INSTRUCTION MANUAL

ARTIFICIAL INTELLIGENCE LAB
(BTCS-704)

Prepared by

Er. Supreet Kaur
Assistant Professor (CSE)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

GURU NANAK DEV ENGINEERING COLLEGE
LUDHIANA – 141006

DECLARATION

This Manual of Artificial Intelligence (BTCS-704) has been prepared by me as per syllabus of Artificial Intelligence (BTCS-704).

Signature

List of Experiments

EXPERIMENT 1

AIM:- Introduction to Prolog.

Prolog, which stands for PROgramming in LOGic, is the most widely available language in the logic programming paradigm. Logic and therefore Prolog is based the mathematical notions of relations and logical inference. Prolog is a declarative language meaning that rather than describing how to compute a solution, a program consists of a data base of facts and logical relationships (rules) which describe the relationships which hold for the given application. Rather then running a program to obtain a solution, the user asks a question. When asked a question, the run time system searches through the data base of facts and rules to determine (by logical deduction) the answer.

Among the features of Prolog are `logical variables' meaning that they behave like mathematical variables, a powerful pattern-matching facility (unification), a backtracking strategy to search for proofs, uniform data structures, and input and output are interchangeable.

Often there will be more than one way to deduce the answer or there will be more than one solution, in such cases the run time system may be asked find other solutions. backtracking to generate alternative solutions. Prolog is a weakly typed language with dynamic type checking and static scope rules.

Prolog is used in artificial intelligence applications such as natural language interfaces, automated reasoning systems and expert systems. Expert systems usually consist of a data base of facts and rules and an inference engine, the run time system of Prolog provides much of the services of an inference engine.

The Structure of Prolog Programs

- A Prolog program consists of a database of facts and rules, and queries (questions).
    - Fact: ... .
    - Rule: ... :- ... .
    - Query: ?- ... .
    - Variables: must begin with an upper case letter.
    - Constants: numbers, begin with lowercase letter, or enclosed in single quotes.

    Lists: append, member

```
list([]).
list([X|L])                    :- [list(L).
```

Abbrev: $[X_1|[...[X_n|[]...] = [X_1,...X_n]$

```
append([],L,L).
append([X|L1],L2,[X|L12]) :- append(L1,L2,L12).
member(X,L)               :- concat(_,[X|_],L).
```

Ancestor

```
ancestor(A,D) :- parent(A,B).
ancestor(A,D) :- parent(A,C),ancestor(C,D).
```

but not

```
ancestor(A,D) :- Ancestor(A,P),
                 parent(P,D).
```

- *since infinite recursion may result.*

- Depth-first search: Maze/Graph traversal
  A database of arcs (we will assume they are directed arcs) of the form:
  a(node_i,node_j).
- Rules for searching the graph:

```
go(From,To,Trail).
go(From,To,Trail) :- a(From,In), not visited(In,Trail),
                     go(In,To,[In|Trail]).
visited(A,T)      :- member(A,T).
```

- I/O: terms, characters, files, lexical analyzer/scanner
  - read(T), write(T), nl.
  - get0(N), put(N): ascii value of character
  - name(Name,Ascii_list).
  - see(F), seeing(F), seen, tell(F), telling(F), told.
- Natural language processing: Context-free grammars may be represented as Prolog rules. For example, the rule
  sentence ::= noun_clause verb_clause
- *can be implemented in Prolog as*

```
sentence(S)     :- append(NC,VC,S), noun_clause(NC),
                   verb_clause(VC).
or in DCG as:
Sentence        -> noun_clause, verb_clause.
?-
sentence(S,[]).
```

- Note that two arguments appear in the query. Both are lists and the first is the sentence to be parsed, the second the remaining elements of the list which in this case is empty.

A Prolog program consists of a data base of facts and rules. There is no structure imposed on a Prolog program, there is no main procedure, and there is no nesting of definitions. All facts and rules are global in scope and the scope of a variable is the fact or rule in which it appears. The readability of a Prolog program is left up to the programmer.

Facts

This chapter describes the basic Prolog facts. They are the simplest form of Prolog predicates, and are similar to records in a relational database. As we will see in the next chapter they can be queried like database records.

The syntax for a fact is

```
pred(arg1, arg2, ... argN).
```

where

pred
: The name of the predicate

arg1, ...
: The arguments.

Viva questions:

1. What is Prolog?
2. What is the difference between clauses and predicates?
3. How can you write facts and rules?

Assignments:

1. What are the different data types in prolog?
2. Explain Data Structure in Prolog

EXPERIMENT 2(a)

AIM: Write simple fact for Nani search problem.

We will now begin to develop Nani Search by defining the basic facts that are meaningful for the game. These include

- The rooms and their connections
- The things and their locations
- The properties of various things
- Where the player is at the beginning of the game

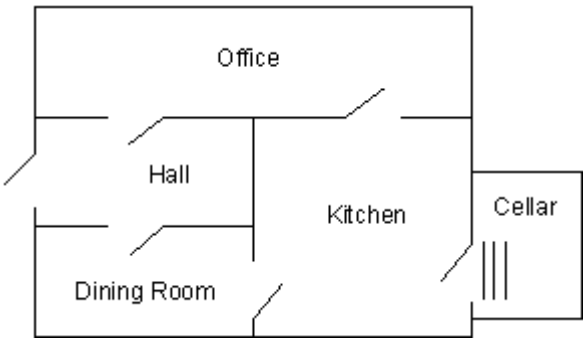


Figure The rooms of Nani Search

Open a new source file and save it as 'myadven.pro', or whatever name you feel is appropriate. You will make your changes to the program in that source file. (A completed version of nanisrch.pro is in the Prolog samples directory, SAMPLES\PROLOG\MISC.)

First we define the rooms with the predicate room/1, which has five clauses, all of which are facts. They are based on the game map in figure 2.1.

```
room(kitchen).
room(office).
room(hall).
room('dining room').
room(cellar).
```

We define the locations of things with a two-argument predicate location/2. The first argument will mean the thing and the second will mean its location. To begin with, we will add the following things.

```
location(desk, office).
location(apple, kitchen).
location(flashlight, desk)
```

```
location('washing machine', cellar).
location(nani, 'washing machine').
location(broccoli, kitchen).
location(crackers, kitchen).
location(computer, office).
```

EXPERIMENT 2(B)

AIM: Write simple fact for following

1. Ram likes mango.
2. Seema is a girl.
3. Bill likes Cindy.
4. Rose is red.
5. John owns gold.

Clauses

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

Goal

```
?- likes (ram,What).
  What = mango.
  1 solution.
```

Viva Questions:

1- First create a source file for the genealogical logicbase application. Start by adding a few members of your family tree. It is important to be accurate, since we will be exploring family relationships. Your own knowledge of who your relatives are will verify the correctness of your Prolog programs.

2- Enter a two-argument predicate that records the parent-child relationship. One argument represents the parent, and the other the child. It doesn't matter in which order you enter the arguments, as long as you are consistent. Often Prolog programmers adopt the convention that parent(A,B) is interpreted "A is the parent of B".

3- Create a source file for the customer order entry program. We will begin it with three record types (predicates). The first is customer/3 where the three arguments are

arg1

Customer name

arg2

City

arg3

Credit rating (aaa, bbb, etc)

4- Next add clauses that define the items that are for sale. It should also have three arguments

arg1
Item identification number

arg2 Item name
arg3
The reorder point for inventory (when at or below this level, reorder)

5- Next add an inventory record for each item. It has two arguments.

arg1
Item identification number (same as in the item record)
arg2
Amount in stock

Assignment:

Aim: Write facts for following:

1. Ram likes apple.
2. Ram is taller then Mohan.
3. My name is Subodh.
4. Apple is fruit.
5. Orange is fruit.
6. Ram is male.

EXPERIMENT 3

AIM: Write simple queries for following facts.

Simple Queries

Now that we have some facts in our Prolog program, we can consult the program in the listener and query, or call, the facts. This chapter, and the next, will assume the Prolog program contains only facts. Queries against programs with rules will be covered in a later chapter.

Prolog queries work by pattern matching. The query pattern is called a **goal**. If there is a fact that matches the goal, then the query succeeds and the listener responds with 'yes.' If there is no matching fact, then the query fails and the listener responds with 'no.'

Prolog's pattern matching is called **unification**. In the case where the logicbase contains only facts, unification succeeds if the following three conditions hold.

- The predicate named in the goal and logicbase are the same.
- Both predicates have the same arity.
- All of the arguments are the same.

Before proceeding, review figure 3.1, which has a listing of the program so far.

The first query we will look at asks if the office is a room in the game. To pose this, we would enter that goal followed by a period at the listener prompt.

```
?- room(office).
yes
```

Prolog will respond with a 'yes' if a match was found. If we wanted to know if the attic was a room, we would enter that goal.

```
?- room(attic).
no
```

Solution:-

clauses

```
likes(ram ,mango).
girl(seema).
red(rose).
likes(bill ,cindy).
```

owns(john ,gold).

queries

?-likes(ram,What).
What= mango
?-likes(Who,cindy).
Who= cindy

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

Viva Questions:

1- Consider the following Prolog logic base

easy(1).
easy(2).
easy(3).
gizmo(a,1).
gizmo(b,3).
gizmo(a,2).
gizmo(d,5).
gizmo(c,3).
gizmo(a,3).
gizmo(c,4).

and predict the answers to the queries below, including all alternatives when the semicolon (;) is entered after an answer.

?- easy(2).
?- easy(X).
?- gizmo(a,X).
?- gizmo(X,3).
?- gizmo(d,Y).
?- gizmo(X,X).

2- Consider this logicbase,

harder(a,1).

```
harder(c,X).
harder(b,4).
harder(d,2).
```

and predict the answers to these queries.

```
?- harder(a,X).
?- harder(c,X).
?- harder(X,1).
?- harder(X,4).
```

3- Enter the listener and reproduce some of the example queries you have seen against location/2. List or print location/2 for reference if you need it. Remember to respond with a semicolon (;) for multiple answers. Trace the query.

