| | |
|---|---|
| | **Study of Prolog** |
| **Ex.No : 1** | |
| **Date:** | |

## PROLOG- PROGRAMMING IN LOGIC

PROLOG stands for Programming, In Logic — an idea that emerged in the early 1970's to use logical programming language. The early developers of this idea included Robert Kowaiski at Edinburgh(on the theoretical side),Marrtenvan Emdenat Edinburgh(experimental demonstration) and Alian Colmerauer at Marseilles(implementation). David D.H.Warren's efficient implementation at Edinburghin 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.

## SYMBOLICLANGUAGE

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 Artificia1 LanguageA1,) 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 PROIOG 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 inference 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(X2)
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 symbol ’?’ statements such as:

?- parent(X,sam)
X ann
?-grandfather(X,Y)
X= jo, Y= sam

**PROLOG IN DESGINING EXPERT SYSTEMS**

An expert system is a set of programs that manipulates encoded knowledge to solve problemsinaspecializeddomainthatnormallyrequireshumanexpertise.Anexpertsystem’sknowledge 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 computation resembling manipulations of natural language.
- Reason with meta-knowledge.

## METAPROGRAMMING

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 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.

**Result:** Thus the study of prolog programming is done successfully

| | |
|---|---|
| | **Write simple fact for following** |
| **Ex.No :2** | |
| **Date:** | |

**Aim:** To write a simple facts for the following sentences

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

**Program:**

**Clauses**

```
likes(ram ,mango).

girl(seema).

red(rose).

likes(bill ,cindy).

owns(john, gold).
```

**Output:** Goal queries

```
?-likes(ram,What).

What=mango

?-likes(Who,cindy).

Who= cindy

?-red(What).

What= rose

?-owns(Who,What).

Who=john
What=gold.
```

**Result** : Thus the prolog program for simple facts is executed successfully and the output is verified.

| | |
|---|---|
| | **Write predicates One converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.** |
| **Ex.No : 3a** | |
| **Date:** | |

**Aim:** To write a prolog program to converts Centigrade to Fahrenheit.

**Procedure:**

**Formula for Centigrade (C) temperatures to Fahrenheit (F) -**
F = C * 9 / 5 + 32

Here are two example predicates.

1. One converts centigrade temperatures to Fahrenheit,

2. the other checks if a temperature is below freezing.

Predicate/Rule:

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

freezing(F) :-
  F =< 32.
```

**Goal to find Fahrenheit temperature and freezing point**
**Output:**

Queries:

```
?-c_to_f(100,X).

X=212

?- freezing(15).

Yes

?- freezing(45).

No
```

**Result** : Thus the prolog program for conversion form centigrade to fahrenheit is executed successfully and the output is verified.

| | |
|---|---|
| **Ex.No : 3b** | **Write a Prolog program to handle arithmetic expression**<br>. |
| **Date:** | |

**Aim:** To write a prolog program to implement arithmetic expressions.

**Procedure:**

Prolog arithmetic expression examples & exercise.

| | |
|---|---|
| addition | + |
| subtraction | - |
| multiplication | * |
| division | / |
| power | ^ |
| mod | mod |

In prolog 'is' has a special functionality in evaluating arithmetic expressions. But with condition that the expression should be on the right side of 'is' otherwise it will give an error.

**Program:**

```
?- X is 3+2.     // expression on right side of 'is'

X = 5.

?- 3+2 is X.      // expression on left side of 'is'

 ERROR:  is/2:  Arguments are not sufficiently instantiated

 ?- X = 3+2.    // just instantiate variable X to value 3+2

X = 3+2.

?- 3+2 = X.

X = 3+2.

?- X is +(3,2).

 X = 5.

?- 5 is 3+2.

true.

?- 3+2 is 5.
```

```
 false.
?- X is 3*2.
 X = 6.
?- X is 3-2.
X = 1.
?- X is -(2,3).
X = -1.
?- X is 5-3-1.
X = 1.
?- X is -(5,3,1).
ERROR: is/2: Arithmetic: `(-)/3' is not a function
?- X is -(-(5,3),1).
 X = 1.
?- X is 5-3-1.
X = 1
?- X is 3/5.
X = 0.6.
 ?- X is 3 mod 5.
X = 3.
?- X is 5 mod 3.
 X = 2.
?- X is 5^3.
X = 125.
?- X is (5^3)^2.
X = 15625.
?- X = (5^3)^2.
 X = (5^3)^2.
```

```
?- 25 is 5^2.

 true.

?- Y is 3+2*4-1.

Y = 10.

?- Y is (3+2)*(4)-(1).

Y = 19.

?- Y is -(*(+(3,2),4),1).

 Y = 19.

?- X is 3*2, Y is X*2.

X = 6, Y = 12.
```

**Result:** Thus the prolog program for arithmetic expressions is executed successfully and the output is verified

| | |
|---|---|
| | **Write a Prolog program, to find the maximum and minimum of two numbers** |
| **Ex.No : 3c** | |
| **Date:** | . |

**Aim:** To find the minimum of two numbers and the maximum of two numbers using prolog

**Procedure:**

First, we will create two predicates, **find_max(X,Y,Max)**. This takes X and Y values, and stores the maximum value into the Max. Similarly **find_min(X,Y,Min)** takes X and Y values, and store the minimum value into the Min variable.

**Predicate:**

```
find_max(X, Y, X) :- X >= Y, !.

   find_max(X, Y, Y) :- X < Y.

find_min(X, Y, X) :- X =< Y, !.

   find_min(X, Y, Y) :- X > Y.
```

**Output:**

```
?- find_max(100,200,Max).

Max = 200

?- find_max(40,10,Max).

Max = 40

?- find_min(40,10,Min).

Min = 10

?- find_min(100,200,Min).

