Prolog Tutorial-3 (Advanced)

Dr A Sahu
Dept of Computer Science & Engineering
IIT Guwahati

Outline

- *Basic Concepts of Prolog: Discussed*
- Advanced Concepts of Prolog
  - Trace
  - Static/dynamic predicates, Manipulating data base,
  - Cut: Green/Red, fail
  - Prolog as its own Meta Language: findall, bagof, setof, operator
- Examples
  - Maze, Map color
  - ***Prolog Programming in Depth, Free downloadable E book from author***
- Using prolog computation in back end
  - Tic toc toe : java front end and Prolog backend
  - Communication through sockets

Trace: Call Trace in Prolog

```
% geo.pl
loc_in(atlanta, georgia).    loc_in(hosutan, texas).
loc_in(austin, texas).       loc_in(toronto, ontario).
loc_in(X,usa):-loc_in(X,georgia).
loc_in(X,usa):-loc_in(X,texas).
loc_in(X,canada):-loc_in(X,ontaria).
loc_in(X, northamerica):-  loc_in(X,canada).
loc_in(X, northamerica):-  loc_in(X,usa).
```

```
?-consult('geo.pl').
?-spy(loc_in/2)    % specify what predicate you are tracing
yes
?-trace.                        %turn on debugger
Yes
?-loc_in(toronto,canada).
**(0) CALL : loc_in(toronto,canada) ? >   <press enter>
**(1) CALL : loc_in(toronto,ontario) ? >   <press enter>
**(1) CALL : loc_in(toronto,ontario) ? >   <press enter>
**(0) CALL : loc_in(toronto,canada) ? >   <press enter>
yes
```

Trace: Call Trace in Prolog

```
?-loc_in(what,texas).

**(0) CALL : loc_in( __0085, texas ) ? >  <press enter>
**(0) EXIT : loc_in( hoston, texas ) ? >  <press enter>
What = houston ->  ;
**(0) REDO : loc_in( houston, texas ) ? >  <press enter>
**(0) EXIT : loc_in( austin, texas ) ? >  <press enter>
What = austin ->  ;
**(0) REDO : loc_in( austin, texas ) ? >  <press enter>
**(0) EXIT : loc_in( __0085, texas ) ? >  <press enter>
no
?- notrace .     % stop trace
```

Passing Function

```
square( X, Y) :- Y is X * X.
maplist( [], _, []).
maplist( [X|Tail], F, [NewX|NewTail]) :-
        G =.. [F, X, NewX],
        call( G),
        maplist(Tail, F, NewTail).
| ?- maplist([2,6,5], square, Square).
Square = [4,36,25]
yes
```

Database Manipulation

- Prolog has five basic database manipulation commands:

  - assert/1
  - asserta/1
  - assertz/1

  - retract/1
  - retractall/1

Database Manipulation

- Prolog has five basic database manipulation commands:

