deerwalk
DWIT College

AI Lab : Prolog

Submitted by:
Suraj Prasai
Roll: 537
Sec: A

Submitted to:
Birodh Sir

LAB 2: Introduction To Prolog

Programming in Logic or **Prolog** is a high-level programming language that has its roots in first-order logic or first-order predicate calculus. It is a programming language commonly associated with computational linguistics and artificial intelligence and is used in expert systems, theorem proving and pattern matching over natural language parse trees and natural language processing.

Implementation:
A file “family.pl” was created with following predicates:

```
parent(abraham,homer).
parent(clancy,marge).
parent(jackie.selma).
parent(homer,bart).
parent(marge,bart).
parent(mona,homer).
parent(jackie, marge).
parent(jackie,patty).
parent(homer,lisa).
parent(marge,lisa).
```

A. Load “family.pl” and find the answer to the following questions:
(a) Is Abraham a parent of Bart?

```
?-  parent(abraham, bart).
false
```

(b) Is Lisa a child of Mona?

```
?- parent(mona, lisa).
false.
```

(c) Who are Bart’s parent?

```
?- parent(X, bart).
```

```
X = homer ;
X = marge.
```

(d) Who are Homer's children?

```
?- parent(homer, X).
X = bart ;
X = lisa.
```

B. Add the following facts to the database using only the parent predicate:
(a) Maggie is the daughter of Homer and Marge.
(b) Selma is the parent of Ling.

Added in 'family.pl':

```
        parent(homer,maggie).
        parent(marge,maggie).
        parent(selma, ling).
```

C. Find the answer to the following queries:
(a) Who are the grandchildren of Abraham?

```
?- parent(abraham,X), parent(X,Y).
X = homer,
Y = bart ;
X = homer,
Y = lisa ;
X = homer,
Y = maggie.
```

So bart, lisa and maggie are the grandchildren of Abraham.

(b) Who are the grandchildren of Clancy who have Marge as a parent?

```
?- parent(clancy,marge),parent(marge,X).
X = bart ;
X = lisa ;
X = maggie.
```

D. Augment the database with predicates to distinguish between male and female persons.

```
male(abraham).
male(homer).
male(jackie).
male(bart).
male(patty).
female(mona).
female(clancy).
female(marge).
female(selma).
female(lisa).
```

E. Query the database to find out:

(a) Who are the male children of Marge?

```
?- parent(marge,X), male(X).
X = bart ;
```

(b) Who is Lisa’s father?

```
?- parent(X,lisa), male(X).
X = homer ;
```

(c) Who is Bart’s grandfather?

```
?- parent(X,Y), parent(Y,bart), male(X).
X = abraham,
Y = homer ;
X = jackie,
Y = marge ;
```

F. Augment the database with rules and predicate for the following relations: (a) mother (b) father (c) grandfather (d) grandmother

```
father(X,Y):- male(X), parent(X,Y).
grandfather(X,Y) :- male(X), grandparent(X,Y).
mother(X,Y):- female(X), parent(X,Y).
grandmother(X,Y):- female(X), grandparent(X,Y).
```

G. Add the different relation to your database, which is true if its two arguments are not the same, and is defined as follows. Do not worry about the definition for now, it will be eventually taught.
different(X,X):-!,fail.
different(X,Y).

Added:

```
different(X,X):- !,fail.
different(X,Y).
```

H. Now, augment the database with rules and predicates for the following relations:
(i) sister: so that sister(X,Y) is true if X is the sister of Y
(ii) brother: so that brother(X,Y) is true if X is the brother of Y
(iii) aunt: so that aunt(X,Y) is true if X is the aunt of Y
(iv) uncle: so that uncle(X,Y) is true if X is the uncle of Y

(v) cousin: so that cousin(X,Y) is true if X is the cousin of Y
(vi) siblings: so that siblings(X,Y) is true if X is the cousin of Y

```
sister(X,Y):- female(X), parent(Z,X), parent(Z,Y).
brother(X,Y) :- male(X), parent(Z,X), parent(Z,Y).
aunt(X,Y):- grandparent(Z,Y), parent(Z,X), not(parent(X,Y)), female(X).
uncle(X,Y):- grandparent(Z,Y), parent(Z,X), not(parent(X,Y)), male(X).
cousin(X,Y):- aunt(Z,Y), parent(Z,X).
cousin(X,Y):- uncle(Z,Y), parent(Z,X).
siblings(X,Y) :- cousin(X,Y).
```

