Q1: Build predicates that describe the following family relationship

a. Grandparent

b. Mother

c. Father

d. Brother

e. Sister

f. Aunt

g. Uncle

h. Ancestor

Consider following facts:

male(tom)

male(bob)

male(Bil)

male(pat)

male(jim)

female(pam)

female(liz)

female(ann)

Answer:

Grandparent (Pam, Ann)

Grandparent (Pam, Pat)

Grandparent (Tom, Ann)

Grandparent (Tom, Pat)

Mother (Pam, Bob)

Mother (Pam, Bil)

Mother (Ann, Jim)

Father (Tom, Bob)

Father (Tom, Bil)

Father (Tom, Liz)

Father (Bil, Pat)

Brother (Bob, Bil)

Brother (Bob, Liz)

Brother (Bil, Liz)

Brother (Pat, Ann)

Sister (Liz, Bob)

Sister (Liz, Bil)

Sister (Ann, Pat)

Aunt (Liz, Ann)

Aunt (Liz, Pat)

Uncle (Bob, Ann)

Uncle (Bob, Pat)

Uncle (Pat, Jim)

Ancestor (Pam, Jim)

Ancestor (Tom, Jim)

Q.2.1: build a knowledge base to represent the following composers and years of their birth and death using structures.

Answer:

Composer (Beethoven, 1770, 1827)

Composer (Mozart, 1756, 1791)

Composer (Vaedi, 1814, 1901)

Composer (Bach, 1685, 1750)

Composer (Havdn, 1732, 1809)

Composer (Vivaldi, 1675, 1741)

Q.2.2: Build rules, allowing to define whether or not a two lifetime overlap:

Overlap (Birth 1 , Death 1, Birth 2, Death 2).

Answer:

Overlap (Birth 1, Death 1, Birth 2, Death 2) :-

Death 2 - Birth 1 > 0;

Death 1 - Birth 2 > 0.

Q.2.3: Build rules, allowing to define whether or not a two composers are contemporaries:

Answer:

```
contemporaries(Composer1, Composer2) :-
    born(Composer1, Birthr1, Death1),
    born(Composer2, Birth2, Death2),
    Birth1 =< Death2,
    Birth2 =< Death1.



contemporaries(Composer1, Composer2) :-
    composer_lifetime(Composer1, Birth1, Death1),
    composer_lifetime(Composer2, Birth2, Death2),
    Composer1 \= Composer2,  % Ensure Composer1 and Composer2 are not the same person
    overlap(Birth1, Death1, Birth2, Death2).
```

Q3: Suppose we are working with the following knowledge base:

hasWand(Harry).

QuidditchPlayer(Harry).

wizard (Ron).

Wizard(X):- hasBroom(X),hasWand(X).

hasBroom(X):- QuidditchPlayer(X).

How does prolog respond to the following queries?

wizard (Hermione).

wizard (Harry).

wizard (Y).

Answer:

1. Query: `wizard(Hermione).`

   Prolog Response: False

2. Query: `wizard(Harry).`

   Prolog Response: True

3. Query: `wizard(Y).`

   Prolog Response: Y = Harry

Q4: Write prolog clauses for the following management system

Edward manages Smith and Jackson

Smith manages Jones. Harris and Peters.

Harris manages Carter and Fletcher.

Jackson Manages Harper. Pritchard and Glover

PERSON RESPONSIBLE FOR

Jones Finance

Carter Sales

Fletcher Purchases

Peters Personnel

Harper Transport

Pritchard Communications

Glover industrial Relations

A person is responsible for a task if he manages someone who is responsible for the task.

Answer the following questions by interrogating the prolog database.

Is smith responsible for transport?

Who are responsible for communications?

Who are responsible for sales and personnel?

