Ex.No :2 **SIMPLE FACT FOR THE STATEMENTS USING PROLOG**

DATE:

AIM:

To write simple fact for the statements using prolog.

ALGORITHM:

Step1: In Prolog syntax, we can write −

Understand logical programming
syntax and semantics. Design
programs in PROLOG language.

Step2: Prolog programs describe relations, defined by means of clauses. Pure

prolog is restricted to horn clauses. There are two types of clauses:
facts and rules. A rule is one of the form **Head:-Body.**
and is read as "Head is true if Body is true".

Step3: A rule's body consists of calls to predicates, which are called the rule's goals.

a. Ram like s mango.
b. Seema is a girl.
c. Bill likes Cindy.
d. Rose is red.
e. John owns gold.

PROGRAM:

```
clauses
likes(ram,mango).
girl(seema).
red(rose).
likes(bill, cindy).
owns(john,gold).
```

OUTPUT:

```
Goal queries
?- likes(ram,What).
What=mango
?- likes(Who,cindy).
Who=bill
?- red(What).
What=rose.
```

```
?- owns(Who,What).
Who=john What=gold.
```

RESULT:

Thus the simple fact for the statements using prolog was created and the output was verified successfully.

Ex.No:3 **CONVERSION FOR WRITING PREDICATES**

DATE:

AIM:

To write predicates one converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.

ALGORITHM:

Step 1: Start the program

Step 2 : Read the input of temperature in Celsius (say C)

Step 3 : F=(9*C)/5+32

Step 4 : Print temperature in fahrenheit is F

Step 5 : print the output.

PROGRAM:

Production Rules:

Arithmetic : c_to_f

f is c * 9 / 5 + 32

freezing f < = 32

Rules:

```
c_to_f(C,F):-
F is C * 9 / 5 + 32.
freezing(F) :- F =< 32.
```

OUTPUT:

```
?-
|    c_to_f(100,X).
X = 212.

?- freezing(15).
true.
```

```
?- freezing(45).
false.
```

RESULT:

Thus the predicates one to converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing was created and the output was verified successfully.

Ex.No:4 **SOLVING THE MONKEY BANANA PROBLEM**

AIM:

To write a program to solve the monkey banana problem using prolog.

ALGORITHM:

The monkey can perform the following actions:-

Step1: Walk on the floor.

Step2: Climb the box.

Step3: Push the box around(if it is beside the box).

Step4: Grasp the banana if it is standing on the box directly under the banana.

Step5: Production Rules:

canget(banana,monkey).

canreach(banana,monkey).

push(banana,monkey).

strong(monkey).

under(banana,chair).

PROGRAM:

RULES:

```
in(room,banana).

at(ceiling,banana).

strong(monkey).

grasp(monkey).

climb(monkey,chair).

push(monkey,chair):-
    strong(monkey).
```

```
under(banana,chair):-
    push(monkey,chair).
canreach(banana,monkey):-
    at(floor,banana);
    at(ceiling,banana),
    under(banana,chair),
    climb(monkey,chair).
canget(banana,monkey):-
    canreach(banana,monkey),grasp(monkey).
```

OUTPUT:

```
?-
% c:/Users/user/Documents/new4.pl compiled 0.00 sec, 13 clauses
?- canget(banana,monkey).
true.
?- canreach(banana,monkey).
true.
?- push(banana,monkey).
false.
?- strong(monkey).
true.
?- grasp(monkey).
true.
?- climb(monkey,chair).
true.
?- push(monkey,chair).
```

true.

?- under(banana,chair).

true.

RESULT:

Thus the program to solve the monkey banana problem using prolog was executed and the output was verified successfully.

Ex.No:5 IMPLEMENT FACTORIAL, FIBONACCI OF A GIVEN NUMBER

AIM:

To write a program to implement factorial, Fibonacci of a given number.

ALGORITHM:

Step1: Start

Step2: Declare variables i,a,b,show

Step3: Intialize the variables a=0,b=1, and show=0

Step4: Enter the number of terms of Fibonacci series to be printed

Step5: Print First two terms of series

Step6: Use loop for the following steps

- Show=a+=b
- a=b
- b=show
- increase value of i each time by 1
- print the value of show

Step7: End

PROGRAM:

RULES: FACTORIAL

```
factorial(0,1).

factorial(N,F) :-

N>0,

N1 is N-1,

factorial(N1,F1),

F is N * F1.
```

OUTPUT:

Goal:

?- factorial(4,X).

X=24

RULES: FIBONACCI

```
fib(0,0).

fib(X,Y) :- X>0,fib(X,Y,_).
```

```
fib(1,1,0).
fib(X,Y1,Y2) :-
X>1,
X1 is X-1,
fib(X1,Y2,Y3),
Y1 is Y2+Y3.
```

OUTPUT:

Goal:

?- fib(10,X).

X=55

RESULT:

Thus the program to implement factorial, Fibonacci of a given number using prolog was executed and the output was verified successfully.