Min = 100

?-
```

**Result:** Thus the prolog program for finding maximum and minimum of two numbers is executed successfully and the output is verified.

| | |
|---|---|
| | Write a prolog program to solve the Monkey Banana problem.<br>. |
| **Ex.No : 4** | |
| **Date:** | |

**Aim:** To write a prolog program to solve the Monkey Banana problem

**Problem Statement:**

Suppose the problem is as given below −

1. A hungry monkey is in a room, and he is near the door.
2. The monkey is on the floor.
3. Bananas have been hung from the center of the ceiling of the room.
4. There is a block (or chair) present in the room near the window.
5. The monkey wants the banana, but cannot reach it.

The monkey can perform the following actions:-

1) Walk on the floor.

2) Climb the box.

3) Push the box around(if it is beside the box).

4) Grasp the banana if it is standing on the box directly under the banana.

**Program:**

```
on(floor, monkey).              %monkey is on the floor

on(floor, box).

in(room, monkey).               %monkey is in the room

in(room, box).

in(room, banana).

at(ceiling, banana).            %banana is at the ceiling

strong(monkey).                 %monkey is strong

grasp(monkey).                  %monkey can grasp

climb(monkey, box).             %monkey climbs the box

                                %operations does monkey performs
```

```
push(monkey, box):-           %monkey pushes the box only if the monkey is strong
strong(monkey).
under(banana, box):-          %box is under the banana only if the monkey pushes the box
push(monkey, box).

canreach(banana, monkey):-          %monkey canreach the banana only if
at(floor, banana).                  %banana is at the floor
at(ceiling, banana).                %banana is at the ceiling
under(banana, box).                 %box is under the banana
climb(monkey, box).                  %monkey climbs the box

canget(banana, monkey):-                       %monkey canget banana only if monkey
canreach(banana, monkey), grasp(monkey). %monkey canreach banana and grasp.
```

**Output:**

```
? - canget(banana, monkey).
true.
? - trace.
true.
[trace] ? - canget(banana, monkey).
call: (8) canget(banana, monkey) ? creep
call: (9) canreach(banana, monkey) ? creep
call: (10) at(floor, banana) ? creep
Fail: (10)
```

**Result:** Thus the prolog program to solve monkey banana problem is executed successfully and the output is verified.

| | **Write a prolog program to find factorial of a number** |
|---|---|
| **Ex.No : 5a** | |
| **Date:** | **.** |

**Aim**: To write a prolog program to find factorial of a number

**Algorithm:**

1. Prolog Factorial function definition is also similar to a normal factorial function.
2. Factorial(0,1) i.e., factorial of 0 is generally 1.
3. Factorial(N,M), if any temporary value N1 is assigned to N-1.
4. Factorial(N1,M1), and is factorial of N1 is M1.
5. M is NM1 i.e., assigning M to N*M1, then value of N is M.
6. The above happens to be the recursive relation between N and factorial M. It reviews rules for particular relation in the top to bottom order.

**Program: Factorial:**

```
factorial(0,1).
factorial(N,F) :-
N&gt;0,
N1 is N-1,
factorial(N1,F1),
F is N * F1.
```

**Output:**

```
?- factorial(0,1) :- true.
?- factorial(5,6) :- false.
```

**Result:** Thus the prolog program to find factorial of a number is executed successfully and the output is verified.

| | |
|---|---|
| | **Write a prolog program to find fibonacci series** |
| **Ex.No : 5b** | |
| **Date:** | **.** |

**Aim :** To write a prolog program to find fibonacci series.

**Procedure:**

The nth-Fibonacci number is the sum of the two preceeding ones, where the Fibonacci number of 0 and 1 are both 1.

**Program:**

```
Fibonacci:

fib(0, 0).

fib(X, Y) :- X &gt; 0, fib(X, Y, _).

fib(1, 1, 0).

fib(X, Y1, Y2) :-

X&gt;1,

X1 is X - 1,

fib(X1, Y2, Y3),

Y1 is Y2 + Y3.
```

**Output:**

```
? - fib(1,2).

false.

? - fib(1,1).

true.
```

**Result:** Thus the prolog program to find fibonacci series is executed successfully and the output is verified.

|  |  |
|---|---|
| Ex.No : 5c | **Write a prolog program for Checking prime number.** |
| Date: |  |

**Aim:** To write a prolog program for Checking prime number.

**Procedure:**

First, we define the predicate % divisible(X+,Y+) is true if X is divisible by Y divisible(0,Y). divisible(X,Y):- X > 0, X1 is X-Y, divisible(X1,Y)

**Program:**

```
divisible(X,Y):-
N is Y*Y,
N =&lt; X,
X mod Y =:= 0.
divisible(X,Y):-


Y &lt; X,
Y1 is Y+1,
divisible(X,Y1).
isprime(X):-
Y is 2, X &gt; 1, \+divisible(X,Y).
```

**Output:**

```
?- Input:  isprime(7).
Yes
```

**Result:** Thus the prolog program to check a prime number or not is executed successfully and the output is verified

|  |  |
|---|---|
|  | **Finding power of a number** |
| **Ex.No : 5d** |  |
| **Date:** |  |

**Aim:** To write a prolog program for Finding power of a number.

**Algorithm:**

1. Implement an interactive Prolog program that compute the Yth power of X using the predicate pow(X,Y,Z).

2. Use the built-in unary predicate integer to check whether the inputs are integers.

3. The program should stop when the user inputs exit or quit.

**Program:**

```
power(0,N,0):- N&gt;0.

power(X,0,1):- X&gt;0.

power(X,N,V):-X&gt;0,N&gt;0,N1 is N-1,power(X,N1,V1), V is V1*X.
```

**Output:**

```
?- power(2,4,P).

