LAB MANUAL
FOR
IS LAB

WCTM

STUDY OF PROLOG

Prolog – Programming in Logic

PROLOG stands for *Programming In Logic* – an idea that emerged in the early 1970s 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 Alain 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 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 indestible code . When the same algorithms were implemented in PROLOG, the result was a cryetal-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:

1. **Facts :** Some facts about family relationships could be written as :

```
sister(sue, bill)
parent(ann, sam)
```


WCTM

```
parent(joe,ann)
male(joe)
female(ann)
```

2. **Rules :** To represent the general rule for grandfather , we write :

```
Grandfather(X,Z):-
        parent(X,Y),
        parent(Y,Z),
        male(X).
```

3. Queries : Given a data of facts and rules such as that above, we mat make queries by tying after a query symbol '?_' statements such as :

```
?_parent(X,sam)
X=ann
?_male(joe)
yes
?_grandfather(X,Y)
X=joe, Y=sam
?_female(joe)
no
```

PROLOG in Designing Expert 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 the expert sources such as texts, journals articles,. databases etc. and coded 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 knowledge rather than data.
- Modification of the knowledge base without recompilation of control programs.
- Capable of explaing conclusions.
- Symbolic computations resembling manipulations of natural language.
- Reason with meta-knowledge.

META-PROGRAMMING

A meta-program is a program that 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 meta-interpreter is an interpreter for PROLOG, itself written in PROLOG.

WCTM

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. In prototyping the emphasis is on bringing new ideas to life quickly and cheaply, so that they can be immediately tested.

On the other hand, there is not much emphasis on efficiency of implementation. Once the ideas are developed, a prototype may have to be re-implemented, possibly in another, more efficient programming language. Even if this is necessary, the prototype is useful because it usually helps to speed up the creative development stage.

WCTM

ALGORITHM TO SOLVE EIGHT QUEENS PROBLEM

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 ()'.

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 go to Step 6 else go to Step 7.

STEP 6 : Increment Xdist by 1.

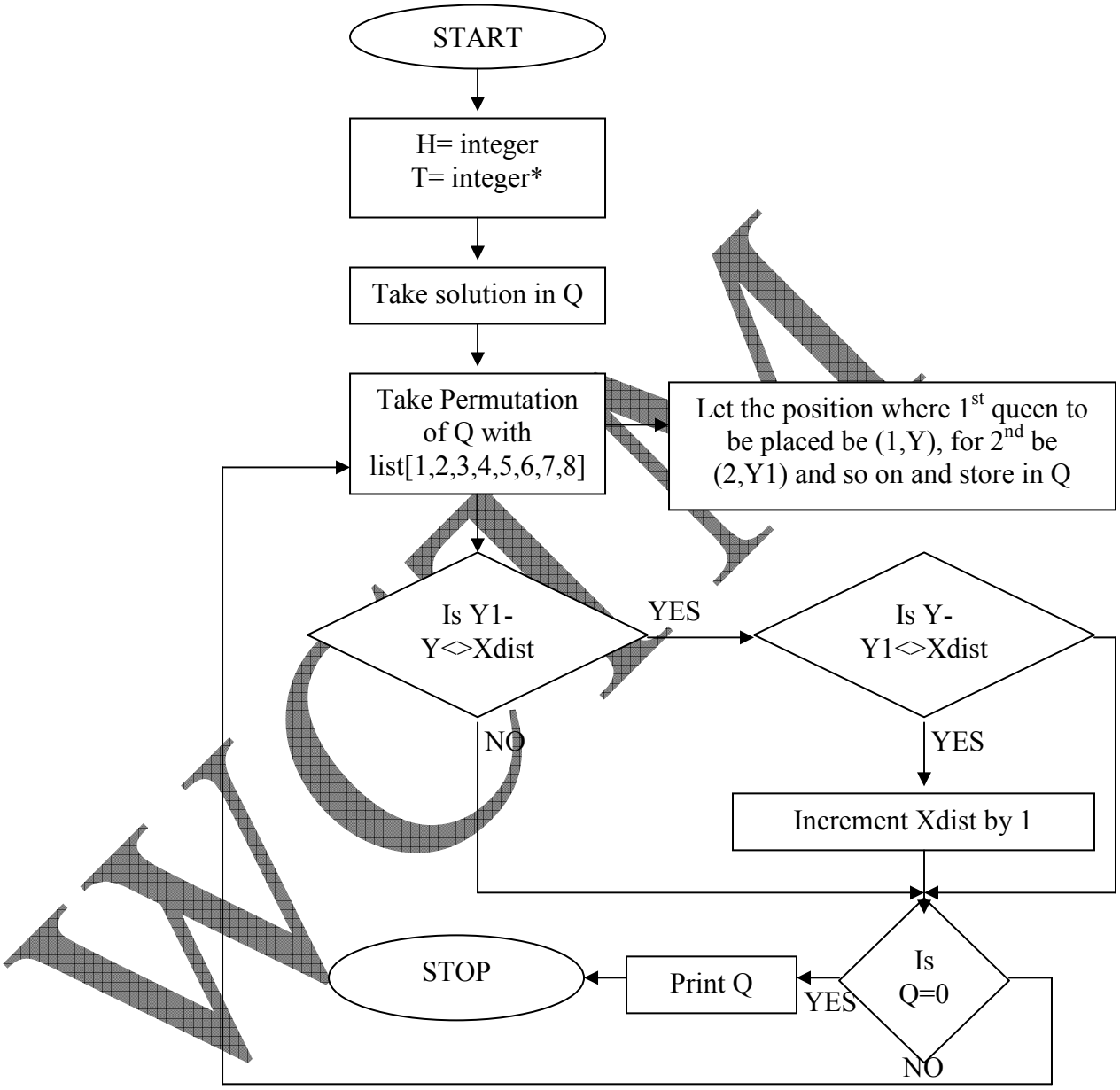
STEP 7 : Repeat above for the rest of the queens , until the end of the list is reached .

