WEEK-1:

Write simple programs in Prolog for facts, rules, and queries.

ALGORITHM:

Step 1: Start.

Step 2: Define Relation structure with fields parent and child.

Step 3: Initialize facts array with known parent–child pairs (e.g., {"john","mary"}, {"mary","susan"}) and set factCount.

Step 4: Define function isParent(x, y):

Step 5: Define function isGrandparent(x, y):

Step 5.1: For i from 0 to factCount-1 do:

Step 6: In main(), call isGrandparent("john","susan").

Step 7: If return value is 1 print result message (program confirms John is grandparent of Susan); else print negative message.

Step 8: End.

PROGRAM:

```
#include <stdio.h>
#include <string.h>

#define MAX 10

// Structure to represent a parent relationship
struct Relation {
   char parent[20];
   char child[20];
};

// Sample facts
struct Relation facts[MAX] = {
   {"john", "mary"},
   {"mary", "susan"}
};

int factCount = 2;

// Function to check if someone is a parent
int isParent(const char* x, const char* y) {
   for (int i = 0; i < factCount; i++) {
      if (strcmp(facts[i].parent, x) == 0 && strcmp(facts[i].child, y) == 0) {
         return 1;
      }
   }
   return 0;
}
```

```
// Rule: grandparent(X, Y) :- parent(X, Z), parent(Z, Y)
int isGrandparent(const char* x, const char* y) {
   for (int i = 0; i < factCount; i++) {
      if (strcmp(facts[i].parent, x) == 0) {
         const char* z = facts[i].child;
         if (isParent(z, y)) {
            return 1;
         }
      }
   }
   return 0;
}

int main() {
   // Query: Is john grandparent of susan?
   if (isGrandparent("john", "susan")) {
      printf("Yes, John is the grandparent of Susan.\n");
   } else {
      printf("No, John is not the grandparent of Susan.\n");
   }

   return 0;
}
```

OUTPUT:

Yes, John is the grandparent of Susan.

RESULT:

Thus, the program successfully determines and displays that John is the grandparent of Susan.

WEEK-2:

Develop a Prolog-based expert system for medical diagnosis or animal identification.

ALGORITHM:

Step 1: Start Program

Step 2: Initialize Knowledge Base

- Create an empty set to store facts.
- Create an empty list to store rules as pairs of (Head, Body). Step 3:

User Selection

- Read the user's choice:
  - If user selects 1, go to Medical Diagnosis.
  - If user selects 2, go to Animal Identification.
  - Else, display "Invalid mode". Step

4A: Medical Diagnosis Mode

- symptom:<symptom_name>.
- Define rules for each disease:
  - Example: diagnosis:flu :- symptom:fever, symptom:cough
  - Define possible goals (e.g., diagnosis:flu, diagnosis:allergy, etc.).
- For each goal, apply backward chaining:
- Display matching diagnoses Step

4B: Animal Identification Mode Step 5:

Backward Chaining Function

- For every rule with head == goal:
  - Check if all subgoals in the body are true by recursive calls.
- If any rule allows the goal to be inferred → return true.
- Else → return false.

Step 6: End Program

- Exit after displaying the diagnosis or identified animal type(s).

PROGRAM:

```
class ExpertSystem:
   def ___init___(self):
   self.facts = set()
      self.rules = []

   def add_fact(self, fact): self.facts.add(fact)

   def add_rule(self, head, body):
      self.rules.append((head, body))
   def backward_chain(self, goal, seen=None): if seen
      is None:
         seen = set()
      if goal in self.facts:
         return True
```

```
        if goal in seen: return
            False
        seen.add(goal)
        for head, body in self.rules: if
            head == goal:
                if all(self.backward_chain(condition, seen.copy()) for condition in
body):
                    return True
        return False

    def infer(self, goals):
        results = []
        for goal in goals:
            if self.backward_chain(goal):
                results.append(goal)
        return results
```