V=16
```

**Result:** Thus the prolog program to find power of a number is executed successfully and the output is verified.

**Ex.No : 6**

**Date:**

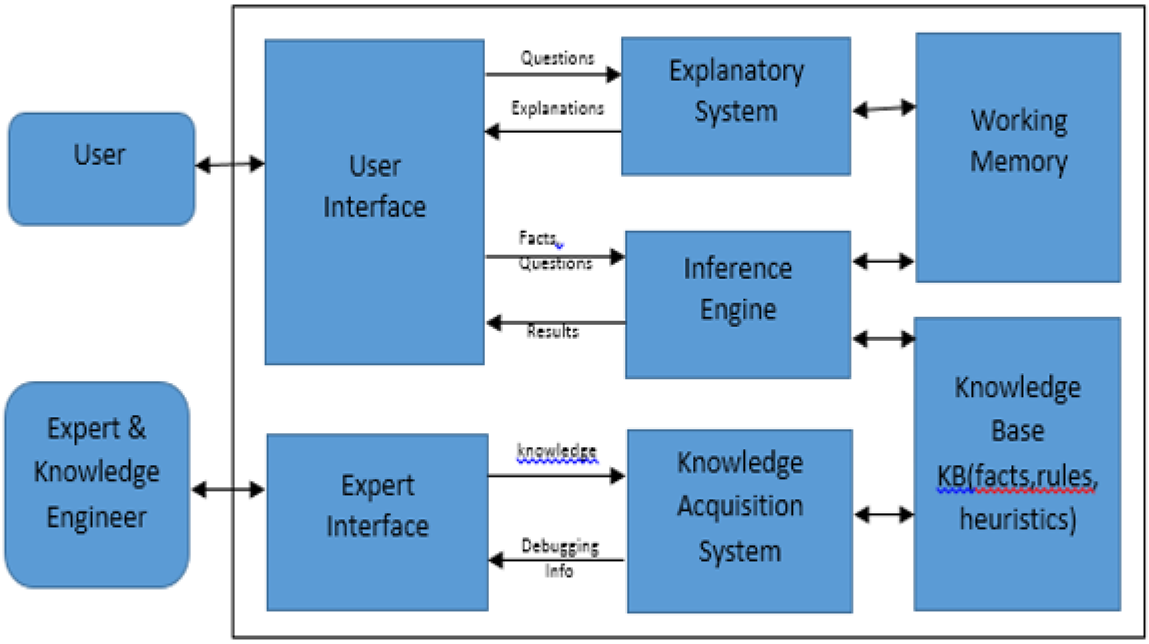
Implementation of Medical Diagnosis in PROLOG

**Aim**: To write a prolog program of Medical Diagnosis.

**Procedure:**

Medical Diagnosis Chosen Statement: The medical expert system will tell the patient the type of disease he/she has on the basis of the symptoms given by the patient.

**Architecture of Expert System**



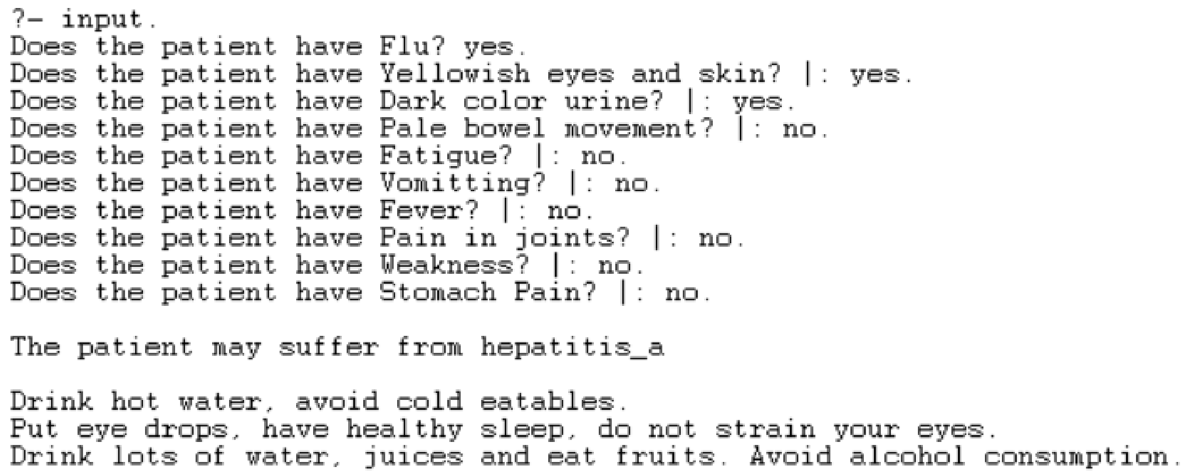
**The Architecture of an Expert System (ES) consists of the following major components:**

**Program:**

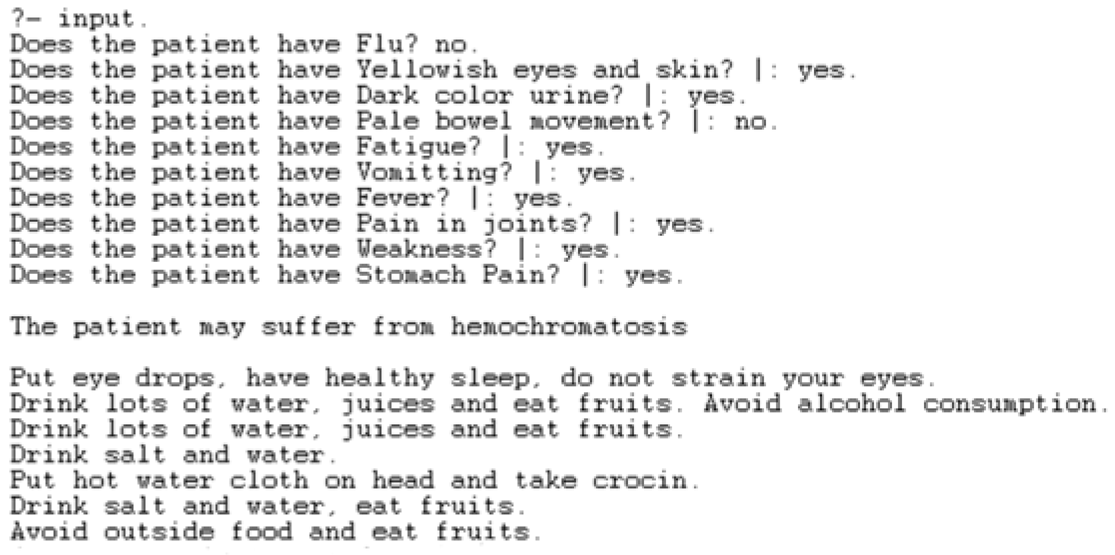
```
symptom('Flu').
symptom('Yellowish eyes and skin').
symptom('Dark color urine').
symptom('Pale bowel movement').
symptom('Fatigue').
symptom('Vomitting').
symptom('Fever').
symptom('Pain in joints').
symptom('Weakness').
symptom('Stomach Pain').

treatment('Flu', 'Drink hot water, avoid cold eatables.').
treatment('Yellowish eyes and skin', 'Put eye drops, have healthy sleep, do not strain your
eyes.').
treatment('Dark color urine', 'Drink lots of water, juices and eat fruits. Avoid alcohol
consumption.').
treatment('Pale bowel movement', 'Drink lots of water and exercise regularly.').
treatment('Fatigue', 'Drink lots of water, juices and eat fruits.').
treatment('Vomitting', 'Drink salt and water.').
treatment('Fever', 'Put hot water cloth on head and take crocin.').
treatment('Pain in Joints', 'Apply pain killer and take crocin.').
treatment('Weakness', 'Drink salt and water, eat fruits.').
treatment('Stomach Pain', 'Avoid outside food and eat fruits.').

