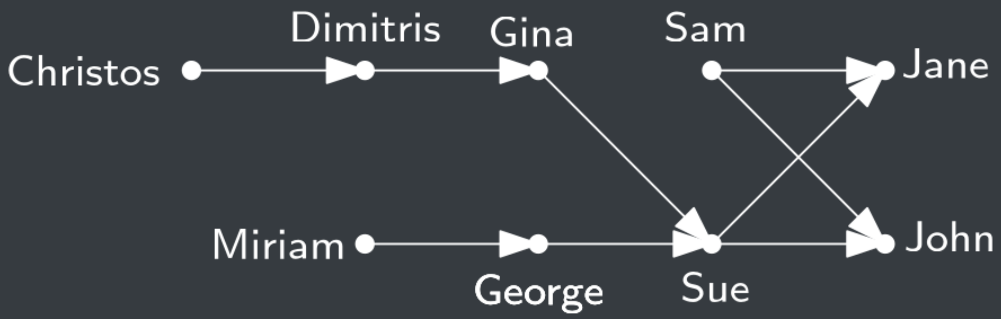
Exercise 3: Recursion

Consider the following expansion to our running `family.lp` example.

```
% family-rec.lp

person(john).
person(sam).
person(george).
person(sue).
person(jane).
person(june).
person(gina).
person(christos).
person(dimitris).
person(miriam).

child(john, sue). child(sue, gina). child(gina, christos). child(christos,
dimitris).
child(jane, sue). child(sue, george). child(georege, miriam).
child(john, sam).
child(jane, sam).

male(john).
male(sam).
male(george).
male(christos).
male(dimitris).



% Negation as Failure
female(X) :- person(X), \+ male(X).

father(X, Y) :- child(Y, X), male(X).
mother(X, Y) :- child(Y, X), female(X).
```

```
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).


grandfather(X, Y) :- father(X, Z), parent(Z, Y).
grandmother(X, Y) :- mother(X, Z), parent(Z, Y).

% Recursion
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

The family graph is extended as follows:



For computing the `ancestor` relation for graphs with *arbitrary diameter*, we need *recursion*. We say that a rule is reursive if the predicate appearing at the head of the rule, also appears at the body of the rule.

Recursion needs two steps. First, we define a *non-recursive base step*, i.e. `ancestor(X, Y) :- parent(X, Y).` . Then, we define the *recursive step*, i.e. `ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).` . Note that each recursion beginning from the latter rule, eventually will result in the base step, for it to terminate. A common mistake that leads to an infinate recursion is forgetting to write a correct rule, for every base case.

Finally, note that, contrary to imperative programming languages, recursion is the *only* way to write an iteration in prolog. Moreover, Prolog is *optimised for recursion*, thus common issues with overflowing heaps in other programming languages (e.g. Python) do not appear here.

Ordering Matters: Avoiding Infinite Loops

We close this tutorial with a remark on avoiding infinite loops. Note that up to this point, our programs *always* terminate. The only way a prolog program may not terminate, is for it to utilize recursion. A good rule o thump for a recursive rule is to use the *"less complicated terms"* earlier in the body of the rule. Prolog respects the order in which we write the predicates in the body of a rule, and tries to prove the *left most* predicate first. Thus, by writing the less complex predicates first *drastically* reduces the search space, leading to a correct execution. E.g. if we change the order of the predicates in the above recursive rule, i.e. `ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y).`, the query `ancestor(X, Y)` would *not* terminate.

Notes

- This notebook became possible by the prolog kernel for jupyter (see here). The kernel has been write as the master thesis of Anne Brecklinghaus, of University of Düsseldorf.
- For more information regarding SWI-prolog, you may visit the site: www.swi-prolog.org
- For the theoretical properties of prolog see SLD resolution, for the Prolog's inference mechanism. See negation as failure for the technicalities involved in implementing negation.
- Please, contact me for any quastions :)