I. Create your own family tree. Only use the parent relation and male/female predicate. Consult your parents if needed.

```
male(gyan).
male(suraj).
female(parbati).
female(reshma).
male(gauri).
female(maya).
male(kashi).
female(tulasa).
parent(parbati, suraj).
parent(gyan, suraj).
parent(parbati, reshma).
parent(gyan, reshma).
parent(kashi, parbati).
parent(tulasa, parbati).
parent(gauri, gyan).
parent(maya, gyan).
```

J. Extra credit: Implement a rule for ancestor relation which is true if X is the ancestor of Y.

```
ancestor(X,Y):- grandparent(X,Z), parent(Z, Y).
_ancestor(X,Y):- parent(X,A), parent(A,B), parent(B,Y)
```

LAB 3: Implementing Recursion in Prolog

A directed graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are directed from one vertex to another. A directed graph is sometimes called a digraph or a directed network.
Suppose we have a following directed graph represented in terms of edge(v1, v2), such that there is an edge from vertex v1 to vertex v2.(The % represents comments in prolog file).

```
% edge list
edge('A','B').
edge('B','C').
edge('C','E').
edge('E','F').
edge('E','D').
edge('D','B').
path(X,Y):- edge(X,Somenode), path(Somenode,Y).
```

The recursive predicate path calls itself recursively until it finds a path from X to Y. If it can find it as a declarative fact in the consulted .pl file, it shows its value as true else it displays false.

We can also implement a production system in prolog. Here is an example of a monkey production system where to reach the monkey has to grab a banana. Here a monkey can walk, move the box, climb on the box, grab the banana but certain conditions have to be fulfilled like to move the box, they have to be in same location.

The production system is designed as follows:

```
% initial state of monkey
% format of predicate state(monkey_location, monkey_on_box_or_floor,
%       location_box, monkey_has_banana)
state(atdoor, onfloor, atwindow, hasnot).
```

```
% goal state
state(_,_,_,has).

% format of predicate do(current_state, action, new_state)
% Action: Grab Banana:

do(state(middle, onbox, middle, hasnot), grab, state(middle, onbox, middle, has)).

% climb the box
do(state(L, onfloor, L, Banana), climb, state(L, onbox, L, Banana)).

% push box from L1 to L2
do(state(L1, onfloor, L1, Banana), push(L1,L2), state(L2, onfloor, L2, Banana)).

% walk from L1 to L2
do(state(L1, onfloor, Box, Banana), walk, state(L2, onfloor, Box, Banana)).

% get state
canget(state(_, _, _, has)).
canget(State1):- do(State1, Move, State2), canget(State2).
```

Above file was consulted and the result of certain queries is as below:

```
?- canget(state(atwindow, onfloor, atwindow, has)).
true ;

?- canget(state(atdoor, onfloor, atwindow, hasnot)).
true .

?- canget(state(atwindow, onbox, atwindow, hasnot)).
false.
```

Tower Of Hanoi is the which consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top.

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack.
3. No disk may be placed on top of a smaller disk.

The minimal number of moves required to solve a Tower of Hanoi puzzle is $2^n - 1$, where n is the number of disks.

In prolog, it was implemented as:

```
move(1,X,Y,_):- write('Move disk from '),write(X),write(' to '),write(Y),nl.

move(N,X,Y,Z):- N>1,M is N-1, move(M,X,Z,Y), move(1,X,Y,_),
move(M,Z,Y,X).

hanoi(N) :- move(N, 'a', 'b', 'c').
```

The solution was obtained as:

```
?- hanoi(3).

Move disk from a to b

Move disk from a to c

Move disk from b to c

Move disk from a to b

Move disk from c to a

Move disk from c to b

Move disk from a to b

true .
```

LAB 5: List, its Operations and Structure in Prolog

List is a common data structure in Prolog. It always start and end with square brackets, and each of the items they contain is separated by a comma.
Prolog also has a special facility to split the first part of the list (called the **head**) away from the rest of the list (known as the **tail**). We can place a special symbol | (pronounced **'bar'**) in the list to distinguish between the first item in the list and the remaining list.
Eg:

```
?-  [Head | Tail] = [1, 2, 3, 4, 5] gives:
Head = 1,
Tail = [2, 3, 4, 5].
```

