DRONACHARYA
College of Engineering

LAB MANUAL OF IS

INDEX

SYLLABUS

CSE-306-F

L	*T*	*P*
-	*-*	*2*

Class Work: 25
Exam: 25
Total: 25
Duration of Exam: 3 Hrs.

1. *Study of PROLOG.*
2. *Write the following programs using PROLOG:*
3. *Write a program to solve 8-queens problem.*
4. *Solve any problem using depth first search.*
5. *Solve any problem using best first search.*
6. *Solve 8- puzzle problem using best first search.*
7. *Solve Robot (traversal) problem using means End Analysis.*
8. *Solve Traveling Salesman problem.*

Note: At least 5 to 10 more exercises to be given by the teacher concerned.

RATIONAL BEHIND THE COVERAGE

This lab is basically suited for problems that involve OBJECTS – in particular, STRUCTURED OBJECTS – and relations between them.

Prolog is a programming language centered around a small set of basic mechanisms, including

- Pattern matching
- Tree based data structuring
- Automatic back tracking.

This small set constitutes a surprisingly powerful and flexible programming framework.

Constraint logic programming(CLP), usually implemented as a part prolog system .CLP extends prolog with constraint processing which has proved in practice to be an exceptionally flexible tool for problems like SCHEDULING and LOGISTIC PLANNING

SOFTWARE REQUIREMENTS:

- Windows 98
- PROLOG

HARDWARE REQUIREMENTS:

- P-1 Processor
- 64 MB RAM
- 4 GB Hard Disk

LIST OF EXPERIMENTS

Program 1

OBJECTIVES: *STUDY OF PROLOG*

PROLOG-PROGRAMMING IN LOGIC

PROLOG stands for *Programming In Logic* – an idea that emerged in the early 1970's to use logic as programming language.The early developers of this idea included Robert Kowalski at Edinburgh (on the theoretical side),Marrten van Emden at Edinburgh (experimental demonstration) and Alian Colmerauer at Marseilles (implementation).
David D.H. Warren's efficient implementation at Edinburgh in the mid - 1970's greatly contributed to the popularity of PROLOG.

PROLOG is a programming language centered around a small set of basic mechanisms,
Including pattern matching, tree based data structuring and automatic backtracking. This
Small set constitutes a surprisingly powerful and flexible programming framework.
PROLOG is especially well suited for problems that involve objects- in particular, structured objects- and relations between them.

SYMBOLIC LANGUAGE

PROLOG is a programming language for symbolic, non-numeric computation. It is
Especially well suited for solving problems that involve objects and relations between objects.

For example, it is an easy exercise in prolog to express spatial relationship between objects, such as the blue sphere is behind the green one. It is also easy to state a more general rule: if object X is closer to the observer than object Y, and object Y is closer than Z, then X must be closer than Z. PROLOG can reason about the spatial relationships and their consistency with respect to the general rule. Features like this make PROLOG a powerful language for *Artificial Language(AI)* and non- numerical programming.

There are well-known examples of symbolic computation whose implementation in other standard languages took tens of pages of indigestible code. When the same algorithms were implemented in PROLOG, the result was a crystal-clear program easily fitting on one page.

FACTS, RULES AND QUERIES

Programming in PROLOG is accomplished by creating a database of facts and rules about objects, their properties, and their relationships to other objects. Queries then can

be posed about the objects and valid conclusions will be determined and returned by the program. Responses to user queries are determined through a form of inferencing control known as *resolution.*

FOR EXAMPLE:

a) **FACTS:**

Some facts about family relationships could be written as :

```
sister(sue,bill)
parent(ann,sam)
male(jo)
female(riya)
```

b)RULES:

To represent the general rule for grandfather, we write:

grandfather(X,Z)
parent(X,Y)
parent(Y,Z)
male(X)

c)QUERIES:

Given a database of facts and rules such as that above, we may make queries by typing after a query a symbol '?_' statements such as:

?_ parent(X,sam)
X=ann
?_grandfather(X,Y)
X=jo, Y=sam

PROLOG IN DESIGINING EPERT SYSTEMS

An expert system is a set of programs that manipulates encoded knowledge to solve problems in a specialized domain that normally requires human expertise. An expert system's knowledge is obtained from expert sources such as texts, journal articles, databases etc. and encoded in a form suitable for the system to use in its inference or reasoning processes. Once a sufficient body of expert knowledge has been acquired, it must be encoded in some form, loaded into knowledge base, then tested, and refined continually throughout the life of the system.

PROLOG serves as a powerful language in designing expert systems because of its following features:

- Use of knowledge rather than data
- Modification of the knowledge base without recompilation of the control programs.
- Capable of explaining conclusion.
- Symbolic computations resembling manipulations of natural language.
- Reason with meta-knowledge.