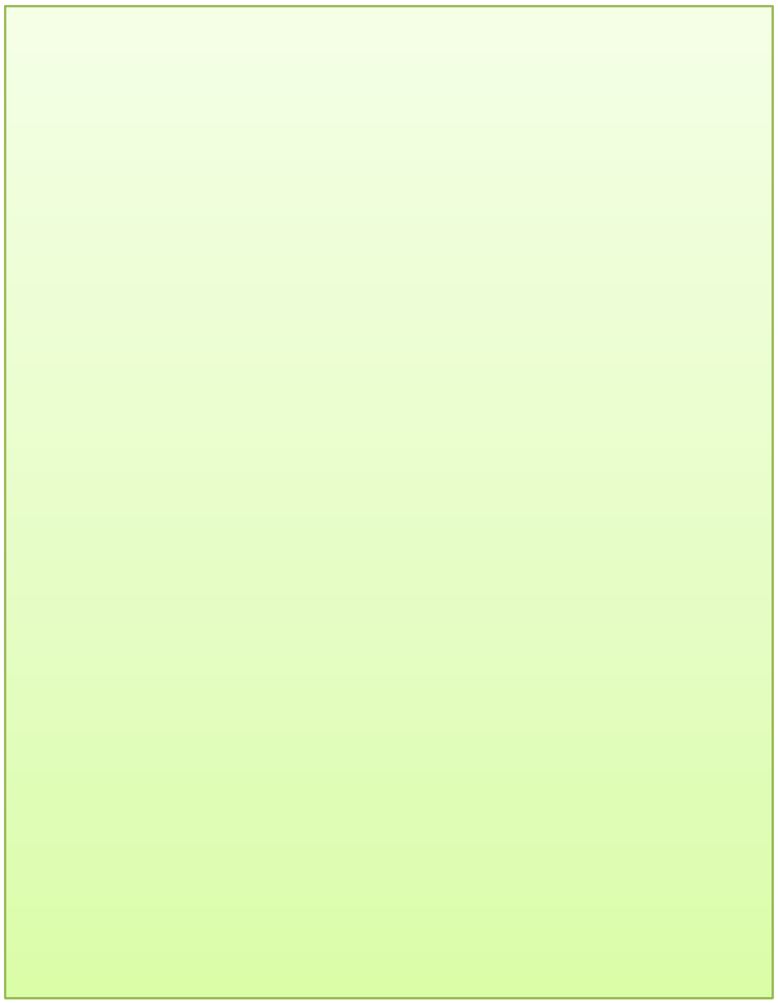
  - assert/1
  - asserta/1
  - assertz/1

  Adding information

  - retract/1
  - retractall/1

  Removing information

Start with an empty database



Start with an empty database

```
?- listing.
yes
```

Using assert/1

```
?- assert(happy(mia)).
yes
```

Using assert/1

```
happy(mia).
```

```
?- assert(happy(mia)).
yes
?-
```

Using assert/1

```
happy(mia).
```

```
?- assert(happy(mia)).
yes
?- listing.
happy(mia).
?-
```

Using assert/1

```
happy(mia).
```

```
?- assert(happy(mia)).
yes
?- listing.
happy(mia).
?- assert(happy(vincent)),
     assert(happy(marsellus)),
     assert(happy(butch)),
    assert(happy(vincent)).
```

Using assert/1

```
happy(mia).
happy(vincent).
happy(marsellus).
happy(butch).
happy(vincent).
```

```
?- assert(happy(mia)).
yes
?- listing.
happy(mia).
?- assert(happy(vincent)),
     assert(happy(marsellus)),
     assert(happy(butch)),
    assert(happy(vincent)).
yes
?-
```

Changing meaning of predicates

- The database manipulations have changed the meaning of the predicate **happy/1**
- More generally:
  - database manipulation commands give us the ability to change the meaning of predicates during runtime

Dynamic and Static Predicates

- Predicates which meaning changing during runtime are called **dynamic** predicates
  - happy/1 is a dynamic predicate
  - Some Prolog interpreters require a declaration of dynamic predicates
- Ordinary predicates are sometimes referred to as **static** predicates

Asserting rules

```
happy(mia).
happy(vincent).
happy(marsellus).
happy(butch).
happy(vincent).
```

```
?- assert( (naive(X):- happy(X)).
```

Asserting rules

```
happy(mia).
happy(vincent).
happy(marsellus).
happy(butch).
happy(vincent).

naive(A):- happy(A).
```

```
?- assert( (naive(X):- happy(X)).
yes
?-
```

Removing information

- Now we know how to add information to the Prolog database
  - We do this with the **assert**/1 predicate
- How do we remove information?
  - We do this with the **retract**/1 predicate, this will remove one clause
  - We can remove several clauses simultaneously with the **retractall**/1 predicate

Using retract/1

```
happy(mia).
happy(vincent).
happy(marsellus).
happy(butch).
happy(vincent).

naive(A):- happy(A).
```

```
?- retract(happy(marsellus)).
```

