3. A Prolog program for decision-making in a game scenario, specifically in a simple text-based adventure game. The program will represent the game's world using facts, define rules for player actions, and allow the inference of possible outcomes based on the player's decisions.

PROGRAM:

```
% World representation
location(kitchen).
location(living_room).
location(bedroom).

item(key, kitchen).
item(sword, bedroom).
item(book, living_room).

character(player, living_room).

% Current game state (dynamic facts)
:- dynamic(at/2).
:- dynamic(has/2).

at(player, living_room).

% Player actions: Move to a location
move(Location) :-
    location(Location),
    retract(at(player, _)),
    assert(at(player, Location)).

% Player actions: Take an item
take(Item) :-
    at(player, Loc),
    item(Item, Loc),
    retract(item(Item, Loc)),
    assert(has(player, Item)).

% Player actions: Drop an item
drop(Item) :-
    has(player, Item),
    at(player, Loc),
    retract(has(player, Item)),
    assert(item(Item, Loc)).

% Check if player can unlock door
can_unlock_door :-
    has(player, key),
    at(player, kitchen).

% Check if player can defeat enemy
can_defeat_enemy :-
    has(player, sword).

% Win condition: Player has key and is in kitchen
game_won :-
    has(player, key),
```

```
    at(player, kitchen).

% Describe current game state
describe_location :-
    at(player, Loc),
    write('You are in: '), write(Loc), nl,
    findall(Item, item(Item, Loc), Items),
    (Items \= [] -> write('Items here: '), write(Items), nl ; write('No items
here.'), nl),
    findall(Item, has(player, Item), Inventory),
    (Inventory \= [] -> write('Inventory: '), write(Inventory), nl ;
write('Inventory empty.'), nl).
```

Query / OUTPUT:

```
at(player, living_room).
true.

move(bedroom).
true.

at(player, X).
X = bedroom.

take(key).
false.

move(kitchen).
true.

take(key).
true.

has(player, key).
true.

can_unlock_door.
true.

move(bedroom).
true.

take(sword).
true.

has(player, sword).
true.

can_defeat_enemy.
true.

move(kitchen).
true.

game_won.
true.
```

```
describe_location.
You are in: kitchen
Items here: []
Inventory: [key, sword]
true.
```

---

4. A Prolog program for a simple medical diagnosis system. This program will represent symptoms, diseases, and their relationships using facts and rules. We will also create rules to infer possible diagnoses based on the symptoms provided.

PROGRAM:

```
% Facts representing symptoms
symptom(fever).
symptom(cough).
symptom(headache).
symptom(rash).
symptom(sneeze).
symptom(runny_nose).

% Facts representing diseases
disease(flu).
disease(cold).
disease(chickenpox).

% Rules linking symptoms to diseases
disease_symptom(flu, fever).
disease_symptom(flu, cough).
disease_symptom(flu, headache).
disease_symptom(cold, cough).
disease_symptom(cold, sneeze).
disease_symptom(cold, runny_nose).
disease_symptom(chickenpox, fever).
disease_symptom(chickenpox, rash).

% Rule to infer possible diagnoses
possible_diagnosis(Disease) :- disease(Disease).
```

Queries / OUTPUT:

```
possible_diagnosis(flu).
true.

possible_diagnosis(cold).
true.

possible_diagnosis(chickenpox).
true.

possible_diagnosis(Disease).
Disease = flu ;
```

```
Disease = cold ;
Disease = chickenpox.

disease_symptom(flu, X).
X = fever ;
X = cough ;
X = headache.
```

---

5. Write Prolog rules to perform basic arithmetic operations like addition, subtraction, multiplication, and division.

PROGRAM:

```
add(X, Y, Z) :- Z is X + Y.
subtract(X, Y, Z) :- Z is X - Y.
multiply(X, Y, Z) :- Z is X * Y.
divide(X, Y, Z) :- Z is X / Y.
```

Qureies / OUTPUT:

```
add(5, 3, Z).
Z = 8.

subtract(5, 3, Z).
Z = 2.

multiply(5, 3, Z).
Z = 15.

divide(6, 2, Z).
Z = 3.
```

---

6. Implement the basic Boolean operations: AND, OR, NOT

PROGRAM:

```
and_op(X, Y, true) :- X = true, Y = true.
and_op(_, _, false).

or_op(X, _, true) :- X = true.
or_op(_, Y, true) :- Y = true.
or_op(_, _, false).

not_op(true, false).
not_op(false, true).
```