STEP 8 : Print Q as answer .

STEP 9 : Exit.

WCTM

FLOWCHART FOR 8 QUEEN'S PROBLEM



PROGRAM FOR 8 QUEEN'S PROBLEM

```
domains
        H=integer
        T=integer*

predicates
        safe(T)
        solution(T)
        permutation(T,T)
        del(H,T,T)
        noattack(H,T,H)

clauses
        del(I,[I|L],L).          /*to take a position from the permutation of list*/
        del(I,[F|L],[F|L1]):-
                del(I,L,L1).

        permutation([],[]).      /*to find the possible positions*/
        permutation([H|T],PL):-
                permutation(T,PT),\
                del(H,PL,PT).

        solution(Q):-            /*final solution is stored in Q*/
                permutation([1,2,3,4,5,6,7,8],Q),
                safe(Q).

        safe([]).                /*Q is safe such that no queens attack each other*/
        safe([Q|others]):-
                safe(others),
                noattack(Q,others,1).

        noattack(_,[],_).        /*to find if the queens are in same row, column or
        diagonal*/
        noattack(Y,[Y1|Ydist],Xdist):-
                Y1-Y<>Xdist,
                Y-Y1<>Xdist,
                dist1=Xdist,
                noattack(Y,Ydist,dist1).
```

WCTM

OUTPUT:-

goal:-solution(Q).
Q=[“3”,”8”,”4”,”7”,”1”,”6”,2”,”5”]

WCTM

ALGORITHM TO IMPLEMENT DEPTH FIRST SEARCH

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.

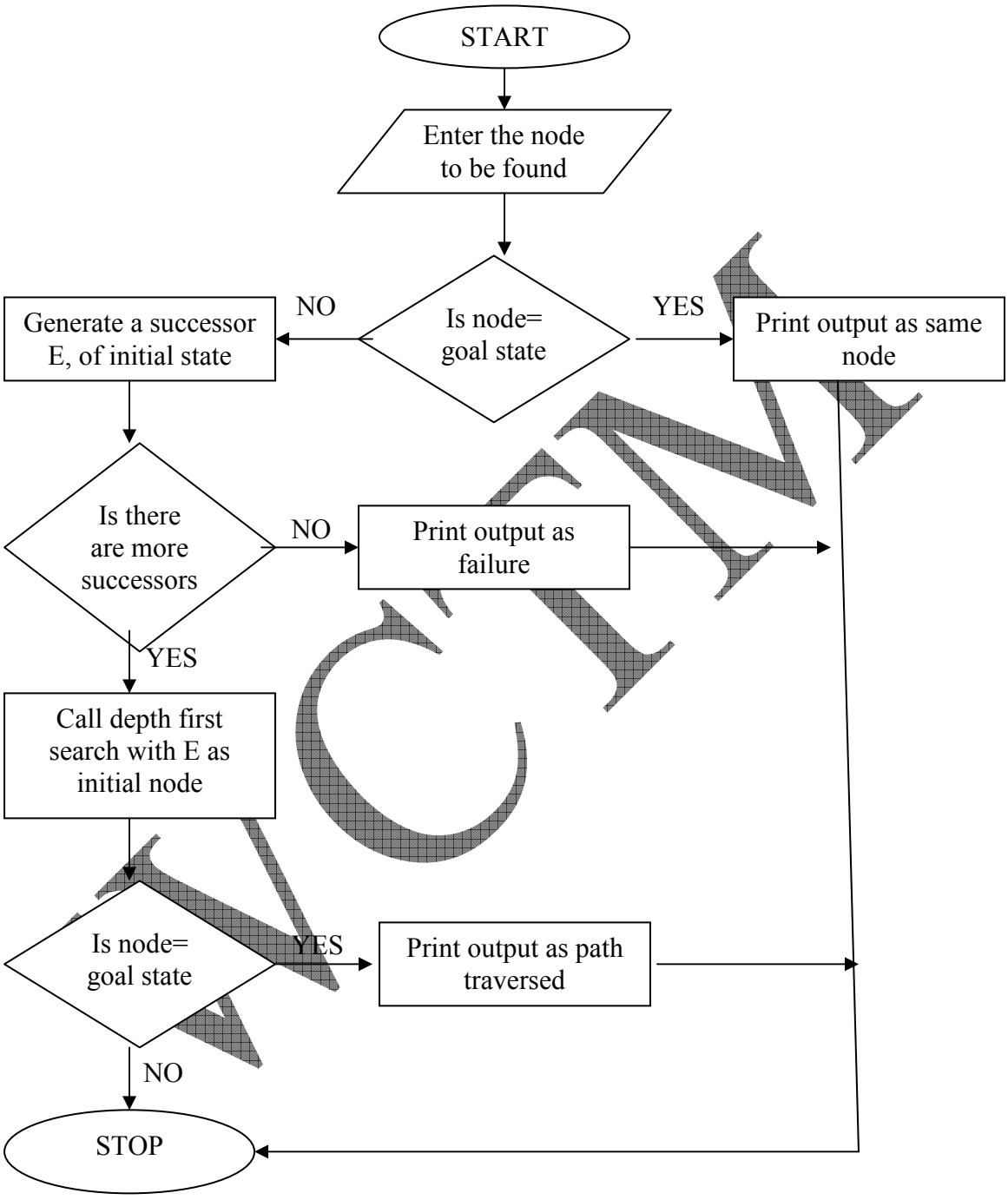
(c) If success is returned , signal success . Otherwise continue in this loop.

STEP 4 : Print the output as the path traversed.

STEP 5 : Exit.