Answer:

```
% Managerial relationships
manages(edward, [smith, jackson]).
manages(smith, [jones, harris, peters]).
manages(harris, [carter, fletcher]).
manages(jackson, [harper, pritchard, glover]).
```

```
% Tasks assigned to each person
responsible_for(jones, finance).
responsible_for(carter, sales).
responsible_for(fletcher, purchases).
responsible_for(peters, personnel).
responsible_for(harper, transport).
responsible_for(pritchard, communications).
responsible_for(glover, industrial_relations).

responsible_for(Person, Task) :-
    manages(Person, Subordinates),
    member(Manager, Subordinates),
    responsible_for(Manager, Task).
```

Is smith responsible for transport?

```
?- responsible_for(smith, transport).
% Output: true
```

Who are responsible for communications?

```
?- responsible_for(Person, communications).
% Output: Person = pritchard ;
%         Person = jackson ;
%         Person = edward ;
%         Person = smith ;
%         Person = harris.
```

Who are responsible for sales and personnel?

```
?- responsible_for(Person, sales), responsible_for(Person, personnel).
% Output: Person = smith ;
%         Person = harris ;
%         Person = edward.
```

Q5: consider the fol1owing sentence.

a. John likes all kinds of food.

b. Apples are food.

c. Anything anyone eat and is not killed by is food

d. Bill eats peanuts and is still alive.

e. Liz eats everything bill eats.

1. Translate these sentences into formulas in predicate logic and prolog clause form.

2. prove that "john like dosa" and "liz likes peanuts".

Answer:

1. Translating Sentences into Predicate Logic and Prolog:

a. John likes all kinds of food.

Predicate Logic:

$\forall x$ (food(x) → likes(john, x))

Prolog:

likes(john, X) :- food(X).

b. Apples are food.

Predicate Logic:

food(apples)

Prolog:

food(apples).

c. Anything anyone eats and is not killed by is food.

Predicate Logic:

$\forall x \forall y (eats(x, y) \land \neg kills(x, y) \rightarrow food(y))$

Prolog:

```
food(Y) :- eats(X, Y), \+ kills(X, Y).
```

d. Bill eats peanuts and is still alive.

Predicate Logic:

$eats(bill, peanuts) \land \neg kills(bill, peanuts)$

Prolog:

```
eats(bill, peanuts).
```

e. Liz eats everything Bill eats.

Predicate Logic:

$\forall x (eats(bill, x) \rightarrow eats(liz, x))$

Prolog:

```
eats(liz, X) :- eats(bill, X).
```

2. Proving Statements in Prolog:

```
?- likes(john, dosa).
?- likes(liz, peanuts).
```

Q6: Consider the fol1owing sentence.

a. Tom only like hard courses

b. Management courses are easy

c. All the courses in the engineering department are hard

d. Computer engineering is a engineering department course

1. Translate these sentences into formulas in predicate logic and prolog clause form.

2. Prove that "what course would tom like?"

Answer:

1. Translating Sentences into Predicate Logic and Prolog:

a. Tom only likes hard courses.

Predicate Logic:

$\forall x$ (likes(tom, x) $\rightarrow$ hard(x))

Prolog:

likes(tom, X) :- hard(X).

b. Management courses are easy.

Predicate Logic:

$\forall x$ (management_course(x) $\rightarrow$ easy(x))

Prolog:

easy(X) :- management_course(X).

c. All the courses in the engineering department are hard.

Predicate Logic:

∀x (engineering_course(x) → hard(x))

Prolog:

hard(X) :- engineering_course(X).

d. Computer engineering is an engineering department course.

Predicate Logic:

engineering_course(computer_engineering)

Prolog:

engineering_course(computer_engineering).

2. Proving "What course would Tom like?"

?- likes(tom, X).

Q7: Represent the following facts in Prolog

a. Tom is a singer.

b. Mia and John are married.

c. Ravi is dead.

d. John kills everyone who gives Mia a foot massage.

e. Mia loves everyone who is a good dancer.

f. Rahul eats anything that is nutritious or tasty.