MAIN PROGRAM

```
def main():
    system = ExpertSystem()

    # --- MODE SELECTION ---
    print("Select Mode:") print("1.
    Medical Diagnosis")
    print("2. Animal Identification")
    mode = input("Enter 1 or 2: ")

    if mode == "1":
        # --- MEDICAL DIAGNOSIS ---
        print("\nMedical Diagnosis Expert System")
        symptoms = input("Enter symptoms (comma-separated, e.g., fever,cough): ").split(',')

        # Add user-reported symptoms for s
        in symptoms:
            system.add_fact(f"symptom:{s.strip().lower()}")

        # Rules for diagnoses
        system.add_rule("diagnosis:flu", ["symptom:fever", "symptom:cough"])
        system.add_rule("diagnosis:common_cold", ["symptom:runny_nose",
"symptom:sneezing"])
        system.add_rule("diagnosis:allergy", ["symptom:itchy_eyes", "symptom:sneezing"])
        system.add_rule("diagnosis:covid19", ["symptom:fever", "symptom:dry_cough",
"symptom:loss_of_taste"])
```

```
        # Possible diagnoses
        diagnoses = ["diagnosis:flu", "diagnosis:common_cold", "diagnosis:allergy",
"diagnosis:covid19"]

        result = system.infer(diagnoses) if
        result:
            print("\nPossible Diagnosis:") for r
            in result:
                print(" -", r.replace("diagnosis:", "").title()) else:
            print("No matching diagnosis found.")

    elif mode == "2":
        # --- ANIMAL IDENTIFICATION ---
        print("\nAnimal Identification Expert System")
        traits = input("Enter traits (comma-separated, e.g., fur,milk,lay_eggs): ").split(',')

        # Add user-reported traits for t
        in traits:
            system.add_fact(f"trait:{t.strip().lower()}")

        # Rules for animals
        system.add_rule("animal:mammal", ["trait:fur", "trait:milk"])
        system.add_rule("animal:bird", ["trait:feathers", "trait:lay_eggs"])
        system.add_rule("animal:reptile", ["trait:scales", "trait:cold_blooded"])
        system.add_rule("animal:amphibian", ["trait:moist_skin", "trait:lay_eggs"])
        system.add_rule("animal:fish", ["trait:gills", "trait:fins"])
        animals = ["animal:mammal", "animal:bird", "animal:reptile", "animal:amphibian",
"animal:fish"]

        result = system.infer(animals) if
        result:
            print("\nIdentified Animal Type:") for r
            in result:
                print(" -", r.replace("animal:", "").title()) else:
            print("No matching animal type found.") else:
        print("Invalid mode selected.")


if __name____== "___main___":
    main()
```

SAMPLE OUTPUT:

Select Mode:

1. Medical Diagnosis
2. Animal Identification

Enter 1 or 2: 2

Animal Identification Expert System

Enter traits (comma-separated, e.g., fur,milk,lay_eggs): fur,milk,lay_eggs

Identified Animal Type:

- Mammal

RESEULT:

Thus, the medical diagnosis or animal identification successfully completed and output verified.

WEEK-3:

Implement Depth-First Search (DFS) and Breadth-First Search (BFS) in Python.

ALGORITHM:

Algorithm: Depth-First Search (DFS) – Recursive

Step 1: Start.

Step 2: Initialize the graph as an adjacency list.

Step 3: Define a recursive function dfs_recursive(graph, node, visited) where visited is a set of visited nodes.

Step 4: If visited is None, initialize it as an empty set.

Step 5: Add the current node to visited and print it.

Step 6: For each neighbor of the current node in the graph:

Step 7: Repeat Step 6 until all reachable nodes are visited.

Step 8: End DFS traversal.

Algorithm: Breadth-First Search (BFS)

Step 1: Start.

Step 2: Initialize the graph as an adjacency list.

Step 3: Define a function bfs(graph, start) to perform BFS.

Step 4: Initialize a set visited to keep track of visited nodes.

Step 5: Initialize a queue and enqueue the start node; add it to visited.

Step 6: While the queue is not empty

Step 7: Repeat Step 6 until the queue is empty.

Step 8: End BFS traversal.

PROGRAM:

```
from collections import deque
# Sample graph as an adjacency list
graph = {
   'A': ['B', 'C'],
   'B': ['D', 'E'],
   'C': ['F'],
   'D': [],
   'E': ['F'],
   'F': []
}
```

```
# Depth-First Search (DFS) - Recursive
def dfs_recursive(graph, node, visited=None):
    if visited is None:
        visited = set()
    visited.add(node)
    print(node, end=' ')
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)

# Breadth-First Search (BFS)
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        node = queue.popleft()
        print(node, end=' ')
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
# --------------------------------
# Main execution
print("Depth-First Search (DFS):")
dfs_recursive(graph, 'A')
print("\nBreadth-First Search (BFS):")
bfs(graph, 'A')
```

Output:

```
Depth-First Search (DFS): A B
D E F C
Breadth-First Search (BFS): A B
C D E F
```

RESULT:

Thus, the program successfully performs Depth-First Search (DFS) and Breadth-First Search (BFS) on the given graph, displaying the order in which nodes are visited.

WEEK-4:

Implement A* Search Algorithm using heuristics in Python.

ALGORITHM:

Step 1: Start.

Step 2: Define the graph as an adjacency list where each node has neighbors and edge costs.

Step 3: Define heuristic values h(n) for each node estimating the cost to reach the goal.

Step 4: Initialize an empty priority queue open_set and add the start node with f(n) = h(start).

Step 5: Initialize dictionaries

Step 6: While open_set is not empty

- Remove the node current with the lowest f(n) from open_set.
- If current is the goal, reconstruct the path using came_from and return it.
- For each neighbor of current

Step 7: If open_set is empty and goal not reached, return "No path found".

Step 8: End.