4- Pose queries against the genealogical logicbase that:

- Confirm a parent relationship such as parent(dennis, diana)
- Find someone's parent such as parent(X, diana)
- Find someone's children such as parent(dennis, X)
- List all parent-children such as parent(X,Y)

5- If parent/2 seems to be working, you can add additional family members to get a larger logicbase. Remember to include the corresponding male/1 or female/1 predicate for each individual added.

EXPERIMENT NO. 4

AIM: Write compound query for Cat and Fish problem.

Compound Queries

Simple goals can be combined to form compound queries. For example, we might want to know if there is anything good to eat in the kitchen. In Prolog we might ask

```
?- location(X, kitchen), edible(X).
```

Whereas a simple query had a single goal, the compound query has a conjunction of goals. The comma separating the goals is read as "and."

Logically (declaratively) the example means "Is there an X such that X is located in the kitchen and X is edible?" If the same variable name appears more than once in a query, it must have the same value in all places it appears. The query in the above example will only succeed if there is a single value of X that can satisfy both goals.

In logical terms, the query says "Find a T and R such that there is a door from the kitchen to R and T is located in R." In procedural terms it says "First find an R with a door from the kitchen to R. Use that value of R to look for a T located in R."

```
?- door(kitchen, R), location(T,R).
R = office
T = desk ;
R = office
T = computer ;
R = cellar
T = 'washing machine' ;
```

Facts are:

1. Cat likes fish.
2. Tuna is fish.
3. Tom is cat.

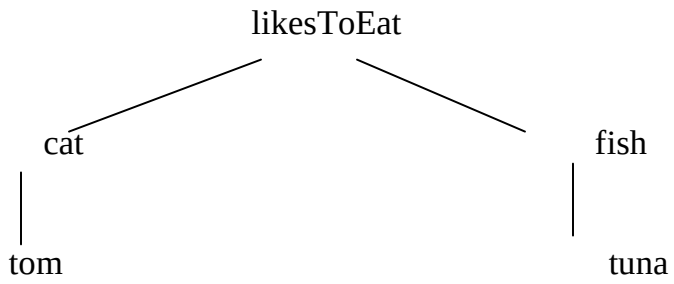
Production Rules:

likesToEat ⟶ cat,fish

cat ⟶ tom

fish ⟶ tuna

Parse tree



Clauses:

likesToEat(X,Y) :- cat(X), fish(Y).
cat(tom)
fish(tuna)

Queries:

?- likesToEat(X, Y), X = tom.
Y = tuna

?- likesToEat(tom, What)
What = tuna

?- likesToEat(Who, tuna)
Who = tom

<u>Viva Questions:</u>

1- Consider the following Prolog logicbase.

easy(1).
easy(2).
easy(3).
gizmo(a,1).
gizmo(b,3).
gizmo(a,2).
gizmo(d,5).
gizmo(c,3).
gizmo(a,3).
gizmo(c,4).

```
harder(a,1).
harder(c,X).
harder(b,4).
harder(d,2).
```

Predict the results of the following queries. Then try them and trace them to see if you were correct.

```
?- gizmo(a,X),easy(X).
?- gizmo(c,X),easy(X).
?- gizmo(d,Z),easy(Z).
?- easy(Y),gizmo(X,Y).
?- write('report'), nl, easy(T), write(T), gizmo(M,T), tab(2), write(M), fail.
?- write('buggy'), nl, easy(Z), write(X), gizmo(Z,X), tab(2), write(Z), fail.
?- easy(X),harder(Y,X).
?- harder(Y,X),easy(X).
```

2- Experiment with the queries you have seen in this chapter.

3- Predict the results of this query before you execute it. Then try it. Trace it if you were wrong.

```
?- door(kitchen, R), write(R), nl, location(T,R), tab(3), write(T), nl, fail.
```

4- Compound queries can be used to find family relationships in the genealogical logicbase. For example, find someone's mother with

```
?- parent(X, someone), female(X).
```

Write similar queries for fathers, sons, and daughters. Trace these queries to understand their behavior (or misbehavior if they are not working right for you).

5- Experiment with the ordering of the goals. In particular, contrast the queries.

```
?- parent(X, someone), female(X).
?- female(X), parent(X, someone).
```

Do they both give the same answer? Trace both queries and see which takes more steps.

Assignment :

Aim: Convert the facts into predicate logic.

1. Ram is vegetarian and eats only his doctor tells him to eat.
2. Cabbage, cauliflower, beans and potatoes are vegetable.
3. Chicken, egg and fish are flesh.
4. Doctor tells him to it vegetables and egg.

EXPERIMENT NO. 5

AIM: Write predicates for following facts

1. Dashrath is Ram's father.
2. Ram is father of Luv.
3. Kush is son of Ram.
4. Koushaliya is wife of Dashrath.

Rules

We said earlier a predicate is defined by clauses, which may be facts or rules. A rule is no more than a stored query. Its syntax is

head :- body.
where

head
a predicate definition (just like a fact)

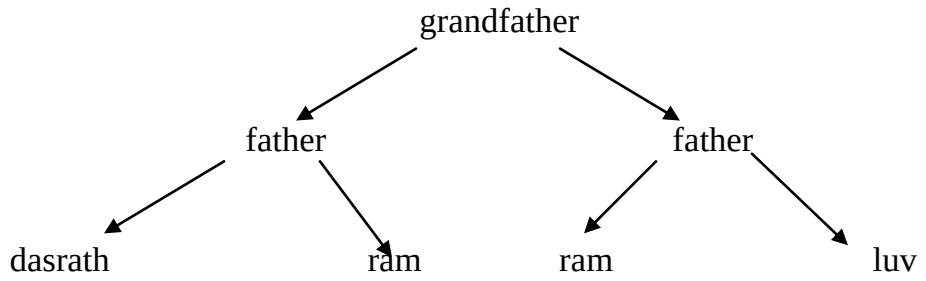
:-
the **neck** symbol, sometimes read as "if"

body
one or more goals (a query)

Production rules:

mother → wife,father
grandfather → father,father/father,son

Parse tree:



Clauses:

father(dashrath, ram).
father(ram, luv).
son(kush, ram).
wife(koushaliya, dashrath).

```
mother(X,Y):-wife(X,Z),father(Z,Y).
grandfather(X,Y):-father(X,Z),father(Z,Y).
grandfather(X,Y):-father(X,Z),son(Y,Z).
```

Queries:

```
?- father(Who,ram).
   Who= Dashrath.
?- mother(Who,ram).
   Who= Koushaliya.
?- grandfather(dashrath,Who).
   Who= luv.
   Who= Kush.
```

Viva Questions:

1- Consider the following Prolog logicbase.

```
a(a1,1).
a(A,2).
a(a3,N).
b(1,b1).
b(2,B).
b(N,b3).
c(X,Y) :- a(X,N), b(N,Y).
d(X,Y) :- a(X,N), b(Y,N).
d(X,Y) :- a(N,X), b(N,Y).
```

Predict the answers to the following queries, then check them with Prolog, tracing.

```
?- a(X,2).
?- b(X,kalamazoo).
?- c(X,b3).
?- c(X,Y).
?- d(X,Y).
```

2- Experiment with the various rules that were developed during this chapter, tracing them all.

3- Write look_in/1 for Nani Search. It should list the things located in its argument. For example, look_in(desk) should list the contents of the desk.

4- Build rules for the various family relationships that were developed as queries in the last chapter. For example

```
mother(M,C):-
  parent(M,C),
  female(M).
```

5- Build a rule for siblings. You will probably find your rule lists an individual as his/her own sibling. Use trace to figure out why.

6- Use the sibling predicate to define additional rules for brothers, sisters, uncles, aunts, and cousins.

Assignment :

Aim: Convert the facts into predicate logic.

1. A person can buy a car if he likes the car and the car is for sell.
2. M-800 is for sell.
3. B.M.W. is for sell.
4. Ram likes B.M.W.
5. Sohan likes M-800.
6. Sita likes Ford.
7. Fait is for sell.

EXPERIMENT NO. 6

AIM: Write predicates One converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.

Arithmetic

Prolog must be able to handle arithmetic in order to be a useful general purpose programming language. However, arithmetic does not fit nicely into the logical scheme of things.

That is, the concept of evaluating an arithmetic expression is in contrast to the straight pattern matching we have seen so far. For this reason, Prolog provides the built-in predicate 'is' that evaluates arithmetic expressions. Its syntax calls for the use of operators, which will be described in more detail in chapter 12.

X is <arithmetic expression>

The variable X is set to the value of the arithmetic expression. On backtracking it is unassigned.

The arithmetic expression looks like an arithmetic expression in any other programming language.

Here is how to use Prolog as a calculator.

```
?- X is 2 + 2.
X = 4

?- X is 3 * 4 + 2.
X = 14
```

Parentheses clarify precedence.

```
?- X is 3 * (4 + 2).
X = 18

?- X is (8 / 4) / 2.     X = 1
```

In addition to 'is,' Prolog provides a number of operators that compare two numbers. These include 'greater than', 'less than', 'greater or equal than', and 'less or equal than.' They behave more logically, and succeed or fail according to whether the comparison is true or false. Notice the order of the symbols in the greater or equal than and less than or

equal operators. They are specifically constructed not to look like an arrow, so that you can use arrow symbols in your programs without confusion.

```
X > Y
X < Y
X >= Y
X =< Y
```

Here are a few examples of their use.

```
?- 4 > 3.
Yes
?- 4 < 3.
No
?- X is 2 + 2, X > 3.
X = 4
?- X is 2 + 2, 3 >= X.
No
?- 3+4 > 3*2.
Yes
```

Production rules:

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.
```

Queries :

```
?- c_to_f(100,X).
X = 212
Yes
?- freezing(15).
Yes
?- freezing(45).
No
```

Viva Questions:

1- Write a predicate valid_order/3 that checks whether a customer order is valid. The arguments should be customer, item, and quantity. The predicate should succeed only if the customer is a valid customer with a good credit rating, the item is in stock, and the quantity ordered is less than the quantity in stock.

2- Write a reorder/1 predicate which checks inventory levels in the inventory record against the reorder quantity in the item record. It should write a message indicating whether or not it's time to reorder.

Assignment:

Aim: Using the following facts answer the question

1. Find car make that cost is exactly 2,00,000/-
2. Find car make that cost is less then 5 lacs.
3. List all the cars available.
4. Is there any car which cost is more then 10 lacs.

EXPERIMENT NO. 7

Aim: Write a prolog program for factorial using recursion.

Recursion

Recursion in any language is the ability for a unit of code to call itself, repeatedly, if necessary. Recursion is often a very powerful and convenient way of representing certain programming constructs.

In Prolog, recursion occurs when a predicate contains a goal that refers to itself.

As we have seen in earlier chapters, every time a rule is called, Prolog uses the body of the rule to create a new query with new variables. Since the query is a new copy each time, it makes no difference whether a rule calls another rule or itself.

A recursive definition (in any language, not just Prolog) always has at least two parts, a boundary condition and a recursive case.

The boundary condition defines a simple case that we know to be true. The recursive case simplifies the problem by first removing a layer of complexity, and then calling itself. At each level, the boundary condition is checked. If it is reached the recursion ends. If not, the recursion continues.

We will illustrate recursion by writing a predicate that can detect things which are nested within other things.

Currently our location/2 predicate tells us the flashlight is in the desk and the desk is in the office, but it does not indicate that the flashlight is in the office.