WCTM

FLOWCHART FOR DEPTH FIRST SEARCH



PROGRAM FOR DEPTH FIRST SEARCH

```
domains
        X=symbol
        Y=symbol*

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

clauses
        child(a,b).                 /*b is child of a*/
        child(a,c).                 /*c is child of a*/
        child(a,d).                 /*d is child of a*/
        child(b,e).                 /*b is child of b*/
        child(b,f).                 /*f is child of b*/
        child(c,g).                 /*g is child of c*/
        path(A,G,[A|Z]):-           /*to find the path from root to leaf*/
                childnode(A,G,Z).
        childnode(A,G,[G]):-        /*to determine whether a node is child of other*/
                child(A,G).
        childnode(A,G,[X|L]):-
                child(A,X),
                childnode(X,G,L).
```

WCTM

OUTPUT:-

goal:-path(a,e,L)
L=[“a”,”b”,”e”]

WCTM

ALGORITHM FOR MENU DRIVEN PROGRAM FOR MEMBER, CONCATENATION, PERMUTATION, ADD AND DELETE FUNCTION

Step 1: Declare the functions for member, concatenation, permutation, add and delete.

Step 2: Enter the choices for above given call functions in X

Step 3: If (choice=1), call the member function.

Step 4: If (choice=2), call the concatenation function.

Step 5: If (choice=3), call the permutation function.

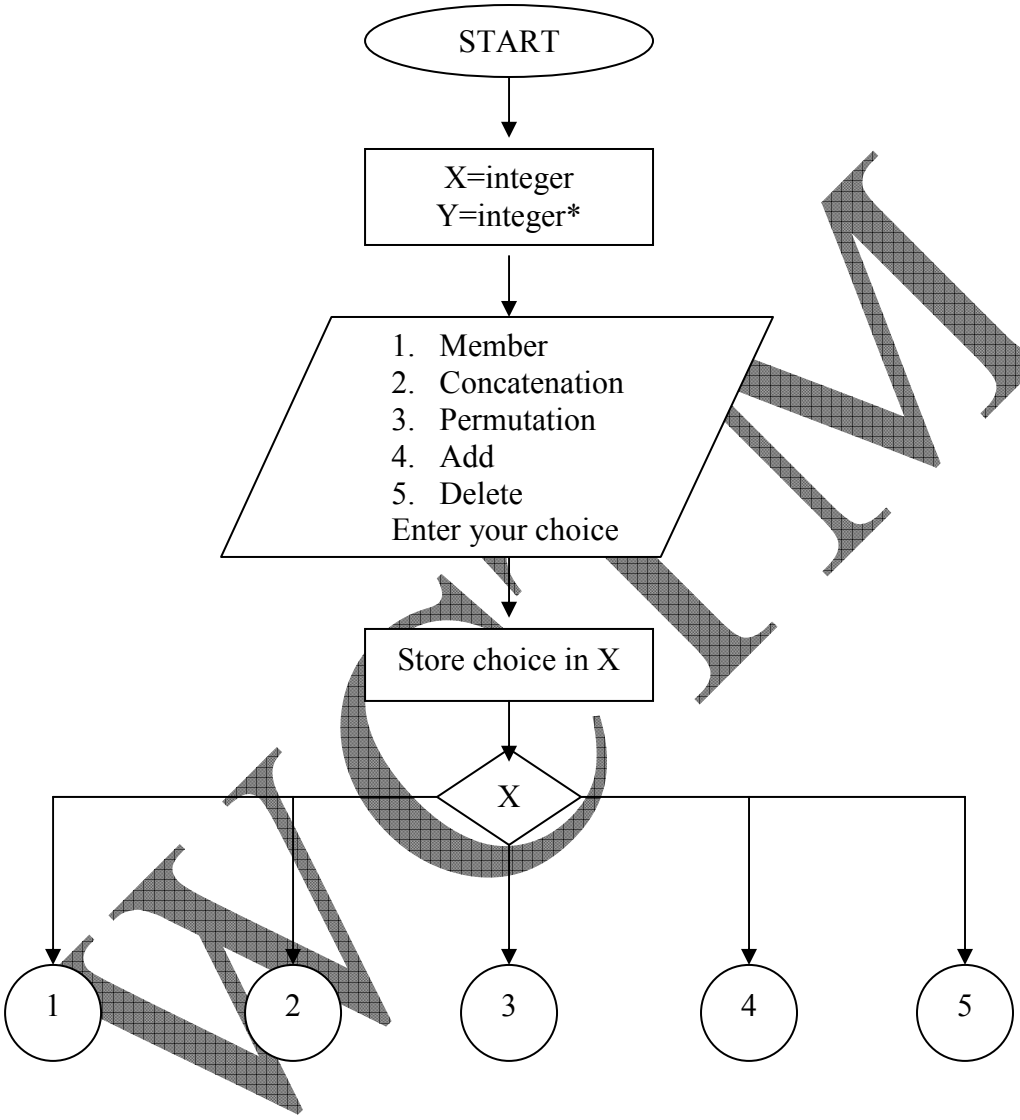
Step 6: If (choice=4), call the add function.

Step 7: If (choice=5), call the delete function.