PROGRAM:

```
from queue import PriorityQueue

# Define the graph as an adjacency
list graph = {
   'A': [('B', 1), ('C', 4)],
   'B': [('D', 2), ('E', 5)],
   'C': [('F', 3)],
   'D': [('G', 2)],
   'E': [('G', 1)],
   'F': [('G', 5)],
   'G': []
}

# Heuristic values (h(n)): estimated cost from each node to the goal 'G'
heuristic = {
   'A': 7,
   'B': 6,
   'C': 5,
   'D': 4,
```

```
  'E': 3,
  'F': 6,
  'G': 0
}

    def a_star_search(start,
                      goal):
open_set = PriorityQueue()
  open_set.put((0, start))  # (f(n), node)

  came_from = {}
  g_score = {node: float('inf') for node in graph}
  g_score[start] = 0

  f_score = {node: float('inf') for node in graph}
  f_score[start] = heuristic[start]

while not open_set.empty():
  _, current = open_set.get()

    if current == goal:
      path = []
      while current in came_from:
        path.append(current)
        current =
      came_from[current]
      path.append(start)
      return path[::-1]
    for neighbor, cost in graph[current]:
      tentative_g = g_score[current] +
      cost if tentative_g <
      g_score[neighbor]:
        came_from[neighbor] = current
        g_score[neighbor] = tentative_g
        f_score[neighbor] = tentative_g + heuristic[neighbor]
        open_set.put((f_score[neighbor], neighbor))

  return None  # If no path

found # Example usage
start_node = 'A'
goal_node = 'G'
path = a_star_search(start_node, goal_node)
if path:
  print("Path found:", " ->
```

```
".join(path)) else:
    print("No path found.")
```

OUTPUT:

Path found: A -> B -> D -> G

RESULT:

The python program successfully finds the shortest path from **A** to **G** using the A* search algorithm, considering both the actual cost g(n) and the heuristic h(n) to reach the goal efficiently.

WEEK-5:

Implement the Minimax algorithm for a simple game (e.g., Tic Tac Toe).

ALGORITHM:

Step 1: Start the program.

Step 2: Initialize the Tic Tac Toe board as a list of 9 empty spaces.

Step 3: Define a function print_board() to display the current board in 3×3 format.

Step 4: Define a function check_winner(board, player) to check if the given player ("X" or "O") has won by checking all rows, columns, and diagonals.

Step 5: Define a function is_draw(board) to check if the board is full and no winner exists.

Step 6: Define a function get_available_moves(board) to return a list of empty cells where a move can be made.

Step 7: Define the minimax(board, depth, is_maximizing) function:

Step 8: Define ai_move() to select the best move for AI using the minimax function and update the board.

Step 9: Define player_move() to take input from the user (1–9), validate the move, and update the board.

Step 10: Define play_game() to run the game loop:

Step 11: End the game when either player wins or the board is full (draw).

PROGRAM:

```
import math
# Define the board
board = [" " for _ in range(9)]
# Print the board
def print_board():
for row in [board[i*3:(i+1)*3] for i in range(3)]:
print("| " + " | ".join(row) + " |")
# Check for winner
def check_winner(brd, player):
win_conditions = [
[0, 1, 2], [3, 4, 5], [6, 7, 8],  # rows
[0, 3, 6], [1, 4, 7], [2, 5, 8],  # columns
[0, 4, 8], [2, 4, 6]              # diagonals
]
for cond in win_conditions:
if all(brd[i] == player for i in cond):
return True
```

```
return False
# Check for draw
def is_draw(brd):
return " " not in brd
# Get available moves
def get_available_moves(brd):
return [i for i in range(9) if brd[i] == " "]
# Minimax algorithm
def minimax(brd, depth, is_maximizing):
if check_winner(brd, "O"):
return 1
elif check_winner(brd, "X"):
return -1
elif is_draw(brd):
return 0
if is_maximizing:
best_score = -math.inf
for move in get_available_moves(brd):
brd[move] = "O"
score = minimax(brd, depth + 1, False)
brd[move] = " "
best_score = max(score, best_score)
return best_score
else:
best_score = math.inf
for move in get_available_moves(brd):
brd[move] = "X"
score = minimax(brd, depth + 1, True)
brd[move] = " "
best_score = min(score, best_score)
return best_score
# AI makes a move
def ai_move():
best_score = -math.inf
best_move = None
for move in get_available_moves(board):
board[move] = "O"
score = minimax(board, 0, False)
board[move] = " "
if score > best_score:
best_score = score
best_move = move
board[best_move] = "O"

# Human move
def player_move():
```

```
while True:
move = int(input("Enter your move (1-9): ")) - 1
if board[move] == " ":
board[move] = "X"
break
else:
print("Invalid move, try again.")
# Game loop
def play_game():
print("Tic Tac Toe - You (X) vs AI (O)")
print_board()
while True:
player_move()
print_board()
if check_winner(board, "X"):
print("You win!")
break
if is_draw(board):
print("It's a draw!")
break
ai_move()
print("AI's Move:")
print_board()
if check_winner(board, "O"):
print("AI wins!")
break
if is_draw(board):
print("It's a draw!")
break

# Run the game
play_game()
```

OUTPUT:

Tic Tac Toe - You (X) vs AI (O)

| | | |

| | | |

| | | |

Enter your move (1-9): 2

| | X | |

| | | |

| | | |

AI's Move:

| O | X | |

| | | |

| | | |

Enter your move (1-9): 4

| O | X | |

| X | | |

| | | |

AI's Move:

| O | X | |

| X | O | |

| | | |

Enter your move (1-9): 1

Invalid move, try again.