```
?- location(flashlight, office).
no
```

Using recursion, we will write a new predicate, is_contained_in/2, which will dig through layers of nested things, so that it will answer 'yes' if asked if the flashlight is in the office.

To make the problem more interesting, we will first add some more nested items to the game. We will continue to use the location predicate to put things in the desk, which in turn can have other things inside them.

```
location(envelope, desk).
location(stamp, envelope).
location(key, envelope).
```

Factorial

To compute the factorial of N we first compute the factorial of N-1 using the same procedure (recursion) and then, usign the result of subproblem, we compute the factorial of N. The recursion stops when a trivial problem is reached, i.e., when a factorial of 0 is computed.

Rules:

```
fact(0,1).
fact(X,FactX):-Y=X-1,
            fact(Y,FactY),
            FactX = X * FactY.
```

Queries:

```
?- fact(6,Z).
  Z = 720

?- fact(X,120)
  X = 5
```

Viva Questions:

1- Trace the two versions of is_contained_in/2 presented at the end of the chapter to understand the performance differences between them.

2- Currently, the can_take/1 predicate only allows the player to take things which are directly located in a room. Modify it so it uses the recursive is_contained_in/2 so that a player can take anything in a room.

3- Use recursion to write an ancestor/2 predicate. Then trace it to understand its behavior. It is possible to write endless loops with recursive predicates. The trace facility will help you debug ancestor/2 if it is not working correctly.

4- Use ancestor/2 for finding all of a person's ancestors and all of a person's descendants. Based on your experience with grandparent/2 and grandchild/2, write a descendant/2 predicate optimized for descendants, as opposed to ancestor/2, which is optimized for ancestors.

Assignment: Aim: Write a prolog program for fibonacci using recursion.

EXPERIMENT NO. 8

Aim: Write a program for implementation of a given tree.

Production rule:

X is_same_level_as Y ⟶ W is_parent X,
Z is_parent Y,
W is_same_level_as Z.

X is_sibling_of Y ⟶ Z is_parent X,
Z is_parent Y,
X \== Y.

node_has_depth D ⟶ mother is_parent Node,
mother has_depth D1,
D is D1 + 1.

locate(Node) ⟶ path(Node),write(Node).

ht(Node,H) ⟶ node is_parent Child,
ht(Child,H1),
H is H1 +1.

Parse tree:

Consider the following tree diagram.

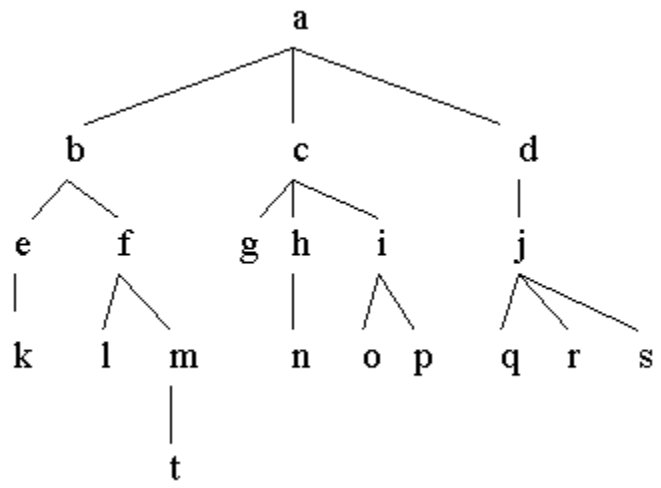


Fig.
The fig. has a representation for this tree and predicate definitions to do some processing of the tree. Note the use of Prolog operators in some of the definitions.

```
/* The tree database */
:- op(500,xfx,'is_parent').
```

```
a is_parent b.    c is_parent g.     f is_parent l.     j is_parent q.
a is_parent c.    c is_parent h.     f is_parent m.     j is_parent r.
a is_parent d.    c is_parent i.     h is_parent n.     j is_parent s.
b is_parent e.    d is_parent j.     i is_parent o.     m is_parent t.
b is_parent f.    e is_parent k.     i is_parent p.
 /* X and  Y are siblings  */
 :- op(500,xfx,'is_sibling_of').
 X is_sibling_of Y :- Z is_parent X,
                      Z is_parent Y,
                      X \== Y.
/* X and Y are on the same level in the tree. */
 :-op(500,xfx,'is_same_level_as').
 X is_same_level_as  X .
 X is_same_level_as  Y :- W is_parent X,
                          Z is_parent Y,
                          W is_same_level_as Z.
/* Depth of node in the tree. */
 :- op(500,xfx,'has_depth').
 a has_depth 0 :- !.
 Node has_depth D :- Mother is_parent Node,
                     Mother has_depth D1,
                     D is D1 + 1.
/* Locate node by finding a path from root down to the node. */
 locate(Node) :- path(Node),
                 write(Node),
                 nl.
 path(a).                          /* Can start at a.      */
 path(Node) :- Mother is_parent Node,  /* Choose parent,       */
               path(Mother),           /*  find path and then  */
               write(Mother),
               write(' --> ').
 /*  Calculate the height of a node, length of longest  path to a leaf under the node. */

height(N,H) :- setof(Z,ht(N,Z),Set),  /* See section 2.8 for 'setof'.  */
                max(Set,0,H).

 ht(Node,0) :- leaf(Node), !.
 ht(Node,H) :- Node is_parent Child,
               ht(Child,H1),
               H is H1 +1.

 leaf(Node) :- not(is_parent(Node,Child)).
```

```
max([],M,M).
max([X|R],M,A) :- (X > M -> max(R,X,A) ;
                   max(R,M,A)).
```

The 'is_sibling_of' relationship tests whether two nodes have a common parent in the tree. For example,

```
?- h is_sibling_of  S.
S=g ;
S=i ;
no
```

Viva Questions:

1- Incorporate the new location into the game. Note that due to data and procedure abstraction, we need only change the low level predicates that deal directly with location. The higher level predicates, such as look/0 and take/1 are unaffected by the change.

2- Use structures to enhance the customer order entry application. For example, include a structure for each customers address.

EXPERIMENT NO. 9

Aim: **Write a prolog program to find length of given list.**

Lists

Lists are powerful data structures for holding and manipulating groups of things.

In Prolog, a list is simply a collection of terms. The terms can be any Prolog data types, including structures and other lists. Syntactically, a list is denoted by square brackets with the terms separated by commas. For example, a list of things in the kitchen is represented as

```
[apple, broccoli, refrigerator]
```

Rather than having separate location predicates for each thing, we can have one location predicate per container, with a list of things in the container.

```
loc_list([apple, broccoli, crackers], kitchen).
loc_list([desk, computer], office).
loc_list([flashlight, envelope], desk).
loc_list([stamp, key], envelope).
loc_list(['washing machine'], cellar).
loc_list([nani], 'washing machine').
```

The special notation for list structures.

```
[X | Y]
```

There is a special list, called the empty list, which is represented by a set of empty brackets ([]). It is also referred to as **nil**. It can describe the lack of contents of a place or thing.

```
loc_list([], hall)
```

Unification works on lists just as it works on other data structures. With what we now know about lists we can ask

```
?- loc_list(X, kitchen).
X = [apple, broccoli, crackers]

?- [_,X,_] = [apples, broccoli, crackers].
X = broccoli
```

Production rule:

length ⟶ increment,length

Clauses:

```
length([ ] , 0).
length([_ | T], N):- length(T,M),
                     N=M+1.
```

Queries:

```
?- length([1,2,3,4,5,6,7], A).
  A= 7

?- length([a,b,c,d,e,f,g,h,i], B).
  B= 9

?- length([“class”, “ram”, “school”],C).
  C= 3
```

Viva Questions:

1- Write list utilities that perform the following functions.

- Remove a given element from a list
- Find the element after a given element
- Split a list into two lists at a given element (Hint - append/3 is close.)
- Get the last element of a list
- Count the elements in a list (Hint - the length of the empty list is 0, the length a non-empty list is 1 + the length of its tail.)

2- Because write/1 only takes a single argument, multiple 'writes' are necessary for writing a mixed string of text and variables. Write a list utility respond/1 which takes as its single argument a list of terms to be written. This can be used in the game to communicate with the player. For example

```
respond(['You can''t get to the', Room, 'from here'])
```

3- Lists with a variable tail are called open lists. They have some interesting properties. For example, member/2 can be used to add items to an open list. Experiment with and trace the following queries.

```
?- member(a,X).
?- member(b, [a,b,c|X]).
```

```
?- member(d, [a,b,c|X]).
?- OpenL = [a,b,c|X], member(d, OpenL), write(OpenL).
```

4- Predict the results of the following queries.