Qureies / OUTPUT:

```
and_op(true, true, Result).
Result = true.
```

```
and_op(true, false, Result).
Result = false.

or_op(true, false, Result).
Result = true.

or_op(false, false, Result).
Result = false.

not_op(true, Result).
Result = false.

not_op(false, Result).
Result = true.
```

---

7. Simple Decision-Making: Find Maximum

PROGRAM:

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

Qureies / OUTPUT:

```
max(10, 5, Max).
Max = 10.

max(3, 7, Max).
Max = 7.

max(8, 8, Max).
Max = 8.
```

---

8. Decision-Making: Classify numbers (positive, negative, zero)

PROGRAM:

```
classify(Number, positive) :- Number > 0.
classify(Number, negative) :- Number < 0.
classify(0, zero).
```

Qureies / OUTPUT:

```
classify(10, Result).
Result = positive.

classify(-5, Result).
Result = negative.
```

```
classify(0, Result).
Result = zero.
```

---

9. Decision-Making: Classify even/odd numbers

PROGRAM:

```
classify(Number, positive_even) :- Number > 0, 0 is Number mod 2.
classify(Number, positive_odd) :- Number > 0, 1 is Number mod 2.
classify(Number, negative_even) :- Number < 0, 0 is Number mod 2.
classify(Number, negative_odd) :- Number < 0, 1 is Number mod 2.
classify(0, zero).
```

Qureies / OUTPUT:

```
classify(4, Result).
Result = positive_even.

classify(5, Result).
Result = positive_odd.

classify(-4, Result).
Result = negative_even.

classify(-5, Result).
Result = negative_odd.

classify(0, Result).
Result = zero.
```

---

10. Factorial (Recursive Decision-Making)

PROGRAM:

```
factorial(0, 1).
factorial(N, Result) :-
    N > 0,
    M is N - 1,
    factorial(M, Temp),
    Result is N * Temp.
```

Qureies / OUTPUT:

```
factorial(5, Result).
Result = 120.

factorial(0, Result).
Result = 1.

factorial(3, Result).
```

```
Result = 6.
```

---

11. Sum of first N natural numbers

PROGRAM:

```
sum(0, 0).
sum(N, Result) :-
    N > 0,
    N1 is N - 1,
    sum(N1, Result1),
    Result is N + Result1.
```

Qureies / OUTPUT:

```
sum(5, Result).
Result = 15.

sum(10, Result).
Result = 55.

sum(0, Result).
Result = 0.
```

---

12. Generate pairs that sum to 15

PROGRAM:

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

add_to_15(X, Y) :-
    between(0, 15, X),
    between(0, 15, Y),
    add(X, Y, 15).
```

Qureies / OUTPUT:

```
add_to_15(X, Y).
X = 0, Y = 15 ;
X = 1, Y = 14 ;
X = 2, Y = 13 ;
X = 3, Y = 12 ;
...
X = 15, Y = 0.
```

---

13. Calculating the length of a list

PROGRAM:

```
list_length([], 0).
list_length([_|Tail], N) :-
    list_length(Tail, N1),
    N is N1 + 1.
```

Qureies / OUTPUT:

```
list_length([1, 2, 3], N).
N = 3.

list_length([a, b, c, d, e], N).
N = 5.

list_length([], N).
N = 0.
```

---

14. Checking if an element is a member of the list

PROGRAM:

```
list_member(X, [X|_]).
list_member(X, [_|Tail]) :-
    list_member(X, Tail).
```

Qureies / OUTPUT:

```
list_member(3, [1, 2, 3, 4]).
true.

list_member(5, [1, 2, 3, 4]).
false.

list_member(X, [a, b, c]).
X = a ;
X = b ;
X = c.
```

---

15. Appending two lists

PROGRAM:

```
list_append([], L, L).
list_append([Head|Tail], L2, [Head|L3]) :-
    list_append(Tail, L2, L3).
```

Qureies / OUTPUT:

```
list_append([1, 2, 3], [4, 5], Result).
Result = [1, 2, 3, 4, 5].
```

```
list_append([a, b], [c, d, e], Result).
Result = [a, b, c, d, e].

list_append([], [1, 2, 3], Result).
Result = [1, 2, 3].
```

---

16. Reversing a list

PROGRAM:

```
list_reverse([], []).
list_reverse([Head|Tail], Reversed) :-
    list_reverse(Tail, ReversedTail),
    list_append(ReversedTail, [Head], Reversed).