META PROGRAMMING

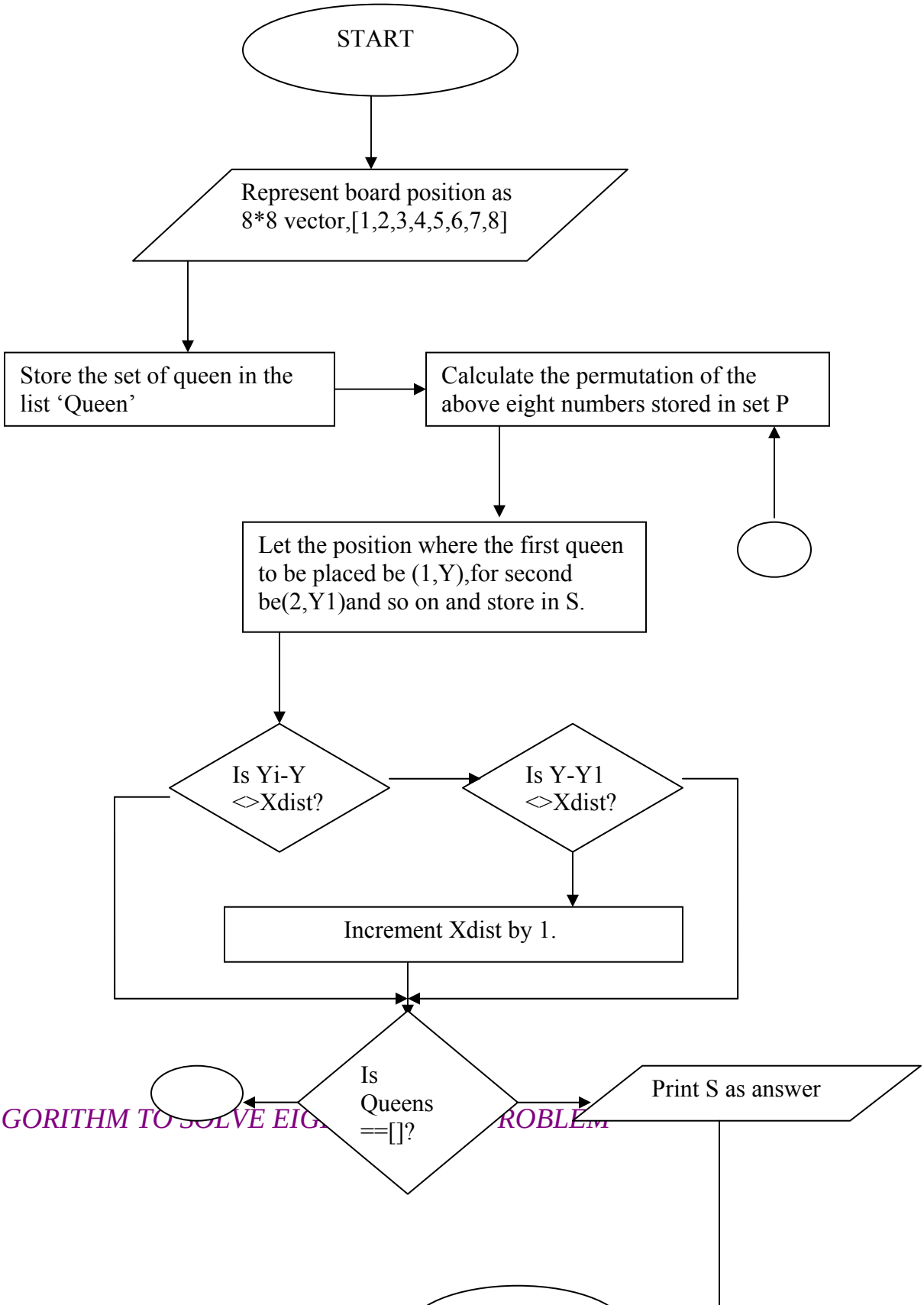
A meta program is a program that takes other programs as data. Interpreters and compilers are examples of meta-programs. Meta-interpreter is a particular kind of meta-program: an interpreter for a language written in that language. So a PROLOG mete- interpreter is an interpreter for PROLOG, itself written in PROLOG.

Due to its symbol- manipulation capabilities, PROLOG is a powerful language for meta-programming. Therefore, it is often used as an implementation language for other languages. PROLOG is particularly suitable as a language for rapid prototyping where we are interested in implementing new ideas quickly. New ideas are rapidly implemented and experimented with.

Program 2

OBJECTIVES:: *PROGRAM TO SOLVE EIGHT QUEENS PROBLEM*

FLOWCHART TO SOLVE EIGHT QUEENS PROBLEM



ALGORITHM TO SOLVE EIG ROBLEM

Start
Step 1: Represent the board position as 8*8 vector, i.e., [1,2,3,4,5,6,7.8]. Store the set of queens in the list ‘Queens’.

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 position in S.

Step 4: Check for the safety of the queens through the predicate, ‘noattack()’.

Step 5: Calculate Y1-y and Y-Y1. If both are not equal to Xdist, which is the X-distance between the first queen and others, then goto step 6 else goto 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 S as answer.

Stop

Source code :

```
domains
        H,F,I,Y,Y1,Xdist,Dist1,Queen=integer
        T,L,L1,PL,PT,Queen,Others=integer*
predicates
        safe(L)
        solution(L)
        permutation(L,L)
        del(I,L,L)
        noattack(I,L,L)
clauses
        solution(Queens):-
                permutation([1,2,3,4,5,6,7,8],Queens),
                safe(Queens).
        permutation([],[]).
        permutation([H|T],PL):-
                permutation(T,PT),
                del(H,PT,PI).
        del(I,[I|L],L).
        del(I,[F|L,[F|L1]):-
                del(I,L,L1).
        safe([]).
        safe([Queen|Others]):-
                safe(Others),
                noattack(Queen,Others,1).
        noattack(_,[],_).
        noattack(Y,[Y1|Ylist],Xdist):-
                Y1-Y<>Xdist,
                Y-Y1<>Xdist,
                Dist1=Xdist+1,
```

Noattack(Y,Ylist,Dist1).