input :- dynamic(patient/2),
   repeat,
   symptom(X),
   write('Does the patient have '),
   write(X),
   write('? '),
   read(Y),
   assert(patient(X,Y)),
   \+ not(X='Stomach Pain'),
   not(output).

disease(hemochromatosis) :-
   patient('Stomach Pain',yes),
   patient('Pain in joints',yes),
   patient('Weakness',yes),
   patient('Dark color urine',yes),
   patient('Yellowish eyes and skin',yes).

disease(hepatitis_c) :-
   not(disease(hemochromatosis)),
   patient('Pain in joints',yes),
   patient('Fever',yes),
   patient('Fatigue',yes),
   patient('Vomitting',yes),
   patient('Pale bowel movement',yes).

disease(hepatitis_b) :-
   not(disease(hemochromatosis)),
   not(disease(hepatitis_c)),
   patient('Pale bowel movement',yes),
   patient('Dark color urine',yes),
   patient('Yellowish eyes and skin',yes).

disease(hepatitis_a) :-
```

```
    not(disease(hemochromatosis)),
    not(disease(hepatitis_c)),
    not(disease(hepatitis_b)),
    patient('Flu',yes),
    patient('Yellowish eyes and skin',yes).

disease(jaundice) :-
    not(disease(hemochromatosis)),
    not(disease(hepatitis_c)),
    not(disease(hepatitis_b)),
    not(disease(hepatitis_a)),
    patient('Yellowish eyes and skin',yes).

disease(flu) :-
    not(disease(hemochromatosis)),
    not(disease(hepatitis_c)),
    not(disease(hepatitis_b)),
    not(disease(hepatitis_a)),
    patient('Flu',yes).

output:-
    nl,
    possible_diseases,
    nl,
    advice.

possible_diseases :- disease(X), write('The patient may suffer from '), write(X),nl.
advice :- symptom(X), patient(X,yes), treatment(X,Y), write(Y),nl, \+ not(X='Stomach Pain').
```

**Output –**



```
?- input.
Does the patient have Flu? yes.
Does the patient have Yellowish eyes and skin? |: yes.
Does the patient have Dark color urine? |: yes.
Does the patient have Pale bowel movement? |: no.
Does the patient have Fatigue? |: no.
Does the patient have Vomitting? |: no.
Does the patient have Fever? |: no.
Does the patient have Pain in joints? |: no.
Does the patient have Weakness? |: no.
Does the patient have Stomach Pain? |: no.

The patient may suffer from hepatitis_a

Drink hot water, avoid cold eatables.
Put eye drops, have healthy sleep, do not strain your eyes.
Drink lots of water, juices and eat fruits. Avoid alcohol consumption.
```



```
?- input.
Does the patient have Flu? no.
Does the patient have Yellowish eyes and skin? |: yes.
Does the patient have Dark color urine? |: yes.
Does the patient have Pale bowel movement? |: no.
Does the patient have Fatigue? |: yes.
Does the patient have Vomitting? |: yes.
Does the patient have Fever? |: yes.
Does the patient have Pain in joints? |: yes.
Does the patient have Weakness? |: yes.
Does the patient have Stomach Pain? |: yes.

The patient may suffer from hemochromatosis

Put eye drops, have healthy sleep, do not strain your eyes.
Drink lots of water, juices and eat fruits. Avoid alcohol consumption.
Drink lots of water, juices and eat fruits.
Drink salt and water.
Put hot water cloth on head and take crocin.
Drink salt and water, eat fruits.
Avoid outside food and eat fruits.
```

**Result:** Thus the prolog program to medical diagnosis is executed successfully and the output is verified.

|  |  |
|---|---|
|  | **Prolog program to solve the 4-3 Gallon Water Jug Problem in Artificial Intelligence** |
| **Ex.No : 7** |  |
| **Date:** |  |

**Aim:** To write a Prolog program to solve the 4-3 Gallon Water Jug Problem in Artificial Intelligence

**Program:**

```
database
visited_state(integer,integer)
predicates
state(integer,integer)
clauses
state(2,0).
state(X,Y):- X &lt; 4,
not(visited_state(4,Y)),
assert(visited_state(X,Y)),
write(&quot;Fill the 4-Gallon Jug: (&quot;,X,&quot;,&quot;,Y,&quot;) --&gt; (&quot;, 4,&quot;,&quot;,Y,&quot;)\n&quot;),
state(4,Y).
state(X,Y):- Y &lt; 3,
not(visited_state(X,3)),
assert(visited_state(X,Y)),
write(&quot;Fill the 3-Gallon Jug: (&quot;, X,&quot;,&quot;,Y,&quot;) --&gt; (&quot;, X,&quot;,&quot;,3,&quot;)\n&quot;),
state(X,3).
state(X,Y):- X &gt; 0,
not(visited_state(0,Y)),
assert(visited_state(X,Y)),
write(&quot;Empty the 4-Gallon jug on ground: (&quot;, X,&quot;,&quot;,Y,&quot;) --&gt;
(&quot;, 0,&quot;,&quot;,Y,&quot;)\n&quot;),
state(0,Y).
state(X,Y):- Y &gt; 0,
not(visited_state(X,0)),
assert(visited_state(X,0)),
write(&quot;Empty the 3-Gallon jug on ground: (&quot;, X,&quot;,&quot;,Y,&quot;) --&gt;
(&quot;, X,&quot;,&quot;,0,&quot;)\n&quot;),
state(X,0).
state(X,Y):- X + Y &gt;= 4,
Y &gt; 0,
NEW_Y = Y - (4 - X),
not(visited_state(4,NEW_Y)),
assert(visited_state(X,Y)),
write(&quot;Pour water from 3-Gallon jug to 4-gallon until it is full: (&quot;,
X,&quot;,&quot;,Y,&quot;) --&gt; (&quot;,
4,&quot;,&quot;,NEW_Y,&quot;)\n&quot;),
state(4,NEW_Y).
state(X,Y):- X + Y &gt;=3,
X &gt; 0,
NEW_X = X - (3 - Y),


not(visited_state(X,3)),
assert(visited_state(X,Y)),
write(&quot;Pour water from 4-Gallon jug to 3-gallon until it is full: (&quot;,
X,&quot;,&quot;,Y,&quot;) --&gt; (&quot;,
NEW_X,&quot;,&quot;,3,&quot;)\n&quot;),

state(NEW_X,3).

state(X,Y):- X + Y &lt;=4,

Y &gt; 0,

NEW_X = X + Y,

not(visited_state(NEW_X,0)),

assert(visited_state(X,Y)),

write(&quot;Pour all the water from 3-Gallon jug to 4-gallon: (&quot;,
X,&quot;,&quot;,Y,&quot;) --&gt; (&quot;,

