**Artificial intelligence program (prolog):**

1. **Prolog (bird)**

```
can_fly(eagle).
can_fly(sparrow).
can_fly(pigeon).

cannot_fly(penguin).
cannot_fly(ostrich).

can_bird_fly(Bird) :- can_fly(Bird),write('.It Can Fly').
can_bird_fly(Bird) :- write('.It cannot Fly'),\+ cannot_fly(Bird).
```

output:

```
?- can_bird_fly(eagle).
It Can Fly
true .

?- can_bird_fly(penguin).
It cannot Fly
true .
```

2. **Prolog ( a DB with NAME, DOB)**

```
dob('abi','04-01-2004',12).

dob('hari','03-06-2004').

dob('shishi','14-05-2003').

dob('devi','26-10-2003').

get_dob(Name,DOB):- dob(Name,DOB).

get_dobage(Name,DOB,AGE):-dob(Name,DOB,AGE).
```

output:

x=abi,y=04-01-2004;

x=hari,y=03-06-2004;

x=shishi,y=14-05-2003;

x=devi,y=26-10-2003;

3. **Prolog(backwardchaining)**

```
has_feathers(bird).
can_fly(bird).
lays_eggs(bird).
has_scales(fish).
swims(fish).

% Define rules for inferring new information.
bird(X) :- has_feathers(X), lays_eggs(X), can_fly(X).
fish(X) :- has_scales(X), swims(X).

% Define a goal for backward chaining.
is_bird(X) :- bird(X).
is_fish(X) :- fish(X).
```

4. **Prolog(best first search algorithm )**

```
:- dynamic visited/1.

% best_first_search(+Start, +Goal, +Edges, +Heuristic, -Path, -Cost)
best_first_search(Start, Goal, Edges, Heuristic, Path, Cost) :-
```

```
    heuristic(Start, Goal, H),
    best_first_search_helper([(H, Start)], Goal, Edges, Heuristic, [Start], Path, Cost).

best_first_search_helper([(Cost, Node)|_], Goal, _, _, Path, [Node|Path], Cost) :-
    Node == Goal, !.

best_first_search_helper([(Cost, Node)|RestQueue], Goal, Edges, Heuristic, Visited, Path, TotalCost) :-
    findall((NewCost, Neighbor), (member((Node, Neighbor, EdgeCost), Edges),
                                  \+ member(Neighbor, Visited),
                                  heuristic(Neighbor, Goal, H),
                                  NewCost is Cost + EdgeCost + H),
            NeighborCostList),
    append(NeighborCostList, RestQueue, NewQueue),
    sort(NewQueue, SortedQueue),
    best_first_search_helper(SortedQueue, Goal, Edges, Heuristic, [Node|Visited], Path, TotalCost).

% Example Heuristic function
% heuristic(+Node, +Goal, -H)
% Define your own heuristic function based on your problem domain
heuristic(_, _, 0).

% Example graph edges
% graph_edge(Node1, Node2, Cost)
% Define your own graph edges based on your problem domain
```

```
graph_edge(a, b, 1).
graph_edge(a, c, 2).
graph_edge(b, d, 3).
graph_edge(c, e, 2).
graph_edge(d, e, 1).
graph_edge(e, f, 3).

% Example query(output):
% ?- best_first_search(a, f, [(a,b,1),(a,c,2),(b,d,3),(c,e,2),(d,e,1),(e,f,3)], heuristic, Path, Cost).
```

5. **prolog (dieting system disease)**

```
% Facts about foods and their glycemic index (GI).
food_gi(apple, 38).
food_gi(banana, 51).
food_gi(carrot, 35).
food_gi(rice, 73).
food_gi(bread, 70).
food_gi(pasta, 45).
food_gi(chocolate, 49).
food_gi(cookies, 57).

% Diet plan suggestions for diabetes.
diet_suggestion(diabetes, [apple, carrot, pasta]).

% Rules to suggest a diet based on disease.
suggest_diet(Disease, Diet) :-
```

```
    diet_suggestion(Disease, Diet),
    write('Recommended diet for '), write(Disease), write(': '), write(Diet),
nl.

% Rules to check if a food is suitable based on glycemic index (GI).
is_suitable_food(Food, GI) :-
    GI =< 55,
    write(Food), write(' is a low GI food.'), nl.

is_suitable_food(Food, GI) :-
    GI > 55,
    write(Food), write(' is a high GI food. Avoid in excess.'), nl.

% Predicate to check the suitability of foods for a specific disease.
check_food_suitability(Disease) :-
    diet_suggestion(Disease, Diet),
    write('Checking suitability of foods for '), write(Disease), write(':'), nl,
    check_food_suitability_helper(Diet).

check_food_suitability_helper([]).
check_food_suitability_helper([Food|Rest]) :-
    ,
    is_suitable_food(Food, GI),
    check_food_suitability_helper(Rest).
```