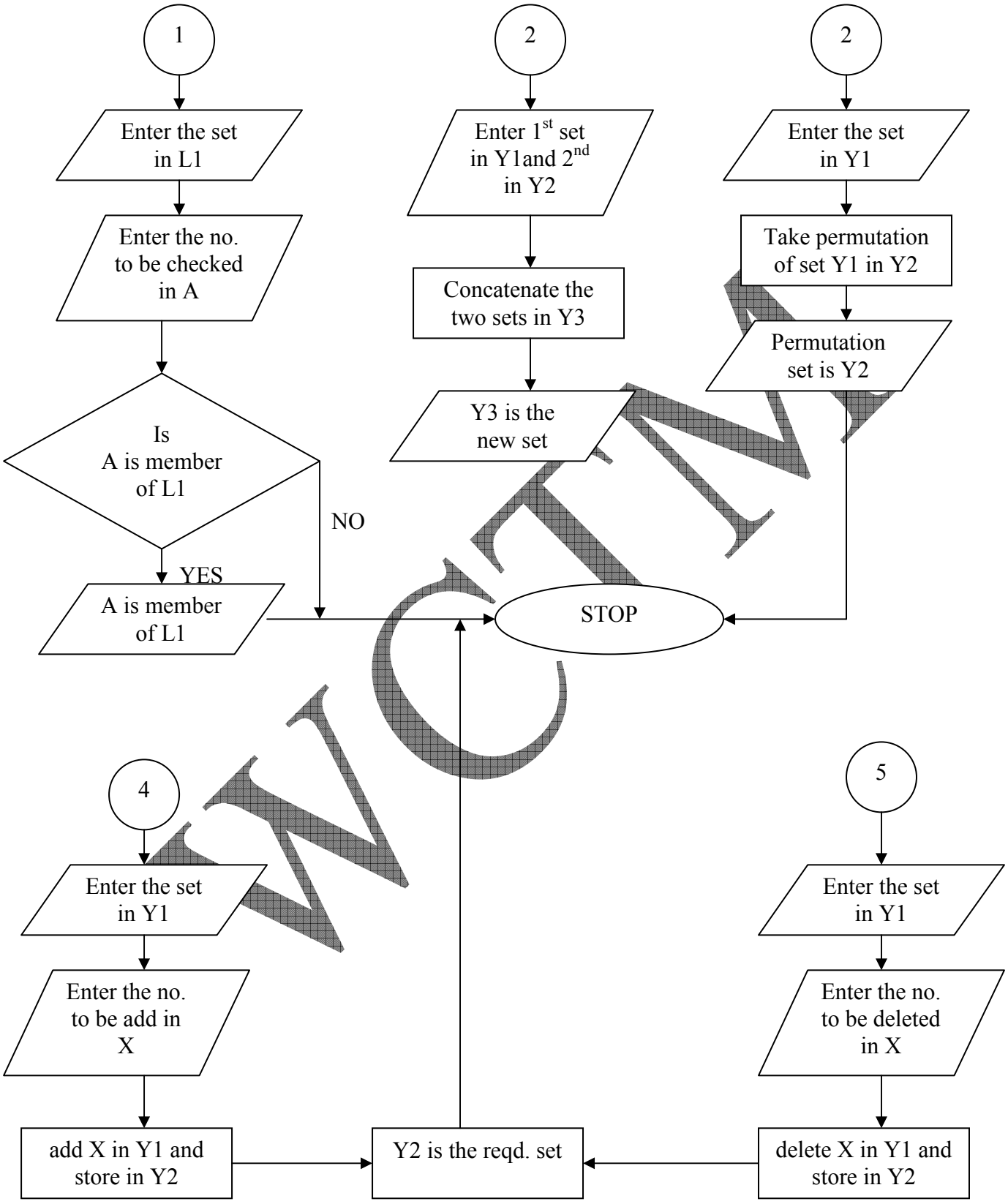
Step 8: Exit.

WCTM

FLOWCHART FOR MENU DRIVEN PROGRAM FOR MEMBER, CONCATENATION, PERMUTATION, ADD AND DELETE FUNCTION



WCTM


1
Enter the set in L1
Enter the no. to be checked in A
Is A is member of L1
NO
YES
A is member of L1
2
Enter 1st set in Y1and 2nd in Y2
Concatenate the two sets in Y3
Y3 is the new set
2
Enter the set in Y1
Take permutation of set Y1 in Y2
Permutation set is Y2
STOP
4
Enter the set in Y1
Enter the no. to be add in X
add X in Y1 and store in Y2
5
Enter the set in Y1
Enter the no. to be deleted in X
delete X in Y1 and store in Y2
Y2 is the reqd. set

WCTM

MENUDRIVEN PROGRAM FOR MEMBER, CONCATENATION, ADD, DELETE AND PERMUTATION FUNCTIONS

```
domains
        X=integer
        Y=integer*

predicates
        member(X,Y)
        concatenation(Y,Y,Y)
        add(X,Y,Y)
        delete(X,Y,Y)
        permutation(Y,Y)
        choice(X)

goal
        makewindow(1, 18, 680, "Menu Driven", 1, 1, 20, 70),
        write("1. Member\n 2. Concatenation\n 3. Permutation\n 4. Add\n 5. Delete\n "),
        write("Enter the choice :: "),
        readint(X),
        choice(X).

clauses
        choice(1):-                                     /*Member function*/
                write("\nEnter the set :: "),
                readterm(Y,L1),
                write("\nEnter the number to be checked :: "),
                readint(A),
                member(A,L1),
                write(A),
                write("is a member of "),
                write(L1),
                write("\n\nEnter your choice again :: "),
                readint(X),
                choice(X).

        choice(2):-                                     /*Concatenate function*/
                write("Enter the 1st set :: "),
                readterm(Y,Y1),
                write("\nEnter the 2nd set :: "),
                readterm(Y,Y2),
                concatenation(Y1,Y2,Y3),
                write(Y3),
                write("is the concatenated list"),
                write("\n\nEnter your choice again :: "),
                readint(X),
```

WCTM

```
        choice(X).

choice(3):-                                  /*Permutation function*/
        write("\nEnter the set :: "),
        readterm(Y,Y1),
        permutation(Y1,Y2),
        write(Y2),
        write("is the permutation list of "),
        write(Y1),
        write("\n\nEnter your choice again :: "),
        readint(X),
        choice(X).

choice(4):-                                  /*Add function*/
        write("\nEnter the set :: "),
        readterm(Y,Y1),
        write("\nEnter the no. to be added :: "),
        readint(X),
        add(X,Y1,Y2),
        write(Y2),
        write("is the new set "),
        write("\n\nEnter your choice again :: "),
        readint(X),
        choice(X).

choice(5):-                                  /*Delete function*/
        write("\nEnter the set :: "),
        readterm(Y,Y1),
        write("\nEnter the no. to be deleted :: "),
        readint(X),
        delete(X,Y1,Y2),
        write(Y2),
        write("is the new set "),
        write("\n\nEnter your choice again :: "),
        readint(X),
        choice(X).

member(X,[X|L]).                             /*to find the member*/
member(X,[F|L]):-
        member(X,L).

concatenation([],L,L).                       /*to concatenate two lists*/
concatenation([H|T],L1,[H|L2]):-
        concatenation(T,L1,L2).
```