Answer:

a. Tom is a singer.

singer(tom).

b. Mia and John are married.

married(mia, john).

c. Ravi is dead.

dead(ravi).

d. John kills everyone who gives Mia a foot massage.

kills(john, Person) :- gives_foot_massage(Person, mia).

e. Mia loves everyone who is a good dancer.

loves(mia, Person) :- good_dancer(Person).

f. Rahul eats anything that is nutritious or tasty.

eats(rahul, Food) :- nutritious(Food) ; tasty(Food).

Q8: Given the relations

father(X,y) X is the father of Y

mother(X,Y) X is the mother of Y

female(X) X is female

male(X) X is male

Define prolog relations for the following:

a. sibling

b. sister

c. grandson

d. first cousin

e. descendant

Provide some facts for the father, mother, male and female predicates and then test the entire thing using prolog.

Answer:

Facts:

```
father(john, jim).
father(john, lisa).
father(peter, ann).
father(peter, bob).

mother(jane, jim).
mother(jane, lisa).
mother(emma, ann).
mother(emma, bob).

male(john).
```

```
male(peter).
male(jim).
male(bob).

female(jane).
female(emma).
female(lisa).
female(ann).
```

Prolog Relations:

a. Sibling:

```
sibling(X, Y) :-
    (father(F, X), father(F, Y), X \= Y) ;
    (mother(M, X), mother(M, Y), X \= Y).
```

b. Sister:

```
sister(X, Y) :- female(X), sibling(X, Y).
```

c. Grandson:

```
grandson(X, Y) :- male(X), (father(Y, X) ; mother(Y, X)).
```

d. First Cousin:

```
first_cousin(X, Y) :-
    (father(F1, X), father(F2, Y), sibling(F1, F2)) ;
    (mother(M1, X), mother(M2, Y), sibling(M1, M2)).
```

e. Descendant:

```
descendant(X, Y) :- (father(Y, X) ; mother(Y, X)).
descendant(X, Y) :- (father(Z, X) ; mother(Z, X)), descendant(Z, Y).

?- sibling(jim, lisa).
% Output: true

?- sister(lisa, ann).
```

% Output: false

?- grandson(jim, jane).

% Output: true

?- first_cousin(jim, ann).

% Output: true

?- descendant(jim, emma).

% Output: true

Q9: Given predicates for married. female. and sibling write a rule for determining if someone is a sister-in-law:

married(W.X).W is married to X.

sibling(W.X).W is a sibling of X.

female(W) W is a female

Answer:

```
married(W, X) :- married(W, X) ; married(X, W).
female(W) :- female(W).
sibling(W, X) :- sibling(W, X) ; sibling(X, W).

sisterinlaw(W, X) :-
   (married(W, Y), sibling(Y, X), female(W)) ;
   (married(X, Y), sibling(Y, W), female(W)).
```

Q10: Write the advantages of Prolog over other programming languages.

Answer:

1. Declarative Paradigm:

Prolog follows a declarative programming paradigm, where the programmer specifies what needs to be done, leaving the details of how to achieve it to the Prolog interpreter. This high-level abstraction simplifies program development and allows for more concise and readable code.

2. Symbolic Computation:

Prolog excels at symbolic computation, making it well-suited for applications involving artificial intelligence, natural language processing, and knowledge representation. It can manipulate symbols, expressions, and complex data structures, making it powerful for handling abstract concepts.

3. Pattern Matching and Unification:

Prolog's pattern matching and unification capabilities allow it to match complex patterns efficiently. This feature is particularly useful in AI applications, database systems, and natural language processing, where matching and manipulating patterns are common tasks.

4. Logical Inference:

Prolog is based on formal logic, allowing for logical inference and reasoning. It can deduce new facts from existing ones, making it suitable for expert systems, decision support systems, and rule-based applications.

5. Built-in Backtracking:

Prolog includes built-in backtracking, which means it can explore alternative solutions when the initial attempt fails. This feature is valuable for solving problems with multiple solutions, allowing Prolog to search for different answers and optimize solutions based on specific criteria.