```
?- [a,b,c,d] = [H|T].
?- [a,[b,c,d]] = [H|T].
?- [] = [H|T].
?- [a] = [H|T].
?- [apple,3,X,'What?'] = [A,B|Z].
?- [[a,b,c],[d,e,f],[g,h,i]] = [H|T].
?- [a(X,c(d,Y)), b(2,3), c(d,Y)] = [H|T].
```

5- Consider the following Prolog program

```
parent(p1,p2).
parent(p2,p3).
parent(p3,p4).
parent(p4,p5).

ancestor(A,D,[A]) :- parent(A,D).
ancestor(A,D,[X|Z]) :-
      parent(X,D),
      ancestor(A,X,Z).
```

6- What is the purpose of the third argument to ancestor?

7- Predict the response to the following queries. Check by tracing in Prolog.

```
?- ancestor(a2,a3,X).
?- ancestor(a1,a5,X).
?- ancestor(a5,a1,X).
```

Assignments:

Aim: Write a prolog program to check whether a given element is member of list or not.

Aim: Write a prolog program to append a list L1 & L2, and create a new list L3.

Aim: Write a prolog program to print the position of element in the list.

Aim: Test whether a set is subset of a given set or not.

EXPERIMENT NO. 10

Aim: Write predicates for natural language sentences.

Introduction to Natural Language

Prolog is especially well-suited for developing natural language systems. In this chapter we will create an English front end for Nani Search.

But before moving to Nani Search, we will develop a natural language parser for a simple subset of English. Once that is understood, we will use the same technology for Nani Search.

The simple subset of English will include sentences such as

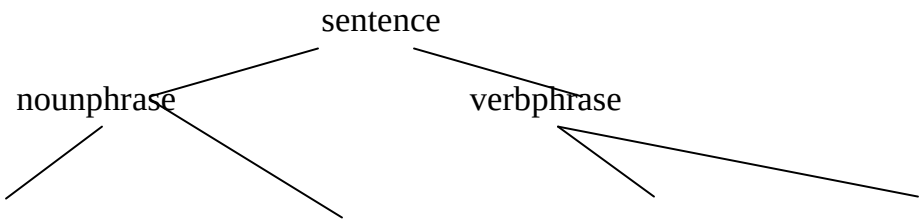
- The dog ate the bone.
- The big brown mouse chases a lazy cat.

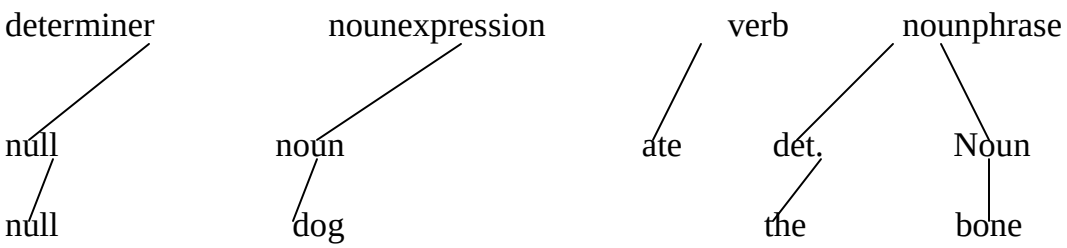
This grammar can be described with the following grammar rules. (The first rule says a sentence is made up of a noun phrase followed by a verb phrase. The last rule says an adjective is either 'big', or 'brown', or 'lazy.' The '|' means 'or.')

Production rules :--

sentence □ noun phrase, verb phrase.
noun phrase □ determiner, noun expression.
noun phrase □ nounexpression.
noun expression □noun.
noun expression □ adjective, noun expression.
verb phrase □ verb, noun phrase.
determiner □ the | a.
noun □ dog | bone | mouse | cat.
verb □ ate | chases.
adjective □ big | brown | lazy.

Parse Tree:-





Clauses

```
sentence(S) :-
  nounphrase(S-S1),
  verbphrase(S1-[]).
noun([dog|X]-X).
noun([cat|X]-X).
noun([mouse|X]-X).
verb([ate|X]-X).
verb([chases|X]-X).
adjective([big|X]-X).
adjective([brown|X]-X).
adjective([lazy|X]-X).
determiner([the|X]-X).
determiner([a|X]-X).
nounphrase(NP-X):-
  determiner(NP-S1),
  nounexpression(S1-X).
nounphrase(NP-X):-
  nounexpression(NP-X).
nounexpression(NE-X):-
  noun(NE-X).
nounexpression(NE-X):-
  adjective(NE-S1),
  nounexpression(S1-X).
verbphrase(VP-X):-
  verb(VP-S1),
  nounphrase(S1-X).
```

Queries:

```
?- noun([dog,ate,the,bone]-X).
X = [ate,the,bone]
?- verb([dog,ate,the,bone]-X).
No
?- sentence([the,lazy,mouse,ate,a,dog]).
Yes
```

```
?- sentence([the,dog,ate]).
No

?- sentence([the,cat,jumps,the,mouse]).
No
```

Viva Questions:

1- Expand the natural language capabilities to handle all of the commands of Nani Search.

2- Expand the natural language front end to allow for compound sentences, such as "go to the kitchen and take the apple," or "take the apple and the broccoli."

3- Expand the natural language to allow for pronouns. To do this the 'noun' predicate must save the last noun and its type. When the word 'it' is encountered pick up that last noun. Then 'take the apple' followed by 'eat it' will work. (You will probably have to go directly to the difference list notation to make sentences such as "turn it on" work.)

4- Build a natural language query system that responds to queries such as "Who are dennis' children?" and "How many nephews does jay have?" Assuming you write a predicate get_query/1 that returns a Prolog query, you can call the Prolog query with the call/1 built-in predicate. For example,

```
main_loop :-
  repeat,
  get_query(X),
  call(X),
  X = end.
```

Assignment:

Write predicates for natural language complex sentences.

EXPERIMENT NO. 11

Aim:- Write a program to solve the Monkey Banana problem.

Imagine a room containing a monkey, chair and some bananas. That have been hanged from the center of ceiling. If the monkey is clever enough he can reach the bananas by placing the chair directly below the bananas and climb on the chair .

The problem is to prove the monkey can reach the bananas.

Production Rules

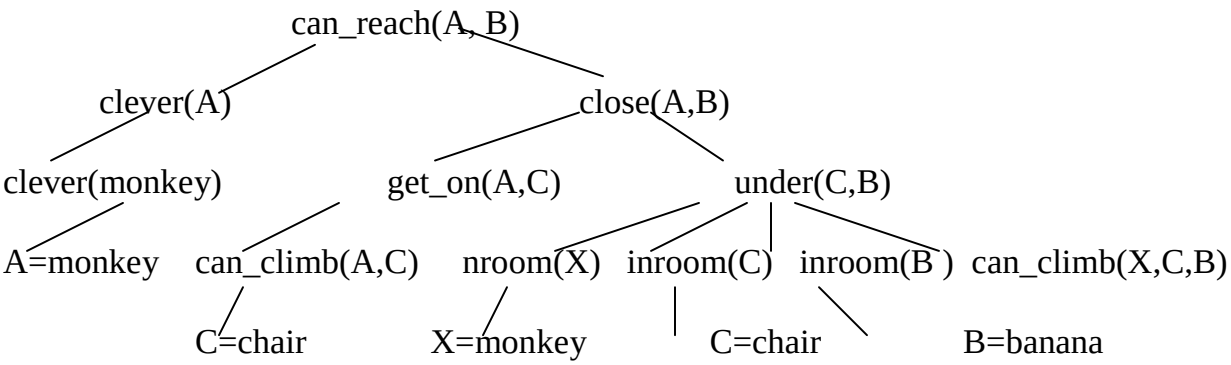
can_reach clever,close.

get_on: can_climb.

under in room,in_room, in_room,can_climb.

Close get_on,under| tall

Parse Tree



So Can_climb(monkey,chair) close(monkey,banana)

A=monkey B=banana

Solution:-

Clauses:

```
in_room(bananas).
in_room(chair).
in_room(monkey).
```

```
clever(monkey).
can_climb(monkey, chair).
tall(chair).
can_move(monkey, chair, bananas).
can_reach(X, Y):-clever(X),close(X, Y).
get_on(X,Y):- can_climb(X,Y).

under(Y,Z):-in_room(X),in_room(Y),
        in_room(Z),can_climb(X,Y,Z).
close(X,Z):-get_on(X,Y), under(Y,Z);
        tall(Y).
```

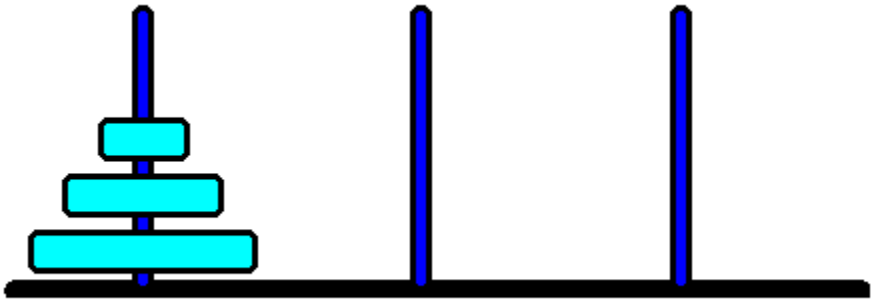
Queries:

```
?- can_reach(A, B).
  A = monkey.
  B = banana.