WCTM

```
permutation([],[]).                    /*to find permutation list of a list*/
permutation([H|T],PL):-
        permutation(T,PT),
        delete(H,PL,PT).

add(X,L,[X|L]).                        /*to add an element in a list*/

delete(X,[X|L],T).                     /*to delete an element from a list*/
delete(X,[H|T],[H|T1]):-
        delete(X,T,T1).
```

WCTM

OUTPUT:-

goal:

1. Member
2. Concatenate
3. Permutation
4. Add
5. Delete

Enter the choice :: 2
Enter the 1st set :: [1,3,5]
Enter the 2nd set :: [2,4,6]
[1,3,5,2,4,6] is the concatenated list.

Enter ur choice again ::

WCTM

ALGORITHM TO FIND THE UNION OF TWO GIVEN LISTS

STEP 1 : Obtain the given lists as L and L1.

STEP 2 : Let H be the Head and T be the Tail of the List L.

STEP 3 : Check whether H is also a member of the other list L1.If yes, goto Step4 else goto Step5.

STEP 4 : check H is the last element of the list L. If yes, goto Step 6 else goto Step 5.

STEP 5 : Compare the rest of the elements of the tail T with that of the other list L1.
Goto Step3.
Do not duplicate the elements.

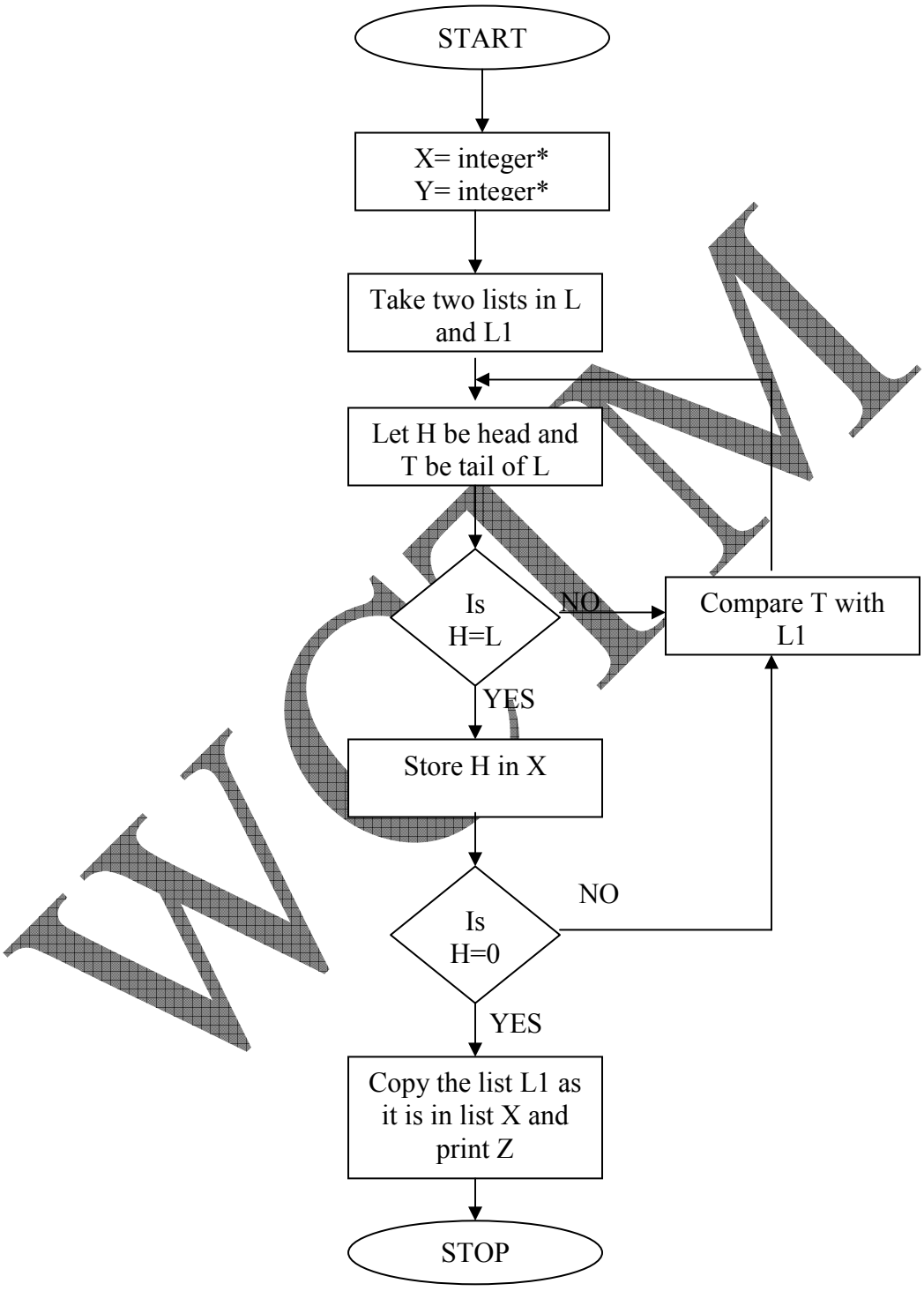
STEP 6 :Copy the list L1 as it is in list Z.

STEP 7 : Print Z as the required list .