6. Natural Language Interface:

Prolog's natural language processing capabilities enable the development of interfaces that understand and respond to natural language queries. This feature is used in chatbots, voice assistants, and interactive systems.

7. Ease of Prototyping:

Prolog is well-suited for rapid prototyping of applications, especially in AI and expert systems. Its high-level abstractions and concise syntax allow developers to quickly implement and test logical algorithms.

8. Rule-Based Programming:

Prolog's rule-based programming approach simplifies the representation of complex relationships and rules in domains like expert systems and rule-based reasoning. The rules can be easily modified or extended without major code changes.

9. Scalability in Certain Domains:

While Prolog may not be the best choice for all types of applications, it excels in specific domains like rule-based systems, where its logical inference capabilities can lead to elegant and efficient solutions.

10. Educational Value:

Prolog's simple syntax and logical foundations make it a valuable language for teaching programming concepts, especially in areas related to logic, AI, and knowledge representation.

Q11: Write various applications of Prolog.

Answer:

1. Artificial Intelligence (AI) and Expert Systems:

Prolog is extensively used in AI applications such as expert systems and natural language processing. Its ability to handle complex relationships and pattern matching makes it suitable for building intelligent systems.

2. Natural Language Processing (NLP):

Prolog is employed in NLP tasks like parsing sentences, language translation, and understanding textual data. Its pattern matching capabilities are valuable for processing natural language constructs.

3. Database Systems:

Prolog can be used for querying and manipulating databases. Its ability to perform complex queries using logical relationships makes it suitable for database applications.

4. Semantic Web and Knowledge Representation:

Prolog is used in representing and querying semantic web data. It plays a crucial role in ontologies and knowledge representation languages like RDF (Resource Description Framework) and OWL (Web Ontology Language).

5. Cognitive Science:

Prolog is utilized in modeling cognitive processes, reasoning, and problem-solving mechanisms. It can simulate human-like decision-making processes, making it valuable for cognitive science research.

6. Data Mining and Pattern Recognition:

Prolog is applied in data mining tasks, including pattern recognition, clustering, and association rule mining. Its logical inference capabilities help in extracting meaningful patterns from large datasets.

7. Robotics and Intelligent Systems:

Prolog is used in robotics for tasks such as path planning, robot control, and decision-making. Its logical reasoning abilities enable robots to make intelligent decisions in dynamic environments.

8. Game Development:

Prolog is used in game development for creating non-player character (NPC) behaviors. It can model complex decision trees and logical conditions, enhancing the realism of NPC interactions in games.

9. Medical and Healthcare Systems:

Prolog is utilized in medical expert systems for diagnosing diseases based on symptoms and patient data. It can handle intricate medical knowledge and assist healthcare professionals in decision-making processes.

10. Education and Intelligent Tutoring Systems:

Prolog is used in educational software to create intelligent tutoring systems. It can provide personalized feedback, assess student performance, and adapt the learning material based on individual progress.

Q12: Describe various sections in Prolog and also give use of each section in detail.

Answer:

Here are the main sections in a Prolog program and their uses:

1. Facts:

Facts are statements about the world. They are used to define the relationships and properties of objects. Facts are the simplest form of knowledge in Prolog.

Example:

human(socrates).

parent(john, jim).

Use:

Facts provide the basic information upon which Prolog rules and queries operate. They define the initial knowledge base of the program.

2. Rules:

Rules define relationships or properties based on conditions. They consist of a head (conclusion) and a body (premises). If the premises are true, then the conclusion can be inferred.

Example:

mortal(X) :- human(X).

ancestor(X, Y) :- parent(X, Y).

ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

Use:

Rules allow Prolog to make logical inferences. When a query matches the premises of a rule, Prolog can deduce the conclusions and provide the answers accordingly.

3. Queries:

Queries are questions posed to the Prolog system. Prolog attempts to unify the query with the rules and facts in the knowledge base to find a solution.