Enter your move (1-9): 6

| O | X | |

| X | O | X |

| | | |

AI's Move:

| O | X | O |

| X | O | X |

| | | |

Enter your move (1-9): 7

| O | X | O |

| X | O | X |

| X | | |

AI's Move:

| O | X | O |

| X | O | X |

| X | | O |

AI wins!

RESULT:

Thus, the above python program has completed successfully and verified.

WEEK-6

Design and implement a two-pass assembler in C.

ALGORITHM:

Step 1: Start the program.

Step 2: Open the source file input.asm for reading.
Step 3: Initialize variables and data structures
Step 4: Read one line from the input file.
If end of file is reached, go to **Step 10**.

Step 5: Split the line into three parts — label, opcode, operand (using sscanf).

Step 6: Check if the first word is actually an opcode:
Step 7: If a label exists, store it in the Symbol Table:
Step 8: Store the instruction into instructions[instrcount] with its address, label, opcode, and operand.
Step 9: Go back to **Step 4** to read the next line.

Step 10: Close the input file after reading all lines (end of Pass 1).

Step 11: Display the Symbol Table showing Label and Address for each symbol.

Step 12: For each instruction in instructions[] (begin Pass 2):
Step 13: After all instructions are processed, display the complete Machine Code Output.

Step 14: Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include <string.h>

#define MAX_SYMBOLS 100
#define MAX_LINES 100
#define MAX_LINE_LEN 100

typedef struct { char
   label[20]; int
   address;
} Symbol;

typedef struct { int
   address; char
   label[20];
   char opcode[10]; char
   operand[20];
} Instruction;

Symbol symtab[MAX_SYMBOLS]; int
```

```
symcount = 0;

Instruction instructions[MAX_LINES]; int
instrcount = 0;

int getOpcodeValue(const char* opcode) {
    if (strcmp(opcode, "LOAD") == 0) return 1; if
    (strcmp(opcode, "STORE") == 0) return 2; if
    (strcmp(opcode, "ADD") == 0) return 3;
    if (strcmp(opcode, "SUB") == 0) return 4; if
    (strcmp(opcode, "JMP") == 0) return 5; if
    (strcmp(opcode, "HLT") == 0) return 0; return -
    1;
}
int searchSymbol(const char* label) { for (int
    i = 0; i < symcount; i++) {
        if (strcmp(symtab[i].label, label) == 0) return
            symtab[i].address;
    }
    return -1;
}

void pass1(FILE* input) { char
    line[MAX_LINE_LEN]; int
    address = 0;

    while (fgets(line, sizeof(line), input)) {
        char label[20] = "", opcode[10] = "", operand[20] = ""; sscanf(line,
        "%s %s %s", label, opcode, operand);

        if (getOpcodeValue(label) != -1) {  // no label, shift everything strcpy(operand,
            opcode);
            strcpy(opcode, label);
            strcpy(label, "");
        }

        if (strlen(label) > 0) {
            strcpy(symtab[symcount].label, label);
            symtab[symcount].address = address;
            symcount++;
        }

        instructions[instrcount].address = address;
        strcpy(instructions[instrcount].label, label);
        strcpy(instructions[instrcount].opcode, opcode);
        strcpy(instructions[instrcount].operand, operand); instrcount++;
```

```
        address++;
    }
}

void pass2() {
    printf("Machine Code Output:\n");
    printf("Address\tMachine Code\n");

    for (int i = 0; i < instrcount; i++) {
        int opcodeVal = getOpcodeValue(instructions[i].opcode); int
        operandAddr = 0;

        if (strcmp(instructions[i].opcode, "HLT") != 0) {
            operandAddr = searchSymbol(instructions[i].operand); if
            (operandAddr == -1) {
                printf("Error: Undefined symbol %s\n", instructions[i].operand); exit(1);
            }
        }
        printf("%d\t%02d%02d\n", instructions[i].address, opcodeVal, operandAddr);
    }
}

void printSymbolTable() {
    printf("\nSymbol Table:\n");
    printf("Label\tAddress\n");
    for (int i = 0; i < symcount; i++) {
        printf("%s\t%d\n", symtab[i].label, symtab[i].address);
    }
}

int main() {
    FILE* input = fopen("input.asm", "r"); if
    (!input) {
        perror("Error opening file"); return
        1;
    }
    pass1(input);
    fclose(input);
    printSymbolTable();
    pass2();
    return 0;
}
```

input.asm

