**Practical List**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Date** | **Title** | **Page No** | **Sign** |
| 1 |  | Given three relations *child(X, Y)* : X is a child of Y, *male(X)* : X is male and *female(X)* : X is female. Write a program to find relations like brother(X, Y), father(X, Y), mother(X, Y), grandfather(X, Y), ancestor(X, Y), sister(X, Y), grandmother(X, Y) between different members of a family. |  |  |
| 2 |  | Given a relation player(“name”, age). Store knowledge about six players of age between 12-15. Write a prolog program to find possible combination of players to organize a two player tournament. |  |  |
| 3 |  | Write a prolog program to check whether a person X can eat Food Y or not. A person X can eat food Y if X likes Y, X is available and X is eatable. |  |  |
| 4 |  | Write a program to represent knowledge of 10 employees of a company, *employee(EmpId,“name”, B\_Date, “Address”)*. Where name of employee and B\_Date are compound objects. Display list of Employees whose birthday is in current month.(Use Compound Obj and fail). |  |  |
| 5 |  | Write a program to perform different operations like add, mul, sub, div etc. (Implementation of switch statement using cut). |  |  |
| 6 |  | Write a program to implement BFS (for 8 puzzle problem or Water Jug problem or any AI search problem) . |  |  |
| 7 |  | Write a program to implement DFS (for 8 puzzle problem or Water Jug problem or any AI search problem) |  |  |
| 8 |  | Write a program to implement Tic-Tac-Toe game problem. |  |  |
| 9 |  | Write a program to implement Single Player Game (Using Heuristic Function) |  |  |
| 10 |  | Write a program to Implement A\* Algorithm. |  |  |
| 11 |  | Write a program to solve N-Queens problem using Prolog. |  |  |
| 12 |  | Write a program to solve 8 puzzle problem using Prolog. |  |  |
| 13 |  | Write a program to solve travelling salesman problem using Prolog. |  |  |
| 14 |  | Convert following Prolog predicates into Semantic Net  cat(tom).  cat(cat1).  mat(mat1).  sat\_on(cat1,mat1).  bird(bird1).  caught(tom,bird1).  like(X,cream) :– cat(X).  mammal(X) :– cat(X).  has(X,fur) :– mammal(X).  animal(X) :– mammal(X).  animal(X) :– bird(X).  owns(john,tom).  is\_coloured(tom,ginger).  10. Write the Conceptual Dependency for following statements.  *(a)* John gives Mary a book  *(b)* John gave Mary the book yesterday |  |  |

**Practical: 1**

**Aim : Given three relations *child(X, Y)* : X is a child of Y, *male(X)* : X is male and *female(X)* : X is female. Write a program to find relations like brother(X, Y), father(X, Y), mother(X, Y), grandfather(X, Y), ancestor(X, Y), sister(X, Y), grandmother(X, Y) between different members of a family.**

predicates

father(symbol,symbol)

mother(symbol,symbol)

son(symbol,symbol)

daughter(symbol,symbol)

brother(symbol,symbol)

sister(symbol,symbol)

cousin(symbol,symbol)

uncle(symbol,symbol)

aunty(symbol,symbol)

grandfa(symbol,symbol)

grandma(symbol,symbol)

couple(symbol,symbol)

male(symbol)

female(symbol)

clauses

male(vijay).

male(ajay).

male(avan).

male(sujay).

male(viren).

male(nilay).

male(jimi).

male(ram).

male(kush).

male(jay).

female(sujata).

female(gita).

female(nita).

female(tina).

female(sita).

female(susmita).

couple(sujata,sujay).

couple(susmita,jay).

couple(sita,ram).

son(vijay,jay).

son(vijay,susmita).

son(sujay,vijay).

son(ajay,vijay).

son(nilay,vijay).

son(jimi,nilay).

son(avan,ajay).

son(viren,sujata).

son(viren,sujay).

son(kush,ram).

son(kush,sita).

daughter(nita,sujata).

daughter(nita,sujay).

daughter(gita,sujata).

daughter(gita,sujay).

daughter(sita,vijay).

daughter(gita,sujay).

daughter(tina,nilay).

father(X,Y):-son(Y,X),male(X).

father(X,Y):-daughter(Y,X),male(X).

mother(X,Y):-son(Y,X),female(X).

mother(X,Y):-daughter(Y,X),female(X).

brother(A,B):-father(X,A),father(X,B),male(A),male(B).

brother(A,B):-father(X,A),father(X,B),male(A),female(B).

sister(A,B):-father(X,A),father(X,B),female(A),female(B).

sister(A,B):-father(X,A),father(X,B),female(A),male(B).

cousin(C,D):-father(X,C),father(Y,D),brother(X,Y).

cousin(C,D):-mother(X,C),brother(Y,X),father(Y,D).

cousin(C,D):-father(X,C),sister(Y,X),mother(Y,D).

aunty(O,P):-father(X,P),brother(X,Y),couple(O,Y).

aunty(O,P):-mother(X,P),brother(Y,X),couple(O,Y).

aunty(O,P):-father(X,P),sister(O,X),female(O).

uncle(M,N):-father(X,N),brother(X,M),male(M).

uncle(M,N):-mother(X,N),brother(M,X),male(M).

grandfa(K,L):-father(X,L),father(K,X),male(K).

grandfa(K,L):-mother(X,L),father(K,X),male(K).

grandma(K,L):-father(X,L),mother(K,X),female(K).

goal

cousin(gita,kush).

**Output :**

yes

**Practical: 2**

**Aim : Given a relation player(“name”, age). Store knowledge about six players of age between 12-15. Write a prolog program to find possible combination of players to organize a two player tournament.**

PREDICATES

player(symbol, integer)

CLAUSES

player(viral,13).

player(dhaval,12).

player(chirag,13).

player(gaurang,13).

player(yogesh,14).

player(keyur,14).

GOAL

player(Person1, 13),

player(Person2, 14),

Person1 <> Person2.

**Output :**

Person1=viral, Person2=yogesh

Person1=viral, Person2=keyur

Person1=chirag, Person2=yogesh

Person1=chirag, Person2=keyur

Person1=gaurang, Person2=yogesh

Person1=gaurang, Person2=keyur

6 Solutions

**Practical: 3**

**Aim : Write a prolog program to check whether a person X can eat Food Y or not. A person X can eat food Y if X likes Y, X is available and X is eatable.**

PREDICATES

food(symbol)

canit(symbol,symbol)

eatable(symbol)

avail(symbol)

person(symbol)

start

CLAUSES

canit(X,Y):-

food(Y),eatable(Y),avail(Y),person(X).

person(viral).

person(gaurang).

person(chirag).

food(pizza).

food(burger).

food(hotdog).

food(brick).

avail(pizza).

avail(burger).

eatable(pizza).

eatable(burger).

eatable(hotdog).

start:-

canit(X,Y), write(X ," can it ",Y ,"\n"),fail.

GOAL

start.

**Output :**

viral can it pizza

gaurang can it pizza

chirag can it pizza

viral can it burger

gaurang can it burger

chirag can it burger

no

**Practical: 4**

**Aim : Write a program to represent knowledge of 10 employees of a company, *employee(EmpId,“name”, B\_Date, “Address”)*. Where name of employee and B\_Date are compound objects. Display list of Employees whose birthday is in current month.(Use Compound Obj and fail).**

DOMAINS

empid = symbol

name = person(symbol,symbol) /\* (First, Last) \*/

birthday = b\_date(symbol,integer,integer) /\* (Month, Day, Year) \*/

address = add(symbol,symbol) /\* (City, State) \*/

PREDICATES

nondeterm emp\_list(symbol,name,address,birthday)

get\_months\_birthdays

convert\_month(symbol,integer)

check\_birthday\_month(integer,birthday)

write\_person(name)

CLAUSES

get\_months\_birthdays:-

write("--------------->> Birthday List <<---------------"),nl,

write(" EmpID\t First name\t Last Name\n"),

write("--------------------------------------------------------------"),nl,

write("--------------------------------------------------------------"),nl,

date(\_, This\_month, \_), /\* Get month from system clock \*/

emp\_list(Empid,Person, \_, Date),

check\_birthday\_month(This\_month, Date),

write(" ", Empid,"\t "),

write\_person(Person),

fail.

get\_months\_birthdays:-

write("\n\n Press any key to continue: "),nl,

readchar(\_).

write\_person(person(First\_name,Last\_name)):-

write(" ",First\_name,"\t\t ",Last\_name),nl.

check\_birthday\_month(Mon,b\_date(Month,\_,\_)):-

convert\_month(Month,Month1),

Mon = Month1.

emp\_list("01", person(viral, prajapati), add(valsad,gujarat), b\_date(jun, 17, 1989)).

emp\_list("02", person(gaurang, talati), add(kanpur,up), b\_date(feb, 5, 1985)).

emp\_list("03", person(chirag, nathwani), add(rajkot,gujarat), b\_date(mar, 3, 1935)).

emp\_list("04", person(dhaval, rana), add(surat,gujarat), b\_date(apr, 29, 1951)).

emp\_list("05", person(yogesh, dangar), add(junagath,gujarat), b\_date(may, 12, 1962)).

emp\_list("06", person(rohini, patel), add(vadodara,gujarat), b\_date(oct, 17, 1980)).

emp\_list("07", person(vinita, shah), add(anand,gujarat), b\_date(jan, 20, 1986)).

emp\_list("08", person(ankita, chawda), add(rajkot,gujarat), b\_date(jul, 16, 1981)).

emp\_list("09", person(nikunj, patel), add(ahemdabad,gujarat), b\_date(aug, 10, 1981)).

emp\_list("10", person(tarun, joshi), add(valsad,gujarat), b\_date(jun, 25, 1981)).

convert\_month(jan, 1).

convert\_month(feb, 2).

convert\_month(mar, 3).

convert\_month(apr, 4).

convert\_month(may, 5).

convert\_month(jun, 6).

convert\_month(jul, 7).

convert\_month(aug, 8).

convert\_month(sep, 9).

convert\_month(oct, 10).

convert\_month(nov, 11).

convert\_month(dec, 12).