?- can_reach(monkey, banana).
  Yes.
```

EXPERIMENT NO. 12

Aim:- Write a program to solve Tower of Hanoi.

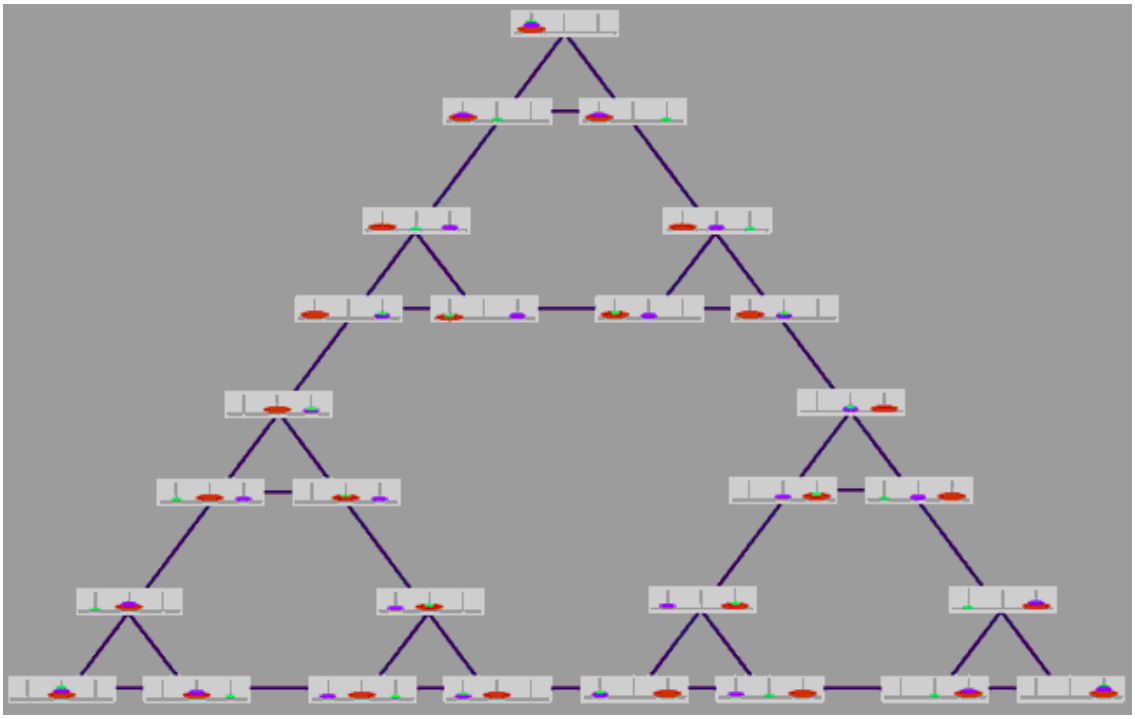
This object of this famous puzzle is to move N disks from the left peg to the right peg using the center peg as an auxiliary holding peg. At no time can a larger disk be placed upon a smaller disk. The following diagram depicts the starting setup for N=3 disks.



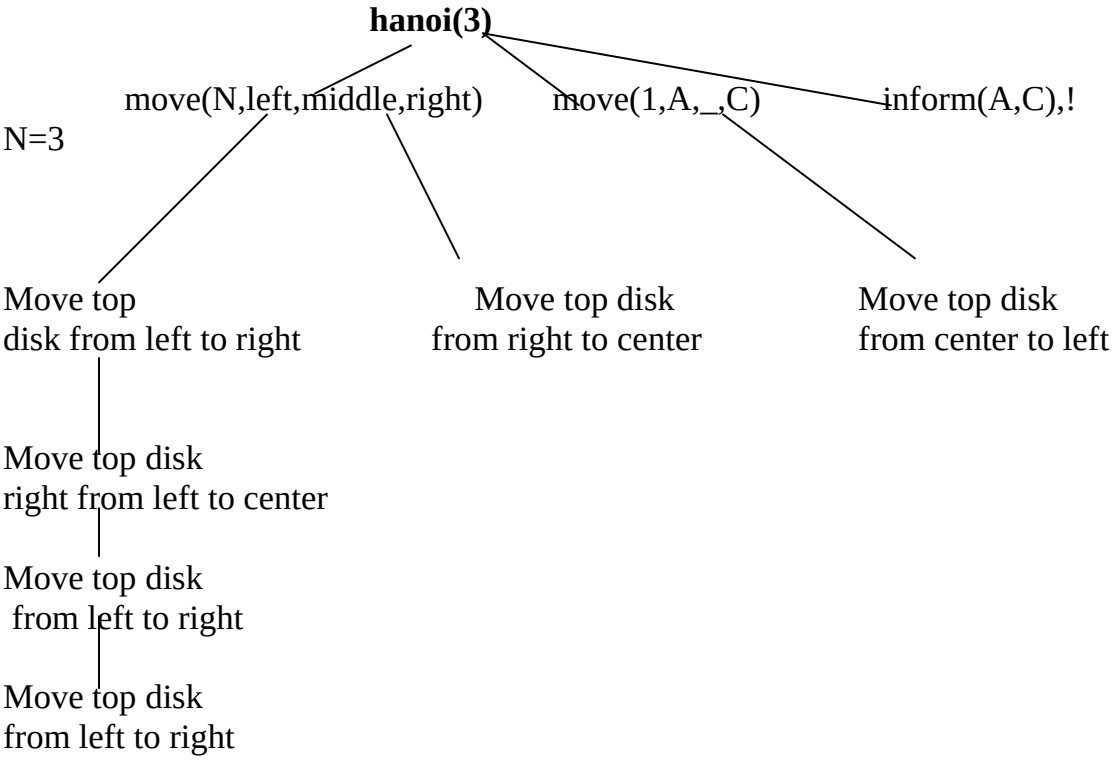
Production Rules

hanoi(N)□move(N,left,middle,right).
move(1,A,_,C)□inform(A,C),fail.
move(N,A,B,C)□N1=N-1,move(N1,A,C,B),inform(A,C),move(N1,B,A,C).

Diagram:-



Parse Tree:-



Solution:-

```
domains
  loc =right;middle;left

predicates
  hanoi(integer)
  move(integer,loc,loc,loc)
  inform(loc,loc)

clauses
  hanoi(N):-
          move(N,left,middle,right).
          move(1,A,_,C):-
          inform(A,C),!.

  move(N,A,B,C):-
```

```
        N1=N-1,
        move(N1,A,C,B),
        inform(A,C),
        move(N1,B,A,C).

inform(Loc1, Loc2):-
        write("\nMove a disk from ", Loc1, " to ", Loc2).
```

goal

```
 hanoi(3).
Move(3,left,right,center).
Move top disk from left to right
Move top disk from left to center
Move top disk from right to center
Move top disk from left to right
Move top disk from center to left
Move top disk from center to right
Move top disk from left to right

Yes
```

Viva Questions:

1 What is Tower of Hanoi problem?

2 What is the role of recursion in Tower of Hanoi problem?

3 what will the effect if the number of disk is 4

Assignment:

Aim: Draw the search tree for Tower of Hanoi for N=4.

EXPERIMENT NO. 13

Aim:- Write a program to solve 8-Puzzle problem.

The title of this section refers to a familiar and popular sliding tile puzzle that has been around for at least forty years. The most frequent older versions of this puzzle have numbers or letters an the sliding tiles, and the player is supposed to slide tiles into new positions in order to realign a scrambled puzzle back into a goal alignment. For illustration, we use the 3 x 3 8-tile version, which is depicted here in goal configuration

7	2	3
4	6	5
1	8	

To represent these puzzle "states" we will use a Prolog term representation employing '/' as a separator. The positions of the tiles are listed (separated by '/') from top to bottom, and from left to right. Use "0" to represent the empty tile (space). For example, the goal is ...
goal(1/2/3/8/0/4/7/6/5).

Production Rules :-

h_function(Puzz,H) □ p_fcn(Puzz,P), s_fcn(Puzz,S),H is P + 3*S.

The 'move' productions are defined as follows.
move(P,C,left) □ left(P,C).
move(P,C,up) □ up(P,C).
move(P,C,right) □ right(P,C).
move(P,C,down) □ down(P,C).

```
p_fcn(A/B/C/D/E/F/G/H/I, P) □   a(A,Pa), b(B,Pb), c(C,Pc),
                                d(D,Pd), e(E,Pe), f(F,Pf),
                                g(G,Pg), h(H,Ph), i(I,Pi),
                               P is Pa+Pb+Pc+Pd+Pe+Pf+Pg+Ph+Pg+Pi.
```

```
s_fcn(A/B/C/D/E/F/G/H/I, S) □1 s_aux(A,B,S1), s_aux(B,C,S2),
                               s_aux(C,F,S3),  s_aux(F,I,S4),
                               s_aux(I,H,S5), s_aux(H,G,S6),
                               s_aux(G,D,S7), s_aux(D,A,S8),
                               s_aux(E,S9),
```

S is S1+S2+S3+S4+S5+S6+S7+S8+S9.

```
s_aux(0,0) □ cut
s_aux(X,Y,0) :- Y is X+1, !.
s_aux(8,1,0) :- !.
```

The heuristic function we use here is a combination of two other estimators: p_fcn, the Manhattan distance function, and s_fcn, the sequence function, all as explained in Nilsson (1980), which estimates how badly out-of-sequence the tiles are (around the outside).

```
h_function(Puzz,H) :- p_fcn(Puzz,P),
                      s_fcn(Puzz,S),
                      H is P + 3*S.
```

The 'move' predicate is defined as follows.

```
move(P,C,left) :-  left(P,C).
move(P,C,up) :-  up(P,C).
move(P,C,right) :-  right(P,C).
move(P,C,down) :-  down(P,C).
```

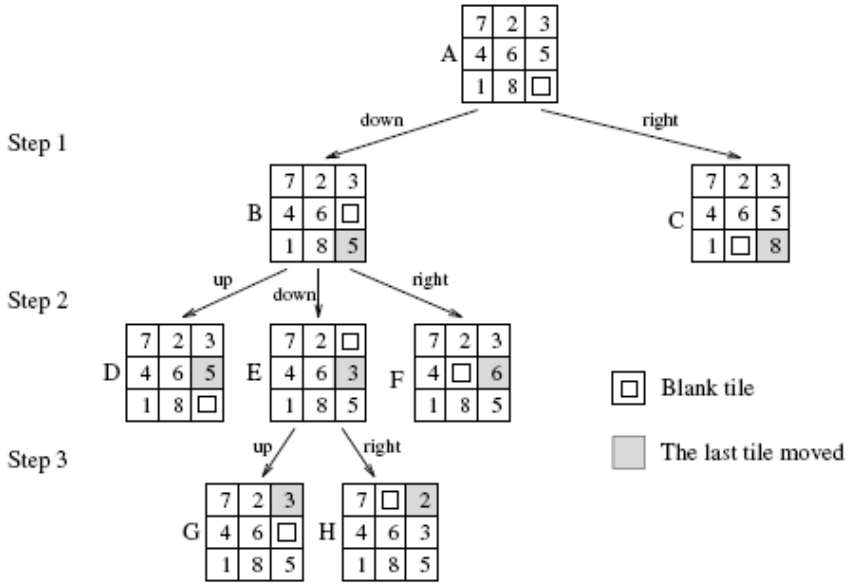
Here is the code for p and s.