STEP 8 : Exit.

WCTM

FLOWCHART FOR UNION OF TWO LIST



WCTM

PROGRAM FOR UNION OF TWO LISTS

```
domains
        X=integer*
        Y=integer*

predicates
        member(X,L)
        union(L,L,L)

clauses
        member(X,[X|_]).            /*to find the member of the list*/
        member(X,[_|T]):-
                member(X,T).

        union([],L,L).              /*to union the two lists*/
        union([H|T],L,L1):-
                member(H,L),!,
                union(T,L,L1).
        union([H|T],L,[H|L1]):-
                union(T,L,L1).
```

WCTM

OUTPUT:-

goal:-
union([1,2,3,4,5],[1,4,5,3,6,7],X)
X=[1,2,3,4,5,6,7]

WCTM

ALGORITHM TO FIND THE INTERSECTION OF TWO GIVEN LISTS

STEP 1 : Obtain the given lists as L and L1.

STEP 2 : Let H be the Head and T be the Tail of the list L.

STEP 3 : Check whether H is also a member of the other list L1. If yes , go to Step 4 else go to Step5.

STEP 4 : Copy the element H as an element of list Z .

STEP 5 : Check H is the last element of the list L. If yes, go to Step7 else go to Step 6.

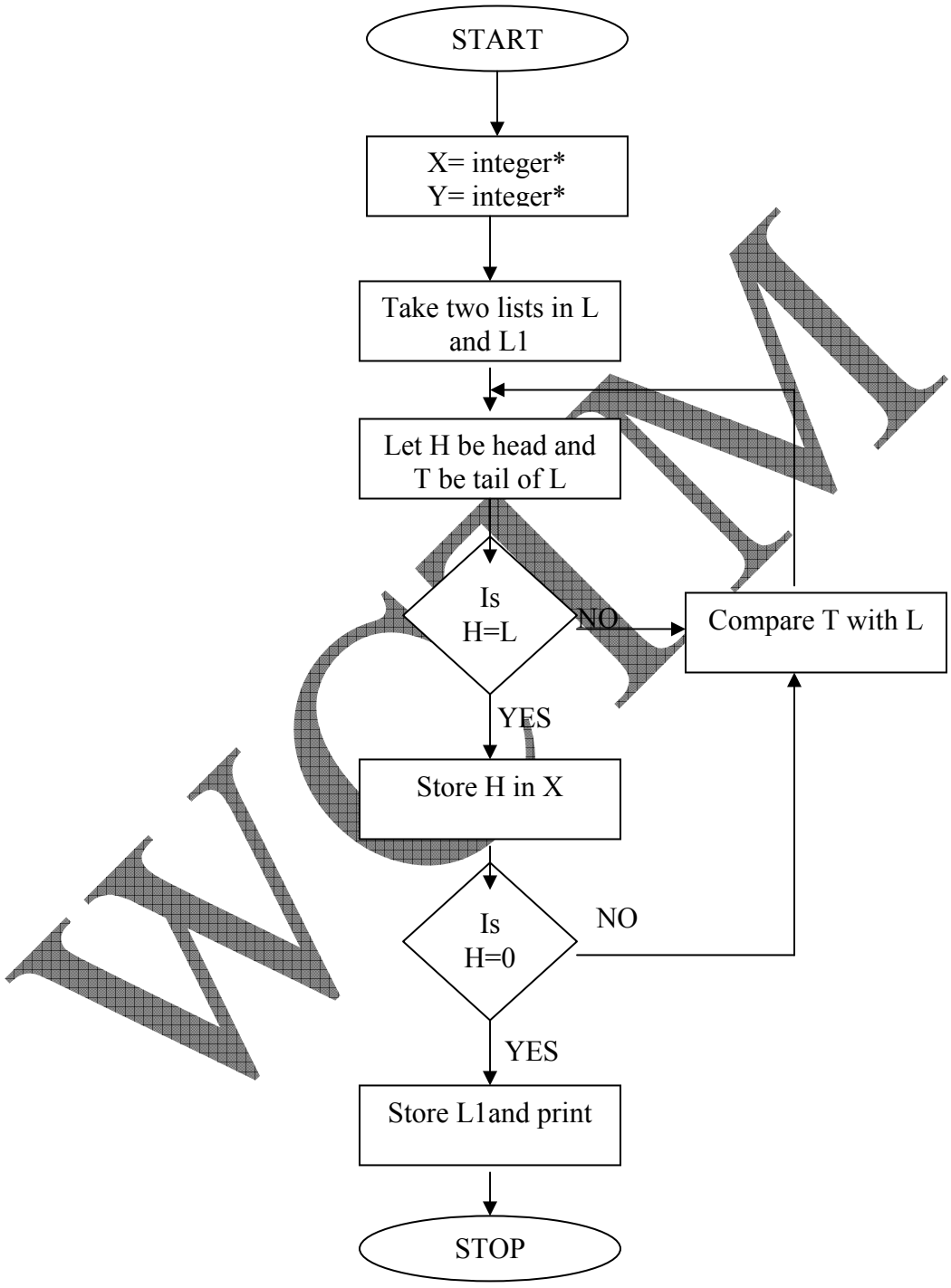
STEP 6 : Compare the rest of the elements of the Tail T with that of the other list L1 . Goto Step 3 .

STEP 7 : Print Z as the required list .

STEP 8 : Exit.

WCTM

FLOWCHART FOR INTERSECTION OF TWO LIST



WCTM

PROGRAM TO PERFORM INTERSECTION OF TWO LISTS

```
domains
        X=integer*
        Y=integer*

predicates
        member(X,L)
        intersection(L,L,L)

clauses
        member(X,[X|_]).                    /*to find the member of the list*/
        member(X,[_|T]):-
                member(X,T).

        intersection([],L,[]).              /*to find the intersection of two lists*/
        intersection([H|T],L,[H|L1]):-
                member(H,L),!,
                intersection(T,L,L1).
        intersection([H|T],L,L1):-
                intersection(T,L,L1).
```

WCTM

OUTPUT:-

goal:-
intersection([3,5,7,4,2],[1,3,2,4,5,6],X)
X=[3,5]

WCTM

ALGORITHM TO FIND THE FACTORIAL OF A NUMBER

Step 1: Enter the integer as X.