GOAL

get\_months\_birthdays.

**Output :**

--------------->> Birthday List <<---------------

EmpID First name Last Name

--------------------------------------------------------------

--------------------------------------------------------------

01 viral prajapati

10 tarun joshi

Press any key to continue:

yes

**Practical: 5**

**Aim : Write a program to perform different operations like add, mul, sub, div etc. (Implementation of switch statement using cut).**

Predicates

op(integer,integer,integer,integer)

readvalues

clauses

readvalues:-

write("Operations: "),nl,

write("Please for add(+) enter 1: "),nl,

write("Please for sub(-) enter 2: "),nl,

write("Please for mul(\*) enter 3: "),nl,

write("Please for div(/) enter 4: "),nl,

write("Enter Choice:"),readint(CH),

write("Enter X:"),readint(X),

write("Enter Y:"),readint(Y),

op(CH,X,Y,Ans),

write("Answer is : ",Ans,"\n").

op(1,X,Y,Ans):-!, Ans=X+Y.

op(2,X,Y,Ans):-!, Ans=X-Y.

op(3,X,Y,Ans):-!, Ans=X\*Y.

op(4,X,Y,Ans):-!, Ans=X/Y.

GOAL

readvalues.

**Output :**

Operations:

Please for add(+) enter 1:

Please for sub(-) enter 2:

Please for mul(\*) enter 3:

Please for div(/) enter 4:

Enter Choice:3

Enter X:50

Enter Y:25

Answer is : 1250

Yes

**Practical: 6**

**Aim : Implement 8-puzzle problem using Best First Search.**

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

int difference(int A[3][3], int B[3][3]);

int alter(int A[3][3], int B[3][3]);

int diffup(int A[3][3], int B[3][3]);

int diffdown(int A[3][3], int B[3][3]);

int diffleft(int A[3][3], int B[3][3]);

int diffright(int A[3][3], int B[3][3]);

int minimum(int a, int b, int c, int d);

void display(int A[3][3]);

int main()

{

int A[3][3] = {{2,8,3}, {1,6,4}, {7,0,5}}; // the initial array

int B[3][3] = {{1,2,3}, {8,0,4}, {7,6,5}};

int i, j;

int d; // difference between the desired array anfd the test array

int steps =0 ;

clrscr();

printf("Target: \n");

display(B);

printf("Original: \n");

display(A);

while(1)

{

d = difference(A, B);

if(d==0)

{

printf("TARGET ACHIEVED AT STEP: %d", steps);

system("PAUSE");

break;

}

steps++;

printf("\nStep: %d \n", steps);

alter(A, B); // changes the array according to whichever suitable

display(A);

system("PAUSE");

getch();

}

getch();

return 0;

}

void display(int A[3][3])

{

int i,j;

for(i=0;i<3;i++)

{

for(j=0;j<3;j++)

printf("%d ", A[i][j]);

printf("\n");

}

}

int difference(int A[3][3], int B[3][3])

{

int counter =0 ,i,j;

for(i=0;i<3; i++)

for( j=0;j<3;j++)

if(A[i][j] != B[i][j]) counter++;

return counter;

}

int alter(int A[3][3], int B[3][3])

{

int dup, ddown, dleft, dright;

int sel,min,temp, i , j, serial=0;

char change,ran[4];

dup = diffup(A, B);

ddown = diffdown(A, B);

dleft = diffleft(A, B);

dright = diffright(A, B);

printf("%d %d %d %d\n", dright, dleft, dup, ddown);

min = minimum(dup, ddown, dleft, dright);

printf("%d\n", min);

if (min == dright)

ran[serial++] = 'r';

if (min == dleft)

ran[serial++] = 'l';

if (min == dup)

ran[serial++] = 'u';

if (min == ddown)

ran[serial++] = 'd';

sel = rand()%serial;

change = ran[sel];

if(change == 'r')

{

for(i=0;i<3;i++)

{

for (j=0;j<2;j++)

{

if(A[i][j]==0)

{

A[i][j] = A[i][j+1];

A[i][j+1] = 0;

printf("right\n");

return 0;

}

}

}

}

else if(change == 'l')

{

for(i=0;i<3;i++)

{

for (j=1;j<3;j++)

{

if(A[i][j]==0)

{

A[i][j] = A[i][j-1];

A[i][j-1] = 0;

printf("left\n");

return 0;

}

}

}

}

else if(change == 'u')

{

for(i=1;i<3;i++)

{

for (j=0;j<3;j++)

{

if(A[i][j]==0)

{

A[i][j] = A[i-1][j];

A[i-1][j] = 0;

printf("up\n");

return 0;

}

}

}

}

else if(change == 'd')

{

for(i=0;i<2;i++)

{

for (j=0;j<3;j++)

{

if(A[i][j]==0)

{

A[i][j] = A[i+1][j];

A[i+1][j] = 0;

printf("down\n");

return 0;

}

}

}

}

return 0;

}

int diffup(int A[3][3], int B[3][3])

{

int temp[3][3], i, j;

for(i=0;i<3;i++) //copying the array A in array temp for testing

{

for (j=0;j<3;j++)

temp[i][j] = A[i][j];

}

for(i=1;i<3;i++) // swapping the space in upward direction

{

for (j=0;j<3;j++)

// starting the array from i=1 insures that the condition i=0 avoided

{

if(A[i][j]==0)

{

temp[i-1][j] = 0;

temp[i][j] = A[i-1][j];

}

}

}

return difference(temp, B);

}

int diffdown(int A[3][3], int B[3][3])

{

int temp[3][3], i, j;

for(i=0;i<3;i++) //copying the array A in array temp for testing

{

for (j=0;j<3;j++)

temp[i][j] = A[i][j];

}

for(i=0;i<2;i++) // swapping the space in downward direction

{

for (j=0;j<3;j++) // ending at i=1 insures that the condition i=0 avoided

{

if(A[i][j]==0)

{

temp[i+1][j] = 0;

temp[i][j] = A[i+1][j];

}

}

}

return difference(temp, B);

}

int diffleft(int A[3][3], int B[3][3])

{

int temp[3][3], i, j;

for(i=0;i<3;i++) //copying the array A in array temp for testing

{

for (j=0;j<3;j++)

temp[i][j] = A[i][j];

}

for(i=0;i<3;i++) // swapping the space in the left direction

{

for (j=1;j<3;j++)

{

if(A[i][j]==0)

{

temp[i][j-1] = 0;

temp[i][j] = A[i][j-1];

}

}

}

return difference(temp, B);

}

int diffright(int A[3][3], int B[3][3])

{

int temp[3][3], i, j;

for(i=0;i<3;i++) //copying the array A in array temp for testing

{

for (j=0;j<3;j++)

temp[i][j] = A[i][j];

}

for(i=0;i<3;i++) // swapping the space in the right direction

{

for (j=0;j<2;j++)

{

if(A[i][j]==0)

{

temp[i][j+1] = 0;

temp[i][j] = A[i][j+1];

}

}

}

return difference(temp, B);

}

int minimum (int a, int b, int c , int d)

{

int min = a;

if(b<min)

min= b;

if(c<min)

min = c;

if(d<min)

min = d;

return min;

}

**Output :**

Target:

1 2 3

8 0 4

7 6 5

Original:

2 8 3

1 6 4

7 0 5

Step: 1

6 6 3 5

3

up

2 8 3

1 0 4

7 6 5

Step: 2

5 4 4 5

4

left

2 8 3

0 1 4

7 6 5

Step: 3

3 4 4 5

3

right

2 8 3

1 0 4

7 6 5

Step: 4

5 4 4 5

4

left

2 8 3

0 1 4

7 6 5

Step: 5

3 4 4 5

3

right

2 8 3

1 0 4

7 6 5

Step: 6

5 4 4 5

4

up

2 0 3

1 8 4

7 6 5

Step: 7

5 3 4 3

3

down

2 8 3

1 0 4

7 6 5

Step: 8

5 4 4 5

4

up

2 0 3

1 8 4

7 6 5

Step: 9

5 3 4 3

3

left

0 2 3

1 8 4

7 6 5

Step: 10

4 3 3 2

2

down

1 2 3

0 8 4

7 6 5

Step: 11

0 2 3 3

0

right

1 2 3

8 0 4

7 6 5

TARGET ACHIEVED AT STEP: 11

**Practical: 7**

**Write a program to implement DFS (for 8 puzzle problem or Water Jug problem or any AI search problem)**

**(Water Jug Problem)**

#include <cstdio>

#include <stack>

#include <map>

#include <algorithm>

using namespace std;

// Representation of a state (x, y)

// x and y are the amounts of water in litres in the two jugs respectively

struct state {

int x, y;

// Used by map to efficiently implement lookup of seen states

bool operator < (const state& that) const {

if (x != that.x) return x < that.x;

return y < that.y;

}

};

// Capacities of the two jugs respectively and the target amount

int capacity\_x, capacity\_y, target;

void dfs(state start, stack <pair <state, int> >& path) {

stack <state> s;

state goal = (state) {-1, -1};