Using retract/1

```
happy(mia).
happy(vincent).
happy(butch).
happy(vincent).

naive(A):- happy(A).
```

```
?- retract(happy(marsellus)).
yes
?-
```

Using retract/1

```
happy(mia).
happy(vincent).
happy(butch).
happy(vincent).

naive(A):- happy(A).
```

```
?- retract(happy(marsellus)).
yes
?- retract(happy(vincent)).
```

Using retract/1

```
happy(mia).
happy(butch).
happy(vincent).

naive(A):- happy(A).
```

```
?- retract(happy(marsellus)).
yes
?- retract(happy(vincent)).
yes
```

Using retract/1

```
happy(mia).
happy(butch).
happy(vincent).

naive(A):- happy(A).
```

```
?- retract(happy(X)).
```

Using retract/1

```
naive(A):- happy(A).
```

```
?- retract(happy(X)).
X=mia;
X=butch;
X=vincent;
no
?-
```

Using asserta/1 and assertz/1

- If we want more control over where the asserted material is placed we can use the variants of assert/1:
  - **asserta**/1
    places asserted matieral at the **beginning** of the database
  - **assertz**/1
    places asserted material at the **end** of the database

Memoisation

- Database manipulation is a useful technique
- It is especially useful for storing the results to computations, in case we need to recalculate the same query
- This is often called **memoisation** or **caching**

Example of memoisation

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).
```

Example of memoisation

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).
```

```
?- addAndSquare(3,7,X).
```

Example of memoisation

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).

lookup(3,7,100).
```

```
?- addAndSquare(3,7,X).
X=100
yes
?-
```

Example of memoisation

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).

lookup(3,7,100).
```

```
?- addAndSquare(3,7,X).
X=100
yes
?- addAndSquare(3,4,X).
```

Example of memoisation

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).

lookup(3,7,100).
lookup(3,4,49).
```

```
?- addAndSquare(3,7,X).
X=100
yes
?- addAndSquare(3,4,X).
X=49
yes
```

Using retractall/1

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
   Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).

lookup(3,7,100).
lookup(3,4,49).
```

```
?- retractall(lookup(_, _, _)).
```

Using retractall/1

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).
```

```
?- retractall(lookup(_, _, _)).
yes
?-
```

Red and Green Cuts: precuttion of using dynamic predicates

Red cut

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).
```

If by incidence : rule 1 got removed , 2nd rule will be kind of broken rule

Red and Green Cuts

If by incidence : rule 1 got removed , 2nd rule will be still complete rule

Red cut

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).
```

Green cuts

```
:- dynamic lookup/3.

addAndSquare(X,Y,Res):-
    lookup(X,Y,Res), !.

addAndSquare(X,Y,Res):-
    \+ lookup(X,Y,Res), !,
    Res is (X+Y) * (X+Y),
    assert(lookup(X,Y,Res)).
```

Collecting solutions

- There may be many solutions to a Prolog query
- However, Prolog generates solutions one by one
- Sometimes we would like to have *all* the solutions to a query in one go
- Needless to say, it would be handy to have them in a neat, usable format

Collecting solutions

- Prolog has three built-in predicates that do this: **findall**/3, **bagof**/3 and **setof**/3
- In essence, all these predicates collect all the solutions to a query and put them into a single list
- But there are important differences between them

Consider this database

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
               descend(Z,Y).
```

```
?- descend(martha,X).
X=charlotte;
X=caroline;
X=laura;
X=rose;
no
```

findall/3

- The query

```
?- findall(O,G,L).
```

  produces a list **L** of all the objects **O** that satisfy the goal **G**

  – Always succeeds

  – Unifies **L** with empty list if **G** cannot be satisfied

findall/3

- **`findall/3`** is the most straightforward of the three, and the most commonly used:

```
| ?-findall(X, member(X,[1,2,3,4]), Results).
        Results = [1,2,3,4]
        yes
```