```
START  LOAD  A
       ADD   B
```

```
      STORE C
LOOP  SUB  A
      JMP  LOOP
      HLT
A     DATA 05
B     DATA 03
C     DATA 00
```

HOW TO COMPILE AND RUN:

gcc assembler.c -o assembler

./assembler

SAMPLE OUTPUT:

Symbol Table:

Label	Address
START	0
LOOP	3
A	6
B	7
C	8

Machine Code Output:

Address	Machine Code
0	0106
1	0307
2	0208
3	0406
4	0503
5	0000

Result:

Thus, a **two-pass assembler** program was successfully implemented in C language.

WEEK-7

Implement a Macro Processor using C for assembly language programs.

ALGORITHM:

Step 1: Start the program.

Step 2: Initialize data structures:

Step 3: Open the input assembly file macro_input.asm for reading and intermediate.asm for writing.

Step 4: Read each line from the input file.

Step 5: If the line contains the keyword MACRO:

Step 6: If the line is not a macro definition, copy it to the intermediate file.

Step 7: Close the input and intermediate files after Pass 1.

Step 8: Open intermediate.asm for reading and expanded.asm for writing.

Step 9: Read each line from intermediate.asm:

Step 10: Close all files after Pass 2.

Step 11: Display **MNT** and **MDT** tables to show macro names and definitions.

Step 12: Display a message indicating that macro expansion is complete.

Step 13: Stop the program.

PROGRAM: macro_processor.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 100

struct MNTEntry {
   char name[20];
   int mdtIndex;
};

struct MNTEntry MNT[MAX];
char MDT[MAX][100];
int mntc = 0, mdtc = 0;

// Function to check if a line starts with a macro name
int isMacroCall(char *line, int *mntIndex) {
   char temp[20];
   sscanf(line, "%s", temp);
   for (int i = 0; i < mntc; i++) {
      if (strcmp(temp, MNT[i].name) == 0) {
         *mntIndex = i;
         return 1;
```

```
        }
    }
    return 0;
}

// Pass 1 - Build MNT and MDT
void pass1() {
    FILE *fin = fopen("macro_input.asm", "r");
    FILE *fout = fopen("intermediate.asm", "w");

    char line[100];
    while (fgets(line, sizeof(line), fin)) {
        char word[20];
        sscanf(line, "%s", word);

        if (strcmp(word, "MACRO") == 0) {
            fgets(line, sizeof(line), fin);
            char macroName[20];
            sscanf(line, "%s", macroName);
            strcpy(MNT[mntc].name, macroName);
            MNT[mntc].mdtIndex = mdtc;
            mntc++;

            while (fgets(line, sizeof(line), fin)) {
                if (strstr(line, "MEND") != NULL) {
                    strcpy(MDT[mdtc++], "MEND");
                    break;
                } else {
                    strcpy(MDT[mdtc++], line);
                }
            }
        } else {
            fputs(line, fout);
        }
    }

    fclose(fin);
    fclose(fout);
}

// Pass 2 - Expand macros
void pass2() {
    FILE *fin = fopen("intermediate.asm", "r");
    FILE *fout = fopen("expanded.asm", "w");

    char line[100];
    while (fgets(line, sizeof(line), fin)) {
        int index;
        if (isMacroCall(line, &index)) {
```

```
            int i = MNT[index].mdtIndex;
            while (strcmp(MDT[i], "MEND") != 0) {
                fputs(MDT[i], fout);
                i++;
            }
        } else {
            fputs(line, fout);
        }
    }

    fclose(fin);
    fclose(fout);
}

// Display tables
void displayTables() {
    printf("MNT:\n");
    for (int i = 0; i < mntc; i++) {
        printf("%s\t%d\n", MNT[i].name, MNT[i].mdtIndex);
    }
    printf("\nMDT:\n");
    for (int i = 0; i < mdtc; i++) {
        printf("%d\t%s", i, MDT[i]);
    }
}

int main() {
    printf("Pass 1: Processing macros...\n");
    pass1();
    printf("Pass 2: Expanding macros...\n");
    pass2();

    printf("\nMacro Name Table (MNT) and Macro Definition Table (MDT):\n");
    displayTables();

    printf("\nMacro expansion complete. See 'expanded.asm' for output.\n");
    return 0;
}
```

Create File Before Run this program

macro_input.asm

```
MACRO
INCR &ARG1
ADD &ARG1, ONE
MEND

START
MOV A, B
INCR A
```

```
SUB B, A
END
```

intermediate.asm

```
START
MOV A, B
INCR A
SUB B, A
END
```

expanded.asm

```
START
MOV A, B
ADD A, ONE
SUB B, A
END
```

COMPILE THE PROGRAM:

Using a terminal or command prompt:

bash

gcc macro_processor.c -o macro_processor

RUN THE PROGRAM

./macro_processor

SAMPE OUTPUT:

Expected Console Output

Pass 1: Processing macros...
Pass 2: Expanding macros...