OUTPUT:

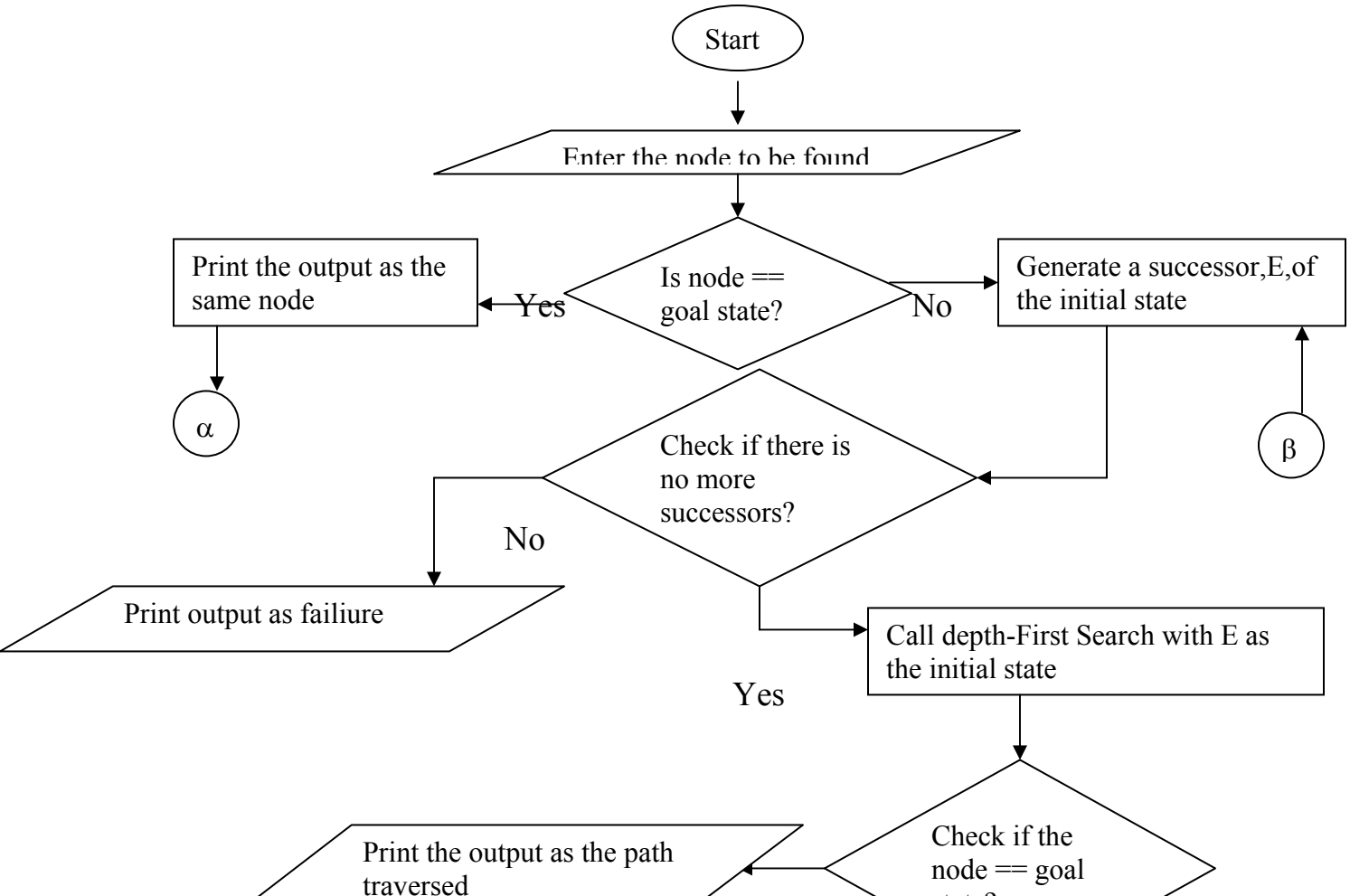
Goal:

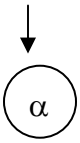
Solutions(s)

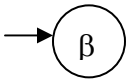
Program 3

OBJECTIVES:*:PROGRAM TO IMPLEMENT DEPTH FIRST SEARCH*

FLOWCHART TO IMPLEMENT DEPTH FIRST SEARCH







Yes

ALGORITHM TO IMPLEMENT DEPTH-FIRST SEARCH

Start
Step 1: Enter the node to be found
Step 2: If the initial state is a goal state, quit and return success
Step 3: Otherwise, do the following until success or failure is signaled.
(a) Generate a successor, E, of the initial state. If there are no more successors, signal failure
(b) Call Depth-First Search with E as the initial state.
(c) If success is retuned, signal success. Otherwise continue in this loop.