- This reads: `find all of the Xs, such that X is a member of the list `[1,2,3,4]` and put the list of results in `Results`'.
- Solutions in the result: Same order in which Prolog finds them.
- If there are duplicated solutions, all are included.

findall/3

- We can use `findall/3` in more sophisticated ways.
- The second argument, which is the goal, might be a compound goal:

```
| ?- findall(X, (member(X,[1,2,3,4]), X>2),
Results).
      Results = [3,4]?
      yes
```

- The first argument can be a term of any complexity:

```
|?- findall(X/Y, (member(X,[1,2,3,4]), Y is X *
X), Results).
    Results = [1/1, 2/4, 3/9, 4/16]?
    yes
```

A findall/3 example

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
                    descend(Z,Y).
```

```
?- findall(X,descend(martha,X),L).
L=[charlotte,caroline,laura,rose]
yes
```

Other findall/3 examples

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
                  descend(Z,Y).
```

```
?- findall(f:X,descend(martha,X),L).
L=[f:charlotte,f:caroline,f:laura,f:rose]
yes
```

Other findall/3 examples

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
                    descend(Z,Y).
```

```
?- findall(X,descend(rose,X),L).
L=[ ]
yes
```

Other findall/3 examples

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
               descend(Z,Y).
```

```
?- findall(d,descend(martha,X),L).
L=[d,d,d,d]
yes
```

findall/3 is sometimes rather crude

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):- child(X,Y).
descend(X,Y):- child(X,Z),
                    descend(Z,Y).
```

```
?- findall(Chi,descend(Mot,Chi),L).
L=[charlotte,caroline,laura, rose,
    caroline,laura,rose,laura,rose,rose]
yes
```

bagof/3

- The query

```
?- bagof(O,G,L).
```

  produces a list **L** of all the objects **O** that satisfy the goal **G**
  - Only succeeds if the goal **G** succeeds
  - Binds free variables in **G**

bagof/3

- that the list of results might contain duplicates, and isn't sorted.

```
| ?- bagof(Child, age(Child,Age),Results).
     Age = 5, Results = [tom,ann,ann] ? ;
     Age = 7, Results = [peter] ? ;
     Age = 8, Results = [pat] ? ;
     no
```

Bag of child with Age 5

Bag of child with Age 7

Bag of child with Age 8

```
age(peter, 7).
age(ann, 5).
age(pat, 8).
age(tom, 5).
age(ann, 5).
```

Using bagof/3

```
child(martha,charlotte).
child(charlotte,caroline).
child(caroline,laura).
child(laura,rose).

descend(X,Y):-
    child(X,Y).
descend(X,Y):-
    child(X,Z),
    descend(Z,Y).
```

```
?- bagof(X ,descend(Y,X),L).
Y=caroline
L=[laura, rose];
Y=charlotte
L=[caroline,laura,rose];
Y=laura
L=[rose];
Y=martha
L=[charlotte,caroline,laura,rose];
no
```

Bag of member satisfy descend relation

setof/3

- The query