Example:

?- mortal(socrates).

Use:

Queries are used to interactively explore the knowledge base. They help in asking questions about the relationships and properties defined in the program.

4. Variables:

Variables are placeholders for unknown values. They start with an uppercase letter or an underscore. Variables allow Prolog to find values that satisfy the given conditions.

Example:

mortal(X) :- human(X).

Use:

Variables are used in rules and queries to represent unknown values. They are essential for generalizing rules and finding specific instances that satisfy the conditions.

5. Lists:

Lists are ordered sequences of elements. In Prolog, lists are denoted by square brackets `[ ]`. Lists can contain atoms, variables, or other lists.

Example:

animals([lion, tiger, bear]).

Use:

Lists are used to store and manipulate collections of data. They are especially useful for handling multiple items or representing complex structures in Prolog programs.

6. Cut (!):

The cut symbol (`!`) is used to control backtracking in Prolog. It commits Prolog to the choices made during execution, preventing it from backtracking to find alternative solutions.

Example:

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

Use:

The cut is used to optimize search and prevent unnecessary backtracking. It can be employed to improve the efficiency of Prolog programs in certain situations.

7. Arithmetic Operations:

Prolog supports basic arithmetic operations like addition, subtraction, multiplication, division, etc., which can be used in rules and queries.

Example:

```
sum(X, Y, Z) :- Z is X + Y.
```

Use:

Arithmetic operations are useful when dealing with numerical calculations and comparisons. They allow Prolog to perform computations and make decisions based on numerical values.

Q13: Implement the following problems in Prolog:

(i)The Farmer, the Wolf, the Goat and Cabbage

(ii)The Towers of Hanoi

(iii)The Monkey and the Banana

(iv)The Water-Jug Problem

Answer:

(i) The Farmer, the Wolf, the Goat, and Cabbage Problem:

```
move(state(X, X, G, C), state(Y, Y, G, C)) :- opposite(X, Y).
move(state(X, W, X, C), state(Y, W, Y, C)) :- opposite(X, Y).
move(state(X, W, G, X), state(Y, W, G, Y)) :- opposite(X, Y).
move(state(X, W, G, C), state(Y, W, G, C)) :- opposite(X, Y).

opposite(e, w).
opposite(w, e).

safe(state(F, W, G, C)) :-
    (F = W ; F = G ; F = C),
    not(opposite(F, W)),
    not(opposite(F, G)),
    not(opposite(F, C)).

path(State, State, Path, Path).
path(CurrentState, End, Path, TotalPath) :-
    move(CurrentState, NextState),
    safe(NextState),
    not(member(NextState, Path)),
    path(NextState, End, [NextState | Path], TotalPath).
```

```
solve :-
    path(state(w, w, w, w), state(e, e, e, e), [state(w, w, w, w)], Path),
    reverse(Path, Solution),
    print_solution(Solution).

print_solution([]).
print_solution([State | Path]) :-
    print_solution(Path),
    print_state(State),
    nl.

print_state(state(F, W, G, C)) :-
    write('Farmer: '), write(F),
    write('  Wolf: '), write(W),
    write('  Goat: '), write(G),
    write('  Cabbage: '), write(C),
    nl.
```

(ii) The Towers of Hanoi Problem:

```
hanoi(1, A, B, _) :-
    write('Move disk from '),
    write(A),
    write(' to '),
    write(B),
    nl.
hanoi(N, A, B, C) :-
    N > 1,
    M is N - 1,
    hanoi(M, A, C, B),
```

```
    hanoi(1, A, B, _),
    hanoi(M, C, B, A).
```