Step 4: Print the output as the path traversed
Stop

Source code :

```
domains

        A,B,X,Y = symbol
         L = symbol*


predicates
        childnode(X,Y)
        child(X,Y,L)
        path(X,Y,L)

clauses
        childnode(a,b).
        childnode(a,c).
        childnode(c,d).
        childnode(c,e).

        path(A,B,[A|L]):-
              child(A,B,L).
```

```
child(A,B,[B|[]):-
        childnode(A,B),!.

child(A,B,[X|L1]):-
        childnode(A,X),
        child(X,B,L1).
```

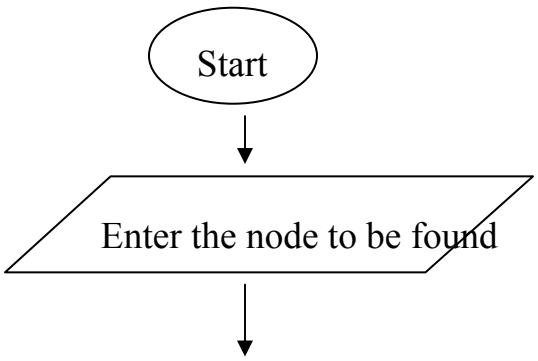
OUTPUT

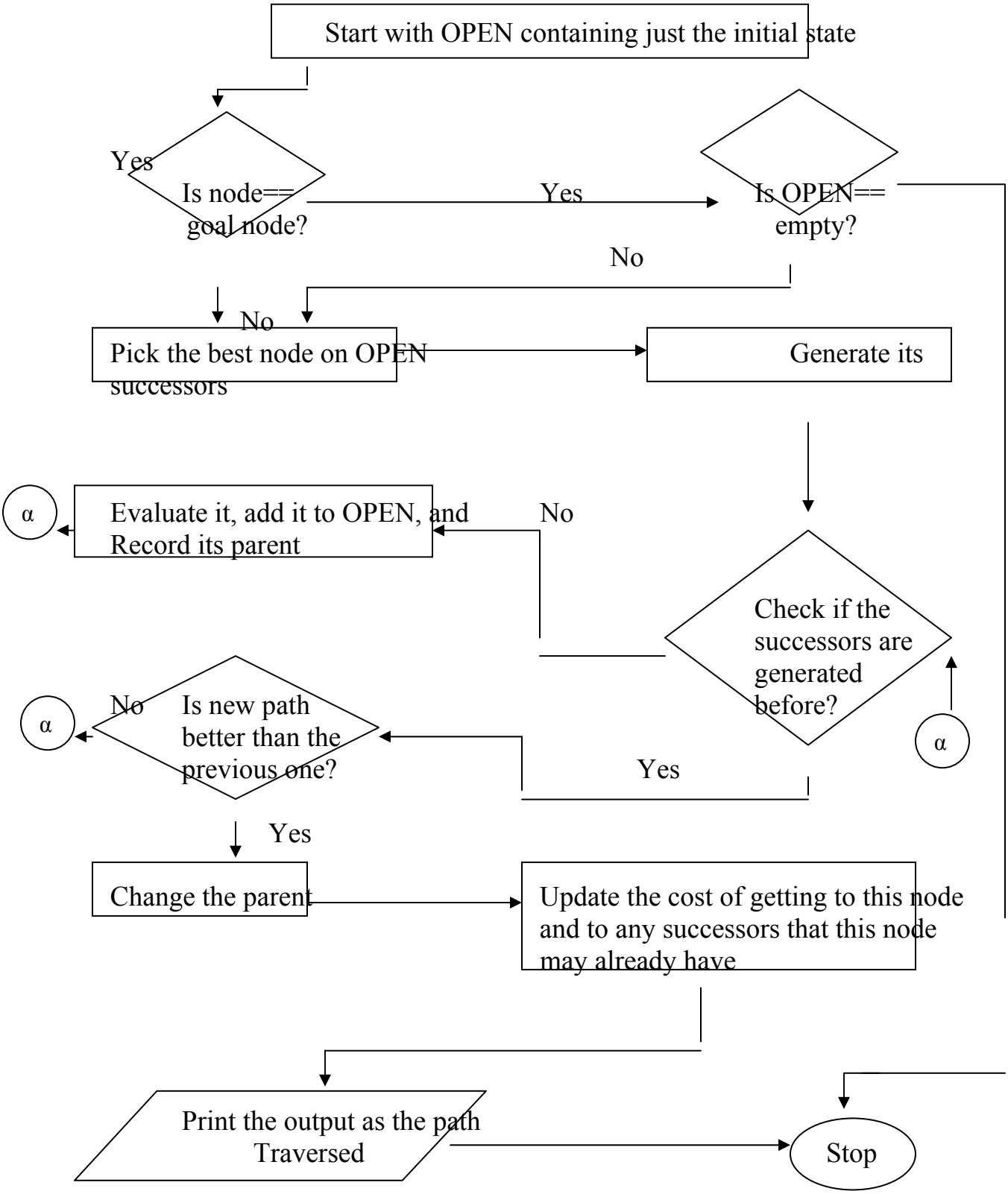
Goal : path(a,e,L)
L= [“a”,”c”,”e”]

Program 4

OBJECTIVES: *PROGRAM TO IMPLEMENT BEST FIRST SEARCH*

FLOWCHART TO IMPLEMENT BEST FIRST SEARCH




Start with OPEN containing just the initial state
Is node== goal node?
Yes
Yes
Is OPEN== empty?
No
No
Pick the best node on OPEN
Generate its successors
α
Evaluate it, add it to OPEN, and Record its parent
No
Check if the successors are generated before?
α
No
Is new path better than the previous one?
α
Yes
Yes
Change the parent
Update the cost of getting to this node and to any successors that this node may already have
Print the output as the path Traversed
Stop

ALGORITHM TO IMPLEMENT BEST FIRST SEARCH

Step 1: Enter the node to be found.
Step 2: Start with OPEN containing just the initial state.
Step 3: Until a goal is found or there are no nodes left on OPEN do:

a) Pick the best node on OPEN.
b) Generate its successors.

c) For each successor do:
    i. If it has not been generated before, evaluate it, add it to OPEN, and record its parent.
    ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

Step 4: Print the output as the path traversed.
Step 5: Exit.