```
?- setof(O,G,L).
```

produces a sorted list **L** of all the objects **O** that satisfy the goal **G**

  - Only succeeds if the goal **G** succeeds
  - Binds free variables in **G**
  - **Remove duplicates from L**
  - **Sorts the answers in L**

setof/3

- **`setof/3`** works very much like `findall/3`, except that:
  - It produces the ***set*** of all results, with any duplicates removed, and the results ***sorted***.
  - If any variables are used in the goal, which do not appear in the first argument, `setof/3` will return a separate result for each possible instantiation of that variable:

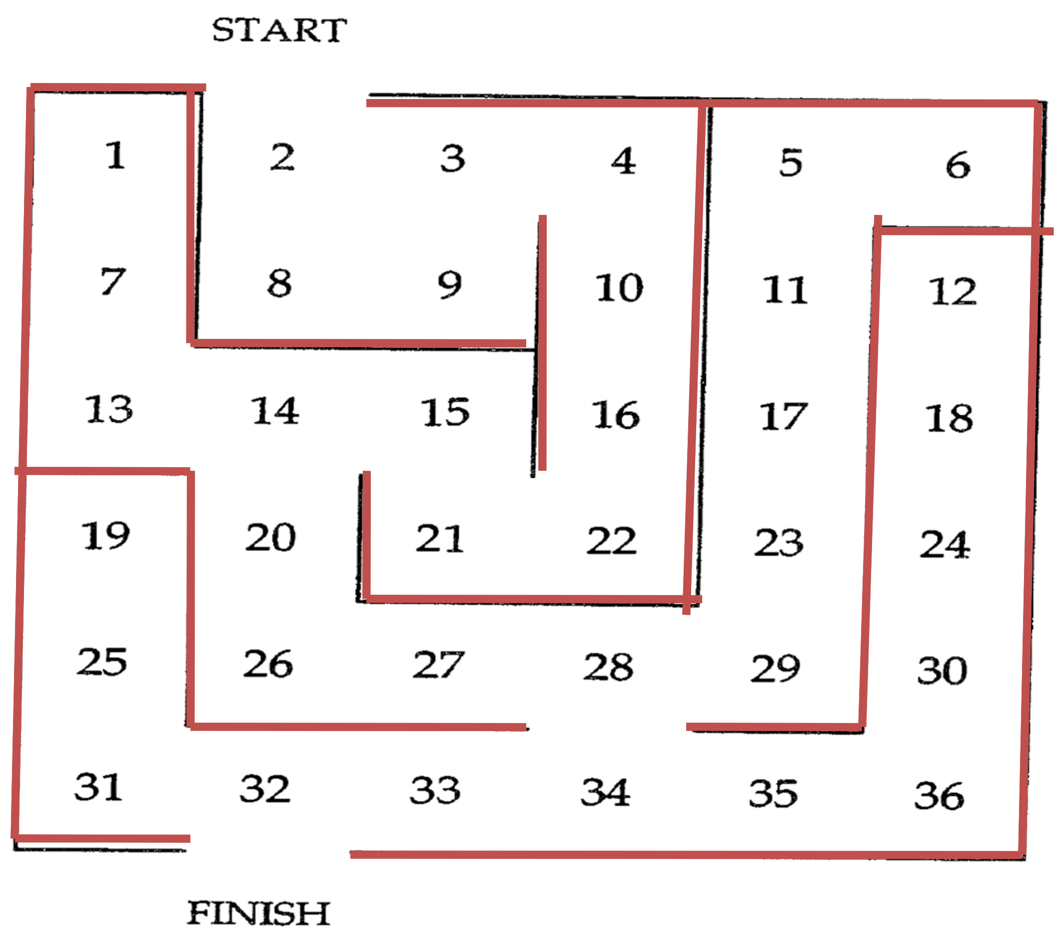
```
|?-setof(Child, age(Child,Age),Results).
    Age = 5, Results = [ann,tom] ? ;
    Age = 7, Results = [peter] ? ;
    Age = 8, Results = [pat] ? ;
          no
```

```
age(peter, 7).
age(ann, 5).
age(pat, 8).
age(tom, 5).
age(ann, 5).
```

Knowledge base

Example : Through the MAZE

- Find a path through the maze from the start to finish
- Represent maze in prolog facts
- Represent the rule



Represent maze in prolog facts

```
connect(start,2).      connect(1,7).          connect(2,8).
connect(3,4).          connect(3,9).          connect(4,10).
connect(5,11).         connect(5,6).          connect(7,13).
connect(8,9).          connect(10,16).        connect(11,17).
connect(12,18).        connect(13,14).        connect(14,15).
connect(14,20).        connect(15,21).        connect(16,22).
connect(17,23).        connect(18,24).        connect(19,25).
connect(20,26).        connect(21,22).        connect(23,29).
connect(24,30).        connect(25,31).        connect(26,27).
connect(27,28).        connect(28,29).        connect(28,34).
connect(30,36).        connect(31,32).        connect(32,33).
connect(33,34).        connect(34,35).        connect(35,36).
connect(32,finish).
```

Represent rules for MAZE