Macro Name Table (MNT) and Macro Definition Table (MDT):
MNT:
INCR 0

MDT:
0 ADD &ARG1, ONE
1 MEND

Macro expansion complete. See 'expanded.asm' for output.

intermediate.asm (After Pass 1)

```
START
MOV A, B
INCR A
SUB B, A
END
```

expanded.asm (Final Output After Pass 2)

```
START
MOV A, B
ADD &ARG1, ONE
SUB B, A
END
```

RESULT:
The program successfully reads macros from the input file, builds **MNT** and **MDT**, and expands macro calls in the program to generate the fully expanded assembly code.

WEEK-8

Develop a simple Linux Shell (command interpreter) using C.

ALGORITHM:

Step 1: Start the program.

Step 2: Initialize variables

Step 3: Enter an infinite loop (while(1)) to repeatedly accept user commands.

Step 5: Read a line of input from the user using fgets().

Step 6: Remove the newline character (\n) from the input line.

Step 7: Check for empty input; if empty, continue to the next iteration.

Step 8: Check for the exit command:

Step 9: Split the input line into arguments using strtok().

Step 10: Create a child process using fork().

Step 11: In the child process (pid == 0):

Step 12: In the parent process (pid > 0):

Step 13: Repeat the loop from **Step 4** for the next command.

Step 14: End the program when the user types "exit" or EOF is received.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define MAX_ARGS 64

int main() {
   char line[1024];
   char *argv[MAX_ARGS];

   while (1) {
      // Print prompt
      printf("myshell$ ");
      fflush(stdout);

      // Read command line
      if (!fgets(line, sizeof(line), stdin)) {
         printf("\n");
         break; // EOF
      }
```

```
        // Remove newline
        line[strcspn(line, "\n")] = '\0';

        // Skip empty input
        if (line[0] == '\0') continue;

        // Exit command
        if (strcmp(line, "exit") == 0) break;

        // Split into arguments
        int argc = 0;
        char *token = strtok(line, " ");
        while (token != NULL && argc < MAX_ARGS - 1) {
            argv[argc++] = token;
            token = strtok(NULL, " ");        }
        argv[argc] = NULL;

        // Fork and execute
        pid_t pid = fork();
        if (pid < 0) {
            perror("fork");
        } else if (pid == 0) {
            execvp(argv[0], argv);
            perror("execvp"); // Only runs if exec fails
            exit(EXIT_FAILURE);
        } else {
            int status;
            waitpid(pid, &status, 0);      }    }

    return 0;
}
```

How to run:

```
gcc simple_shell.c -o myshell

./myshell
```

OUTPUT:

```
myshell$ ls
simple_shell.c
myshell$ echo Hello World
Hello World
myshell$ exit
```

RESULT:

Thus, the simple Linux Shell (command interpreter) using C program successfully completed and output verified.

WEEK-9

Write shell scripts for file operations, process creation, and monitoring.

ALGORITHM:

Step 1: Start the script.

Step 2: Create a file sample.txt with some content.

Step 3: Display the contents of sample.txt.

Step 4: Copy sample.txt to sample_copy.txt.

Step 5: Rename sample_copy.txt to renamed_sample.txt.

Step 6: Delete renamed_sample.txt.

Step 7: Start a background process sleep 60 and capture its PID.

Step 8: Check if the background process is running and display its status.

Step 9: Kill the background process.

Step 10: Verify whether the process has stopped and display the result.

Step 11: End the script.

PROGRAM:

nanomyscript.sh

```
echo "===== FILE OPERATIONS ====="
# Create a file
echo "Creating file sample.txt ..."
echo "This is a sample file." > sample.txt

# Display file contents
echo "Contents of sample.txt:"
cat sample.txt

# Copy the file
cp sample.txt sample_copy.txt
echo "File copied as sample_copy.txt"

# Rename the file
mv sample_copy.txt renamed_sample.txt
echo "File renamed to
renamed_sample.txt"

# Delete renamed file
rm renamed_sample.txt
```

```
echo "renamed_sample.txt deleted"

echo
echo "===== PROCESS CREATION ====="

# Create a background process
sleep 60 &
PID=$!
echo "Background process 'sleep 60' started with PID: $PID"

echo
echo "===== PROCESS MONITORING ====="

# Check if process is
running if ps -p $PID >
/dev/null  then
   echo "Process $PID is
running" else
   echo "Process $PID is not running"
fi

# Kill the process
kill $PID
echo "Process $PID killed"
# Verify
if ps -p $PID > /dev/null
then
   echo "Process $PID is still running"
else
   echo "Process $PID has stopped"
fi

echo "===== SCRIPT COMPLETED ====="
```

Sample Output:

```
===== FILE OPERATIONS =====
Creating file sample.txt ...
Contents of sample.txt:
This is a sample file.
File copied as sample_copy.txt
File renamed to renamed_sample.txt
renamed_sample.txt deleted