Source code :

```
bestfirst(S,SL):-
        expand([],1(S,0/0),_,yes,SL).

expand(P,1(N,_),_,_,yes,[N|P]):-
        goal(N).

expand(P,1(N,F/G),B,T1,SV,SL1):-
        F=<B,
        (bagof(M/C,(s(N,M,C),not member(M,P)),SC),
        !,
        succlist(G,SC,TS),
        bestf(TS,F1),
        expand(P,t(N,F1/G,TS),B,T1,SV,SL1)
        ;
        SV=never
        ).

expand(P,t(N,F/G,[T|TS],B,T1,SV,SL1):-
        F<=B,
        bestf(TS,BF),min(B,BF,B1),
        expand([N|P],T,B1,T1,SV,S),
        continue(P,t(N,F/G,[T1|TS],B,T1,S1,S,S1).

expand(_,t(_,_,[]),_,_,never,_):- !

expand(_,T,B,T,no,_):-
        f(T,F),
        F>B.

continue(_,_,_,_,yes,yes,SL1).

continue(P,t(N,F/G,[TI|TS]),B,T1,no,SV,SL):-
        insert(T1,TS,NTS),
        bestf(NTs,F1).

expand(P,t(N,F1/G,NTS),B,T1,SV,SL).

continue(P,t(N,F1/G,[_|TS]),B,T1,never,SV,SL1):-
        bestf(TS,F1),
        expand(P,t(N,F1/G,TS),B,T1,SV,SL1).
```

```
succlist(_,[],[]).

succlist(G0,[N/C|NCS],TS):-
        G=G0+C,
        h(N,H),
        F=G+H,
        succlist(G0,NCS,TS1),
        insert(1(N,F/G),TS1,TS).

insert(T,TS,[T1|TS]):-
        F(T,F),
        bestf(TS,F1),
        F=<F1,
        !.

insert(T,[T1|TS],[T1|TS1]):-
        insert(T,TS,TS1).

f(1(_,F/_),F).

f(t(_,F/_,_),F).

bestf([T|_],F):-
        f(T,F).

bestf([],9999).

member(X,[X|T]).

member(X,[H|T]):-
        member(X,T).
```

Program 5

OBJECTIVES *Solve 8- puzzle problem using best first search.*

Clauses

```
        move(State(middle,onbox,middle,hasnot),grasp,state(middle,onbox,m
iddle,has)).

        move(State(P,onfloor,P,H),climb,state(P,onbox,P,H)).

        move(State(P1,onfloor,P1,H),push(P1,P2),state(P2,onfloor,P2,H)).



move(state(P1,onfloor,B,H),
        walk(P1,P2),                                    % Walk from
P1 to P2
        state(P2,onfloor,B,H)).

% canget(State): monkey can get banana in State.

        canget(state(_,_,_,has)).                       % can1: Monkey already
has it.

        canget(State1):-                                % can2: Do some work to
get it
                move(State1,Move,State2),               % Do something
                canget(State2).                         % Get it now
```

```
delete_all([],_,[]).

delete_all([X|L1],L2,Diff):-
        member(X,L2),!,
        delete_all(L1,L2,Diff).

delete_all([X|L1],L2,[X|Diff]):-
        delete_all(L1,L2,Diff).

% member function

member(X,[X|Tail]).

member(X,[Head|Tail]):-
        member(X,Tail).

%Concatenation function

conc([],L,L).

conc([X|L1],L2,[X|L3]):-
        conc(L1,L2,L3)
```

Program 6

OBJECTIVES *PROGRAM TO SOLVE ROBOT TRAVERSAL PROBLEM USING MEANS END ANALYSIS*

```
% plan(State,Goals,Plan,FinalState)
%The way plan is decomposed into stages by conc,the precondition
plan(Preplan) is
%found in breadth-first fashion. However, the length of the rest of plan is
not restricted
%and goals are achieved in depth-first style.
plan(State,Goals,Plan,FinalState):-
conc(Preplan,[Action|Postplan],Plan), %Divide plan
select(State,Goals,Goal), %Select goal
achieves(Action,Goal), %Relevant action
can(Action,Condition),
plan(State,Condition,Preplan,MidState1), %Enable action
apply(MidState1,Action,MidState2), %Apply action
plan(MidState2,Goals,Postplan,FinalState). %Achieve remainig goals

%satisfied(State,Goals):Goals are true in State
satisfied(State,[Goal|Goals]):-
member(Goal,State),
satisfied(State,Goals).
Select(State,Goals,Goal):-
Member(Goal,Goals),
Not member(Goal,State). %Goal not satisfied already
```

```
%achieves(Action,Goal): Goal is add-list of action

achieves(Action,Goal):-
adds(Action,Goals),
member(Goal,Goals).

%apply(State,Action,NewState):Action executed in state produces NewState

apply(State,Action,NewState):-
deletes(Action,DelList),
delete_all(State,DelList,State1),!,
adds(Action,AddList),
conc(AddList,State1,NewState).

% delete_all(L1,L2,Diff) if Diff is set-difference of L1 and L2

delete_all([],_,[]).

delete_all([X|L1],L2,Diff):-
        member(X,L2),!,
        delete_all(L1,L2,Diff).

delete_all([X|L1],L2,[X|Diff]):-
        delete_all(L1,L2,Diff).

% member function

member(X,[X|Tail]).

member(X,[Head|Tail]):-
        member(X,Tail).

%Concatenation function

conc([],L,L).

conc([X|L1],L2,[X|L3]):-
        conc(L1,L2,L3)
```

