

1_IntroProlog

October 18, 2022

1 Introduction to Prolog

1.0.1 Propositions

Prolog programs consist of clauses. A clause is always terminated by a dot (.). The simplest clauses are facts. Here we define two propositions to be true:

```
[1]: rains.  
     no_hat.
```

```
% Asserting clauses for user:rains/0
```

```
% Asserting clauses for user:no_hat/0
```

We can now ask the Prolog system whether it rains:

```
[2]: ?-rains.
```

```
true
```

More complicated clauses make use of the implication operator `:-`. They are also called rules. Logically they stipulate that the left-hand side of the clause must be true if the right-hand side is true. The right-hand side can contain multiple propositions separated by commas. The comma can be read as a logical conjunction (and).

```
[3]: carry_umbrella :- rains, no_hat.
```

```
% Asserting clauses for user:carry_umbrella/0
```

```
[4]: ?- carry_umbrella.
```

```
true
```

1.0.2 Predicates

Instead of propositions we can also use predicates with arguments within our clauses. The arguments to predicates denote objects for which the predicate is true. Arguments which start with an upper-case letter are logical variables. Below `X` is such a variable and it can stand for any object.

```
[5]: human(sokrates).  
     human(schopenhauer).  
     human(locke).  
  
     tiger(hobbes).  
  
     mortal(X) :- human(X).  
     mortal(X) :- animal(X).  
  
     animal(X) :- tiger(X).
```

```
% Asserting clauses for user:human/1
```

```
% Asserting clauses for user:tiger/1
```

```
% Asserting clauses for user:mortal/1
```

```
% Asserting clauses for user:animal/1
```

You can now ask questions about logical consequences of your logic program. In simple queries you provide all arguments:

```
[8]: ?-human(locke).
```

```
true
```

```
[9]: ?- human(hobbes).
```

```
false
```

```
[10]: ?- animal(hobbes).
```

```
true
```

You can also use variables in queries, and Prolog will find values for the variables so that the result is a logical consequence of you program:

```
[11]: ?- mortal(X).
```

```
X = sokrates
```

In the standard Prolog console you can type a semicolon (;) to get more answers. Here in Jupyter we need to use `jupyter:retry`.

```
[13]: jupyter:retry
```

```
% Retrying goal: mortal(X)
```

```
X = schopenhauer
```

```
[14]: jupyter:retry
```

```
% Retrying goal: mortal(X)
```

```
X = locke
```

```
[15]: jupyter:retry
```

```
% Retrying goal: mortal(X)
```

```
X = hobbes
```

```
[16]: jupyter:retry
```

```
% Retrying goal: mortal(X)
```

```
false
```

Jupyter provides a feature to compute all solutions of a goal and display them in a table:

```
[28]: jupyter:print_table(mortal(X))
```

X
sokrates
schopenhauer
locke
hobbes

```
true
```

Prolog also has a built-in predicate called `findall` which can be used to find all solutions in one go:

```
[17]: ?-findall(X,mortal(X),Results).
```

```
Results = [sokrates,schopenhauer,locke,hobbes]
```

1.0.3 Prolog lists and using append

The result is a Prolog list. Lists play an important role in Prolog and they can be written using square brackets. `[]` denotes the empty list. The built-in predicate `append` can be used to concatenate two lists:

```
[18]: ?-append([sokrates,locke],[hobbes],R).
```

```
R = [sokrates,locke,hobbes]
```

Lists can contain any kind of object, e.g., numbers but also other lists:

```
[19]: ?-append([1,2,sokrates,3],[4,[sokrates],4],Out).
```

```
Out = [1,2,sokrates,3,4,[sokrates],4]
```

One nice feature of logic programming is that the input/output relation is not pre-determined. One can run predicates backwards, meaning one can use `append` to deconstruct a list:

```
[20]: ?-append(X,Y,[1,2,3]).
```

```
X = [],
```

```
Y = [1,2,3]
```

```
[21]: jupyter:retry.
```

```
% Retrying goal: append(X,Y,[1,2,3])
```

```
X = [1],
```

```
Y = [2,3]
```

```
[22]: jupyter:retry.
```

```
% Retrying goal: append(X,Y,[1,2,3])
```

```
X = [1,2],
```

```
Y = [3]
```

```
[23]: jupyter:retry.
```

```
% Retrying goal: append(X,Y,[1,2,3])
```

```
X = [1,2,3],
```

```
Y = []
```

```
[24]: jupyter:retry.
```

```
% Retrying goal: append(X,Y,[1,2,3])
```

```
false
```

Variables can also appear multiple times in clauses or queries. Here we check if we can split a list in half:

```
[26]: ?-append(X,X,[a,b,a,b]).
```

```
X = [a,b]
```

With the underscore we indicate that we are not interested in an argument; it is an anonymous logical variable. Here we use this to find the last element of a list:

```
[27]: ?-append(_,[X],[a,b,c,d]).
```

```
X = d
```

1.0.4 Family tree example

We now load a Prolog file describing the family tree of “Game of Thrones”.

```
[39]: :- consult('prolog_files/1_got_family_tree.pl').
```

It contains facts for four basic predicates male/1, female/1, child/2 and couple/2.

```
[40]: ?-male(X).
```

```
X = Aegon V Targaryen
```