// Stores seen states so that they are not revisited and

// maintains their parent states for finding a path through

// the state space

// Mapping from a state to its parent state and rule no. that

// led to this state

map <state, pair <state, int> > parentOf;

s.push(start);

parentOf[start] = make\_pair(start, 0);

while (!s.empty()) {

// Get the state at the front of the stack

state top = s.top();

s.pop();

// If the target state has been found, break

if (top.x == target || top.y == target) {

goal = top;

break;

}

// Find the successors of this state

// This step uses production rules to produce successors of the current state

// while pruning away branches which have been seen before

// Rule 1: (x, y) -> (capacity\_x, y) if x < capacity\_x

// Fill the first jug

if (top.x < capacity\_x) {

state child = (state) {capacity\_x, top.y};

// Consider this state for visiting only if it has not been visited before

if (parentOf.find(child) == parentOf.end()) {

s.push(child);

parentOf[child] = make\_pair(top, 1);

}

}

// Rule 2: (x, y) -> (x, capacity\_y) if y < capacity\_y

// Fill the second jug

if (top.y < capacity\_y) {

state child = (state) {top.x, capacity\_y};

if (parentOf.find(child) == parentOf.end()) {

s.push(child);

parentOf[child] = make\_pair(top, 2);

}

}

// Rule 3: (x, y) -> (0, y) if x > 0

// Empty the first jug

if (top.x > 0) {

state child = (state) {0, top.y};

if (parentOf.find(child) == parentOf.end()) {

s.push(child);

parentOf[child] = make\_pair(top, 3);

}

}

// Rule 4: (x, y) -> (x, 0) if y > 0

// Empty the second jug

if (top.y > 0) {

state child = (state) {top.x, 0};

if (parentOf.find(child) == parentOf.end()) {

s.push(child);

parentOf[child] = make\_pair(top, 4);

}

}

// Rule 5: (x, y) -> (min(x + y, capacity\_x), max(0, x + y - capacity\_x)) if y > 0

// Pour water from the second jug into the first jug until the first jug is full

// or the second jug is empty

if (top.y > 0) {

state child = (state) {min(top.x + top.y, capacity\_x), max(0, top.x + top.y - capacity\_x)};

if (parentOf.find(child) == parentOf.end()) {

s.push(child);

parentOf[child] = make\_pair(top, 5);

}

}

// Rule 6: (x, y) -> (max(0, x + y - capacity\_y), min(x + y, capacity\_y)) if x > 0

// Pour water from the first jug into the second jug until the second jug is full

// or the first jug is empty

if (top.x > 0) {

state child = (state) {max(0, top.x + top.y - capacity\_y), min(top.x + top.y, capacity\_y)};

if (parentOf.find(child) == parentOf.end()) {

s.push(child);

parentOf[child] = make\_pair(top, 6);

}

}

}

// Target state was not found

if (goal.x == -1 || goal.y == -1)

return;

// backtrack to generate the path through the state space

path.push(make\_pair(goal, 0));

// remember parentOf[start] = (start, 0)

while (parentOf[path.top().first].second != 0)

path.push(parentOf[path.top().first]);

}

int main() {

stack <pair <state, int> > path;

printf("Enter the capacities of the two jugs : ");

scanf("%d %d", &capacity\_x, &capacity\_y);

printf("Enter the target amount : ");

scanf("%d", &target);

dfs((state) {0, 0}, path);

if (path.empty())

printf("\nTarget cannot be reached.\n");

else {

printf("\nNumber of moves to reach the target : %d\nOne path to the target is as follows :\n", path.size() - 1);

while (!path.empty()) {

state top = path.top().first;

int rule = path.top().second;

path.pop();

switch (rule) {

case 0: printf("State : (%d, %d)\n#\n", top.x, top.y);

break;

case 1: printf("State : (%d, %d)\nAction : Fill the first jug\n", top.x, top.y);

break;

case 2: printf("State : (%d, %d)\nAction : Fill the second jug\n", top.x, top.y);

break;

case 3: printf("State : (%d, %d)\nAction : Empty the first jug\n", top.x, top.y);

break;

case 4: printf("State : (%d, %d)\nAction : Empty the second jug\n", top.x, top.y);

break;

case 5: printf("State : (%d, %d)\nAction : Pour from second jug into first jug\n", top.x, top.y);

break;

case 6: printf("State : (%d, %d)\nAction : Pour from first jug into second jug\n", top.x, top.y);

break;

}

}

}

return 0;

}

**Practical: 8**

**Aim: Write a program to implement Tic-Tac-Toe game problem.**

#include<stdio.h>

#include<conio.h>

void Board();

voidPlayerX();

voidPlayerO();

voidPlayer\_win();

void check();

int win=0,wrong\_X=0,wrong\_O=0,chk=0;

charname\_X[30];

charname\_O[30];

intpos\_for\_X[3][3];

intpos\_for\_O[3][3];

intpos\_marked[3][3];

void main()

{

inti,ch,j;

charans;

/\* clrscr();

printf("\n\t\t\t\tTIC TAC TOE");

printf("\n\t\t\t\t");

for(i=1;i<=11;i++)

{

delay(10000);

printf("\*");

}\*/

do

{

clrscr();

printf("\n\t\t\t\tTIC TAC TOE");

printf("\n\t\t\t\t");

for(i=1;i<=11;i++)

{

delay(10000);

printf("\*");

}

printf("\n1.Start The Game");

printf("\n2.Quit The Game");

printf("\nEnter your choice(1-2) : ");

scanf("%d",&ch);

switch(ch)

{

case 1:

chk=0;

win=0;

for(i=1;i<=3;i++)

{

for(j=1;j<=3;j++)

{

pos\_for\_X[i][j]=0;

pos\_for\_O[i][j]=0;

pos\_marked[i][j]=0;

}

}

printf("\n\n");

clrscr();

printf("\nEnter the name of the player playing for \'X\': ");

fflush(stdin);

gets(name\_X);

printf("\nEnter the name of the player playing for \'O\': ");

fflush(stdin);

gets(name\_O);

Board();

for(;;)

{

if(win==1)

break;

check();

if(chk==9)

{

printf("\n\t\t\t MATCH DRAWS!!");

printf("\nPress any key....");

break;

}

else

chk=0;

printf("\nTURN FOR %s:",name\_X);

PlayerX();

do

{

if(wrong\_X!=1)

break;

wrong\_X=0;

printf("\nTURN FOR %s:",name\_X);

PlayerX();

}while(wrong\_X==1);

check();

if(chk==9)

{

printf("\n\t\t\tMATCH DRAWS");

printf("\nPress any key....");

break;

}

else

chk=0;

printf("\nTURN FOR %s:",name\_O);

PlayerO();

do

{

if(wrong\_O!=1)

break;

wrong\_O=0;

printf("\nTURN FOR %s:",name\_O);

PlayerO();

}while(wrong\_O==1);

}

Board();

if(win!=1)

{

printf("\n\t\t\tMATCH DRAWS!!");

printf("\nPress any key.......");

}

getch();

break;

case 2:

printf("\n\n\n\t\t\tThank You For Playing The Game.");

printf("\n\t\t\t###############################");

getch();

exit(1);

break;

}

printf("\nWant To Play(Y/N) ? ");

fflush(stdin);

scanf("%c",&ans);

}while(ans=='y' || ans=='Y');

}

void Board()

{

inti,j;

clrscr();

printf("\n\t\t\t\tTIC TAC TOE BOARD");

printf("\n\t\t\t\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

printf("\n\n\n");

printf("\n\t\t\t 1\t 2\t 3");

for(i=1;i<=3;i++)

{

printf("\n \t\t\t \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

printf("\n \t\t\tº\t º\t º\t º");

printf("\n\t\t%d\t",i);

for(j=1;j<=3;j++)

{

if(pos\_for\_X[i][j]==1)

{

printf(" X");

printf(" ");

}

else if(pos\_for\_O[i][j]==1)

{

printf(" O");

printf(" ");

}

else

{

printf(" ");

continue;

}

}

printf("\n\t\t\tº\t º\t º\t º");

}

printf("\n\t\t\t------------------------------");

Player\_win();

}

voidPlayerX()

{

introw,col;

if(win==1)

return;

printf("\nEnter the row no. : ");

fflush(stdin);

scanf("%d",&row);

printf("Enter the column no. : ");

fflush(stdin);

scanf("%d",&col);

if(pos\_marked[row][col]==1 || row<1 || row>3 || col<1 || col>3)

{

printf("\nWRONG POSITION!! Press any key.....");

wrong\_X=1;

getch();

Board();

}

else

{

pos\_for\_X[row][col]=1;

pos\_marked[row][col]=1;

Board();

}

}

voidPlayerO()

{

introw,col;

if(win==1)

return;

printf("\nEnter the row no. : ");

scanf("%d",&row);

printf("Enter the column no. : ");

scanf("%d",&col);

if(pos\_marked[row][col]==1 || row<1 || row>3 || col<1 || col>3)

{

printf("\nWRONG POSITION!! Press any key....");

wrong\_O=1;

getch();

Board();

}

else

{

pos\_for\_O[row][col]=1;

pos\_marked[row][col]=1;

Board();

}

}

voidPlayer\_win()

{

int i;

for(i=1;i<=3;i++)

{

if(pos\_for\_X[i][1]==1 &&pos\_for\_X[i][2]==1 &&pos\_for\_X[i][3]==1)

{

win=1;

printf("\n\nRESULT: %s wins!!",name\_X);

printf("\nPress any key............");

return;

}

}

for(i=1;i<=3;i++)

{

if(pos\_for\_X[1][i]==1 &&pos\_for\_X[2][i]==1 &&pos\_for\_X[3][i]==1)

