Term Work - 04

Problem Statement

Write a PROLOG for a MENUDRIVEN PROGRAM for member, concatenation, add, delete and Permutation function.

Theory

In Prolog, a menu-driven program can be implemented using a combination of facts, rules, and predicates. Here's general outline of how you can structured a menu-driven program in prolog:

1. Define the menu option:
Create facts to represent each menu option, where the facts structured represent the option's name & any associated parameter or argument.

2. Implement menu handling rules:
→ Write rules that define the behaviour for menu option
→ Each rule should have a head that matches the selected menu option & a body that specifies the action to be performed.

3. Implement the main menu loop:
→ Write a predicate that displays the menu options & prompts the user for input.
→ Repeate the main menu loop until the user chooses to exit the program.

PROGRAM

```
% Helper predicates
concatenate([], L, L).
concatenate([H|T], L, [H|R]) :- concate(T, L, R).

add_element(X, L, [X|L]).

delete_element(_, [], []).
delete_element(X, [X|T], T).
delete_element(X, [H|T], [H|R]) :- X \= H, delete_element(X, T, R).

permute([], []).
permute([H|T], R) :- permute(T, X), select(H, R, X).

list_member(X, [X|_])
list_member(X, [_|T]) :- list_member(X, T)

% menu_driven function
menu :-
    write('MENU'), nl,
    write('1. concatenate list'), nl,
    write('2. Add element'), nl
    write('3. Delete element'), nl,
    write('4. Permute list'), nl,
    write('5. Check member'), nl,
    write('6. Quit'), nl,
    write('Enter the number of your choice :')
    read(choice),
    Process(choice).
```

```
Process(1):-
    write('Enter first list:'),
    read(L1),
    write('Enter second list:');
    read(L2),
    concatenate(L1, L2, Result),
    write('Concatenated list:'),
    write(Result), nl.
    menu.
Process(2):-
    write('Enter an element:'),
    read(X)
    write('Enter a list:'),
    read(L),
    add_element(X, L, Result),
    write('List after adding element:'),
    write(Result), nl,
    menu.
Process(3):-
    write('Enter an element:'),
    read(X),
    write('Enter a list:'),
    read(L),
    delete_element(X, L, Result),
    write('List after deleting element:'),
    write(Result), nl,
    menu.
Process(4):-
    write('Enter a list:'),
    read(L),
```

```
findall (X, Permute (L, X), Result),
write ('Permutations: '), nl, maplist (writeln, Results),
menu.
Process (5):-
write ('Goodbye!'), nl,

% Main Predicate
main :-
menu.

Process (5):-
write(' Enter the list').
read (L)
write ('Enter element to be searched / checked').
read (L1)
list_member (L, L1, Result).
write (Result).
menu.
Process (6):-
write ('Goodbye!'). nl

% Main Predicate
main :-
menu
```

Output

MENU
1. Concatenate lists
2. Add element
3. Queue element
4. Permute list
5. Check member
6. Quit.

Enter the number of your choice: 1

Enter the first list: [1, 2, 3, 4]
Enter the second list: [5, 6, 7, 8]
Concatenated list: [1, 2, 3, 4, 5, 6, 7, 8].

Enter the first number of your choice: 2
Enter an element: 2,
Enter a list: 4, 5, 6
List after adding element: [4, 5, 6, 2]

Enter the number of your choice: 3
Enter an element: 5
Enter a list: 4, 5, 6, 2
List after queuing element: [4, 6, 2]

TERMWORK

PROBLEM STATEMENT - Design an algorithm for TO implement Depth first search and develop a PROLOG program for the same.

Theory

Depth - first search or DFS algorithm is a recursive algorithm that uses the backtracking principle. It entails conducting exhaustive searches of all nodes by moving forward if possible & backtracking, if necessary. To visit the next node, pop the top node from the stack & push all of its nearby nodes into a stack. Topological sorting, Scheduling problems, graph cycle detection, & solving puzzles with just one solution, such as a maze or a sudoku puzzle.

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited.
2. Not visited.

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows.

1. Start by putting any one of the graph's vertices on top of a stack.

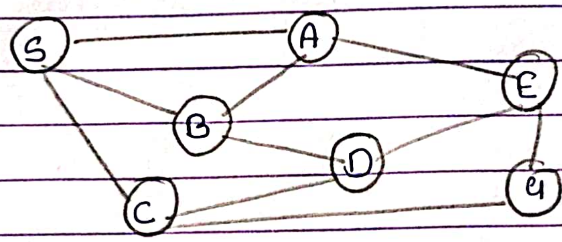
2. Take the top item of the stack & add it to the visited list.

3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.

4. Keep repeating steps 2 & 3 until the stack is empty.

Example

Start



Goal.

Open	Closed
(S, Nil)	( )
(A, S) (B, S) (C, S)	(S, Nil)
(E, A) (B, S) (C, S)	(A, S), (S, Nil)
(D, E), (G, E) (B, S) (C, S)	(E, A), (A, S) (S, Nil)
(G, E), (B, S), (C, S)	(D, E), (E, A) (A, S) (S, Nil)

Path is G → E, E → A, A → S

G → E → A → S   G ← E ← A ← S

S → A → E → S

Algorithm

```
DepthFirstSearch()
  open ← ((start NIL))
  closed ← ()
  while not Null(open)
     do nodepair ← Head(open)
        node ← Head(nodePair)
        if GoalTest(node) = TRUE
          then return ReconstructPath(nodePair, closed)
          else closed ← cons(nodepair, closed)
               children ← MoveGen(node)
               noLoops ← RemoveSeen(children, open, closed).
               new ← MakePairs(noLoops, node)
               open ← Append(new, Tail(open))
return "No solution found"

RemoveSeen(nodeList, openList, closedList)
   if Null(nodeList)
     then return ()
       else n ← Head(nodeList)
         if (OccursIn(n, openList) OR OccursIn(n, closedList))
           then return RemoveSeen(Tail(nodeList), openList,
                  closedList).
           else return cons(n, RemoveSeen(Tail(nodeList),
                openList, closedList).

OccursIn(node, listOfPairs)
  if Null(listOfPairs)
     then return FALSE
```

```
else if  n = Head ( Head ( list of Pairs )
        then return  TRUE.
        else return  occursIn ( node , Tail ( list of Pairs ) )

MakePairs ( list , parent )
    if  Null ( list )
        then return ( )
        else   return   cons ( MakeList ( Head ( list ) , parent ) ,
                    MakePairs ( Tail ( list ) , parent ) ) .
```

PROGRAM

```
child(S, E)
child(S, A)
child(S, B)
child(B, A)
child(A, E)
child(C, D)
child(D, E)
child(E, G)

path(A, G, [A|Z]):-        /* to find the path from root to leaf */
childnode(A, G, Z).
childnode(A, G, [G]):-      /* to determine whether a node is child of
                               other */
child(A, G).
childnode(A, G, [X|L]):-
child(A, X).
childnode(X, G, L).
```