Program 7

OBJECTIVES: *IMPLEMENTATION OF TRAVELING SALESMAN PROBLEM*

This is a naïve method of solving the Travelling Salesman Problem.

```
/* tsp(Towns, Route, Distance) is true if Route is an optimal solution of  */
/*   length Distance to the Travelling Salesman Problem for the Towns,     */
/*   where the distances between towns are defined by distance/3.          */
/*   An exhaustive search is performed using the database.  The distance   */
/*   is calculated incrementally for each route.                           */
/* e.g. tsp([a,b,c,d,e,f,g,h], Route, Distance)                            */


tsp(Towns, _, _):-
  retract_all(bestroute(_)),
  assert(bestroute(r([], 2147483647))),
```

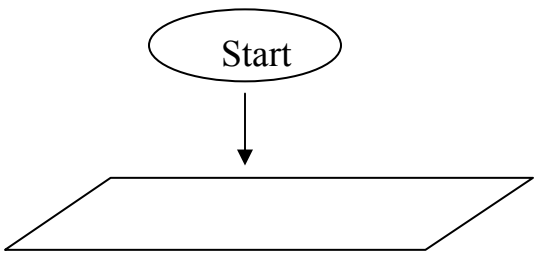
```
  route(Towns, Route, Distance),
  bestroute(r(_, BestSoFar)),
  Distance < BestSoFar,
  retract(bestroute(r(_, BestSoFar))),
  assert(bestroute(r(Route, Distance))),
  fail.
tsp(_, Route, Distance):-
  retract(bestroute(r(Route, Distance))), !.

/* route([Town|OtherTowns], Route, Distance) is true if Route starts at    */
/*   Town and goes through all the OtherTowns exactly once, and Distance
*/
/*   is the length of the Route (including returning to Town from the last */
/*   OtherTown) as defined by distance/3.                                  */
route([First|Towns], [First|Route], Distance):-
  route_1(Towns, First, First, 0, Distance, Route).

route_1([], Last, First, Distance0, Distance, []):-
  distance(Last, First, Distance1),
  Distance is Distance0 + Distance1.
route_1(Towns0, Town0, First, Distance0, Distance, [Town|Towns]):-
  remove(Town, Towns0, Towns1),
  distance(Town0, Town, Distance1),
  Distance2 is Distance0 + Distance1,
  route_1(Towns1, Town, First, Distance2, Distance, Towns).

distance(X, Y, D):-X @< Y, !, e(X, Y, D).
distance(X, Y, D):-e(Y, X, D).

retract_all(X):-retract(X), retract_all(X).
retract_all(X).

/*
 * Data: e(From,To,Distance) where From @< To
 */
e(a,b,11). e(a,c,41). e(a,d,27). e(a,e,23). e(a,f,43). e(a,g,15). e(a,h,20).
e(b,c,32). e(b,d,16). e(b,e,21). e(b,f,33). e(b,g, 7). e(b,h,13).
e(c,d,25). e(c,e,49). e(c,f,35). e(c,g,34). e(c,h,21).
e(d,e,26). e(d,f,18). e(d,g,14). e(d,h,19).
e(e,f,31). e(e,g,15). e(e,h,34).
```

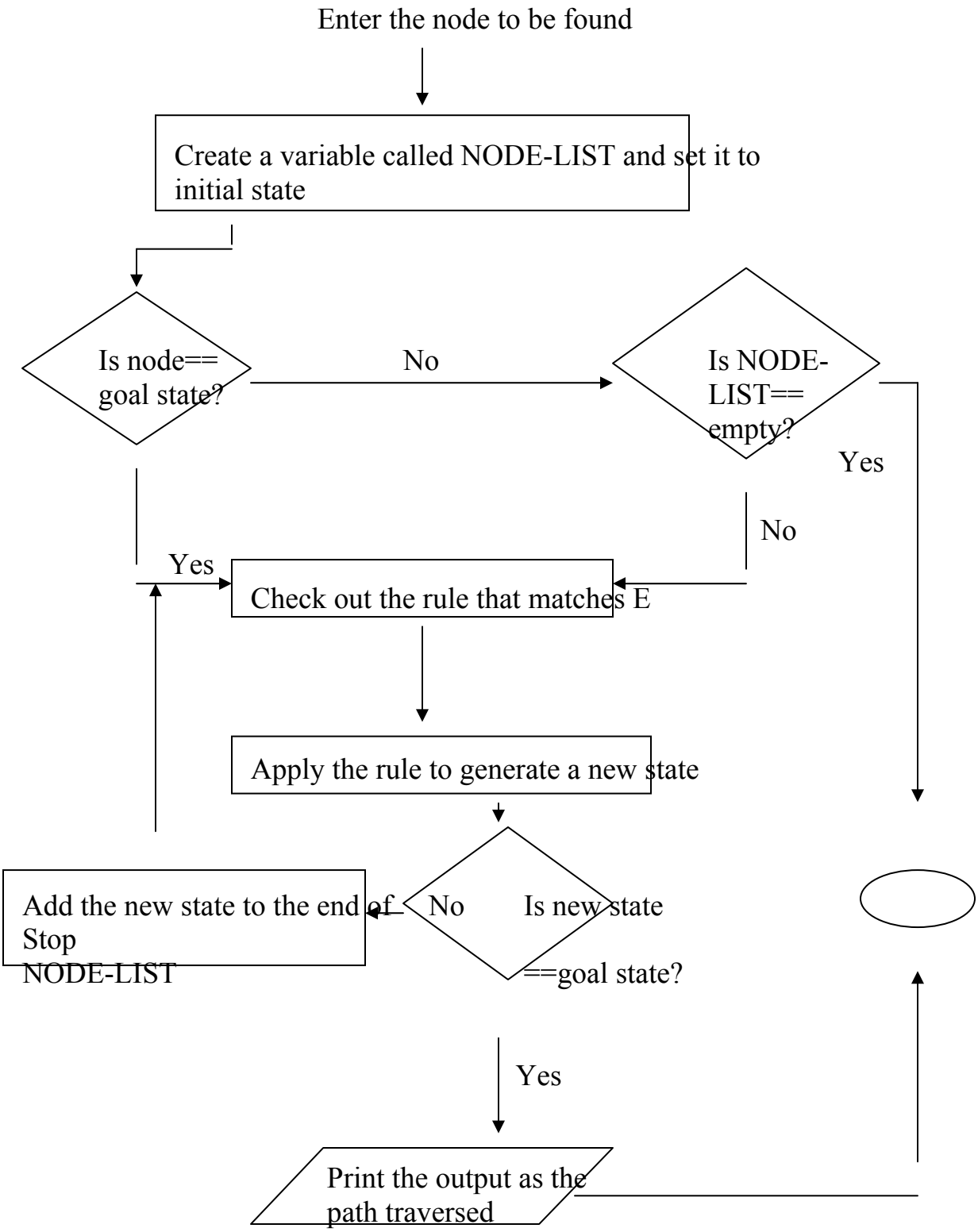
e(f,g,28). e(f,h,36).
e(g,h,19).