NEW_X,&quot;,&quot;,0,&quot;)\n&quot;),

state(NEW_X,0).

state(X,Y):- X+Y&lt;=3,

X &gt; 0,

NEW_Y = X + Y,

not(visited_state(0,NEW_Y)),

assert(visited_state(X,Y)),

write(&quot;Pour all the water from 4-Gallon jug to 3-gallon: (&quot;,
X,&quot;,&quot;,Y,&quot;) --&gt; (&quot;,

0,&quot;,&quot;,NEW_Y,&quot;)\n&quot;),

state(0,NEW_Y).

state(0,2):- not(visited_state(2,0)),

assert(visited_state(0,2)),

write(&quot;Pour 2 gallons from 3-Gallon jug to 4-gallon: (&quot;,
0,&quot;,&quot;,2,&quot;) --&gt; (&quot;,

2,&quot;,&quot;,0,&quot;)\n&quot;),

state(2,0).

state(2,Y):- not(visited_state(0,Y)),

assert(visited_state(2,Y)),
write(&quot;Empty 2 gallons from 4-Gallon jug on the ground: (&quot;,
2,&quot;,&quot;,Y,&quot;) --&gt; (&quot;,

0,&quot;,&quot;,Y,&quot;)\n&quot;),

state(0,Y).

goal

makewindow(1,2,3,&quot;4-3 Water Jug Problem&quot;,0,0,25,80),

state(0,0).
```

**Output:**

```
+----------------------------4-3 Water Jug Problem-------------------------+

¦Fill the 4-Gallon Jug: (0,0) --&gt; (4,0) ¦

¦Fill the 3-Gallon Jug: (4,0) --&gt; (4,3) ¦

¦Empty the 4-Gallon jug on ground: (4,3) --&gt; (0,3) ¦

¦Pour all the water from 3-Gallon jug to 4-gallon: (0,3) --&gt; (3,0) ¦


¦Fill the 3-Gallon Jug: (3,0) --&gt; (3,3) ¦

¦Pour water from 3-Gallon jug to 4-gallon until it is full: (3,3) --&gt; (4,2) ¦

¦Empty the 4-Gallon jug on ground: (4,2) --&gt; (0,2) ¦

¦Pour all the water from 3-Gallon jug to 4-gallon: (0,2) --&gt; (2,0) ¦

¦ ¦Press the SPACE bar ¦
```

**Result:** Thus the prolog program to solve the 4 - 3 gallon water jug problem is executed successfully and the output is verified.

| | |
|---|---|
| | **Write a Prolog Program to solve n queen problem** |
| **Ex.No : 8** | |
| **Date:** | |

**Aim:** To write a Prolog Program to solve n queen problem.

**Procedure:**

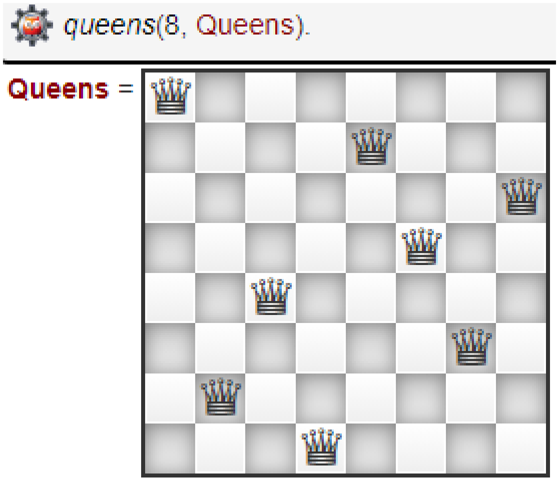
In the 4 Queens problem the object is to place n queens on a chessboard in such a way that no queens can capture a piece. This means that no two queens may be placed on the same row, column, or diagonal.

**Program:**

```
:- use_rendering(chess).
        queens(N, Queens) :-
           length(Queens, N),
        board(Queens, Board, 0, N, _, _),
        queens(Board, 0, Queens).


        board([], [], N, N, _, _).
        board([_|Queens], [Col-Vars|Board], Col0, N, [_|VR], VC) :-
        Col is Col0+1,
        functor(Vars, f, N),
        constraints(N, Vars, VR, VC),
        board(Queens, Board, Col, N, VR, [_|VC]).


        constraints(0, _, _, _) :- !.
        constraints(N, Row, [R|Rs], [C|Cs]) :-
        arg(N, Row, R-C),
        M is N-1,
        constraints(M, Row, Rs, Cs).


        queens([], _, []).

        queens([C|Cs], Row0, [Col|Solution]) :-

        Row is Row0+1,

        select(Col-Vars, [C|Cs], Board),

        arg(Row, Vars, Row-Row),

        queens(Board, Row, Solution)
```

**__Output:__**

?- queens(8, Queens).



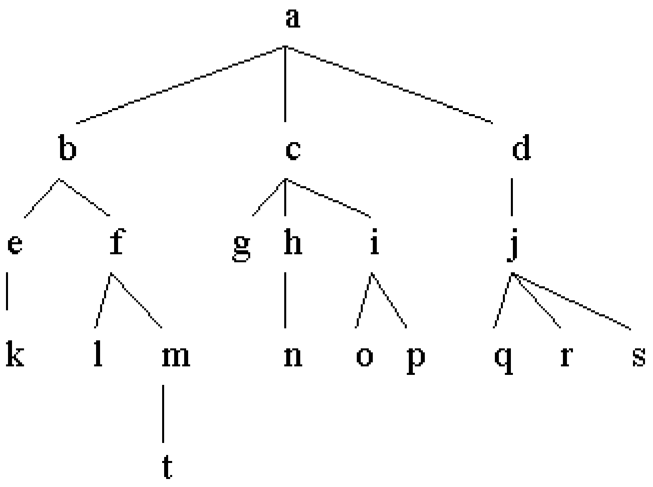
```
queens(8, Queens).