Ex.No:6 **N QUEEN PROBLEM**

Date:

AIM:

To write a program to implement N Queen problem of the given set.

ALGORITHM:

STEP 1: Represent the board positions as 8*8 vector,i.e., (1,2,3,4,5,6,7,8).Store the set of queens in the list 'Q'.

STEP 2: Calculate the permutation of the above eight numbers stored in set P.

STEP 3: Let the position where the first queen to be placed be (1,Y), for second be (2,Y1) and so on and store the positions in Q.

STEP 4: Check for the safety of the queens through the predicate, 'noattack 0’.

STEP 5: Calculate YI-Y and Y-Y1. If both are not equal to Xdist, which is the X- distance between the first queen and others, then go to Step 6: Else go to Step 7.

STEP 6: Increment Xdist by 1.

STEP 7: Repeat above for the rest of the queens, until the end of the list is reached.

STEP 8: Print Q as answer.

STEP 9: Exit.

PROGRAM:

```
use_module(library(lists)).
n_queen(N, Solution) :-
        length(Solution, N),
        queen(Solution, N).
up2N(N,N,[N]) :-!.
up2N(K,N,[K|Tail]) :- K < N, K1 is K+1, up2N(K1, N, Tail).
queen([],_).
queen([Q|Qlist],N) :-
        queen(Qlist, N),
        up2N(1,N,Candidate_positions_for_queenQ),
        member(Q, Candidate_positions_for_queenQ),
        check_solution(Q,Qlist, 1).
check_solution(_,[], _).
check_solution(Q,[Q1|Qlist],Xdist) :-
        Q =\= Q1,
        Test is abs(Q1-Q),
        Test =\= Xdist,
        Xdist1 is Xdist + 1,
        check_solution(Q,Qlist,Xdist1).
```

OUTPUT:

?- n_queen(4,Solution).
Solution = [3, 1, 4, 2]

RESULT:

Thus the program to implement N queen problem using prolog was executed and the output was verified successfully.

Ex.No:7 **BREADTH FIRST SEARCH (BFS)**

Date:

AIM:

To write a program to solve any program using breadth first search.

ALGORITHM:

STEP 1: SET STATUS = 1 (ready state) for each node in G

STEP 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

STEP 3: Repeat Steps 4 and 5 until QUEUE is empty

STEP 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

STEP 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2

(waiting state)

[END OF LOOP]

STEP 6: EXIT

PROGRAM:

```
% breadth_first_search(+Start, +Goal, -Path)
% Breadth-first search algorithm. Finds a path from Start to Goal.
breadth_first_search(Start, Goal, Path) :-
    bfs([[Start]], Goal, RevPath),
    reverse(RevPath, Path).

% bfs(+Paths, +Goal, -Path)
% Breadth-first search implementation.
bfs([[Goal|Path]|_], Goal, [Goal|Path]).
bfs([Path|Paths], Goal, Result) :-
    Path = [Node|_],
    findall([NewNode, Node|Path],
```

```
        (edge(Node, NewNode), \+ member(NewNode, Path)),
        NewPaths),
    append(Paths, NewPaths, NextPaths),
    bfs(NextPaths, Goal, Result).

% edge(+Node1, -Node2)
% The edge/2 predicate defines the graph we are searching.
% Here, the graph is defined by the edge/2 facts.
edge(a, b).
edge(b, c).
edge(c, d).
edge(c, e).
edge(e, f).
edge(f, g).
```

OUTPUT:

```
?- breadth_first_search(a, g, Path).
Path = [a, b, c, e, f, g].
```

RESULT:

Thus the program to implement Breadth first search (BFS) using prolog was executed and the output was verified successfully.

Ex.No:6 **SOLVING WATER JUG PROBLEM**

Date:

AIM:

To write a prolog program to implement water jug problem.

ALGORITHM:

STEP 1: Fill the 4 gallon jug if x<4.

STEP 2: Fill the 3 gallon jug if x<3.

STEP 3: Pour some water out of the 4 gallon jug (if x>0).

STEP 4: Pour some water out of the 3 gallon jug (if x>0).

STEP 5: Empty the 4 gallon jug on the ground (if x>0).

STEP 6: Empty the 3 gallon jug on the ground (if y>0).

STEP 7: Pour the water from the 3 gallon jug into the 4 gallon jug until the 4 gallon jug is full (if x+y>=4 and y>0).

STEP 8: Pour the water from the 4 gallon jug into the 3 gallon jug until the 3 gallon jug is full (if x+y>=3 and x>0).

STEP 9: Pour all the water from 3 gallon jug into the 4 gallon jug (if x+y<=4 and y>0).

STEP 10: Pour all the water from 4 gallon jug into the 3 gallon jug (if x+y<=3 and x>0).