Step 2: Initialize A=1.

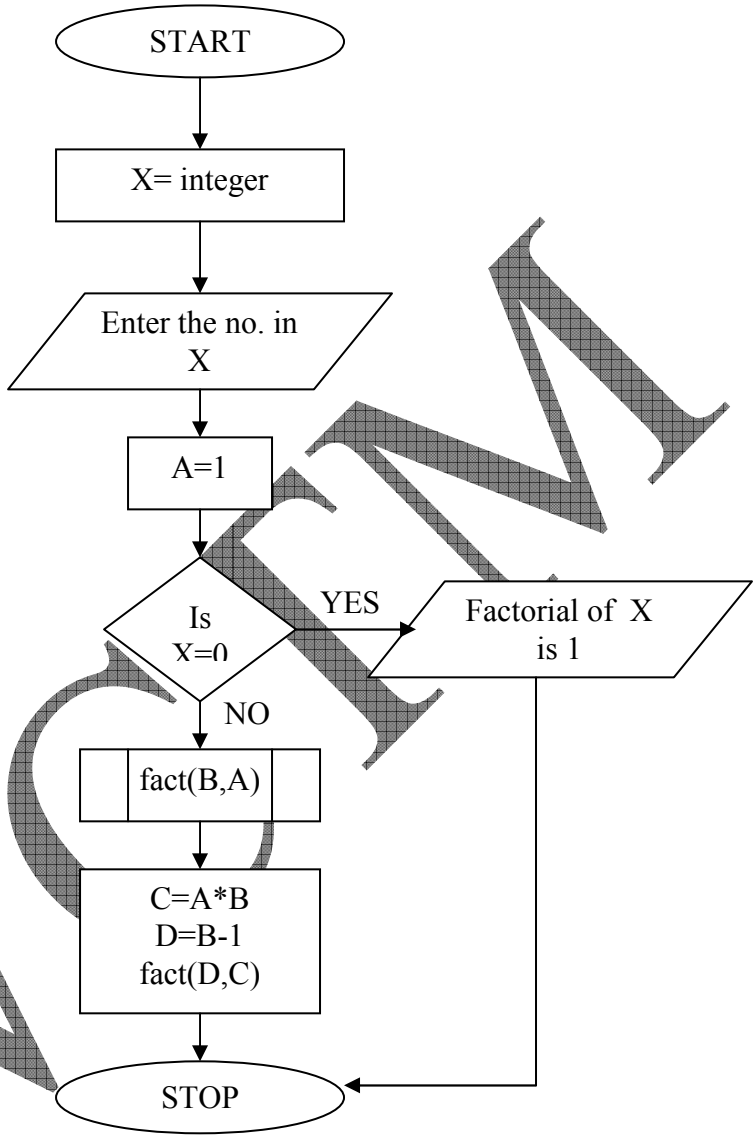
Step 3: If X=0, then print Factorial is A.

Step 4: If X! =0 i.e. =B, then perform factorial of B and print the result.

Step 5: Exit.

WCTM

FLOWCHART TO FIND THE FACTORIAL OF A NUMBER



WCTM

PROGRAM TO FIND FACTORIAL OF A NUMBER

```
domains
        X=integer

predicates
        go
        factX,X)

clauses
        go:-
                write("\nEnter the no. :: "),
                readint(X),
                A=1.

        fact(0,A).                      /*factorial of 0 is 1*/
        fact(X,A):-
                write("Factorial of "),
                write(X),
                write(" is " A),
        fact(B,A):-                     /*find the factorial by multiplying no. by its
                                        predeccessors*/
                C=A*B,
                D=B-1,
                fact(D,C).
```

WCTM

OUTPUT:-

goal:-
go
Enter the no. :: 4
Factorial of 4 is 16

WCTM

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 :