```
  %%% Manhattan distance
p_fcn(A/B/C/D/E/F/G/H/I, P) :-
    a(A,Pa), b(B,Pb), c(C,Pc),
    d(D,Pd), e(E,Pe), f(F,Pf),
    g(G,Pg), h(H,Ph), i(I,Pi),
    P is Pa+Pb+Pc+Pd+Pe+Pf+Pg+Ph+Pg+Pi.

a(0,0). a(1,0). a(2,1). a(3,2). a(4,3). a(5,4). a(6,3). a(7,2). a(8,1).
b(0,0). b(1,0). b(2,0). b(3,1). b(4,2). b(5,3). b(6,2). b(7,3). b(8,2).
c(0,0). c(1,2). c(2,1). c(3,0). c(4,1). c(5,2). c(6,3). c(7,4). c(8,3).
d(0,0). d(1,1). d(2,2). d(3,3). d(4,2). d(5,3). d(6,2). d(7,2). d(8,0).
e(0,0). e(1,2). e(2,1). e(3,2). e(4,1). e(5,2). e(6,1). e(7,2). e(8,1).
f(0,0). f(1,3). f(2,2). f(3,1). f(4,0). f(5,1). f(6,2). f(7,3). f(8,2).
g(0,0). g(1,2). g(2,3). g(3,4). g(4,3). g(5,2). g(6,2). g(7,0). g(8,1).
h(0,0). h(1,3). h(2,3). h(3,3). h(4,2). h(5,1). h(6,0). h(7,1). h(8,2).
i(0,0). i(1,4). i(2,3). i(3,2). i(4,1). i(5,0). i(6,1). i(7,2). i(8,3).

  %%% the out-of-cycle function
s_fcn(A/B/C/D/E/F/G/H/I, S) :-
    s_aux(A,B,S1), s_aux(B,C,S2), s_aux(C,F,S3),
    s_aux(F,I,S4), s_aux(I,H,S5), s_aux(H,G,S6),
    s_aux(G,D,S7), s_aux(D,A,S8), s_aux(E,S9),
    S is S1+S2+S3+S4+S5+S6+S7+S8+S9.
```

```
s_aux(0,0) :- !.
s_aux(_,1).

s_aux(X,Y,0) :- Y is X+1, !.
s_aux(8,1,0) :- !.
s_aux(_,_,2).
```

The Prolog program from the previous section and the program outlined in this section can be used as an 8-puzzle solver.

```
?- solve(0/8/1/2/4/3/7/6/5, S).
```



Solution:

```
left( A/0/C/D/E/F/H/I/J , 0/A/C/D/E/F/H/I/J ).
left( A/B/C/D/0/F/H/I/J , A/B/C/0/D/F/H/I/J ).
left( A/B/C/D/E/F/H/0/J , A/B/C/D/E/F/0/H/J ).
left( A/B/0/D/E/F/H/I/J , A/0/B/D/E/F/H/I/J ).
left( A/B/C/D/E/0/H/I/J , A/B/C/D/0/E/H/I/J ).
left( A/B/C/D/E/F/H/I/0 , A/B/C/D/E/F/H/0/I ).

up( A/B/C/0/E/F/H/I/J , 0/B/C/A/E/F/H/I/J ).
up( A/B/C/D/0/F/H/I/J , A/0/C/D/B/F/H/I/J ).
up( A/B/C/D/E/0/H/I/J , A/B/0/D/E/C/H/I/J ).
```

up(A/B/C/D/E/F/0/I/J , A/B/C/0/E/F/D/I/J).
up(A/B/C/D/E/F/H/0/J , A/B/C/D/0/F/H/E/J).
up(A/B/C/D/E/F/H/I/0 , A/B/C/D/E/0/H/I/F).

right(A/0/C/D/E/F/H/I/J , A/C/0/D/E/F/H/I/J).
right(A/B/C/D/0/F/H/I/J , A/B/C/D/F/0/H/I/J).
right(A/B/C/D/E/F/H/0/J , A/B/C/D/E/F/H/J/0).
right(0/B/C/D/E/F/H/I/J , B/0/C/D/E/F/H/I/J).
right(A/B/C/0/E/F/H/I/J , A/B/C/E/0/F/H/I/J).
right(A/B/C/D/E/F/0/I/J , A/B/C/D/E/F/I/0/J).

down(A/B/C/0/E/F/H/I/J , A/B/C/H/E/F/0/I/J).
down(A/B/C/D/0/F/H/I/J , A/B/C/D/I/F/H/0/J).
down(A/B/C/D/E/0/H/I/J , A/B/C/D/E/J/H/I/0).
down(0/B/C/D/E/F/H/I/J , D/B/C/0/E/F/H/I/J).
down(A/0/C/D/E/F/H/I/J , A/E/C/D/0/F/H/I/J).
down(A/B/0/D/E/F/H/I/J , A/B/F/D/E/0/H/I/J).

Viva Questions:

1. What is 8-puzze problem.

2. What can be the next states if the board is in following position?

a)

1	2	3
8		4
7	6	5

b)

2	**5**	**1**
3	**7**	**6**
4	**8**	

Assignment:

Aim: - Solve the program for the sequence (8,7,6,5,4,1,2,3,0).

EXPERIMENT NO. 14

Aim:-Write a program to solve the Farmer/Wolf/Goat/

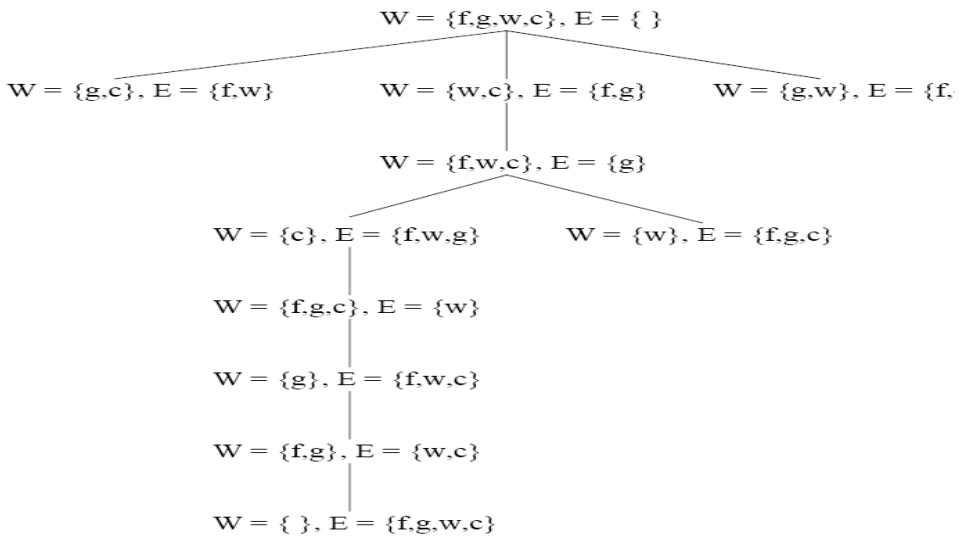
Cabbage problem.

A farmer and his goat, wolf, and cabbage come to a river that they wish to cross. There is a boat, but it has only room for two, and the farmer is the only one that can row. If the goat and the cabbage get in the boat at the same time, the goat will eat the cabbage. Similarly, if the wolf and the goat are together without the farmer, the goat will be eaten. Devise a series of crossings of the river so that all concerned make it across safely.

Production rules :-

```
go(StartState,GoalState)  path(StartState,GoalState,[StartState],Path),
                          write("A solution is:\n"), write_path(Path).
 path(StartState,GoalState,VisitedPath,Path)  move(StartState,NextState),
not( unsafe(NextState) ), not( member(NextState,VisitedPath) ),
path(NextState,GoalState,[NextState|VisitedPath],Path),!.
  path(GoalState,GoalState,Path,Path).
  move(state(X,X,G,C),state(Y,Y,G,C))opposite(X,Y).
move(state(X,W,X,C),state(Y,W,Y,C))opposite(X,Y).
  move(state(X,W,G,X),state(Y,W,G,Y))opposite(X,Y
move(state(X,W,G,C),state(Y,W,G,C))opposite(X,Y).
```

Parse Tree:-



In words:

1. Farmer takes goat across the river. W = {w,c}, E = {f,g}
2. Farmer comes back alone. W = {f,w,c}, E = {g}
3. Farmer takes wolf across. W = {c}, E = {f,g,w}
4. Farmer comes back with goat. W = {f,g,c}, E = {w}
5. Farmer takes cabbage across. W = {g}, E = {f,w,c}
6. Farmer comes back alone. W = {f,g}, E = {w,c}
7. Farmer takes goat across. W = {∅}, E = {f,g,w,c}

Solution:-

```
domains
  LOC   = east ; west
  STATE = state(LOC farmer,LOC wolf,LOC goat,LOC cabbage)
  PATH  = STATE*
predicates
  go(STATE,STATE)              % Start of the algorithm
  path(STATE,STATE,PATH,PATH)   % Finds a path from one state to another
  nondeterm move(STATE,STATE)   % Transfer a system from one side to another
  opposite(LOC,LOC)            % Gives a location on the opposite side
  nondeterm unsafe(STATE)       % Gives the unsafe states
  nondeterm member(STATE,PATH)  % Checks if the state is already visited
  write_path(PATH)
  write_move(STATE,STATE)
clauses
  go(StartState,GoalState):-
       path(StartState,GoalState,[StartState],Path),
       write("A solution is:\n"),
       write_path(Path).

  path(StartState,GoalState,VisitedPath,Path):-
       move(StartState,NextState),                  % Find a move
```