Queens =
```

**Result:** Thus the n queen problem using prolog program is implemented successfully and the output is verified.

| | |
|---|---|
| Ex.No : 9 | **Write a Prolog program to implement BFS and DFS** |
| Date: | |

**Aim:**

To implement BFS and DFS using prolog program.



**Program:**

```
:- 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.


getchildren(Parent, Children) :-

   setof(Child, Parent^is_parent(Parent, Child), Children), !.

getchildren(_, []).
```

```
depthfirst([]) :- !.

depthfirst([Node|Frontier]) :-

    format('~p ', [Node]),

    getchildren(Node, Children),

    append(Children, Frontier, NewFrontier),

    depthfirst(NewFrontier).


breadthfirst([]) :- !.

breadthfirst([Node|Frontier]) :-

    format('~p ', [Node]),

    getchildren(Node, Children),

    append(Frontier, Children, NewFrontier),

    breadthfirst(NewFrontier).
```

**Output:**

```
?- depthfirst([a]).

a b e k f l m t c g h n i o p d j q r s

true


?-  breadthfirst([a]).

a b c d e f g h i j k l m n o p q r s t

true
```

**Result:** Thus the prolog program for breadth first search and depth first search is implemented successfully and the output is verified.

|  |  |
|---|---|
|  | **Write a program to solve Missionaries and Cannibal problem** |
| **Ex.No : 10** |  |
| **Date:** |  |

**Aim**: To write a program to solve Missionaries and Cannibal problem.

**Procedure**:

"Missionaries and cannibals problem” is well known problem in Computer Science.

The idea is the following: You have equal number of cannibals and missioners on one side of the river. You have to move them all to other side of river, by using boat. Boat can take 1 cannibal and 1 missioner or 1 cannibal and 0 missioners or 1 missioner and 0 cannibals.

This problem falls into AI (Artificial Intelligence) area which solved relatively easily with Prolog. The pattern is the following .

1. Define the “state”

2. Define now to move from state A to state B

3. Define the “Goal” state

4. Define the search , that checks all possible legal “moves” that bring from initial state to the goal.

**Program**:

```
%       ?- ['missionaries_and_cannibals.pl'].

%       ?- find.

% Represent a state as [CL,ML,B,CR,MR]

start([3,3,left,0,0]).

goal([0,0,right,3,3]).

legal(CL,ML,CR,MR) :-

        % is this state a legal one?

        ML>=0, CL>=0, MR>=0, CR>=0,
        (ML>=CL ; ML=0),
        (MR>=CR ; MR=0).
% Possible moves:
move([CL,ML,left,CR,MR],[CL,ML2,right,CR,MR2]):-
        % Two missionaries cross left to right.
        MR2 is MR+2,
        ML2 is ML-2,
        legal(CL,ML2,CR,MR2).
move([CL,ML,left,CR,MR],[CL2,ML,right,CR2,MR]):-
        % Two cannibals cross left to right.
        CR2 is CR+2,
        CL2 is CL-2,
        legal(CL2,ML,CR2,MR).
move([CL,ML,left,CR,MR],[CL2,ML2,right,CR2,MR2]):-
        %  One missionary and one cannibal cross left to right.
        CR2 is CR+1,
        CL2 is CL-1,
        MR2 is MR+1,
        ML2 is ML-1,
        legal(CL2,ML2,CR2,MR2).
move([CL,ML,left,CR,MR],[CL,ML2,right,CR,MR2]):-
        % One missionary crosses left to right.
        MR2 is MR+1,
        ML2 is ML-1,
        legal(CL,ML2,CR,MR2).
move([CL,ML,left,CR,MR],[CL2,ML,right,CR2,MR]):-
        % One cannibal crosses left to right.
        CR2 is CR+1,
        CL2 is CL-1,
        legal(CL2,ML,CR2,MR).
move([CL,ML,right,CR,MR],[CL,ML2,left,CR,MR2]):-
        % Two missionaries cross right to left.
        MR2 is MR-2,
        ML2 is ML+2,
        legal(CL,ML2,CR,MR2).
move([CL,ML,right,CR,MR],[CL2,ML,left,CR2,MR]):-
        % Two cannibals cross right to left.
        CR2 is CR-2,
        CL2 is CL+2,
        legal(CL2,ML,CR2,MR).
move([CL,ML,right,CR,MR],[CL2,ML2,left,CR2,MR2]):-
        %  One missionary and one cannibal cross right to left.
        CR2 is CR-1,
        CL2 is CL+1,
        MR2 is MR-1,
        ML2 is ML+1,
        legal(CL2,ML2,CR2,MR2).
move([CL,ML,right,CR,MR],[CL,ML2,left,CR,MR2]):-
        % One missionary crosses right to left.
        MR2 is MR-1,
        ML2 is ML+1,
        legal(CL,ML2,CR,MR2).
move([CL,ML,right,CR,MR],[CL2,ML,left,CR2,MR]):-
        % One cannibal crosses right to left.
        CR2 is CR-1,
        CL2 is CL+1,
        legal(CL2,ML,CR2,MR).

% Recursive call to solve the problem

path([CL1,ML1,B1,CR1,MR1],[CL2,ML2,B2,CR2,MR2],Explored,MovesList) :-

  move([CL1,ML1,B1,CR1,MR1],[CL3,ML3,B3,CR3,MR3]),

  not(member([CL3,ML3,B3,CR3,MR3],Explored)),


path([CL3,ML3,B3,CR3,MR3],[CL2,ML2,B2,CR2,MR2],[[CL3,ML3,B3,CR3,MR3]|Explore
d],[ [[CL3,ML3,B3,CR3,MR3],[CL1,ML1,B1,CR1,MR1]] | MovesList ]).

% Solution found

path([CL,ML,B,CR,MR],[CL,ML,B,CR,MR],_,MovesList):-

        output(MovesList).

% Printing

Output([]) :- nl.

Output([[A,B]|MovesList]) :-

        output(MovesList),

        write(B), write(' -> '), write(A), nl.