{

win=1;

printf("\n\nRESULT: %s wins!!",name\_X);

printf("\nPress any key............");

return;

}

}

if(pos\_for\_X[1][1]==1 &&pos\_for\_X[2][2]==1 &&pos\_for\_X[3][3]==1)

{

win=1;

printf("\n\nRESULTL: %s wins!!",name\_X);

printf("\nPress any key......");

return;

}

else if(pos\_for\_X[1][3]==1 &&pos\_for\_X[2][2]==1 &&

pos\_for\_X[3][1]==1)

{

win=1;

printf("\n\nRESULT: %s wins!!",name\_X);

printf("\nPress any key.....");

return;

}

for(i=1;i<=3;i++)

{

if(pos\_for\_O[i][1]==1 &&pos\_for\_O[i][2]==1 &&pos\_for\_O[i][3]==1)

{

win=1;

printf("\n\nRESULT: %s wins!!",name\_O);

printf("\nPress any key.....");

return;

}

}

for(i=1;i<=3;i++)

{

if(pos\_for\_O[1][i]==1 &&pos\_for\_O[2][i]==1 &&pos\_for\_O[3][i]==1)

{

win=1;

printf("\n\nRESULT: %s wins!!",name\_O);

printf("\nPress any key.....");

return;

}

}

if(pos\_for\_O[1][1]==1 &&pos\_for\_O[2][2]==1 &&pos\_for\_O[3][3]==1)

{

win=1;

printf("\n\nRESULT: %s wins!!",name\_O);

printf("\nPress any key.....");

return;

}

else if(pos\_for\_O[1][3]==1 &&pos\_for\_O[2][2]==1 &&

pos\_for\_O[3][1]==1)

{

win=1;

printf("\n\nRESULT: %s wins!!",name\_O);

printf("\nPress any key.....");

return;

}

}

void check()

{

inti,j;

for(i=1;i<=3;i++)

{

for(j=1;j<=3;j++)

{

if(pos\_marked[i][j]==1)

chk++;

else

continue;

}

}

}

**Practical: 09**

**Write a program to implement Single Player Game (Using Heuristic Function)**

role(player).

succ(a,b).

. . .

succ(g,h).

init(cell(a,single)).

init(cell(Y,single)) :- succ(X,Y).

legal(P,jump(X,Y)) :- true(cell(X,single)), true(cell(Y,single)),

twobetween(X,Y).

legal(P,jump(X,Y)) :- true(cell(X,single)), true(cell(Y,single)),

twobetween(Y,X).

next(cell(X,nocoin)) :- does(player,jump(X,Y)).

next(cell(X,double)) :- does(player,jump(Y,X)).

next(cell(X,Number)) :- true(cell(X,Number)), does(player,jump(Y,Z)),

distinct(X,Y), distinct(X,Z).

terminal :- not continuable

continuable :- legal(player,Move).

goal(player,100) :- not lonelycoin.

goal(player, 50) :- lonelycoin, not threelonelycoins.

goal(player, 0) :- threelonelycoins.

lonelycoin :- true(cell(X,single)).

threelonelycoins :- true(cell(X,single)), true(cell(Y,single)),

true(cell(Z,single)), distinct(X,Y),

distinct(X,Z), distinct(Y,Z).

twobetween(X,Y) :- succ(X,Z), true(cell(Z,nocoin)), twobetween(Z,Y).

twobetween(X,Y) :- succ(X,Z), true(cell(Z,single)), onebetween(Z,Y).

twobetween(X,Y) :- succ(X,Z), true(cell(Z,double)), nilbetween(Z,Y).

onebetween(X,Y) :- succ(X,Z), true(cell(Z,nocoin)), onebetween(Z,Y).

onebetween(X,Y) :- succ(X,Z), true(cell(Z,single)), nilbetween(Z,Y).

nilbetween(X,Y) :- succ(X,Z), true(cell(Z,nocoin)), nilbetween(Z,Y).

nilbetween(X,Y) :- succ(X,Y).

**Practical: 10**

**Write a program to Implement A\* Algorithm.**

Open := {Start}

Closed := {}

while Open is not empty loop

find a Node in Open minimising Cost(Node)+H(Node)

(if there are several such nodes, prefer a goal node)

Open := Open \ {Node}

Closed := Closed ∪ {Node}

if Goal(Node) then

Path := [Node]

while Parent(Node) is defined loop

Node := Parent(Node)

Path := [Node|Path]

end loop

return Path

end if

for each (Child,StepCost) in Expander(Node) loop

if Child in Open+Closed then

if Cost(Node) + StepCost < Cost(Child) then

Cost(Child) := Cost(Node) + StepCost

Parent(Child) := Node

if Child in Closed then

Open := Open ∪ {Child}

Closed := Closed \ {Child}

end if

end if

else

Parent(Child) := Node

Cost(Child) := Cost(Node) + StepCost

Open := Open ∪ {Child}

end if

end loop

end loop

return failure

**Practical – 11**

**Write a program to solve N-Queens problem using Prolog.**

% This program finds a solution to the 8 queens problem. That is, the problem of placing 8

% queens on an 8x8 chessboard so that no two queens attack each other. The prototype

% board is passed in as a list with the rows instantiated from 1 to 8, and a corresponding

% variable for each column. The Prolog program instantiates those column variables as it

% finds the solution.

% Programmed by Ron Danielson, from an idea by Ivan Bratko.

% 2/17/00

queens([]). % when place queen in empty list, solution found

queens([ Row/Col | Rest]) :- % otherwise, for each row

queens(Rest), % place a queen in each higher numbered row

member(Col, [1,2,3,4,5,6,7,8]), % pick one of the possible column positions

safe( Row/Col, Rest). % and see if that is a safe position

% if not, fail back and try another column, until

% the columns are all tried, when fail back to

% previous row

safe(Anything, []). % the empty board is always safe

safe(Row/Col, [Row1/Col1 | Rest]) :- % see if attack the queen in next row down

Col =\= Col1, % same column?

Col1 - Col =\= Row1 - Row, % check diagonal

Col1 - Col =\= Row - Row1,

safe(Row/Col, Rest). % no attack on next row, try the rest of board

member(X, [X | Tail]). % member will pick successive column values

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

member(X, Tail).

board([1/C1, 2/C2, 3/C3, 4/C4, 5/C5, 6/C6, 7/C7, 8/C8]). % prototype board

**Practical – 12**

**Write a program to solve 8 puzzle problem using Prolog.**

domains

value, row, col, gval, hval, pval, sval, parent, nodeno = integer

nodevalue=ndval(value,row,col)

nodelist=nodevalue\*

loclist=value\*

poslist=posval(loclist,value)

nodestruct=ndstruct(nodelist,value,value,value,value)

hvallist=nodestruct\*

database

opennodeinfo(nodelist,value,value,value,value).

closenodeinfo(nodelist,value,value,value,value).

bestnodeinfo(nodelist,value,value,value,value).

nvalue(value,row,col).

rowcolCounter(value).

nodeNo(value).

currParent(value).

predicates

displayPuzzle.

displayNodeList(nodelist).

calHvalue(nodelist,value).

calPvalue(nodelist,value).

getNodeInfo(nodevalue,value,row,col).

finalnode(nodelist,hval).

pvalue(value, poslist).

findStepsFar(value,poslist,value).

memberOfLocList(value,loclist).

moveLeft.

moveRight.

moveUp.

moveDown.

findValue(row,col,value).

findBlank(row,col).

setNewPos(value,row,col).

setCurrNode(nodelist).

emptyCurrNode.

genNodeList(nodelist,nodelist).

genHvalList(hvallist,hvallist).

input.

performSearch.

processCurrNode.

checkInOpen(nodelist,value).

checkInClose(nodelist,value).

bubblesort(hvallist,hvallist).

swap(hvallist,hvallist).

insertSortedList(hvallist).

clauses

finalnode([ndval(1,0,0),ndval(2,0,1),ndval(3,0,2),ndval(8,1,0),

ndval(0,1,1),ndval(4,1,2),ndval(7,2,0),ndval(6,2,1),

ndval(5,2,2)],0).

pvalue(1,posval([0],0)).

pvalue(1,posval([1,3],1)).

pvalue(1,posval([2,6,4],2)).

pvalue(1,posval([7,5],3)).

pvalue(1,posval([8],4)).

pvalue(2,posval([1],0)).

pvalue(2,posval([0,2,4],1)).

pvalue(2,posval([3,5,7],2)).

pvalue(2,posval([6,8],3)).

pvalue(3,posval([2],0)).

pvalue(3,posval([1,5],1)).

pvalue(3,posval([0,4,8],2)).

pvalue(3,posval([3,7],3)).

pvalue(3,posval([6],4)).

pvalue(4,posval([5],0)).

pvalue(4,posval([2,4,8],1)).

pvalue(4,posval([1,3,7],2)).

pvalue(4,posval([0,6],3)).

pvalue(5,posval([8],0)).

pvalue(5,posval([5,7],1)).

pvalue(5,posval([2,4,6],2)).

pvalue(5,posval([1,3],3)).

pvalue(5,posval([0],4)).

pvalue(6,posval([7],0)).

pvalue(6,posval([4,6,8],1)).

pvalue(6,posval([1,3,5],2)).

pvalue(6,posval([0,2],3)).

pvalue(7,posval([6],0)).

pvalue(7,posval([3,7],1)).

pvalue(7,posval([0,4,8],2)).

pvalue(7,posval([1,5],3)).

pvalue(7,posval([2],4)).

pvalue(8,posval([3],0)).

pvalue(8,posval([0,4,6],1)).

pvalue(8,posval([1,5,7],2)).