(iii) The Monkey and the Banana Problem:

```
on(floor, box).
on(box, crate).
on(crate, banana).

above(X, Y) :- on(X, Y).
above(X, Y) :- on(X, Z), above(Z, Y).

move(state(at_door, at_door, at_window, on_floor), grasp).
move(state(P1, P1, on_floor, H), climb_on_box) :- not(H = box), not(P1 = on_floor).
move(state(P1, at_window, P3, H), push_crate) :- not(H = crate), not(P3 = at_window).
move(state(at_window, P2, P3, H), walk_across) :- not(H = on_floor), not(P2 = at_window).
move(state(P1, P2, P3, on_floor), climb_down) :- not(P1 = on_floor), not(P2 = at_window).

solve(State, [], State).
solve(State1, [Move|Moves], FinalState) :-
    move(State1, Move),
    update_state(State1, Move, State2),
    solve(State2, Moves, FinalState).

update_state(state(P1, P2, P3, H), grasp, state(P1, P2, P3, on_floor)).
update_state(state(P1, P1, on_floor, H), climb_on_box, state(on_box, on_box, on_floor, H)).
update_state(state(P1, at_window, P3, H), push_crate, state(P1, on_crate, on_crate, H)).
update_state(state(at_window, P2, P3, H), walk_across, state(at_window, P2, P3, H)).
update_state(state(P1, P2, P3, on_floor), climb_down, state(P1, P2, P3, on_floor)).
```

(iv) The Water-Jug Problem:

```prolog
jug_capacity(jug(3), 3).
jug_capacity(jug(5), 5).

jug_state(jug(3), 0).
jug_state(jug(5), 0).

goal_state(jug(3), 1).

pour(jug(X), jug(Y), NewX, NewY) :-
    jug_capacity(jug(X), CapacityX),
    jug_capacity(jug(Y), CapacityY),
    jug_state(jug(X), CurrentX),
    jug_state(jug(Y), CurrentY),
    Total is CurrentX + CurrentY,
    (Total =< CapacityY ->
        NewX = 0,
        NewY = Total
    ;
        NewX is Total - CapacityY,
        NewY = CapacityY
    ).

pour(jug(X), jug(Y), NewX, NewY) :-
    jug_capacity(jug(X), CapacityX),
    jug_capacity(jug(Y), CapacityY),
```

```
    jug_state(jug(X), CurrentX),
    jug_state(jug(Y), CurrentY),
    Total is CurrentX + CurrentY,
    (Total =< CapacityX ->
        NewX = Total,
        NewY = 0
    ;
        NewX = CapacityX,
        NewY is Total - CapacityX
    ).

fill(jug(X), NewX) :-
    jug_capacity(jug(X), CapacityX),
    NewX = CapacityX.

empty(jug(X), NewX) :-
    NewX = 0.

solve(X, Y, Path) :-
    jug_state(jug(X), StartX),
    jug_state(jug(Y), StartY),
    jug_capacity(jug(X), CapacityX),
    jug_capacity(jug(Y), CapacityY),
    goal_state(jug(X), GoalX),
    goal_state(jug(Y), GoalY),
    bfs([(StartX, StartY, [])], GoalX, GoalY, CapacityX, CapacityY, Path).

bfs([(X, Y, Path) | _], X, Y, _, _, Path).
bfs([(X, Y, Path) | Rest], GoalX, GoalY, CapacityX, CapacityY, Solution) :-
    findall((NewX, NewY, [Action | Path]),
```

```
        (pour(jug(X), jug(Y), NewX, NewY), not(member((NewX, NewY, _), [Path | Rest])),
        (NewX = GoalX, NewY = GoalY ;
         fill(jug(X), NewX), not(member((NewX, NewY, _), [Path | Rest])) ;
         fill(jug(Y), NewY), not(member((NewX, NewY, _), [Path | Rest])) ;
         empty(jug(X), NewX), not(member((NewX, NewY, _), [Path | Rest])) ;
         empty(jug(Y), NewY), not(member((NewX, NewY, _), [Path | Rest])))),
        Children),
    append

(Rest, Children, Queue),
    bfs(Queue, GoalX, GoalY, CapacityX, CapacityY, Solution).
```