% Find the solution for the missionaries and cannibals problem

find :-

path([3,3,left,0,0],[0,0,right,3,3],[[3,3,left,0,0]],_).
```

**Result:** Thus the prolog program to solve Missionaries and Cannibal problem is implemented successfully and the output is verified.

| | |
|---|---|
| | **Case study – expert system** |
| **Ex.No : 11** | |
| **Date:** | |

## INTRODUCTION

An Artificial Intelligence System created to solve problems in a particular domain is called an Expert System. An expert system is a knowledge intensive program to solve the problem in a domain that require considerable amount of technical expertise. An expert system is computer program that simulates the judgment and behavior of a human or an organization that has expert knowledge and experience in a particular field. Expert systems are an emerging technology ,past applications range from MYCIN, used in the medical field to diagnose of bacterial infection and effectively handles uncertain data, XCON/RI, used to configure computer systems. Dendral, does the inferring process of structure elucidation of chemical compounds. Expert systems are typically very domain specific. The developer of such a system must limit his or her scope of the system to just what is needed to solve the target problem. Special tools or programming languages are often needed to accomplish the specific objectives of the system.

## I. EXPERT SYSTEM ARCHITECTURE

Expert system derives complex decisions from the combination of factual and heuristic knowledge. In order for the computer to be able to retrieve and effectively use heuristic knowledge, the knowledge must be organized in an easily accessible format that distinguishes among data, knowledge, and control structures. The process of building expert systems is often called knowledge engineering. The knowledge engineer is involved with all components of an expert system:

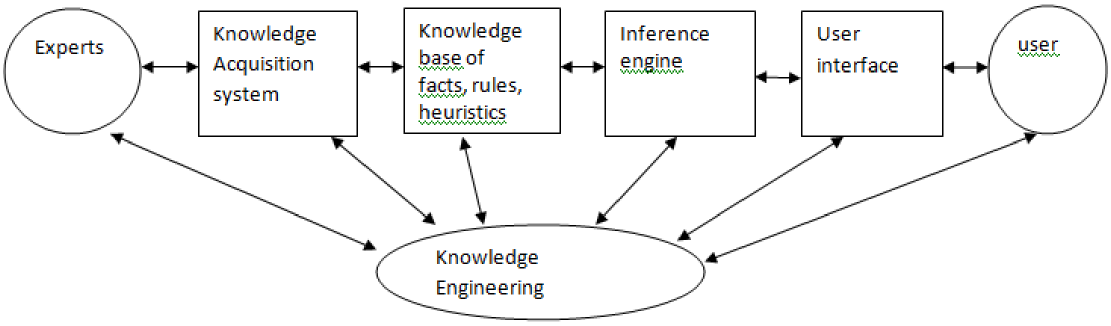

Experts
Knowledge Acquisition system
Knowledge base of facts, rules, heuristics
Inference engine
User interface
user
Knowledge Engineering

Figure 1

Building expert systems is generally an iterative process. The components and their interaction will be refined over the course of numerous meetings of the knowledge engineer with the experts and users. So the fundamental modules of expert system are:

1. **Knowledge base** a core module of expert system. It is a warehouse of domain specific knowledge, consists of problem-solving rules, procedures, and intrinsic data relevant to the problem domain. A knowledge base is created by knowledge engineers, who translate the knowledge of real human experts into rules and strategies. These rules and strategies can change depending on the prevailing problem scenario. The knowledge base constitutes the problem-solving rules, facts, or intuition that a human expert might use in solving problems in a given problem domain. The knowledge base is usually stored in terms of if–then rules(production rules used in MYCIN,DENDRAL,XON).There are many others ways of representing the knowledge ie logic, semantic nets, frames ,conceptual dependency, production rules and scripts.

2. **WORKING MEMORY** refers to task-specific data for the problem under consideration.

3. **INFERENCE ENGINE** is a generic control mechanism that act as “rule interpreter” applies the axiomatic knowledge in the knowledge base to the task-specific data to arrive at some solution or conclusion.

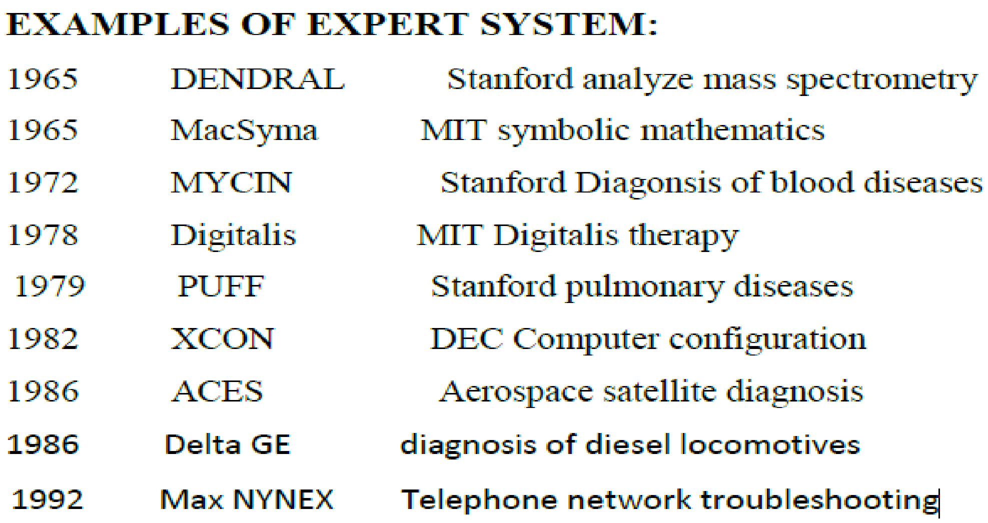
Recall the steps in the basic Recognize Act Cycle:

1. Match the premise patterns of the rules against elements in the working memory. Generally the rules will be domain knowledge built into the system, and the working memory will contain the case based facts entered into the system, plus any new facts that have been derived from them.

2. If there is more than one rule that can be applied, use a conflict resolution strategy to choose one to apply. Stop if no further rules are applicable.

3. Activate the chosen rule, which generally means adding/deleting an item to/from working memory. Stop if a terminating condition is reached, or return to step 1.Early production systems spent over 90% of their time doing pattern matching, but there is now a solution to this efficiency problem .

**EXAMPLES OF EXPERT SYSTEM:**



**EXAMPLES OF EXPERT SYSTEM:**

| | | |
|---|---|---|
| 1965 | DENDRAL | Stanford analyze mass spectrometry |
| 1965 | MacSyma | MIT symbolic mathematics |
| 1972 | MYCIN | Stanford Diagonsis of blood diseases |
| 1978 | Digitalis | MIT Digitalis therapy |
| 1979 | PUFF | Stanford pulmonary diseases |
| 1982 | XCON | DEC Computer configuration |
| 1986 | ACES | Aerospace satellite diagnosis |
| 1986 | Delta GE | diagnosis of diesel locomotives |
| 1992 | Max NYNEX | Telephone network troubleshooting |

**Figure 2**

### 2.1 Uses of Expert Systems

1. Experts systems have vast quantity of domain specific knowledge.

2. Expert system plays three major role:Role of a problem solver,tutor and archive.

3. Experts systems are always available and can be used anywhere, any time.

4. Cost effective

5. Expert System has increased accessibility than human experts.

6. Expert system are highly advantageous in interdisciplinary domains.

7. Expert System are easy to develop and modify

8. Human experts are not 100% reliable or consistent as moods may lead to a default assumptions or irrelevant.

**2.2 Limitation of present Expert Systems**

1. Limited domain specific knowledge.

2. Lack of knowledge representation mechanism.

3. User has to describe the problem in formal language only.

4. Systems are not always up to date.

5. Lack of understanding as “no common-sense”.

6. Experts needed to setup and maintain system.

7. Possibility of error as they don’t learn reason and cannot refine its own knowledge base as it needs knowledge engineering.

8. Development cost may be too high as lot of recourses is needed today.

**2.3 Expert System Applications**

**1. PUFF:** Expert System for the interpretation of pulmonary function tests for patients with lung disease.PUFF was probably the first AI system to have been used in clinical practice.It was developed by Stanford University and Pacific Presbyterian Medical Center (Janice Aikins, John Kunz, Ted Shortliffe, Robert Fallat) PUFF can diagnose the presence and severity of lung disease and produce reports for the patient's file

PUFF was the first system developed using EMYCIN (Essential MYCIN, van Melle, 1979). It included the domain-independent features of MYCIN:

□ rule interpreter