pvalue(8,posval([2,8],3)).

pvalue(0,posval([4],0)).

pvalue(0,posval([1,3,5,7],1)).

pvalue(0,posval([0,2,6,8],2)).

input:-

assert(opennodeinfo([ndval(2,0,0),ndval(1,0,1),

ndval(6,0,2),ndval(4,1,0),ndval(0,1,1),

ndval(8,1,2),ndval(7,2,0),ndval(5,2,1),

ndval(3,2,2)],0,0,0,0)),

assert(rowcolCounter(8)),

assert(nodeNo(1)),

assert(currParent(0)).

displayPuzzle:-

genNodeList([],Nodelist),

bestnodeinfo(Nodelist,Hval,Gval,Parent,Nodeno),

displayNodeList(Nodelist),

write("h(n) : ",Hval),nl,

write("g(n) : ",Gval),nl,

write("parent(n) : ",Parent),nl,

write("nodno(n) : ",Nodeno).

displayNodeList([]).

displayNodeList([ndval(Value1,\_,\_),ndval(Value2,\_,\_),ndval(Value3,\_,\_)|Tail]):-

write(Value1," "), write(Value2," "), write(Value3," "),nl,

displayNodeList(Tail).

getNodeInfo(ndval(Value,Row,Col),Value,Row,Col).

emptyCurrNode:-

retractall(nvalue(\_,\_,\_)).

setCurrNode([]):- !.

setCurrNode([ndval(Value,Row,Col)|Tail]):-

assert(nvalue(Value,Row,Col)),

setCurrNode(Tail).

calHvalue(NodeList,Hval):-

calPvalue(NodeList,Pval),

Hval=Pval.

calPvalue([],0):- !.

calPvalue([Head|Tail],Pval):-

getNodeInfo(Head,Value,Row,Col),

CurrPos=(Row\*3)+Col,

pvalue(Value,PosList),

findStepsFar(CurrPos,PosList,Steps),

calPvalue(Tail,NewPval),

Pval=Steps+NewPval.

findStepsFar(CurrPos,posval(LocList,Steps),Steps):-

memberOfLocList(CurrPos,LocList).

memberOfLocList(CurrPos,[CurrPos|\_]) :- !.

memberOfLocList(CurrPos,[\_|Rest]):-

memberOfLocList(CurrPos,Rest).

moveLeft:-

findBlank(Row,Col),

Col>0,

NewCol=Col-1,

findValue(Row,NewCol,Value),

setNewPos(Value,Row,Col),

setNewPos(0,Row,NewCol),

write("\nleft successor"),

processCurrNode,

setNewPos(0,Row,Col),

setNewPos(Value,Row,NewCol),!.

moveLeft.

moveRight:-

findBlank(Row,Col),

Col<2,

NewCol=Col+1,

findValue(Row,NewCol,Value),!,

setNewPos(Value,Row,Col),

setNewPos(0,Row,NewCol),

write("\nright successor"),

processCurrNode,

setNewPos(0,Row,Col),

setNewPos(Value,Row,NewCol),!.

moveRight.

moveUp:-

findBlank(Row,Col),

Row>0,

NewRow=Row-1,

findValue(NewRow,Col,Value),

setNewPos(Value,Row,Col),

setNewPos(0,NewRow,Col),

write("\nup successor"),

processCurrNode,

setNewPos(0,Row,Col),

setNewPos(Value,NewRow,Col),!.

moveUp.

moveDown:-

findBlank(Row,Col),

Row<2,

NewRow=Row+1,

findValue(NewRow,Col,Value),

setNewPos(Value,Row,Col),

SetNewPos(0,NewRow,Col),

write("\ndown successor"),

processCurrNode,

setNewPos(0,Row,Col),

setNewPos(Value,NewRow,Col),!.

moveDown.

processCurrNode:-

genNodeList([],Nodelist),

checkInOpen(Nodelist,Oval),

Oval=1,

checkInClose(Nodelist,Cval),

Cval=1,

calHvalue(Nodelist,Hval),

currParent(Parent),

Gval=Parent+1,

nodeNo(Nodeno),

NewNodeno=Nodeno+1,

retract(nodeNo(Nodeno)),

assert(nodeNo(NewNodeno)),

assert(opennodeinfo(Nodelist,Hval,Gval,Parent,Nodeno)),

nl,displayNodeList(Nodelist), write("Parent : ",Parent," Nodeno : ",Nodeno),

readchar(\_),

genHvalList([],HvalList),

bubblesort(HvalList,SortedList),

insertSortedList(SortedList).

processCurrNode.

checkInOpen(Nodelist,Oval):-

Oval=1,

not(opennodeinfo(Nodelist,\_,\_,\_,\_)),!.

checkInOpen(Nodelist,Oval):-

opennodeinfo(Nodelist,\_,\_,\_,\_),

Oval=0.

checkInClose(Nodelist,Cval):-

Cval=1,

not(closenodeinfo(Nodelist,\_,\_,\_,\_)),!.

checkInClose(Nodelist,Cval):-

closenodeinfo(Nodelist,\_,\_,\_,\_),

Cval=0.

findValue(Row,Col,Value):-

nvalue(Value,Row,Col).

findBlank(Row,Col):-

nvalue(0,Row,Col).

setNewPos(Value,Row,Col):-

retract(nvalue(\_,Row,Col)),

assert(nvalue(Value,Row,Col)).

genNodeList(L,NodeList):-

rowcolCounter(Cnt),

Cnt>=0,

NewCnt=Cnt-1,

retract(rowcolCounter(Cnt)),

assert(rowcolCounter(NewCnt)),

Row=Cnt div 3,

Col=Cnt mod 3,

nvalue(Value,Row,Col),

NewList=[ ndval(Value,Row,Col) | L ],!,

genNodeList(NewList,NodeList).

genNodeList(NodeList,NodeList):-

rowcolCounter(Cnt),

retract(rowcolCounter(Cnt)),

assert(rowcolCounter(8)),!.

genHvalList(L,Hvallist):-

retract(opennodeinfo(Nodelist,Hval,Gval,Parent,Nodeno)),

Newlist=[ndstruct(Nodelist,Hval,Gval,Parent,Nodeno)|L],

genHvalList(Newlist,Hvallist).

genHvalList(Hvallist,Hvallist):- !.

insertSortedList([ndstruct(Nodelist,Hval,Gval,Parent,Nodeno)|Tail]):-

assert(opennodeinfo(Nodelist,Hval,Gval,Parent,Nodeno)),

insertSortedList(Tail).

insertSortedList([]) :- !.

performSearch:-

retract(bestnodeinfo(\_,\_,\_,\_,\_)),

opennodeinfo(Nodelist,Hval,Gval,Parent,Nodeno),

Hval<> 0,

retract(currParent(\_)),

assert(currParent(Nodeno)),

assert(bestnodeinfo(Nodelist,Hval,Gval,Parent,Nodeno)),

emptyCurrNode,

setCurrNode(Nodelist),

write("\ncurrent best node\n"),

displayNodeList(Nodelist),

write("h(n) : ",Hval),nl,

write("g(n) : ",Gval),nl,

write("parent(n) : ",Parent),nl,

write("nodno(n) : ",Nodeno),

readchar(\_),

assert(closenodeinfo(Nodelist,Hval,Gval,Parent,Nodeno)),

retract(opennodeinfo(Nodelist,Hval,Gval,Parent,Nodeno)),

moveLeft,

moveUp,

moveRight,

moveDown,

performSearch.

performSearch.

bubblesort(List, Sorted) :-

swap(List, List1), !,

bubblesort(List1, Sorted).

bubblesort(Sorted, Sorted).

swap([ndstruct(A,X,B,C,D),ndstruct(E,Y,F,G,H)|Rest], [ndstruct(E,Y,F,G,H),ndstruct(A,X,B,C,D)|Rest]) :- X > Y.

swap([ndstruct(A,Z,B,C,D)|Rest], [ndstruct(A,Z,B,C,D)|Rest1]) :- swap(Rest, Rest1).

goal

makewindow(1,2,3,"8-Puzzle Problem",0,0,25,80),

input,

opennodeinfo(Nodelist,\_,\_,\_,\_),

calHvalue(Nodelist,Hval),

nodeNo(Nodeno),

NewNodeno=Nodeno+1,

retract(nodeNo(Nodeno)),

assert(nodeNo(NewNodeno)),

retract(opennodeinfo(Nodelist,\_,\_,\_,\_)),

assert(opennodeinfo(Nodelist,Hval,0,0,Nodeno)),

assert(bestnodeinfo(Nodelist,Hval,0,0,Nodeno)),

setCurrNode(Nodelist),

emptyCurrNode,

performSearch.

**Practical: 13**

**Aim:** **Write a program to solve travelling salesman problem using Prolog.**

domains

/\* will allow us cooperate with better names, for me this is like #typedef in C++ \*/

town = symbol

distance = unsigned

rib = r(town,town,distance)

tlist = town\*

rlist = rib\*

predicates

nondeterm way(town,town,rlist,distance)

nondeterm route(town,town,rlist,tlist,distance)

nondeterm route1(town,tlist,rlist,tlist,distance)

nondeterm ribsmember(rib,rlist)

nondeterm townsmember(town,tlist)

nondeterm tsp(town,town,tlist,rlist,tlist,distance)

nondeterm ham(town,town,tlist,rlist,tlist,distance)

nondeterm shorterRouteExists(town,town,tlist,rlist,distance)

nondeterm alltown(tlist,tlist)

nondeterm write\_list(tlist)

clauses

/\*

Nothing special with write\_list.

If list is empty we do nothing,

and if something there we write head and call ourselves for tail.