PROGRAM:

```
member(X,[X|_]).

member(X,[Y|Z]):-member(X,Z).

move(X,Y,_):-X=:=2,Y=:=0,write('done'),!.

move(X,Y,Z):-X<4,\+member((4,Y),Z),write("fill 4 jug"),nl,move(4,Y,[(4,Y)|Z]).

move(X,Y,Z):-Y<3,\+member((X,3),Z),write("fill 3 jug"),nl,move(X,3,[(X,3)|z]).

move(X,Y,Z):-X>0,\+member((0,Y),Z),write("pour 4 jug"),nl,move(0,Y,[(0,Y)|Z]).

move(X,Y,Z):-Y>0,\+member((X,0),Z),write("pour 3 jug"),nl,move(X,0,[(X,0)|Z]).

move(X,Y,Z):-P is X+Y,P>=4,Y>0,K is 4-X,M is Y-K,\+member((4,M),Z),write("pour from 3jug to 4jug"),nl,move(4,M,[(4,M)|Z]).
```

```
move(X,Y,Z):-P is X+Y,P>=3,X>0,K is 3-Y,M is X-K,\+member((M,3),Z),write("pour from 4jug to 3jug"),nl,move(M,3,[(M,3)|Z]).

move(X,Y,Z):-K is X+Y,K<4,Y>0,\+member((K,0),Z),write("pour from 3jug to 4jug"),nl,move(K,0,[(K,0)|Z]).

move(X,Y,Z):-K is X+Y,K<3,X>0,\+member((0,K),Z),write("pour from 4jug to 3jug"),nl,move(0,K,[(0,K)|Z]).
```

OUTPUT:

```
?- move(0,0,[(0,0)])

fill 4 jug
fill 3 jug
pour 4 jug
pour 3 jug
fill 4 jug
pour from 4jug to 3jug
pour 3 jug
pour from 4jug to 3jug
fill 4 jug
pour from 4jug to 3jug
pour 3 jug
done
```

RESULT:

Thus the program to implement water jug problem using prolog was executed and the output was verified successfully.

Ex.No:9 **SOLVING TRAVELLING SALESMAN PROBLEM**

Date:

AIM:

To write a program to implement travelling salesman problem.

ALGORITHM:

STEP 1: Define the list of cities to be visited.

STEP 2: Define the starting city.

STEP 3: Create a list of all possible routes that can be taken.

STEP 4: Calculate the length of each route.

STEP 5: Select the route with the shortest length.

PROGRAM:

```
/*This is the data set.*/
edge(a, b, 3).
edge(a, c, 4).
edge(a, d, 2).
edge(a, e, 7).
edge(b, c, 4).
edge(b, d, 6).
edge(b, e, 3).
edge(c, d, 5).
edge(c, e, 8).
edge(d, e, 6).
edge(b, a, 3).
edge(c, a, 4).
edge(d, a, 2).
edge(e, a, 7).
edge(c, b, 4).
edge(d, b, 6).
```

```
edge(e, b, 3).
edge(d, c, 5).
edge(e, c, 8).
edge(e, d, 6).
edge(a, h, 2).
edge(h, d, 1).

/* Finds the length of a list, while there is something in the list it increments N
when there is nothing left it returns.*/
len([], 0).
len([H|T], N):- len(T, X), N is X+1 .

/*Best path, is called by shortest_path.  It sends it the paths found in a
 path, distance format*/
best_path(Visited, Total):- path(a, a, Visited, Total).

/*Path is expanded to take in distance so far and the nodes visited */
path(Start, Fin, Visited, Total) :- path(Start, Fin, [Start], Visited, 0, Total).

/*This adds the stopping location to the visited list, adds the distance and then calls recursive
to the next stopping location along the path */
path(Start, Fin, CurrentLoc, Visited, Costn, Total) :-
    edge(Start, StopLoc, Distance), NewCostn is Costn + Distance, \+ member(StopLoc,
CurrentLoc),
    path(StopLoc, Fin, [StopLoc|CurrentLoc], Visited, NewCostn, Total).

/*When we find a path back to the starting point, make that the total distance and make
sure the graph has touch every node*/
path(Start, Fin, CurrentLoc, Visited, Costn, Total) :-
    edge(Start, Fin, Distance), reverse([Fin|CurrentLoc], Visited), len(Visited, Q),
    (Q\=7 -> Total is 100000; Total is Costn + Distance).
```

```
/*This is called to find the shortest path, takes all the paths, collects them in holder.
Then calls pick on that holder which picks the shortest path and returns it*/
shortest_path(Path):-setof(Cost-Path, best_path(Path,Cost), Holder),pick(Holder,Path).

/* Is called, compares 2 distances. If cost is smaller than bcost, no need to go on. Cut it.*/
best(Cost-Holder,Bcost-_,Cost-Holder):- Cost<Bcost,!.
best(_,X,X).

/*Takes the top path and distance off of the holder and recursively calls it.*/
pick([Cost-Holder|R],X):- pick(R,Bcost-Bholder),best(Cost-Holder,Bcost-Bholder,X),!.
pick([X],X).
```

OUTPUT:

```
?- shortest_path(Path).
Path = 20-[a, h, d, e, b, c, a].
```

RESULT:

Thus the program to implement travelling salesman problem using prolog was executed and the output was verified successfully.