We can now find the parents of X:

```
[41]: ?-male(X),child(X,Y).
```

```
X = Aegon V Targaryen,
```

```
Y = Maekar Targaryen
```

```
[42]: jupyter:retry.
```

```
% Retrying goal: male(X),child(X,Y)
```

```
X = Aegon V Targaryen,
```

```
Y = Dyanna Dayne
```

Let us now define derived predicates for father and mother:

```
[43]: father(A,B) :- child(B,A),male(A).  
mother(A,B) :- child(B,A),female(A).
```

```
% Asserting clauses for user:father/2
```

```
% Asserting clauses for user:mother/2
```

```
[44]: ?-father(A,'Sansa Stark').
```

```
A = Eddard Stark
```

We can visualise the father/mother relationships in graphical way in Jupyter:

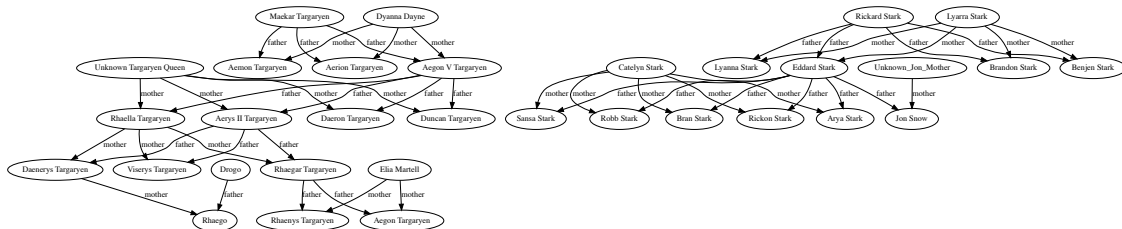
```
[45]: parent_relation(A,B,'father') :- father(A,B).
parent_relation(A,B,'mother') :- mother(A,B).
```

Previously defined clauses of user:parent_relation/3 were retracted:
:- dynamic parent_relation/3.

```
parent_relation(A, B, father) :-
    father(A, B).
parent_relation(A, B, mother) :-
    mother(A, B).
```

```
% Asserting clauses for user:parent_relation/3
```

```
[33]: jupyter:print_transition_graph(parent_relation/3, 1, 2, 3).
```



```
true
```

Let us now define the grandfather and grandmother relationships:

```
[46]: grandfather(A,B) :- child(B,C) , child(C,A) , male(A).
grandmother(A,B) :- child(B,C) , child(C,A) , female(A).
```

```
% Asserting clauses for user:grandfather/2
```

```
% Asserting clauses for user:grandmother/2
```

```
[47]: ?-grandfather(GF, 'Sansa Stark').
```

```
GF = Rickard Stark
```

Finally let us use recursion in Prolog to define arbitrary ancestors:

```
[50]: parent(A,B) :- child(B,A).  
      ancestor(A,B):- parent(A,B).  
      ancestor(A,B):- parent(C,B), ancestor(A,C).
```

```
% Asserting clauses for user:parent/2
```

```
Previously defined clauses of user:ancestor/2 were retracted:  
:- dynamic ancestor/2.
```

```
ancestor(A, B) :-  
    parent(A, B).  
ancestor(A, B) :-  
    parent(C, B),  
    ancestor(A, C).
```

```
% Asserting clauses for user:ancestor/2
```

```
[51]: ?-ancestor(GF, 'Sansa Stark').
```

```
GF = Eddard Stark
```

```
[52]: jupyter:print_table(ancestor(X, 'Sansa Stark'))
```

X
Eddard Stark
Catelyn Stark
Rickard Stark
Lyarra Stark

```
true
```

1.0.5 Send More Money Puzzle

Prolog is also a natural language to solve constraint satisfaction problems. In particular the CLP(FD) library is very useful here. It allows to solve constraints over finite domains.

```
[53]: :- use_module(library(clpfd)).
```

The library provides a few operators like `#=`, `ins` or `all_different`:

```
[55]: ?- X #= Y+Z, [Y,Z] ins 0..9.
```

```
X in 0..18,
```

```
Y in 0..9,
```

```
Z in 0..9
```

To find solutions one needs to call `labeling`:

```
[56]: X #= Y+Z, [Y,Z] ins 0..9, labeling([], [Y,Z]).
```

```
X = 0,
```

```
Y = 0,
```

```
Z = 0
```

```
[57]: X #= Y+Z, [Y,Z] ins 0..9, all_different([X,Y,Z]), labeling([], [Y,Z]).
```

```
X = 3,
```

```
Y = 1,
```

```
Z = 2
```

Let us now solve the Send More Money puzzle, where we have to find distinct digits such that this equation holds:

```
      S E N D
+     M O R E
=  M O N E Y
```

```
[61]: ?- L = [S,E,N,D,M,O,R,Y],
      L ins 0..9, % all variables are digits
      S#>0, M#>0, % S and M cannot be 1
      all_different(L), % all variables are different
      1000*S + 100*E + 10*N + D
      +
      1000*M + 100*O + 10*R + E
      #= 10000*M + 1000*O + 100*N + 10*E + Y,
      labeling([], L).
```

```
L = [9,5,6,7,1,0,8,2],
```

```
S = 9,
```

```
E = 5,
```

```
N = 6,
```

```
D = 7,
```

```
M = 1,
```

```
O = 0,
```

```
R = 8,
```

```
Y = 2
```

```
jupyter:retry
```