list_append([], List, List).
list_append([Head|Tail], List, [Head|Rest]) :-
    list_append(Tail, List, Rest).
```

Qureies / OUTPUT:

```
list_reverse([1, 2, 3], X).
X = [3, 2, 1].

list_reverse([a, b, c, d], Reversed).
Reversed = [d, c, b, a].

list_reverse([], X).
X = [].
```

---

17. Sum of list elements

PROGRAM:

```
list_sum([], 0).
list_sum([Head|Tail], Sum) :-
    list_sum(Tail, SumTail),
    Sum is Head + SumTail.
```

Qureies / OUTPUT:

```
list_sum([1, 2, 3, 4], Sum).
Sum = 10.

list_sum([5, 10, 15], Sum).
Sum = 30.

list_sum([], Sum).
Sum = 0.
```

---

18. Find maximum element in list

PROGRAM:

```
list_max([X], X).
list_max([Head|Tail], Max) :-
    list_max(Tail, MaxTail),
    Max is max(Head, MaxTail).
```

Qureies / OUTPUT:

```
list_max([3, 1, 4, 1, 5], Max).
Max = 5.

list_max([10], Max).
Max = 10.

list_max([2, 8, 1, 9, 3], Max).
Max = 9.
```

---

19. Flatten nested list

PROGRAM:

```
list_flatten([], []).
list_flatten([Head|Tail], FlatList) :-
    list_flatten(Head, FlatHead),
    list_flatten(Tail, FlatTail),
    list_append(FlatHead, FlatTail, FlatList).
list_flatten(NonList, [NonList]) :-
    NonList \= [],
    NonList \= [_|_].

list_append([], L, L).
list_append([H|T], L, [H|R]) :- list_append(T, L, R).
```

Qureies / OUTPUT:

```
list_flatten([1, [2, 3], [4, [5, 6]]], X).
X = [1, 2, 3, 4, 5, 6].

list_flatten([a, [b, [c, d]], e], X).
X = [a, b, c, d, e].

list_flatten([], X).
X = [].
```

---

20. Increment each element in list

PROGRAM:

```
list_increment([], []).
list_increment([Head|Tail], [IncrementedHead|IncrementedTail]) :-
    IncrementedHead is Head + 1,
    list_increment(Tail, IncrementedTail).
```

Qureies / OUTPUT:

```
list_increment([1, 2, 3], Result).
Result = [2, 3, 4].

list_increment([5, 10, 15], Result).
Result = [6, 11, 16].

list_increment([], Result).
Result = [].
```

---

21. Filter even numbers from list

PROGRAM:

```
list_filter_even([], []).
list_filter_even([Head|Tail], [Head|FilteredTail]) :-
    0 is Head mod 2,
    list_filter_even(Tail, FilteredTail).
list_filter_even([_|Tail], FilteredTail) :-
    list_filter_even(Tail, FilteredTail).
```

Qureies / OUTPUT:

```
list_filter_even([1, 2, 3, 4, 5, 6], Result).
Result = [2, 4, 6].

list_filter_even([7, 9, 11], Result).
Result = [].

list_filter_even([2, 4, 6, 8], Result).
Result = [2, 4, 6, 8].
```

---

22. Pair elements from two lists

PROGRAM:

```
list_pairs([], [], []).
list_pairs([X|Xs], [Y|Ys], [(X, Y)|Pairs]) :-
    list_pairs(Xs, Ys, Pairs).
```

Qureies / OUTPUT:

```
list_pairs([1, 2, 3], [a, b, c], Result).
Result = [(1, a), (2, b), (3, c)].
```

```
list_pairs([a, b], [x, y], Result).
Result = [(a, x), (b, y)].

list_pairs([], [], Result).
Result = [].
```

---

23. Split list into front and back parts

PROGRAM:

```
split_list(List, 0, [], List).
split_list([Head|Tail], N, [Head|Front], Back) :-
    N > 0,
    N1 is N - 1,
    split_list(Tail, N1, Front, Back).
```

Qureies / OUTPUT:

```
split_list([1, 2, 3, 4, 5], 2, Front, Back).
Front = [1, 2],
Back = [3, 4, 5].

split_list([a, b, c, d], 0, Front, Back).
Front = [],
Back = [a, b, c, d].

split_list([1, 2, 3], 3, Front, Back).
Front = [1, 2, 3],
Back = [].

split_list([a, b, c, d, e], 2, Front, Back).
Front = [a, b],
Back = [c, d, e].
```