Operations on list:

1. **To cycle through elements of list, we use write as:**

```
% cycle through list
write_list([]).
write_list([Head|Tail]) :-     write(Head),  nl,    write_list(Tail).
```

Above predicate is called as:

```
?- write_list([1,2,3,4,5]).
1
2
3
4
5
True.
```

2. ?- append([1,2], [3,4] ,X).

```
X = [1, 2, 3, 4].
```

3. ?- reverse([1,2,3,4],X).

```
X = [4, 3, 2, 1].
```

```
4. ?- member(a,[a,b,c]).
true .
```

5.

```
%take nth element from list and check
take(1,[H|_],H).
take(N,[_|T],X) :- N1 is N-1,take(N1,T,X).

 ?- take(4,[4,3,2,1],1).
true .
```

6.

```
% take sum of values in list
sum([], 0).
sum([X|L], Sum) :- sum(L, SL), Sum is X + SL.

?- sum([1,2,3],X).
X = 6.
```

Here is an example of how a **DFA** machine can be made in Prolog. In the below example, we have starting state 0 and final state 1. The rules that define the transition is given by t/3, where 3 defines the number of predicates it takes as its input.

% Code snippet begin

```
t(0,a,1).
t(1,a,1).
t(2,a,2).
t(0,b,2).
t(1,b,1).
t(2,b,2).
```

```
% 0 is a starting state
% 1 is a final state
startstate(0).
finalstate(1).

% DFA to accept string

checkinput(1,_).
checkinput(Start, Input):- [Head|Tail] = Input,
    startstate(Start), t(Start, Head, NextState),
    checkinput(NextState,Tail).
```

In above recursive equation, the input list is splitted in to Head part and Tail part. If the provided state is start state and there is a transition on given input to certain state, the above recursion continues until all inputs end or the checkinput(1,_), gives true which is given when the state machine reaches its final state.

Some tests are as follows:

```
?- checkinput(0,[a,b,a,b]).
true .

?- checkinput(0,[b,b]).
false.
```

We can use trace to see what's going on:

```
[trace]  ?- checkinput(0,[a,b,a,b,a]).
   Call: (7) checkinput(0, [a, b, a, b, a]) ? creep
   Call: (8) [a, b, a, b, a]=[_G3465|_G3466] ? creep
   Exit: (8) [a, b, a, b, a]=[a, b, a, b, a] ? creep
   Call: (8) startstate(0) ? creep
   Exit: (8) startstate(0) ? creep
   Call: (8) t(0, a, _G3477) ? creep
   Exit: (8) t(0, a, 1) ? creep
   Call: (8) checkinput(1, [b, a, b, a]) ? creep
   Exit: (8) checkinput(1, [b, a, b, a]) ? creep
   Exit: (7) checkinput(0, [a, b, a, b, a]) ? creep
true .
```

We can see that after it called transition t(0,a,1) which is declared in our knowledge base, so we now reach state 1 and so above query reaches final state.

Similarly, the family tree in previous lab can be restructured as follows:

```
family(
person(homer,simpson,date(7,may,1960),works(inspector,6000)),
person(marge,simpson,date(7,may,1965),housewife),
[ person(bart,simpson,date(7,may,1967),student),
person(lisa,simpson,date(7,may,1965),student) ]).
```

Some queries on above predicate are:
Using the family predicate, implement the following relation as rules:

A. husband(X) : true if X is someone's husband

```
husband(X):-
    family(X,_,[_,_]).
```

B. wife(X) : true if X is someone's wife

```
wife(X):-    family(_,X,[_,_]).
```

C. child(X) : true if X is someone's child

```
child(X):-    family(_,_,Y), member(X,Y).
```

D. exists(Person) : true if the person is in the database

```
exists(Person):- family(Person,_,_).
exists(Person):- family(_,Person,_).
exists(Person):- family(_,_,Y), member(Person,Y).
```

Conclusion:

Prolog can be used to state facts in terms of predicates. The value of these predicates are either true or false if it is stated as a fact or if it can be inferred from other true facts. The recursive method can be used if we have to use same type of operation again but with different inputs as its field values. Furthermore with the use of lists, the common data structure of Prolog, and the operations that is possible in it, a wide range of representation can be done using Prolog.