```
con_sym(Locx,Locy) :- connect(Locx,Locy).
con_sym(Locx,Locy) :- connect(Locy,Locx).


path([finish|RestOfPath],[finish|RestOfPath]).
path([CurrentLoc|RestOfPath],Solution) :-
        con_sym(CurrLoc,NextLoc),
        \+ member(NextLoc,RestOfPath),
        path([NextLoc,CurrLoc|RestOfPath],Solution).
/*  if path reaches a point where it cannot find a new position it will
    back track
    Position will be dropped off the front of the path we have built
    until we reach a a point where new position can be reached  */
solve_maze :- path([start],Solution), write(Solution).
```

Map Color

- Assigning colors to country
- No two adjacent country have same color
- Tail recursive procedure
  - List of (color country) pair made so far

Coloring a MAP: Facts

```
country(argentina).  country(bolivia). country(brazil).
country(columbia). country(chile). country(paraguay). country(peru).
country(uruguay). country(venezuela).

beside(argentina,bolivia).  beside(argentina,brazil).
beside(argentina,chile).  beside(argentina,paraguay).
beside(argentina,uruguay).  beside(bolivia,brazil).
beside(bolivia,chile). beside(bolivia,paraguay).
beside(bolivia,peru).  beside(brazil,columbia).
beside(brazil,paraguay).  beside(brazil,peru).
beside(brazil,uruguay). beside(brazil,venezuela).
beside(chile,peru). beside(columbia,peru).
beside(columbia,venezuela). beside(guyana,venezuela).
```

Rules for Map coloring

```
borders(Country,Neighbor) :-
   beside(Country,Neighbor).
borders(Country,Neighbor) :-
   beside(Neighbor,Country).


prohibited(Country,Hue,Sofar) :-
    borders(Country,Neighbor),
    member([Neighbor,Hue],Sofar).
```

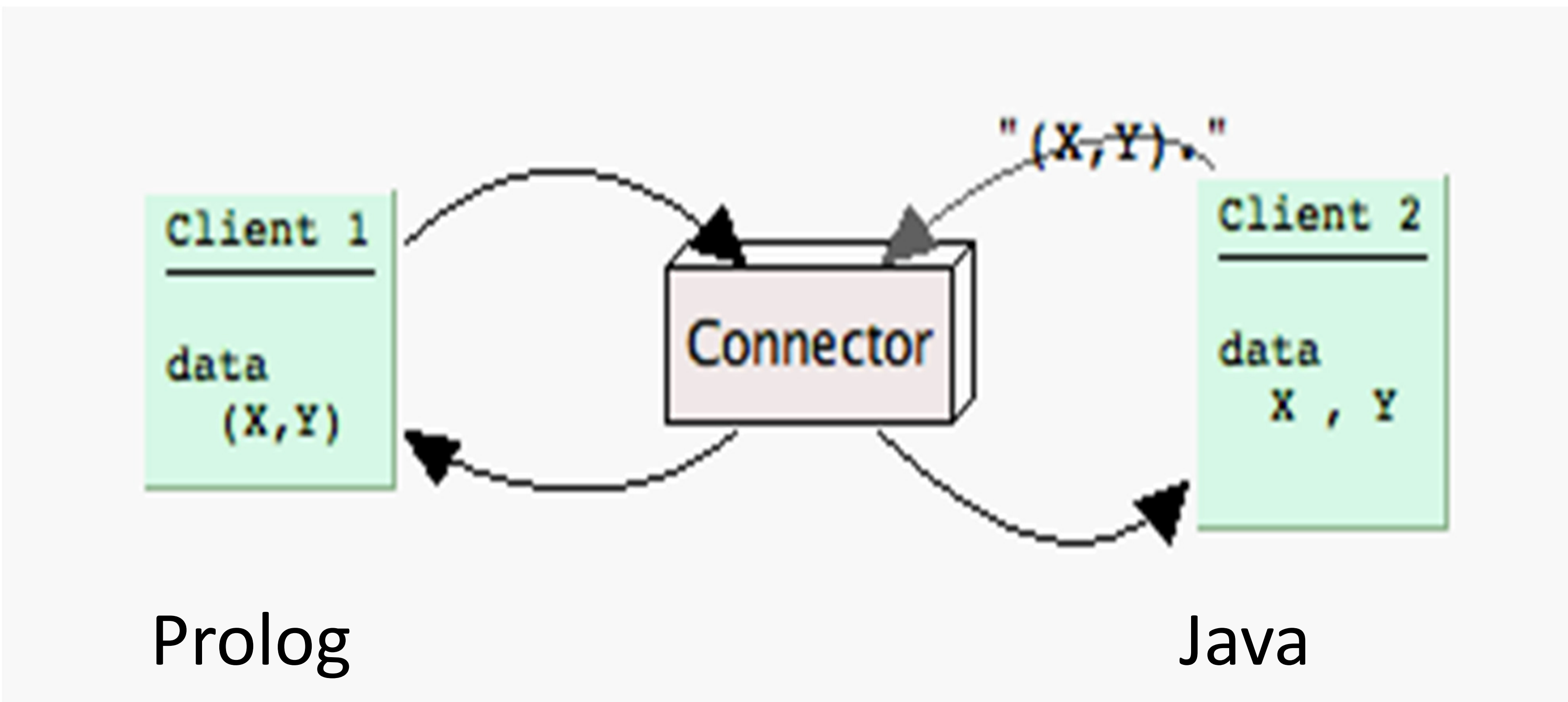
Rules for Map coloring