EXTRA PROGRAMS**:**

Program 1

OBJECTIVES: *PROGRAM TO IMPLEMENT BREADTH FIRST SEARCH*




Enter the node to be found
Create a variable called NODE-LIST and set it to initial state
Is node== goal state?
No
Is NODE-LIST== empty?
Yes
No
Yes
Check out the rule that matches E
Apply the rule to generate a new state
Add the new state to the end of NODE-LIST
No
Is new state ==goal state?
Stop
Yes
Print the output as the path traversed

ALGORITHM TO IMPLEMENT BREADTH FIRST SEARCH

Step 1:Enter the node to be found

Step 2: Create a variable called NODE-LIST and set it to the initial state

Step 3: Until a goal state is found or NODE-LIST is empty do:

a) Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty,quit.

b) For each way that each rule can match the state described in E do:

1)Apply the rule to generate a new state

2)If the new state is goal state,quit and return this state.

3)Otherwise ,add the new state to the end of NODE-LIST.

Step 4: Print the output as the path traversed

Step 5: Exit

Source code

```
domains
        X,H,N,ND = symbol
        P,L,T,Z,Z1,L1,L2,L3,PS,NP,ST,SOL= symbol*
```

```
predicates
        solve (L,L)
        extend (L,L)
        conc (X,L,L)
        breadthfirst (L,L)

clauses

       solve (S,SL):-
                breadthfirst ([[S]| Z],Z,SL).

      breadthfirst ( [ [ N|P ] | _ ] , _, [ N|P ]):-
                Goal(N).

      breadthfirst([P |PS ],Z,SL):-
                extend(P,NP),
                conc(NP ,Z1, Z),
                P\==Z1,
                breathfirst (P, Z1, SL).

extend( [N|P],NP):-
                Bagof( [ND,N |P],(s(N,ND), notmember (ND,[N|P])) ,NP), !.

extend(P,[ ]).

conc([ ],L ,L).

conc ([X| L1],L2, [X|L3]):-
                Conc(L1 ,L2 ,L3).

member(X,[ X|T]).

member(X,[H|T]):-
                member(X,T).
```

OUTPUT

Goal : solve([a,e],S)

S = [“a” ,”b”,”c” ,”e”]

Goal : solve ([a,h],S)

S = [“a”, “b” ,“c”, “d” ,”e” ,“f” ,”g”, “h”]

Program 2

OBJECTIVES: *PROGRAM TO FIND OUT UNION AND INTERSECTION OF TWO LISTS.*

ALGORITHM TO FIND OUT UNION OF TWO LISTS.

Step 1. Start
Step 2. If first list is empty return second list as the union of two lists and goto step 6.
Step 3. If head of first list is not a member of second list add this element in the union of both list produced by calling union function with and goto step 5.
Step 4. If head of first list is a member of second list call the union function for tail of first list and second list
Step 5. Exit

ALGORITHM TO FIND OUT INTERSECTION OF TWO LISTS.

Step 1. Start
Step 2. If first list is empty return empty list as intersection of both lists.
Step 3. If head of first list is a member of second list then add the head to the list obtained by calling intersection function with tail of first list and second list and goto step 5.
Step 4. If head of first list is not a member of second list then call intersection function with tail of first list and second list.
Step 5. Exit

Source code

```
domains
X = symbol*
Y = symbol

predicates
union(X,X,X)
inters(X,X,X)
member(Y,X)

clauses
union([],L,L).
union([H|T],L,[H|U]):-
        union(T,L,U),
        not(member(H,U)),!.
union([_|T],L,U):-
        union(T,L,U).

inters([],_,[]).
inters([H|T],L,[H|Lr]):-
        member(H,L),
        inters(T,L,Lr),!.
inters([H|T],L,Lr):-
        not(member(H,L)),
        inters(T,L,Lr).

member(H,[H|_]).
member(H,[_|L]):-
        member(H,L).
```