6. **Prolog family tree**

```
male(john).
```

```
male(alex).
male(bob).

female(lisa).
female(emma).

parent(john, alex).
parent(john, lisa).
parent(lisa, emma).
parent(lisa, bob).

father(X, Y) :- male(X), parent(X, Y).
mother(X, Y) :- female(X), parent(X, Y).

grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
grandfather(X, Z) :- male(X), grandparent(X, Z).
grandmother(X, Z) :- female(X), grandparent(X, Z).
```

output:

```
?- grandfather(X, emma).
X = john ;
false.

?- grandmother(X, emma).
X = lisa ;
false.
```

**7.prolog (forward chaining)**

```
has_fur(tiger).
has_feathers(bird).
can_fly(bird).
lays_eggs(bird).
has_scales(fish).
swims(fish).

% Define rules for inferring new information.
mammal(X) :- has_fur(X).
mammal(X) :- gives_milk(X).
bird(X) :- has_feathers(X), lays_eggs(X), can_fly(X).
reptile(X) :- has_scales(X), lays_eggs(X), not(mammal(X)).
fish(X) :- has_scales(X), lays_eggs(X), not(mammal(X)), not(bird(X)).

% Query to perform forward chaining and infer new information.
inferred_mammal(X) :- mammal(X).
inferred_bird(X) :- bird(X).
inferred_reptile(X) :- reptile(X).
inferred_fish(X) :- fish(X).
```

7. **Prolog(fruit backtrack)**

```
% Define fruits and their possible colors
fruit_color(apple, red).
fruit_color(banana, yellow).
```

```
fruit_color(grape, purple).
fruit_color(orange, orange).
fruit_color(watermelon, green).
fruit_color(strawberry, red).
fruit_color(blueberry, blue).
fruit_color(kiwi, brown).

% Predicate to check if a given fruit has a specified color
has_color(Fruit, Color) :-
    fruit_color(Fruit, Color).

% Predicate to find all fruits with a specified color using backtracking
fruits_with_color(Color) :-
    fruit_color(Fruit, Color),
    write(Fruit), write(' is '), write(Color), nl,
    fail. % Backtrack to find more fruits with the same color

% Example queries(output):
% To find the color of a specific fruit:
% ?- has_color(apple, Color).
%
% To find all fruits with a specified color:
% ?- fruits_with_color(red).
```

8. **Prolog (medical diagnosis)**

```
% Symptoms database
```

```
symptom(fever, cold).
symptom(cough, cold).
symptom(runny_nose, cold).
symptom(sore_throat, cold).
symptom(headache, flu).
symptom(fever, flu).
symptom(body_aches, flu).
symptom(fatigue, flu).
symptom(rash, measles).
symptom(fever, measles).
symptom(cough, measles).
symptom(conjunctivitis, measles).

% Rules for diagnosis
diagnosis(Patient, Disease) :-
    symptom(Symptom, Disease),
    has_symptom(Patient, Symptom).

% Predicates to check if patient has symptoms
has_symptom(Patient, Symptom) :-
    ask_patient(Patient, Symptom).

ask_patient(Patient, Symptom) :-
    format('Does ~w have ~w? (yes/no): ', [Patient, Symptom]),
    read(Response),
    Response = yes.
```

```
% Predicates to suggest treatment based on diagnosis
treatment(cold) :-
    write('Treatment for cold: Rest, fluids, and over-the-counter cold medications.'),
    nl.
treatment(flu) :-
    write('Treatment for flu: Rest, fluids, and antiviral medications prescribed by a doctor.'),
    nl.
treatment(measles) :-
    write('Treatment for measles: Rest, fluids, and over-the-counter fever reducers. Vaccination can prevent measles.'),
    nl.

% Example query(output):
% ?- diagnosis(john, Disease).
Disease = cold.
```