```
color_map(Sofar,Solution) :-
        country(Country),
         \+ member([Country,_],Sofar),
        color(Hue),
        \+ prohibited(Country,Hue,Sofar),
        color_map([[Country,Hue]|Sofar],Solution).

color_map(Solution,Solution).
```

Using Prolog computation in back end

- Tic-Tac Toe Example
- Tic toc toe : java front end and Prolog backend
- Communication through sockets



Tic-tac-toe

- Java is responsible for win_draw calculation
- Both prolog and Java maintain the database
- Moves of Java is from **User,** Prolog moves is based on computation (System move)

```
Initially board is empty
while (not (win or draw)) {
       User_ input_ his_"X"_sign()
       Calculate_win_or_draw()
       Send_Data_to_Prolog_client();
       Prolog _calculate _new_move();
       Send_back_new _move_to_JAVA_client();
       Calculate_ win_or_draw();
      }
```

Connecting through TCP Ports

```
connect(Port) :-
   tcp_socket(Socket),
   gethostname(Host),  % local host
   tcp_connect(Socket,Host:Port),   tcp_open_socket(Socket,INs,OUTs),
   assert(connectedReadStream(INs)),
   assert(connectedWriteStream(OUTs)).
:- connect(54321).  % connecting to local host port 54321
ttt :-
   connectedReadStream(IStream),
   read(IStream,(X,Y)),
   record(x,X,Y),    board(B),
   alpha_beta(o,2,B,-200,200,(U,V),_Value),     record(o,U,V),
   connectedWriteStream(OStream),    write(OStream,(U,V)),
   nl(OStream), flush_output(OStream),
   ttt.

:- ttt.          % instantiating ttt
```

Java side

- Why Java
  - Completely objected oriented
  - We will use in Concurrent programming
  - Java have good memory model (Software Transactional Memory)
  - A programming Hands on………
- Simple threading
  - Two thread to handle client1 (java prolog) and client 2 (prolog)
- Some GUI to handle from Java
- Solution and movement from Prolog

Lets see the Demo

Thanks