\*/

write\_list([]).

write\_list([H|T]):-

write(H,‘ ‘),

write\_list(T).

/\* Is true if town X is in list of towns… \*/

townsmember(X,[X|\_]).

townsmember(X,[\_|L]):-

townsmember(X,L).

/\* Is true if rib X is in list of ribs… \*/

ribsmember(r(X,Y,D),[r(X,Y,D)|\_]).

ribsmember(X,[\_|L]):-

ribsmember(X,L).

/\* Is true if Route consists of all Towns presented in second argument \*/

alltown(\_,[]).

alltown(Route,[H|T]):-

townsmember(H,Route),

alltown(Route,T).

/\* Is true if there is a way from Town1 to Town2, and also return distance between them \*/

way(Town1,Town2,Ways,OutWayDistance):-

ribsmember(r(Town1,Town2,D),Ways),

OutWayDistance = D.

%/\*

/\* If next is uncommented then we are using non-oriented graph\*/

way(Town1,Town2,Ways,OutWayDistance):-

ribsmember(r(Town2,Town1,D),Ways), /\*switching direction here…\*/

OutWayDistance = D.

%\*/

/\* Is true if we could build route from Town1 to Town2 \*/

route(Town1,Town2,Ways,OutRoute,OutDistance):-

route1(Town1,[Town2],Ways,OutRoute,T1T2Distance),

%SWITCH HERE

way(Town2,Town1,Ways,LasDist), /\* If you want find shortest way comment this line\*/

OutDistance = T1T2Distance + LasDist. /\* And make this: OutDistance = T1T2Distance.\*/

route1(Town1,[Town1|Route1],\_,[Town1|Route1],OutDistance):-

OutDistance = 0.

/\* Does the actual finding of route. We take new TownX town and if it is not member of PassedRoute,

we continue searching with including TownX in the list of passed towns.\*/

route1(Town1,[Town2|PassedRoute],Ways,OutRoute,OutDistance):-

way(TownX,Town2,Ways,WayDistance),

not(townsmember(TownX,PassedRoute)),

route1(Town1,[TownX,Town2|PassedRoute],Ways,OutRoute,CompletingRoadDistance),

OutDistance = CompletingRoadDistance + WayDistance.

shorterRouteExists(Town1,Town2,Towns,Ways,Distance):-

ham(Town1,Town2,Towns,Ways,\_,Other),

Other < Distance.

/\* calling tsp(a,a,…. picks any one connected to a town and calls another tsp\*/

tsp(Town1,Town1,Towns,Ways,BestRoute,MinDistance):-

way(OtherTown,Town1,Ways,\_),

tsp(Town1,OtherTown,Towns,Ways,BestRoute,MinDistance).

/\*Travelling Salesman Problem is Hammilton way which is the shortes of other ones.\*/

tsp(Town1,Town2,Towns,Ways,BestRoute,MinDistance):-

ham(Town1,Town2,Towns,Ways,Route,Distance),

not(shorterRouteExists(Town1,Town2,Towns,Ways,Distance)),

BestRoute = Route,

MinDistance = Distance.

/\*Hammilton route from Town1 to Town2 assuming that Town2->Town1 way exists.\*/

ham(Town1,Town2,Towns,Ways,Route,Distance):-

route(Town1,Town2,Ways,Route,Distance),

%SWITCH HERE

alltown(Route,Towns), % if you want simple road without including all towns you could uncomment this line

write\_list(Route),

write(” tD = “,Distance,“n“).

% fail.

goal

/\* EXAMPLE 1

AllTowns = [a,b,c,d],

AllWays = [r(a,b,1),r(a,c,10),r(c,b,2),r(b,c,2),r(b,d,5),r(c,d,3),r(d,a,4)],

\*/

/\* EXAMPLE 2 \*/

AllTowns = [a,b,c,d,e],

AllWays = [r(a,c,1),r(a,b,6),r(a,e,5),r(a,d,8),r(c,b,2),r(c,d,7),r(c,e,10),r(b,d,3),r(b,e,9),r(d,e,4)],

tsp(a,a,AllTowns,AllWays,Route,Distance),

%SWITCH HERE

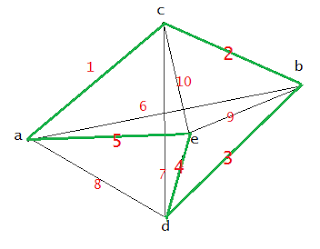
% tsp(a,b,AllTowns,AllWays,Route,Distance),

write(“Finally:n“),

write\_list(Route),

write(” tMIN\_D = “,Distance,“n“).

Let’s take a look on the following graph:

[](http://1.bp.blogspot.com/_kD34xgIwKhc/Sxwx_lREcVI/AAAAAAAAADw/SXXMu0m_HDI/s1600-h/TSP_graph.png)

output :

  a e d b c       D = 15  
  a e d b c       D = 15  
  a d e b c       D = 24  
  a e b d c       D = 25  
  a b e d c       D = 27  
  a d b e c       D = 31

  a b d e c       D = 24

Finally:  
 a e d b c       MIN\_D = 15

**Practical: 14**

**Convert following Prolog predicates into Semantic Net**

**cat(tom).**

**cat(cat1).**

**mat(mat1).**

**sat\_on(cat1,mat1).**

**bird(bird1).**

**caught(tom,bird1).**

**like(X,cream) :– cat(X).**

**mammal(X) :– cat(X).**

**has(X,fur) :– mammal(X).**

**animal(X) :– mammal(X).**

**animal(X) :– bird(X).**

**owns(john,tom).**

**is\_coloured(tom,ginger).**

**Write the Conceptual Dependency for following statements.**

***(a)* John gives Mary a book**

***(b)* John gave Mary the book yesterday**

# Semantic Nets

Semantic networks are an alternative to predicate logic as a form of knowledge representation. The idea is that we can store our knowledge in the form of a graph, with nodes representing objects in the world, and arcs representing relationships between those objects. For example, the following:

is intended to represent the data:

Tom is a cat.

Tom caught a bird.

Tom is owned by John.

Tom is ginger in colour.

Cats like cream.

The cat sat on the mat.

A cat is a mammal.

A bird is an animal.

All mammals are animals.

Mammals have fur.

It is argued that this form of representation is closer to the way humans structure knowledge by building mental links between things than the predicate logic we considered earlier. Note in particular how all the information about a particular object is concentrated on the node representing that object, rather than scattered around several clauses in logic.

There is, however, some confusion here which stems from the imprecise nature of semantic nets. A particular problem is that we haven’t distinguished between nodes representing classes of things, and nodes representing individual objects. So, for example, the node labelled Cat represents both the single (nameless) cat who sat on the mat, and the whole class of cats to which Tom belongs, which are mammals and which like cream. The is\_a link has two different meanings – it can mean that one object is an individual item from a class, for example Tom is a member of the class of cats, or that one class is a subset of another, for example, the class of cats is a subset of the class of mammals. This confusion does not occur in logic, where the use of quantifiers, names and predicates makes it clear what we mean so:

Tom is a cat is represented by Cat(Tom)

The cat sat on the mat is represented by ∃x∃y(Cat(x)∧Mat(y)∧SatOn(x,y))

A cat is a mammal is represented by ∀x(Cat(X)→Mammal(x))

We can clean up the representation by distinguishing between nodes representing individual or instances, and nodes representing classes. The is\_a link will only be used to show an individual belonging to a class. The link representing one class being a subset of another will be labelled a\_kind\_of, or ako for short. The names instance and subclass are often used in the place of is\_a and ako, but we will use these terms with a slightly different meaning in the section on Frames below.



Note also the modification which causes the link labelled is\_owned\_by to be reversed in direction. This is in order to avoid links representing passive relationships. In general a passive sentence can be replaced by an active one, so “Tom is owned by John” becomes “John owns Tom”. In general the rule which converts passive to active in English converts sentences of the form “X is Yed by Z” to “Z Ys X”. This is just an example (though often used for illustration) of the much more general principle of looking beyond the immediate surface structure of a sentence to find its deep structure.

The revised semantic net is:



Note that where we had an unnamed member of some class, we have had to introduce a node with an invented name to represent a particular member of the class. This is a process similar to the Skolemisation we considered previously as a way of dealing with existential quantifiers. For example, “Tom caught a bird” would be represented in logic by ∃x(bird(x)∧caught(Tom,x)), which would be Skolemised by replacing the x with a Skolem constant; the same thing was done above where bird1 was the name given to the individual bird that Tom caught.

There are still plenty of issues to be resolved if we really want to represent what is meant by the English phrases, or to be really clear about what the semantic net means, but we are getting towards a notation that can be used practically (one example of a thing we have skated over is how to deal with mass nouns like “fur” or “cream” which refer to things that come in amounts rather than individual objects).

A direct Prolog representation can be used, with classes represented by predicates, thus:

cat(tom). cat(cat1). mat(mat1).

sat\_on(cat1,mat1).

bird(bird1). caught(tom,bird1).

like(X,cream) :– cat(X). mammal(X) :– cat(X). has(X,fur) :– mammal(X). animal(X) :– mammal(X). animal(X) :– bird(X).

owns(john,tom). is\_coloured(tom,ginger).

So, in general, an is\_a link between a class c and an individual m is represented by the fact c(m). An a\_kind\_of link between a subclass c and a superclass s is represented by s(X) :- c(X). If a property p with further arguments a1, … ,an is held by all members of a class c, it is represented by p(X,a1,…,an) :- c(X). If a property p with further arguments a1, … ,an is specified as held by an individual m, rather than a class to which m belongs, it is represented by p(m,a1,…,an).