□ explanation

□ knowledge acquisition.

This provided a mechanism for representing domain-specific knowledge in the form of production rules, and for performing consultations in that domain. PUFF was originally written in Interlisp using the SUMEX-AIM computer facility and had to be rewritten in

BASIC before it could be installed at PMC. PUFF does not require direct interaction with a physician, avoiding the human engineering problem. Knowledge Representation: there are 75 clinical paramenters (representing pulmonary function test results such as "total lung capacity" and "residual volume"). Control Structure: PUFF is primarily a goal-directed, backward chaining system employing some 400 production rules. If it cannot conclude the value for a parameter, it asks the user. A pulmonary physiologist reviews the PUFF report, and if necessary modifies it on-line before printing it out. The report was found not to require modification in 85% of cases. Modifications, where made, often consisted of comments such as "This is consistant with last visit". The PUFF basic knowledge base was incorporated into the commercial "Pulmonary Consult" product. Several hundred copies were sold in the 1980s and used around the world.

**2.SHYSTER** is a legal expert system derieved from the “Shyler” a slang word for Who acts in a disreputable, unethical, or unscrupulous way, especially in the practice of law and politics. SHYSTER is a specific example of a general category of legal expert systems, broadly defined as systems that make use of artificial intelligence (AI) techniques to solve legal problems.It was developed at the Australian National University in Canberra in 1993. It was written as the doctoral dissertation of James Popple under the supervision of Robin Stanton . Although SHYSTER attempts to model the way in which lawyers argue with cases, it does not attempt to model the way in which lawyers decide which cases to use in those arguments. SHYSTER is of a general design, permitting its operation in different legal domains. It was designed to provide advice in areas of case law that have been specified by a legal expert using a bespoke specification language. Its knowledge of the law is acquired, and represented, as information about cases. It produces its advice by examining, and arguing about, the similarities and differences between cases. SHYSTER was tested in four different and disparate areas of case law.

**3. PROSPECTOR** This expert system help in evaluation of the mineral potential of a geological site or region. PROSPECTOR: consultation system to assist geologists working in mineral exploration. It was developed by Hart and Duda of SRI International attempts to represent the knowledge and reasoning processes of experts in the geological domain. It is intended user is an exploration geologist in the early stages of investigating a possible drilling site

4. **MYCIN**: MYCIN was developed at Stanford University in the mid-1970s. It was designed to aid physicians in the diagnosis and treatment of meningitis and bacteraemia infections. MYCIN was strictly a research system. Medical system for diagnosing blood disorders.

5.**DESIGN ADVISOR**: The Expert Design Advisor (EDA) is a decision-aided toolset for use by systems engineers in facilitating the development of large, complex systems involving Mission Critical Computing Resources (MCCR).Gives advice to designers of processor chips.

**6.Dendral:** It was an influential pioneer project in artificial intelligence (AI) of the 1960s, and the computer software expert system that it produced. Its primary aim was to study hypothesis formation and discovery in science. For that, a specific task in science was chosen: help organic chemists in identifying unknown organic molecules, by analyzing their mass spectra and using knowledge of chemistry. It was done at Stanford University by Edward Feigenbaum, Bruce G. Buchanan, Joshua Lederberg, and Carl Djerassi, along with a team of highly creative research associates and students. It began in 1965 and spans approximately half the history of AI research .The software program Dendral is considered the first expert system because it automated the decision-making process and problem-solving behavior of organic chemists.

## III. EXPERT SYSTEM USED IN INDIA

1. **Indian Institute of Horticultural Research Institute, Bangalore:** The first software for use by the grape cultivators was prepared by the Indian Institute of Horticultural Research Institute, Bangalore. This spontaneous response made them to undertake similar software for providing guidance to mushroom cultivators, which became extremely popular and a large number of growers using it regularly for getting solutions to their problems. The Institute has launched into an effort to give a comprehensive package of practices of about 148 horticulture crops for cultivation in the 4 Southern states of Kerala, Tamilnadu, Karnataka and Andhra Pradesh.

2.**AGREX :** Center for Informatics Research and Advancement, Kerala has prepared an Expert System called AGREX to help the Agricultural field personnel give timely and correct advice to the farmers. These Expert Systems find extensive use in the areas of fertilizer application, crop protection, irrigation scheduling, diagnosis of diseases in paddy and post harvest technology of fruits and vegetables.

3.**Farm Advisory System:** Punjab Agricultural University, Ludhiana, has developed the Farm Advisory System to support agri-business management. The conversation between the system and the user is arranged in such a way that the system asks all the questions from user one by one which it needs to give recommendations on the topic of farm Management. The inputs are encouraging and acceptance by farmers is very good.

4. **TDP Technologies Pvt. Ltd**. In Chennai is using MYCIN technique for diagnosing blood disorders..

5. **Tata Memorial Hospital** in Mumbai is using PUFF for diagnosis of respiratory conditions.

**Result**: Thus the case study for Expert Systems is done successfully.

# CONTENT BEYOND SYLLABUS

# IMPLEMENTATION OF TRAVELLING SALESMAN PROBLEM

**Aim:** To implement travelling salesman problem using prolog program.

**Procedure:**

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) */
```

**Program:**

```
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 */
```

**Output:**

```
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).
```

**Result:** Thus the travelling salesman problem using prolog program is implemented successfully and the output is verified.