===== PROCESS CREATION =====
Background process 'sleep 60' started with PID: 12345
```

===== PROCESS MONITORING =====
Process 12345 is running
Process 12345 killed
Process 12345 has stopped
===== SCRIPT COMPLETED =====

RESULT:

Thus, the above python program has completed successfully and verified.

WEEK- 10.

Demonstrate inter-process communication using pipes and signals in Linux.

ALGORITHM:

Step 1: Start Program

Step 2: Initialize variables

Create a pipefd[2] array to hold the read and write ends of the pipe.

Step 3: Create Pipe

Call pipe(pipefd) to create a unidirectional communication channel.

If pipe() fails, print error and exit.

Set Signal Handler in Parent

Register signal(SIGUSR1, signal_handler) so the parent can respond when the child sends a signal.

Step 4: Create Child Process

Call fork().

If fork() fails, print error and exit.

In Child Process (pid == 0):

In Parent Process (pid > 0):

Step 5: End Program

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

int pipefd[2];  // pipe file descriptors

// Signal handler for parent process
void signal_handler(int sig) {
   if (sig == SIGUSR1) {
      printf("Parent: ✅ Received signal from child (SIGUSR1)\n");
   }
}

int main() {
   pid_t pid;
   char msg[] = "Hello from Parent via Pipe!";
   char buffer[100];

   // Create pipe
   if (pipe(pipefd) == -1) {
      perror("pipe");
      exit(1);
   }

   // Register signal handler in parent
   signal(SIGUSR1, signal_handler);
```

```
    pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) {
        // ---------------- Child Process ----------------
        close(pipefd[1]);  // Close write end (child only reads)
        read(pipefd[0], buffer, sizeof(buffer));
        printf("Child: 📥 Message from parent = %s\n", buffer);
        close(pipefd[0]);

        // Send signal to parent after reading
        kill(getppid(), SIGUSR1);

        exit(0);
    } else {
        // ---------------- Parent Process ----------------
        close(pipefd[0]);  // Close read end (parent only writes)
        write(pipefd[1], msg, strlen(msg) + 1);
        close(pipefd[1]);

        // Wait for signal from child
        pause(); // Suspends until a signal arrives
    }

    return 0;
}
```

How to Compile & Run in Linux:

Open terminal in the directory containing the file:

```
gcc ipc_pipe_signal.c -o ipc_demo
./ipc_demo
```

OUTPUT:

Child: Message from parent = Hello from Parent via Pipe!

Parent: Received signal from child (SIGUSR1)

RESULT:

Thus, the above python program has completed successfully and verified.

WEEK-11

Integrate AI logic (search/expert system) into a shell script or system utility for task automation

Algorithm:

1. Start the program.
2. Display system information (disk usage, memory, uptime).
3. Apply AI logic (rule-based decision-making):
4. IF disk usage > 80% → suggest cleanup.
5. IF uptime > 7 days → suggest reboot.
6. IF free memory < 500MB → suggest closing background apps.
7. Display AI recommendations.
8. Ask user for permission to execute suggested actions.
9. Perform selected task automatically.
10. End program.

PROGRAM: system_utility.sh

```
#!/bin/bash
# AI-Powered System Utility (Expert System Logic)

echo "========================================="
echo "     AI System Utility - Expert Logic     "
echo "========================================="

disk_usage=$(df / | tail -1 | awk '{print $5}' | sed 's/%//')
uptime_days=$(uptime -p | grep -o '[0-9]* day' | awk '{print $1}')
free_mem=$(free -m | awk '/Mem:/ {print $4}')

echo "Disk Usage     : $disk_usage%"
echo "Uptime (days)  : ${uptime_days:-0}"
echo "Free Memory    : $free_mem MB"
echo "-----------------------------------------"

echo "AI System Analysis and Recommendations:"
if [ "$disk_usage" -gt 80 ]; then
echo "High disk usage detected! Suggest: Run cleanup."
action="cleanup"
elif [ "${uptime_days:-0}" -gt 7 ]; then
echo "System uptime is high. Suggest: Reboot system."
action="reboot"
elif [ "$free_mem" -lt 500 ]; then
echo "Low free memory detected! Suggest: Close background apps."
```

```
action="free_memory"
else
echo "System running optimally. No action needed."
action="none"
fi

if [ "$action" != "none" ]; then
read -p "Do you want to perform the suggested action? (y/n): " choice
if [ "$choice" == "y" ]; then
case $action in
"cleanup")
echo "Performing cleanup..."
sudo apt-get clean
sudo rm -rf /tmp/*
;;
"reboot")
echo "Rebooting system..."
sudo reboot
;;
"free_memory")
echo "Clearing memory cache..."
sudo sync; echo 3 | sudo tee /proc/sys/vm/drop_caches
;;
esac
echo "Action completed successfully."
else
echo " Action skipped."
fi
fi

echo "----------------------------------------"
echo "AI System Utility finished."
```

Sample Output:

```
========================================
    AI System Utility - Expert Logic
========================================
Disk Usage     : 85%
Uptime (days)  : 10
Free Memory    : 430 MB
----------------------------------------
AI System Analysis and Recommendations:
High disk usage detected! Suggest: Run cleanup.
Do you want to perform the suggested action? (y/n): y
```

Performing cleanup...
Reading package lists... Done
Action completed successfully.