**9. Prolog (monkey banana)**

```
% Initial state: monkey is at the door, chair is at the middle of the room,
% and banana is hanging from the ceiling.
at(door, s0).
at(chair, s0).
at(banana, s0).

% Actions available to the monkey.
% move(X, Y): move from X to Y.
```

```
% climb(X): climb on object X.
% grasp(X): grasp object X.
action(move(X, Y), s0, s1) :-
    at(X, s0),
    connected(X, Y).

action(climb(X), s0, s1) :-
    at(X, s0),
    on_floor(s0),
    clear(X).

action(grasp(X), s0, s1) :-
    at(X, s0),
    on_floor(s0),
    holding(s0, X).

% Define the problem-solving goal: getting the banana.
goal_state(s1) :-
    holding(s1, banana).

% Define the state transitions.
% A state transition is possible if the action is applicable in the current state.
% The result is the next state.
result(A, S, S1) :-
    action(A, S, S1).

% Define connected rooms or objects.
```

```
connected(door, chair).
connected(chair, middle).
connected(middle, banana).

% Define predicates to check if an object is clear or the monkey is on the floor.
clear(X) :-
    \+ holding(s0, X).

on_floor(s0).

% Define holding predicate.
holding(s1, X) :-
    action(grasp(X), s0, s1).
```

## 10. Prolog (planet)

```
% Facts: Planets and their distances from the sun (in astronomical units, AE)
ae(mercury, 0.39).
ae(venus, 0.72).
ae(earth, 1).
ae(mars, 1.52).
ae(jupiter, 5.20).
ae(saturn, 9.54).
ae(uranus, 19.22).
ae(neptune, 30.06).
```

```
% Moons orbiting planets
orbits(moon, earth).
orbits(deimos, mars).
orbits(phobos, mars).
orbits(ganymede, jupiter).
orbits(callisto, jupiter).
orbits(io, jupiter).
orbits(europa, jupiter).
orbits(titan, saturn).
orbits(enceladus, saturn).
orbits(titania, uranus).
orbits(oberon, uranus).
orbits(umbriel, uranus).
orbits(ariel, uranus).
orbits(miranda, uranus).
orbits(triton, neptune).

% Predicate to find moons of a given planet
moons_of_planet(Planet, Moons) :-
    findall(Moon, orbits(Moon, Planet), Moons).
```

output:

```
 moons_of_planet(jupiter, JupiterMoons).
JupiterMoons = [ganymede, callisto, io, europa].
```

## 11. Prolog (student_techer_subject code)

```
teaches(john, math).
teaches(lisa, english).
teaches(ali, physics).

enrolled(bob, math).
enrolled(bob, english).
enrolled(sara, physics).

course_code(math, 101).
course_code(english, 102).
course_code(physics, 103).

/* Rules */
student_teacher_course(Student) :-
    enrolled(Student, Course),
    teaches(Teacher, Course),
    course_code(Course, Code),
    format('~w is enrolled in ~w taught by ~w. Course code: ~w~n',
[Student, Course, Teacher, Code]).
```

output:

```
Teacher = john_doe, Subject = math ;
Teacher = jane_smith, Subject = science ;
Teacher = alex_jones, Subject = english ;
```

## 12. Prolog(sum of integer 1 to n )

```
sum_integers(1,1).
```

```
sum_integers(0,0).
sum_integers(N,Sum):-
    N > 1,
    N1 is N-1,
    sum_integers(N1,SubSum),
    Sum is SubSum + N.
```

output:

```
Sum(5, Sum).
Sum = 15
```

## 13. Prolog (tower of Hanoi)

```
% hanoi(+N, +Start, +End, +Extra)
% This predicate represents the solution to the Tower of Hanoi problem.
% N: Number of disks
% Start: The starting peg
% End: The ending peg
% Extra: The auxiliary peg

hanoi(1, Start, End, _) :-
    format('Move disk from ~w to ~w~n', [Start, End]).

hanoi(N, Start, End, Extra) :-
    N > 1,
    M is N - 1,
    hanoi(M, Start, Extra, End),
    hanoi(1, Start, End, _),
```

```
    hanoi(M, Extra, End, Start).
```

output:

```
?- hanoi(3, left, middle, right).
Move disk from left to right
Move disk from left to middle
Move disk from right to middle
Move disk from left to right
Move disk from middle to left
Move disk from middle to right
Move disk from left to right
true.
```