OUTPUT

```
Goal: union([a,b,c],[c,d,e],X)
X=["a","b","c","d","e"]
1 solution

Goal: intersection([a,b,c],[c,d,e],X)
X=["c"]
1 solution
```

Program 3

OBJECTIVES: *PROGRAM TO CALCULATE FACTORIAL OF A GIVEN NUMBER*

ALGORITHM TO CALCULATE FACTORIAL OF A GIVEN NUMBER

Step 1. Start
Step 2. If the number is 0 then return 1 and goto step 5.
Step 3. Call the function with a number one less than the original number
Step 4. Return product of number received in step 3 and number itself as factorial
Step 5. Exit

Source code

```
domains
X = integer

predicates
fact(X,X)

clauses
fact(0,1):-!.
fact(No,Fac):-
        N = No -1,
        Fact(N,F),
        Fac = No*F.
```

OUTPUT:

```
Fact(5,X)

X=120
```

Program 4

OBJECTIVES: *PROGRAM TO FLATTEN A LIST*

ALGORITHM TO FLATTEN A LIST

Start

STEP 1: Obtain the given list as L.

STEP 2 Let H be the Head abd T be the Tail of the list L

STEP 3: Let list be the null list then
The resulted list is null list and cut

STEP 4: If H is head of the list and an atom,then
Write it to the resulted list and cut.

STEP 5: Else divide H the list head into two parts H1 and T1
And write H1 to the resulted list and repeat from step 2 for tail T1.

STEP 6: Repeat from step 2 for the tail T of the list.

STEP 7: Print the resulted list and Exit.

Stop

Source code ;

```
domains
        L=symbol*

predicates
        display(L)
        display1(L)

clauses
        display([ ]):-!.

        display([H|T]):-
                Display1(H),
                   Display(T).



        dispaly1([ ]):-!.
        display1(H):-write(H).
         display 1([H|T]):-
                write(H).
                tab(1),
                  display1(T).
```

OUTPUT

Goal: display(a,[b,c,d],m)

'a','b','c','d','m'

Program 5

OBJECTIVES:*PROGRAM TO SOLVE MONEY BANANA PROBLEM*

```
%move(State1,move,State2):making move in State1 resuklts in State2;
% a state is represented by the term:
% state (Monkey Horizontal,Monkey Vertical,Box Position,HasBanana).


Domains

        State1,State2,MH,MV,BP,HB,P1,P2=symbol
        Move=symbol*

Predicates

        move(State1,Move,State2).
        state(MH,MV,BP,HB).
        push(P1,P2).
        walk(P1,P2).
        grasp.
        climb.

Clauses
```

```
        move(State(middle,onbox,middle,hasnot),grasp,state(middle,onbox,m
iddle,has)).

        move(State(P,onfloor,P,H),climb,state(P,onbox,P,H)).

        move(State(P1,onfloor,P1,H),push(P1,P2),state(P2,onfloor,P2,H)).



move(state(P1,onfloor,B,H),
        walk(P1,P2),                                    % Walk from
P1 to P2
        state(P2,onfloor,B,H)).

% canget(State): monkey can get banana in State.

        canget(state(_,_,_,has)).                       % can1: Monkey already
has it.

        canget(State1):-                                % can2: Do some work to
get it
                move(State1,Move,State2),               % Do something
                canget(State2).                         % Get it now
```

REFRENCES:

1. PROLOG PROGRAMMING FOR AI BY: IVAN BRATKO

2. ARTIFICIAL INTELLIGENCE : ELAIN RICH & KEVIN KNIGHT

3. A MODERN APPROACH : STAURT RUSSEL &PETER NORVING

NEW IDEAS /EXPERIMENTS

Apart from university syllabus following programs are performed by students to upgrade their knowledge in prolog programming::

1) program to find intersection of two lists.

2) program to find union of two lists.

3) program to flatten a list.

4) program to find a factorial of a number.

5) program to solve problem using breadth first search.

6) program to solve Monkey Banana Problem.

FREQUENTLY ASKED QUESTIONS

Q 1) What is prolog?

Q 2) What is the structure of a prolog program?

Q 3) What are the different data types in prolog?

Q 4) What are the sections in the prolog program?

Q 5) How to ask queries in prolog explain using examples?

Q 6) Difference between rule and fact?

Q 7)Difference between variable and constants representation in prolog?

Q 8) What is a list? How it can be manipulated?

Q 9)What is the logic to insert a element in a list?

Q10)What is the logic to delete a element in a list?

Q11) What is the logic to find the product of the elements in the given list?

Q12) What is the logic tyo find the largest number in the given list?

Q13)What is cut?

Q14)What are the uses of cut?

Q15)What are the advantages and disadvantages of cut?

Q16)What do you mean by fail predicate?

Q17) What are the advantages and disadvantages of fail?

Q18) What is difference between fail predicate and cut predicate?

Q19)What is anonymous variable and how it is represented?

Q20)What do you mean by bagoff?

Q21)What are bound and free variables in prolog?

Q22)What is the logic for concatenation of two list?

Q23)How will you check whether the list is a sublist of another list or not?

Q24)How will you declare a list?

Q25)Explain the difference between depth first search and breadth first search?

Q26)What is monkey banana problem?

Q27)What are the features of prolog?

Q28)What are the various built in predicates in prolog?

Q29)Difference between prolog and lisp?

Q30)What is expert system?Also tell its characterstics?