AI System Utility finished.

RUN:
bash ai_system_utility.sh

Result:

The shell script was successfully developed and executed to integrate AI logic (expert system) for system task automation.

WEEK-12

1. Title for Final Mini Project:

DEVELOP AN AI-POWERED SYSTEM UTILITY (E.G., INTELLIGENT FILE MANAGER, AI BOT FOR CLI COMMANDS).

2. Objective

The objective of this project is to develop a smart file management system that can understand simple natural-language commands (like *find*, *list*, *delete*) and perform file operationsintelligently.

This utility helps users interact with their computer's file system in a human-friendly way using AI-based command interpretation.

3. Existing System

In the existing system, users have to manually enter command-line instructions or navigate folders to perform file operations like listing, finding, or deleting files.
This process can be time-consuming and requires technical knowledge of system commands.

4. Proposed System

The proposed system introduces an AI-powered intelligent file manager that understands natural language inputs such as

- "List files"
- "Find report"
- "Delete temp files"

and executes appropriate actions automatically.
This system uses simple AI logic (rule-based understanding) to interpret user intent and interact with the operating system.

5. Software Requirements

- Programming Language: Python 3
- IDE: IDLE / VS Code / PyCharm
- Operating System: Windows / Linux
- Libraries Used: Built-in os library

6. Hardware Requirements

- Processor: Intel i3 or above
- RAM: 2 GB minimum
- Storage: 100 MB minimum

7. Methodology / Working Principle

1. **Input:**
   The user types a natural language command (e.g., "find report").
2. **Processing:**
   - The program converts the command to lowercase.
   - It identifies keywords such as *find*, *list*, *delete*.

- Based on the keyword, it performs the corresponding file operation.

3. **Output:**
   The system displays the result — files found, listed, or deleted (simulated).

ALGORITHM:

Step 1: Start the program.
Step 2: Display a welcome message and sample commands to the user.
Step 3: Repeat until the user enters "exit":

1. Read the user input command.
2. Convert the command to lowercase to ensure case-insensitive comparison.
3. Check for keywords in the command:
   - If the command contains "list" or "show":
     - Display all files in the current directory using os.listdir().
   - Else if the command contains "find":
     - Extract the keyword following "find".
     - Search for files in the current directory whose filenames contain that keyword.
     - Display all matching filenames.
   - Else if the command contains "delete":
     - Extract the keyword following "delete".
     - Display all filenames that would be deleted (simulation only).
     - *(No actual deletion is performed for safety.)*
   - Else:
     - Display a message: "Sorry, I don't understand that command."
4. Return to Step 3 to get the next command.

Step 4: When the user types "exit", display "Goodbye!" and stop the program.
Step 5: End the program.

PROGRAM: ai_file.py

```
import os
def ai_file_manager(command):
command = command.lower()

if "list" in command or "show" in command:
print("Listing all files in current directory:")
for f in os.listdir():
print("-", f)

elif "find" in command:
word = command.replace("find", "").strip()
print(f"Searching for files containing '{word}'...")
for f in os.listdir():
if word in f.lower():
print("Found:", f)
```

```
elif "delete" in command:
word = command.replace("delete", "").strip()
print(f"Deleting files containing '{word}' (simulation)...")
for f in os.listdir():
if word in f.lower():
print("Would delete:", f)

else:
print("Sorry, I don't understand that command.")

# ---------- Main Program ----------
print("🤖 AI File Manager (Simple Version)")
print("Type commands like:")
print("  list files")
print("  find report")
print("  delete temp")
print("Type 'exit' to quit.\n")

while True:
cmd = input("Enter command: ")
if cmd.lower() == "exit":
print("Goodbye!")
break
ai_file_manager(cmd)
```

SAMPLE OUTPUT:

```
AI File Manager (Simple Version)
Type commands like:
list files
find report
delete temp
Type 'exit' to quit.

Enter command: list files
Listing all files in current directory:
- ai_file_manager.py
- report1.txt
- temp_data.csv

Enter command: find report
Searching for files containing 'report'...
Found: report1.txt

Enter command: delete temp
```

Deleting files containing 'temp' (simulation)...
Would delete: temp_data.csv

Enter command: exit
Goodbye!

10. Result
The system successfully performs basic intelligent file operations using simple AI-based logic. It interprets user commands in natural language and executes the corresponding action safely.

11. Conclusion
The AI-Powered Intelligent File Manager provides a simple and efficient way for users to interact with their system using natural language.
It serves as a foundation for developing more advanced AI-based command utilities that make computing easier for everyone.