# Inheritance

This Prolog equivalent captures an important property of semantic nets, that they may be used for a form of inference known as inheritance. The idea of this is that if an object belongs to a class (indicated by an is\_a link) it inherits all the properties of that class. So, for example as we have a likes link between cats and cream, meaning “all cats like cream”, we can infer that any object which has an is\_a link to cats will like cream. So both Tom and Cat1 like cream. However, the is\_coloured link is between Tom and ginger, not between cats and ginger, indicating that being ginger is a property of Tom as an individual, and not of all cats. We cannot say that Cat1 is ginger, for example; if we wanted to we would have to put another is\_coloured link between

Cat1 and ginger.

Inheritance also applies across the a\_kind\_of links. For example, any property of mammals or animals will automatically be a property of cats. So we can infer, for example, that Tom has fur, since Tom is a cat, a cat is a kind of mammal, and mammals have fur. If, for example, we had another subclass of mammals, say dogs, and we had, say, Fido is\_a dog, Fido would inherit the property has fur from mammals, but not the property likes cream, which is specific to cats. This situation is shown in the diagram below:



# Reification

An alternative form of representation considers the semantic network directly as a graph. We have already seen ways of representing graphs in Prolog. We could represent each edge in the semantic net graph by a fact whose predicate name is the label on the edge. The nodes in this graph, whether they represent individuals or classes are represented by arguments to the facts representing edges. This gives the following representation for our initial graph:

is\_a(mat1,mats). is\_a(cat1,cats). is\_a(tom,cats). is\_a(bird1,birds). caught(tom,bird1). ako(cats,mammals). ako(mammals,animals). ako(birds,animals). like(cats,cream). owns(john,tom). sat\_on(cat1,mat1).

is\_coloured(tom,ginger).

have(mammals,fur).

Alternatively, the graph could be built using the cells or pointers of an imperative language. There are also special purpose knowledge representation languages which provide a notation which translates directly to this sort of graph.

This process of turning a predicate into an object in a knowledge representation system is known as reification. So, for example, the constant symbol cats represents the set of all cats, which we can treat as just another object.

# The Case for Case

We have shown how binary relationships may be represented by arcs in graphs, but what about relationships with more than two arguments? For example, what about representing the sentence

“John gave the book to Mary”? In predicate logic, we could have a 3-ary predicate gave, whose first argument is the giver, second argument the object given and third argument the person to whom it was given, thus gave(John,Book1,Mary). The way this can be resolved is to consider the act of giving a separate object (remember how in the first set of notes we saw how the pronoun “it” could in some contexts be taken to refer to a previously mentioned action itself rather than to an object involved in the action), thus it is further reification. We can than say that any particular act of giving has three participants: the donor, the recipient, and the gift, so the semantic net representing the sentence is:



In fact the three different roles correspond to what is known in natural language grammar as subjective (the object doing the action, in this case John), objective (the object to which the action is being done, in this case the book) and dative (the recipient of the action, in this case Mary). These different roles of objects in a sentence are known as cases.

The fact that various natural languages make this case distinction can be used to support using it in artificial knowledge representation. The “case for case” is associated with the linguist Charles Fillmore whose work has been influential among AI workers in knowledge representation. The idea is that all sentences can be analysed as an action plus a number of objects filling the roles in the action, with there being a fixed set of roles (though not every role will always be filled). Other roles suggested as fundamental include the locative indicating where the action is done, and the instrumental, indicating the means by which an action is done.

In some natural languages the different roles which a word may fill are indicated by the ending or inflexion of the word. A well-known example of such an inflexional language is Latin (but some modern languages, such as Russian are equally as inflexional), where, for example “Dog bites man” is “Canis hominem mordet” while “Man bites dog” is “Homo canem mordet”. The word for “dog” is “canis” if it is the object of the sentence, but “canem” if it is the subject, while for “man” it is “homo” if he is the object of the sentence and “hominem” if he is the subject. If something were being given to a dog, the word used would be “cane”, if a dog were being used for something the word used would be “cani”. In English the objective and subjective roles are indicated by word order, with the object coming before the verb and the object coming after. In Latin, it is the case endings, not the word order that indicates a role, so “Hominem canis mordet” is just another way of saying “Dog bites man”. You could perhaps compare it the programming languages where the relationship of arguments to formal parameter names in procedure calls may be indicated by their position, but in some cases (e.g. Modula-3) a facility is available for named arguments.

In English the dative is occasionally indicated by word order (for example in “John gave Mary the book”), but more often by prefixing the word indicating the dative item with the preposition “to”, as in “John gave the book to Mary”. Other cases are always indicated by prepositions, for example the locative with “at” (e.g. “John gave the book to Mary at school”) and the instrumental with “by” or “with” (“John sent the book to Mary by post”, “Mary hit John with the book”). Most inflexional languages have a limited range of cases, and use prepositions to extend the range. In fact the argument for case as innate is damaged by the fact that different languages have different case structures, and it is by no means certain which cases are fundamental and which are just variants of others. For example, in sentences involving the concept of movement linguists distinguish the ablative case (the source of the movement, in English indicated by the preposition “from”) and the allative case (the destination of the movement), but should the latter be considered just another form of the dative role?

Using the concept of a semantic network in which nodes represent individual actions, with arcs representing objects having roles in these actions, it is possible to build up complex graphs representing complete scenarios. For example, the story:

“John gave Mary a book. The book was not the one Mary likes, so she punched John. That made her feel sorry for him, so she then kissed him”

is represented by the graph on the next page. The class nodes are omitted as the graph is complex enough without them. The arcs are labelled with sub and obj, for the subject and object of the action, ind.obj and instr for the case where there is an indirect object (i.e. dative in the terminology used previously) and an instrument. There are also arcs representing time relationships – note that individual times are represented by nodes as well, and reasons why an act was performed.

Note that in the graph some English words are translated to an equivalent, thus “punch” is represented as “hit with fist” (we might also, for example, have represented “kiss” by “touch with lips”, though this perhaps illustrates why this sort of attempt to find an underlying representation can miss some of the subtleties of human language!). Similarly, if we are trying to represent underlying meanings, we have not only to convert passive forms to active forms as suggested previously, but also to note forms where one verb is equivalent to another, except with the roles in a different order. For example, the sentence “X buys Y from Z” is essentially equivalent to “Z sells Y to X”, so we could therefore convert all sentences involving selling to the equivalent involving buying and make them instances of the buying class. Work on trying to find underlying primitives to aid network representation of the meaning of natural language semantics is associated with the AI researcher Roger Schank.

facts we gave as the first representation of the previous graph. The advantage of the graph notation is that it may be more intuitive, and in particular it brings together all the information associated with a particular individual. Drawing inferences from a semantic net involves searching for particular patterns. For example, the question “Who kissed John?” from the above graph involves searching for a node which links to the class node kissings with an is\_a link (this is one of the links not shown), and has an object link to the node representing John. The answer to the question is then found from the subject link of that node. In Prolog this would be the query:



is\_a(K,kissings), object(K,john), subject(K,Answer).

Note that the graph may represent a scenario where John is kissed more than once, in which case there would be more than one node fitting the conditions, and the query could be made to backtrack to give alternate answers.

A “whom” question is a search for the object of a node given the subject, thus “Mary kissed whom?” (modern English is more likely to phrase this “Who did Mary kiss?”, the distinction between “who” as a query for a subject and “whom” as a query for an object being lost) is represented by:

is\_a(K,kissings), subject(K,mary), object(K,Answer).

Similarly a “to whom” question is a search for an indirect object given a subject and object, so “John gave the book to whom?” or “Who did John give the book to?” is represented by:

is\_a(G,givings), subject(G,john), object(G,B), is\_a(B,book),

indirect\_object(B,Answer).

A “how” question might be considered equivalent to a “with what” question, so it is returns the instrumental link of the relevant node. A “where” question returns the locative link.

A “why” question is a search for a reason link, so “Why did Mary kiss John?” is represented by:

is\_a(K,kissings), subject(K,mary), object(K,john), reason(K,Answer)

In this case, however, the answer will not be an individual but simply a name assigned to an node representing a feeling\_sorry\_for action. A more correct report would need to give the complete sentence represented by the node to which the reason link points.

Similarly, a “when” question is a search for a time link. Time links may point to nodes actually storing times and dates. However, as in our example, it is more likely to be a time which is relative to another, so again the answer given must involve looking beyond just the node pointed to by the time link. For example, with our above graph the question “When did Mary feel sorry for John” would be answered by finding that the time link from the node sorry1 links to time time3. It can then be noted that time3 is the subject of one after node, and the object of another, so the answer could be given as both “After Mary hit John” and “Before Mary kissed John”. If two different action realistic treatment of time would deal with time intervals which have a start and finish time.



# Frames, Slots and Fillers

Consideration of the use of cases suggests how we can tighten up on the semantic net notation to give something which is more consistent, known as the frame notation. In the place of an arbitary number of arcs leading from a node there are a fixed number of slots representing attributes of an object. Every object is a member or instance of a class, which it may be thought of as linking to with an is\_a link as we saw before. The class indicates the number of slots that an object has, and the name of each slot. In the case of a giving object, for instance, the class of giving objects will indicate that it has at least three slots: the donor, the recipient and the gift. There may be further slots indicated as necessary in the class, such as ones to give the time and location of the action. The time slot may be considered a formalisation of the tense of the verb in a sentence.

The idea of inheritance is used, with some slots being filled at class level, and some at instance level. Where a slot is filled at class level the idea is that this represents attributes which are common to all members of that class. Where it is filled at instance level, it indicates that the value of that attribute varies among members of that class. Slots may be filled with values or with pointers to other objects. This is best illustrated by an example.