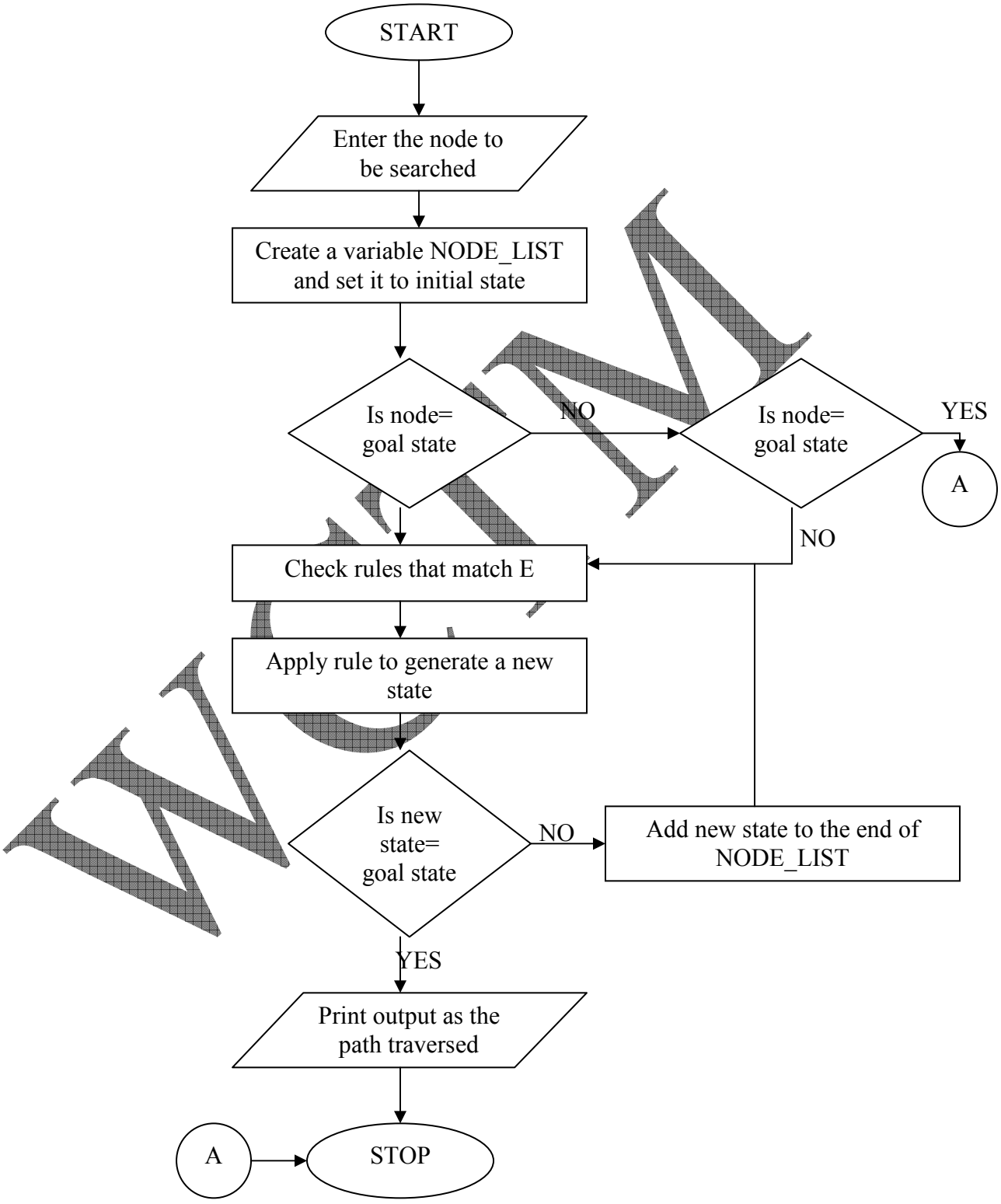
i. Apply the rule to generate a new state .
ii. If the new state is a goal state, quit and return this state.
iii. Otherwise, add the new state to the end of NODE-LIST.

STEP 4 : Print the output as the path traversed.

STEP 5 : Exit.

WCTM

FLOWCHART FOR BREADTH FIRST SEARCH



PROGRAM TO IMPLEMENT BREADTH FIRST SEARCH

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

predicates
        solve(L, L)
        member(X,L)
        extend(L, L)
        conc(X, L, L)
        breadthfirst(L, L)
        goal(X)

clauses
        solve(start, solution):-        /*solution is a state from start to a goal*/
                breadthfirst([[start]],solution).

        breadthfirst([[node|path]| _ ] ,[node|path]):-  /*solution is an extension to a goal*/
                                                        /*of one of path*/
                goal(node).

        breadthfirst([path|paths], solution):-
                extend(path,newpaths),
                conc(paths,newpaths,path1),
                breadthfirst(path1,solution).

        extend([node|path],newpaths):-
                bagof([newnode, node|path],(s(node,
                        newnode),notmember(newnode,[node|path])), newpaths),!.

        extend(path, []).

        conc([], L, L).

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

        member(X, [X|T]).

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

WCTM

OUTPUT:-

goal: solve([a, e], S)
L= [“a”, “b”, ”c”, “d”, ”e”]

goal: solve([a, h],S)
L= [“a”, “b”, ”c”, “d”, ”e”, ”f”, “g”, “h”]

WCTM

PROGRAM TO SOLVE MONKEY BANANA PROBLEM

```
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).
        graps.
        climb.

clauses
        move(state(middle,onbox,middle,hasnot),      /*Before move*/
             grasp,                                  /*Grasp banana*/
             state (middle,onbox,middle,has)).       /*After move*/

        move(state(P,onfloor,P,H),
             climb,                                  /*Climb box*/
             state(P,onbox,P,H)).

        move(state(P1,onfloor,P1,H),
             push(P1,P2),                            /*Push box from P1 to 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)).                    /*can 1:Monkey already has it.*/

        canget(state1):-                             /*can 2:Do some work to get it*/
             move(State1,Move,State2),               /*Do something*/
             canget(State2).                         /*Get it now.   */
```

WCTM

OUTPUT:-

goal: canget(atdoor,atfloor,window,hasnot)
No solution.

WCTM

PROGRAM TO FIND PERMUTATION OF A SET

```
domains
        X=integer
        Y=integer*

predicates
        permute(Y,Y)
        delete(X,Y,Y)

clauses
        delete(X,[X|T],T).
        delete(X,[H|T],[H|T1]):-
                delete(X,T,T1).

        permute([],[]).
        permute([H|T],PL):-
                permute(T,PT),
                delete(H,PL,PT).
```

WCTM

OUTPUT:-

goal:permute([1,2],A)
A=[1,2]
A=[2,1]
2 Solutions

WCTM

PROGRAM TO CONCATENATE TWO SETS

```
domains
        X=integer
        Y=integer*

predicates
        concatenate(Y,Y,Y)

clauses
        concatenate([],[]).
        concatenate([H|T],L,[H|T1]):-
                concatenate(T,L1,L2).
```

WCTM

OUTPUT:-

goal:concatenate([1,2,3],[4,5],A)
A=[1,2,3,4,5]

WCTM

PROGRAM TO FIND MEMBER OF A SET

```
domains
        X=integer
        Y=integer*

predicates
        member(X,Y)

clauses
        member(X,[X|T]).
        member(X,[F|L]):-
                member(X,L).
```

WCTM

OUTPUT:-

goal:member(2,[2,3,4])
Yes

WCTM