```
        not( unsafe(NextState) ),                    % Check that it is not unsage
        not( member(NextState,VisitedPath) ),        % Check that we have not had this
situation before
        path( NextState,GoalState,[NextState|VisitedPath],Path),!.
  path(GoalState,GoalState,Path,Path).               % The final state is reached

  move(state(X,X,G,C),state(Y,Y,G,C)):-opposite(X,Y). % Move FARMER + WOLF
  move(state(X,W,X,C),state(Y,W,Y,C)):-opposite(X,Y). % Move FARMER + GOAT
  move(state(X,W,G,X),state(Y,W,G,Y)):-opposite(X,Y). % Move FARMER +
CABBAGE
  move(state(X,W,G,C),state(Y,W,G,C)):-opposite(X,Y). % Move ONLY FARMER
  opposite(east,west).
  opposite(west,east).
  unsafe( state(F,X,X,_) ):- opposite(F,X),!.  % The wolf eats the goat
  unsafe( state(F,_,X,X) ):- opposite(F,X),!.  % The goat eats the cabbage
  member(X,[X|_]):-!.
  member(X,[_|L]):-member(X,L).
  write_path( [H1,H2|T] ) :-
        write_move(H1,H2),
        write_path([H2|T]).
  write_path( [] ).
  write_move( state(X,W,G,C), state(Y,W,G,C) ) :-!,
    write("The farmer crosses the river from ",X," to ",Y),nl.
  write_move( state(X,X,G,C), state(Y,Y,G,C) ) :-!,
    write("The farmer takes the Wolf from ",X," of the river to ",Y),nl.
  write_move( state(X,W,X,C), state(Y,W,Y,C) ) :-!,
    write("The farmer takes the Goat from ",X," of the river to ",Y),nl.
  write_move( state(X,W,G,X), state(Y,W,G,Y) ) :-!,
    write("The farmer takes the cabbage from ",X," of the river to ",Y),nl.
```

goal

```
 go(state(east, east, east, east ),
   state(west ,west,west,west)),
    write("solved").
```

A solution is:
The farmer takes the Goat from west of the river to east
The farmer crosses the river from east to west
The farmer takes the cabbage from west of the river to east
The farmer takes the Goat from east of the river to west
The farmer takes the Wolf from west of the river to east

The farmer crosses the river from east to west
The farmer takes the Goat from west of the river to east

Viva Question:

1. Can we have some another solution for farmer-wolf problem.

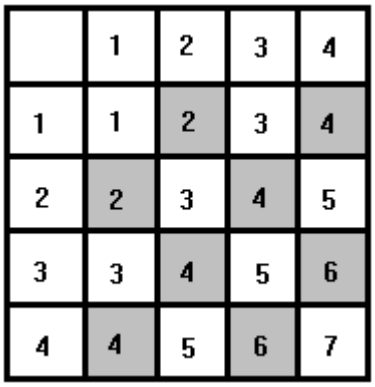
Assignment:

Aim:- Draw one level parse tree for Missionary-cannibal problem.

EXPERIMENT NO. 15

Aim:- **Write a program to solve 4-Queen problem**.

In the 4 Queens problem the object is to place *4* queens on a chessboard in such a way that no queens can capture a piece. This means that no two queens may be placed on the same row, column, or diagonal.



The N Queens Chessboard

```
domains
  queen    = q(integer, integer)
  queens   = queen*
  freelist = integer*
  board    = board(queens, freelist, freelist, freelist, freelist)
predicates
  nondeterm placeN(integer, board, board)
  nondeterm place_a_queen(integer, board, board)
  nondeterm nqueens(integer)
  nondeterm makelist(integer, freelist)
  nondeterm findandremove(integer, freelist, freelist)
  nextrow(integer, freelist, freelist)
clauses
  nqueens(N):-
       makelist(N,L),
       Diagonal=N*2-1,
       makelist(Diagonal,LL),
       placeN(N,board([],L,L,LL,LL),Final),
       write(Final).

  placeN(_,board(D,[],[],D1,D2),board(D,[],[],D1,D2)):-!.
  placeN(N,Board1,Result):-
       place_a_queen(N,Board1,Board2),
```

```
    placeN(N,Board2,Result).

 place_a_queen(N,
          board(Queens,Rows,Columns,Diag1,Diag2),
          board([q(R,C)|Queens],NewR,NewC,NewD1,NewD2)):-
    nextrow(R,Rows,NewR),
    findandremove(C,Columns,NewC),
    D1=N+C-R,findandremove(D1,Diag1,NewD1),
    D2=R+C-1,findandremove(D2,Diag2,NewD2).

 findandremove(X,[X|Rest],Rest).
 findandremove(X,[Y|Rest],[Y|Tail]):-
    findandremove(X,Rest,Tail).

 makelist(1,[1]).
 makelist(N,[N|Rest]) :-
    N1=N-1,makelist(N1,Rest).

 nextrow(Row,[Row|Rest],Rest).
```

goal

```
  nqueens(4),nl.

board([q(1,2),q(2,4),q(3,1),q(4,3),[],[],[7,4,1],[7,4,1])
yes
```

Viva Questions:

1. Explain N-Queen problem.
2. What do you mean by integer *?

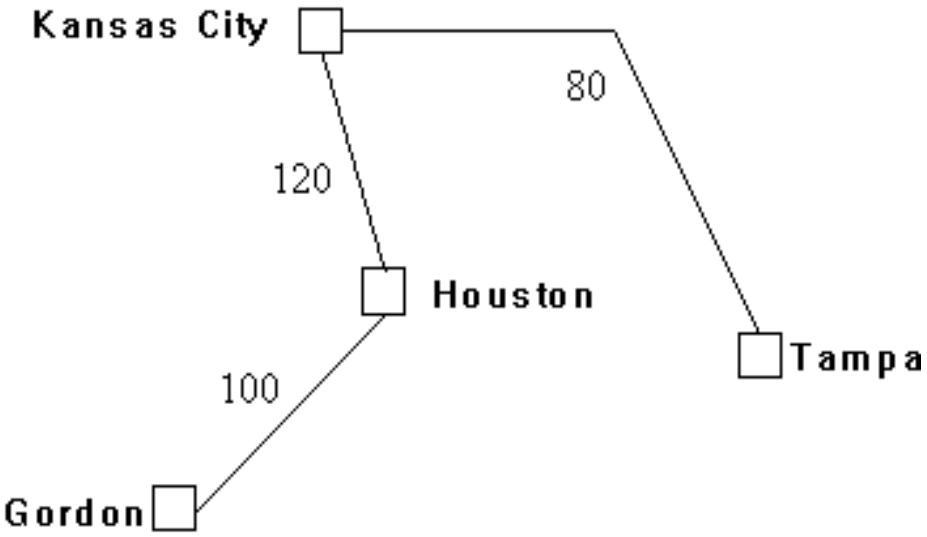
Assignment:

Aim:- Write a program to solve N-Queen problem.

EXPERIMENT NO. 16

Aim:-Write a program to solve traveling salesman problem.

The following is the simplified map used for the prototype:



Production Rules:-

route(Town1,Town2,Distance)□ road(Town1,Town2,Distance).
route(Town1,Town2,Distance)□ road(Town1,X,Dist1),
route(X,Town2,Dist2),
Distance=Dist1+Dist2,

```
domains
  town     = symbol
  distance = integer

predicates
  nondeterm road(town,town,distance)
  nondeterm route(town,town,distance)

clauses
  road("tampa","houston",200).
  road("gordon","tampa",300).
  road("houston","gordon",100).
  road("houston","kansas_city",120).
  road("gordon","kansas_city",130).
 route(Town1,Town2,Distance):-
       road(Town1,Town2,Distance).
   route(Town1,Town2,Distance):-
```

```
    road(Town1,X,Dist1),
    route(X,Town2,Dist2),
    Distance=Dist1+Dist2,
    !.

goal
   route("tampa", "kansas_city", X),
    write("Distance from Tampa to Kansas City  is ",X),nl.
```

Distance from Tampa to Kansas City is 320
X=320
1 Solution

Viva Questions:

1. What do you mean by Traveling Salesman problem?
2. Why you use domain?
3. What is the output of following:
   route(kansascity, gordon, X)).

Assignment :

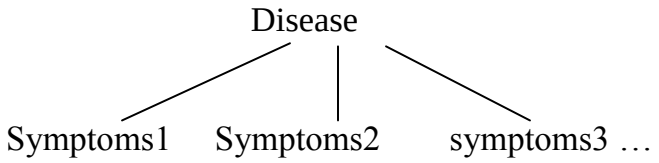
Write a program given the knowledge base,
If Town x is connected to Town y by highway z and bikes are allowed on z, you can get to y from x by bike.
If Town x is connected to y by z then y is also connected to x by z.
If you can get to town q from p and also to town r from town q, you can get to town r from town p.
Town A is connected to Town B by Road 1. Town B is connected to Town C by Road 2.
Town A is connected to Town C by Road 3. Town D is connected to Town E by Road 4.
Town D is connected to Town B by Road 5. Bikes are allowed on roads 3, 4, 5.
Bikes are only either allowed on Road 1 or on Road 2 every day.Convert the following into wff's, clausal form and deduce that `One can get to town B from townD'.

EXPERIMENT NO. 17

Aim:- Write a program for Medical diagnosis.

A medical diagnosis problem will tell you the name of disease we have if we input symptoms.

Parse Tree



Solution:-

```
predicates
        nondeterm disease(symbol).
        nondeterm symptoms(symbol).
clauses
        disease(jaundice):-
         X="indigestion",Y="pale",symptoms(X),symptoms(Y).
        disease(diabetes):-
        X="tiredness",Y="weakness",symptoms(X),symptoms(Y).
        disease(malaria):-
         X="fever",Y="shivering",symptoms(X),symptoms(Y).
        disease(typhoid):-X="fever",Y="lowbp",symptoms(X),symptoms(Y).
        disease(mygrane):-
         X="headache",Y="weakness",symptoms(X),symptoms(Y).
        symptoms(fever).
        symptoms(lowbp).
        symptoms(weakness).
        symptoms(shivering).
        symptoms(tiredness).