In our example we have a general class of birds, and all birds have attributes flying, feathered and colour. The attributes flying and feathered are boolean values and are fixed to true at this level, which means that for all birds the attribute flying is true and the attribute feathered is true. The attribute colour, though defined at this level is not filled, which means that though all birds have a colour, their colour varies. Two subclasses of birds, pet\_canaries and ravens are defined. Both have the colour slot filled in, pet\_canaries with yellow, ravens with black. The class pet\_canaries has an additional slot, owner, meaning that all pet canaries have an owner, though it is not filled at this level since it is obviously not the case that all pet canaries have the same owner. We can therefore say that any instance of the class

pet\_canary has attributes colour yellow, feathered true, flying true, and owner, the last of these varying among instances. Any instance of class raven has colour black, feathered true, flying true, but no attribute owner. The two instances of pet\_canary shown, Tweety and Cheepy have owners John and Mary who are separate instances of the class person, for simplicity no attributes have been given for class person. The instance of pet\_canary Cheepy has an attribute which is restricted to itself, vet (since not all pet canaries have their own vet), which is



We can define a general set of rules for making inferences on this sort of frame system. We can say that an object is an instance of a class if it is a member of that class, or if it is a member of a class which is a subclass of that class. A class is a subclass of another class if it is a kind of that class, or if it is a kind of some other class which is a subclass of that class. An object has a particular attribute if it has that attribute itself, or if it is an instance of a class that has that attribute. In Prolog:

aninstance(Obj,Class) :– is\_a(Obj,Class).

aninstance(Obj,Class) :– is\_a(Obj,Class1), subclass(Class1,Class).

subclass(Class1,Class2) :– a\_kind\_of(Class1,Class2). subclass(Class1,Class2) :– a\_kind\_of(Class1,Class3), subclass(Class3,Class2).

We can then say that an object has a property with a particular value if the object itself has an attribute slot with that value, or it is an instance of a class which has an attribute slot with that value, in Prolog:

value(Obj,Property,Value) :– attribute(Obj,Property,Value).

value(Obj,Property,Value):– aninstance(Obj,Class), attribute(Class,Property,Value).

The diagram above is represented by the Prolog facts:

attribute(birds,flying,true). attribute(birds,feathered,true).

attribute(pet\_canaries,colour,yellow). attribute(ravens,colour,black). attribute(tweety,owner,john). attribute(cheepy,owner,mary). attribute(cheepy,vet,sally). a\_kind\_of(pet\_canaries,birds). a\_kind\_of(ravens,birds). a\_kind\_of(vet,person). is\_a(edgar,ravens). is\_a(tweety,pet\_canaries). is\_a(cheepy,pet\_canaries). is\_a(sally,vet). is\_a(john,person). is\_a(mary,person).

Note in particular how we have used reification leading to a representation of classes like birds, pet\_canaries and so on by object constants, rather than by predicates as would be the case if we represented this situation in straightforward predicate logic. The term superclass may also be used, with X being a superclass of Y whenever Y is a subclass of X.

Using the Prolog representation, we can ask various queries about the situation represented by the frame system, for example if we made the Prolog query:

| ?- value(tweety,colour,V). we would get the response:

V = yellow ?

while

| ?- value(john,feathered,V). gives the response

no

indicating that feathered is not an attribute of John. Note that the no indicates that this is something which is not recorded in the system. If we wanted to actually store the information that persons are not feathered we would have to add:

attribute(person,feathered,true). then the response would have been:

V = false ?

The only thing that has not been captured in this Prolog representation is the way that an attribute can be defined at one level and filled in lower down, like the colour attribute of birds.

# Demons and Object-Oriented Programming

Some frame systems have an additional facility in which a slot may be filled not by a fixed attribute but by a procedure for calculating the value of some attribute. This procedure is known as a demon (the name coming from the idea that it “lurks around waiting to be invoked”). A demon may be attached to a class, but make use of information stored in a subclass or an instance.

For instance, in the above example, we might want to have an attribute maintenance representing maintenance costs attached to the subclass pet\_canaries, which should return £5 for a pet canary without its own vet, but £5+vet’s fees for a canary with a vet. However, if we do this we will need to have a way to refer to the individual instance of a class at the class level. We do this through the use of a variable conventionally called Self. We then need to add the reference to Self to our rules for determining the value of some property:

value(Obj,Property,Value) :– attribute(Obj,Obj,Property,Value).

value(Obj,Property,Value):–

aninstance(Obj,Class), attribute(Obj,Class,Property,Value).

The first argument to attribute here is the reference to Self. Our previous attributes do not depend on the value of Self, so we can just add it as an anonymous variable:

attribute(\_,birds,flying,true). attribute(\_,pet\_canaries,colour,yellow).

and so on for the other attributes. For our example, we must have the attribute fees attached to vets (it will vary from vet to vet so it will be filled in at instance level), so we will also add to our example: attribute(\_,sally,fees,20).

Now, to add our demon, which we will name eval\_maintenance, we add:

attribute(Self,pet\_canaries,maintenance,Costs) :– eval\_maintenance(Self,Costs).

eval\_maintenance(Self,Costs) :– value(Self,vet,SelfsVet), !, value(SelfsVet,fees,VetFees), Costs is VetFees+5.

eval\_maintenance(Self,5).

The use of the cut here is because the only way we can find out if a pet canary doesn’t have a vet is to see if fails, but we don’t want backtracking for a pet canary that does have a vet to give an alternative value for maintenance costs.

The introduction of demons brings our knowledge representation method close to that of objectoriented programming. Several object-oriented programming language have been developed which give mechanisms directly for expressing classes with attached procedures and inheritance. The most successful examples are C++ and Smalltalk. Development of the idea of demons into full procedures which may change the values stored with an object moves away from the declarative ideas of knowledge representation, so we shall not develop it further here, but those taking the course in Object-Oriented Programming will be able to build the connection.

Defaults and Overrides

One of the problems we mentioned with predicate logic is that it does not provide us with a way of saying that some particular conclusion may be drawn unless we can show otherwise. We had to add the idea of negation as failure to deal with this, and even then if we want to draw a conclusion we have to show that all the conditions that would cause that conclusion to fail are false. For example, we know that in general birds can fly. So we can write in Prolog:

flies(X) :– bird(X).

But suppose we want to deal with special cases of birds that cannot fly. We know that kiwis and penguins cannot fly, for instance. We also know that any bird with a broken wing cannot fly. So strictly we would have to say:

flies(X) :– bird(X), \+kiwi(X), \+penguin(X), \+broken\_wing(X).

We can summarise this as: flies(X) :– bird(X), \+ab(X).

where ab(X) means “X is an abnormal bird”. We could list the factors that make X an abnormal bird in respect to flying:

ab(X) :– kiwi(X). ab(X) :– penguin(X). ab(X) :– broken\_wing(X). but there might always be circumstances we had not thought of (other species of birds that don’t fly, birds whose wings are not broken but whose feet are trapped, etc.). As we mentioned in a previous set of notes forms of default logic exist which enable us to say that some conclusion holds on the assumption that there are no facts known to indicate why they should not. So we might say that bird(x) is true with assumption set {¬ab(x)}. This is non-monotonic reasoning, since the addition of a fact which makes some assumption false will make a conclusion false. For example, i f

we have ostrich(ossie) and bird(X):–ostrich(X) we can assume flies(ossie), but if we add ab(X):–ostrich(X), this reasoning fails. In practice there will have to be a separate form of ab for every rule.

Another way of putting this is to say that the default is for any bird x, flies(x) is true.

Default reasoning is easily added to the frame system of representation. The idea used is that an attribute at class level is inherited only if it is not cancelled out or overridden by the same attribute slot occurring in a subclass of that class or in an individual instance with a different value. For example, we could add the class of kiwis as a subclass of birds in our diagram above, and indicate that kiwis cannot fly. The additional attributes to create a class of kiwis with one instance kevin are:

a\_kind\_of(kiwis,birds). attribute(kiwis,flying,false). attribute(kiwis,colour,brown). is\_a(kevin,kiwis).

We have to add a colour attribute for kiwis as this was a slot in its superclass, birds. For simplicity we have gone back to the representation which does not allow for the possibility of demons.

The following arcs are added to our diagram:



Now it will be seen that for X=tweety, cheepy or edgar, | ?- value(X,flying,V).

will give the response butV = true ?

| ?- value(kevin,flying,V). will give the response

V = false ?

One problem is that if we typed ; in response to this we would get:

V = true ?

In order to prevent this possibility we need to put cuts in our inference rules, so that when the property is found it is not possible to backtrack and search higher in the inheritance tree for a value for the same property:

value(Obj,Property,Value) :– attribute(Obj,Property,Value), !.

value(Obj,Property,Value):– aninstance(Obj,Class), attribute(Class,Property,Value), !.

The presence of the cut indicates that we have lost the strict declarative reading, and the result we get will depend on the ordering of the rules. This will become more apparent when we consider multiple inheritance next.

The result of adding the possibility of overrides is that the information stored at class level no longer represents attributes held by all members of that class, but can be taken as being the attributes held by the typical member of that class. Sometimes the class level node in the inheritance tree is said to represent the prototype member of that class. All new instances of that class are constructed by taking the prototype and altering the defaults as required.

In order to establish coherency, sometimes a distinction is made between defining attributes which cannot be overridden, and default attributes which can. Any attempt to add a node to the inheritance graph which overrode a defining attribute would be flagged as an error. Without this feature it would, for example, be possible to define a subclass in which all the attributes of a superclass are overridden.