goal
 A="weakness",B="tiredness",disease(Z).
Z=diabetes.
```

Viva Questions:

1. What is the use of this program?
2. Do you know any other practical use of AI in medical field?

3. What will be the output if the symptoms not match?

Assignment:

Aim:- Write a prolog program to prescribe medicine for corresponding disease.

EXPERIMENT NO. 18

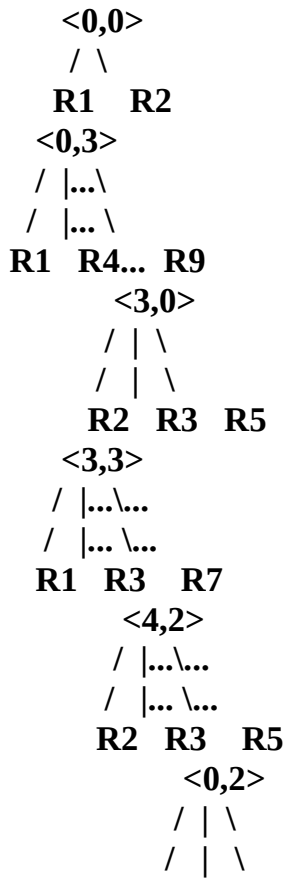
Aim:- Write a program for water jug problem.

"You are given two jugs, a 4-gallon one and a 3-gallon one. Neither have any measuring markers on it. There is a tap that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?".

Production Rules:-

R1: (x,y) --> (4,y) if x < 4
R2: (x,y) --> (x,3) if y < 3
R3: (x,y) --> (x-d,y) if x > 0
R4: (x,y) --> (x,y-d) if y > 0
R5: (x,y) --> (0,y) if x > 0
R6: (x,y) --> (x,0) if y > 0
R7: (x,y) --> (4,y-(4-x)) if x+y >= 4 and y > 0
R8: (x,y) --> (x-(3-y),y) if x+y >= 3 and x > 0
R9: (x,y) --> (x+y,0) if x+y =< 4 and y > 0
R10: (x,y) --> (0,x+y) if x+y =< 3 and x > 0

Parse Tree



R1 R7 R9
<2,0>

Solution:-

Clauses

```
water_jugs :-
   SmallCapacity = 3,
   LargeCapacity = 4,
   Reservoir is SmallCapacity + LargeCapacity + 1,
   volume( small, Capacities, SmallCapacity ),
   volume( large, Capacities, LargeCapacity ),
   volume( reservoir, Capacities, Reservoir ),
   volume( small, Start, 0 ),
   volume( large, Start, 0 ),
   volume( reservoir, Start, Reservoir ),
   volume( large, End, 2 ),
   water_jugs_solution( Start, Capacities, End, Solution ),
   phrase( narrative(Solution, Capacities, End), Chars ),
   put_chars( Chars ).

water_jugs_solution( Start, Capacities, End, Solution ) :-
   solve_jugs( [start(Start)], Capacities, [], End, Solution ).

solve_jugs( [Node|Nodes], Capacities, Visited, End, Solution ) :-
        node_state( Node, State ),
        ( State = End ->
                Solution = Node
        ; otherwise ->
                findall(
                        Successor,
                        successor(Node, Capacities, Visited, Successor),
                        Successors
                        ),
                append( Nodes, Successors, NewNodes ),
                solve_jugs( NewNodes, Capacities, [State|Visited], End, Solution )
        ).

successor( Node, Capacities, Visited, Successor ) :-
        node_state( Node, State ),
        Successor = node(Action,State1,Node),
```

```
        jug_transition( State, Capacities, Action, State1 ),
        \+ member( State1, Visited ).
jug_transition( State0, Capacities, empty_into(Source,Target), State1 ) :-
        volume( Source, State0, Content0 ),
        Content0 > 0,
        jug_permutation( Source, Target, Unused ),
        volume( Target, State0, Content1 ),
        volume( Target, Capacities, Capacity ),
        Content0 + Content1 =< Capacity,
        volume( Source, State1, 0 ),
        volume( Target, State1, Content2 ),
        Content2 is Content0 + Content1,
        volume( Unused, State0, Unchanged ),
        volume( Unused, State1, Unchanged ).
jug_transition( State0, Capacities, fill_from(Source,Target), State1 ) :-
        volume( Source, State0, Content0 ),
        Content0 > 0,
        jug_permutation( Source, Target, Unused ),
        volume( Target, State0, Content1 ),
        volume( Target, Capacities, Capacity ),
        Content1 < Capacity,
        Content0 + Content1 > Capacity,
        volume( Source, State1, Content2 ),
        volume( Target, State1, Capacity ),
        Content2 is Content0 + Content1 - Capacity,
        volume( Unused, State0, Unchanged ),
        volume( Unused, State1, Unchanged ).

volume( small, jugs(Small, _Large, _Reservoir), Small ).
volume( large, jugs(_Small, Large, _Reservoir), Large ).
volume( reservoir, jugs(_Small, _Large, Reservoir), Reservoir ).


jug_permutation( Source, Target, Unused ) :-
        select( Source, [small, large, reservoir], Residue ),
        select( Target, Residue, [Unused] ).


node_state( start(State), State ).
node_state( node(_Transition, State, _Predecessor), State ).


narrative( start(Start), Capacities, End ) -->
```

```
        "Given three jugs with capacities of:", newline,
        literal_volumes( Capacities ),
        "To obtain the result:", newline,
        literal_volumes( End ),
        "Starting with:", newline,
        literal_volumes( Start ),
        "Do the following:", newline.
narrative( node(Transition, Result, Predecessor), Capacities, End ) -->
        narrative( Predecessor, Capacities, End ),
        literal_action( Transition, Result ).

literal_volumes( Volumes ) -->
        indent, literal( Volumes ), ";", newline.

literal_action( Transition, Result ) -->
        indent, "- ", literal( Transition ), " giving:", newline,
        indent, indent, literal( Result ), newline.

literal( empty_into(From,To) ) -->
        "Empty the ", literal( From ), " into the ",
        literal( To ).
literal( fill_from(From,To) ) -->
        "Fill the ", literal( To ), " from the ",
        literal( From ).
literal( jugs(Small,Large,Reservoir) ) -->
        literal_number( Small ), " gallons in the small jug, ",
        literal_number( Large ), " gallons in the large jug and ",
        literal_number( Reservoir ), " gallons in the reservoir".
literal( small ) --> "small jug".
literal( large ) --> "large jug".
literal( reservoir ) --> "reservoir".

literal_number( Number, Plus, Minus ) :-
        number( Number ),
        number_chars( Number, Chars ),
        append( Chars, Minus, Plus ).

indent --> "  ".

newline --> "  ".
```

Goal

?- water_jugs.

Given three jugs with capacities of:
3 gallons in the small jug, 4 gallons in the large jug and 8 gallons in the reservoir;
To obtain the result:
0 gallons in the small jug, 2 gallons in the large jug and 6 gallons in the reservoir;
Starting with:
0 gallons in the small jug, 0 gallons in the large jug and 8 gallons in the reservoir;
Do the following:
- Fill the small jug from the reservoir giving:
  3 gallons in the small jug, 0 gallons in the large jug and 5 gallons in the reservoir
- Empty the small jug into the large jug giving:
  0 gallons in the small jug, 3 gallons in the large jug and 5 gallons in the reservoir
- Fill the small jug from the reservoir giving:
  3 gallons in the small jug, 3 gallons in the large jug and 2 gallons in the reservoir
- Fill the large jug from the small jug giving:
  2 gallons in the small jug, 4 gallons in the large jug and 2 gallons in the reservoir
- Empty the large jug into the reservoir giving:
  2 gallons in the small jug, 0 gallons in the large jug and 6 gallons in the reservoir
- Empty the small jug into the large jug giving:
  0 gallons in the small jug, 2 gallons in the large jug and 6 gallons in the reservoir

yes

Viva Questions:

1. Witch type of search algorithm you used?
2. Can you use any other algorithm to implement it?
3. Can I use this program to solve other water jug problem?
4. Write 2 more solution for the water jug problem.

Assignment:

Aim :- Write a program for water jug problem 5-lit and 2-lit jug, at the end 5-lit jug have

1 liter of water**.**

EXPERIMENT NO. 19

Aim:- Write a program to solve

```
  SEND
 +MORE
 MONEY
```

We have to assign digit values to each letter coming in the three words, no two letters can have same digit value.

Now we start making some assumption as SEND and MORE have 4 letters but MONEY have 5 so there is a carry so M=1. and so on…

S+M+C3>9 to generate carry so S+C3>8

C3=0 C3=1

Then S=9 Then S=8

For S=9 or S=8 , S+M +C3=10
O=0. only

Now when O=0 then C2+O+E=N
C2 can not be 0 becoz if C2=0 then E=N which is not allowed so N=E+1(C2=1)

Solution:-

Predicates

```
nondeterm digit(integer).
nondeterm good(integer,integer,integer,integer,integer,integer,integer,integer).
nondeterm sum(integer,integer,integer,integer,integer,integer,integer,integer).
```

Clauses

```
digit(1).
digit(2).
digit(3).
digit(4).
digit(5).
digit(6).
digit(7).
```

```
digit(8).
digit(9).
digit(0).
good(S,E,N,D,M,O,R,Y) :- digit(S),S<>0,

                digit(E),E<>S,
                digit(N),N<>E,N<>S,
                digit(D),D<>N,D<>E,D<>S,
digit(M),M<>D,M<>N,M<>E,M<>S,M<>0,
digit(O),O<>M,O<>D,O<>N,O<>E,O<>S,
digit(R),R<>O,R<>M,R<>D,R<>N,R<>E,R<>S,
digit(Y),Y<>R,Y<>O,Y<>M,Y<>D,Y<>N,Y<>E,Y<>S,
                sum(S,E,N,D,M,O,R,Y).
sum(S,E,N,D,M,O,R,Y) :- N1 = 1000*S+100*E+10*N+D,
                N2 = 1000*M+100*O+10*R+E,
              N3=10000*M+1000*O+100*N+10*E+Y,
                N3 = N1+N2.
```

Goal

```
?- good(S,E,N,D,M,O,R,Y).
S=9, E=5, N=6, D=7, M=1, O=0, R=8, Y=2
```

1 Solution

Viva Questions:

1. Answer the following:
   a) digit(X).
   b) sum(S,E,N,D,M,O,R,Y).
2. Why sum is defined as "nondeterm"?
3. Is any other solution possible?

Assignment:

Aim:- Write a prolog program for Adding DONALD + GERALD =ROBERT.