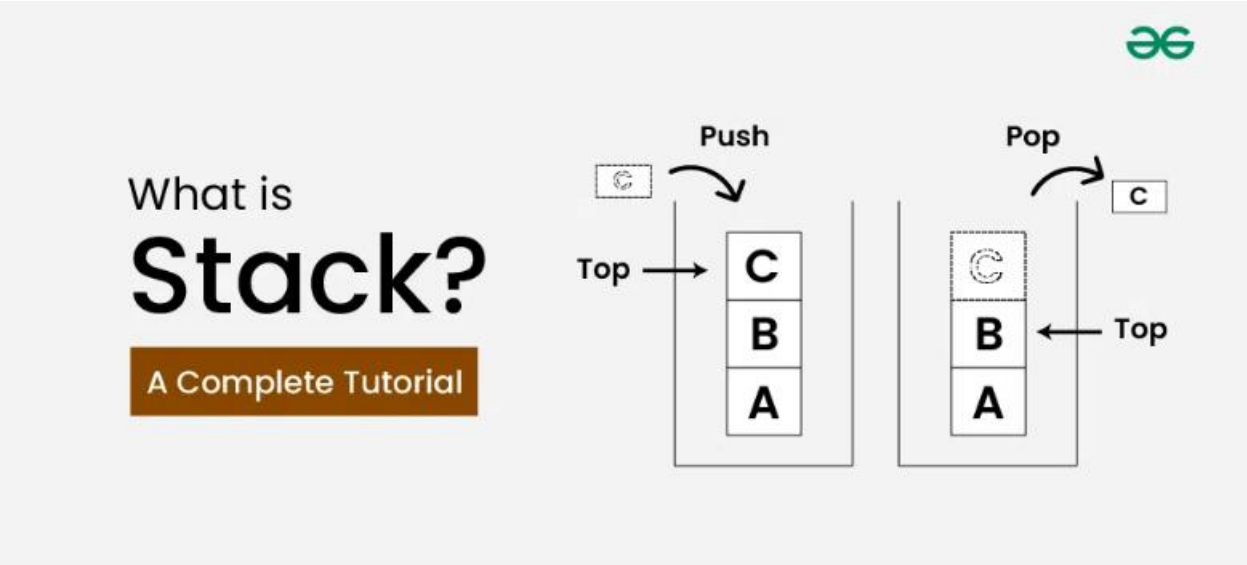
EXPERIMENT-1(a)

AIM: Write a program in c to perform following operation on a stack.

1. Push()
2. Pop()
3. Overflow
4. Underflow

THEORY:

Stack is a linear data structure that follows LIFO (Last In First Out) Principle, the last element inserted is the first to be popped out. It means both insertion and deletion operations happen at one end only.



CODE:

```
#include <stdio.h>

#include <stdlib.h>

#define MAX_SIZE 100

int stack[MAX_SIZE];

int top = -1;

int isEmpty() {

  return top == -1;
```

```
}
int isFull() {
   return top == MAX_SIZE - 1;}
void push(int data) {
   if (isFull()) {
      printf("Stack Overflow\n");    }
 else {
      top++;
      stack[top] = data;
      printf("%d pushed to stack\n", data);}}
int pop() {
   if (isEmpty()) {
      printf("Stack Underflow\n");
      return -1; // Return an error value  }
 else {
      int data = stack[top];
      top--;
      return data;}}
void display() {
   if (isEmpty()) {
      printf("Stack is empty\n");}
    else {
      printf("Stack: ");
      for (int i = top; i >= 0; i--) {
         printf("%d ", stack[i]);}
      printf("\n");}}
```

```
int main() {
   int choice, data;
   while (1) {
      printf("\n1. Push\n");
      printf("2. Pop\n");
      printf("3. Display\n");
      printf("4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);
      switch (choice) {
         case 1:
            printf("Enter data to push: ");
            scanf("%d", &data);
            push(data);
            break;
         case 2:
            data = pop();
            if (data != -1) {
               printf("%d popped from stack\n", data);}
            break;
         case 3:
            display();
            break;
         case 4:
            exit(0);
         default:
```

```
            printf("Invalid choice\n");}}
    return 0;}
```

Output:

1. Push

2. Pop

3. Display

4. Exit

Enter your choice: 1

Enter data to push: 10

10 pushed to stack

1. Push

2. Pop

3. Display

4. Exit

Enter your choice: 1

Enter data to push: 20

20 pushed to stack

1. Push

2. Pop

3. Display

4. Exit

Enter your choice: 3

Stack: 20 10

1. Push

2. Pop

3. Display

4. Exit

Enter your choice: 2

20 popped from stack

1. Push

2. Pop

3. Display

4. Exit

Enter your choice: 3

Stack: 10

1. Push

2. Pop

3. Display

4. Exit

Enter your choice: 2

10 popped from stack

1. Push

2. Pop

3. Display

4. Exit

Enter your choice: 3

Stack is empty

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 2

Stack Underflow

1. Push
2. Pop
3. Display
4. Exit

Enter your choice: 4

EXPERIMENT-1(b)

AIM: Write a program in c to calculate no. of tokens in a code.

THEORY:

C program to detect tokens in a C program

As it is known that Lexical Analysis is the first phase of compiler also known as scanner. It converts the input program into a sequence of Tokens.

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol.

For Example:

```
1) Keywords:
Examples- for, while, if etc.

2) Identifier
Examples- Variable name, function name etc.

3) Operators:
Examples- '+', '++', '-' etc.

4) Separators:
Examples- ', ' ';' etc
```

CODE:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>


int countTokens(char *code) {

  int count = 0;

  int i, len = strlen(code);


  // Skip leading spaces
```

```
    while (isspace(code[0])) {
        code++;
        len--;
    }

    for (i = 0; i < len; i++) {
        // Check if current character is the start of a token
        if (!isspace(code[i]) && (i == 0 || isspace(code[i - 1]))) {
            count++;
        }
    }

    return count;
}

int main() {
    char code[1000];

    printf("Enter the code: ");
    fgets(code, sizeof(code), stdin);

    int numTokens = countTokens(code);
    printf("Number of tokens: %d\n", numTokens);

    return 0;
}
```

Output:

```
Enter the code: int main() {
    int a = 10 + 5;
    return 0;
}

Number of tokens: 12
```

EXPERIMENT-1(c)

AIM: Write a program to identify constants , keywords and identifier in a given string.

THEORY:

A C compiler can break down a source code into tokens, which include keywords, identifiers, constants, operators, and punctuation symbols. The compiler uses these tokens to understand the program's structure and functionality.

- **Keywords**: Reserved for special use, keywords cannot be used as identifiers.
- **Identifiers**: Created by specifying them in the declaration of a variable, type, or function. For example, result is an identifier for an integer variable.
- **Constants**: Values that are not altered by the program during normal execution. For example, pi is a constant that can be defined using #define.

CODE:

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>

int isKeyword(char *str) {
  char keywords[][10] = {"auto", "break", "case", "char", "const", "continue", "default",
                "do", "double", "else", "enum", "extern", "float", "for", "goto",
                "if", "int", "long", "register", "return", "short", "signed",
                "sizeof", "static", "struct", "switch", "typedef", "union",
                "unsigned", "void", "volatile", "while"};
  int i;
  for (i = 0; i < 32; i++) {
    if (strcmp(str, keywords[i]) == 0) {
      return 1; // It's a keyword
    }
```

```
    }
    return 0; // Not a keyword}
int isIdentifier(char *str) {
    int i;
    if (!isalpha(str[0]) && str[0] != '_') {
        return 0; // First character must be letter or underscore}
    for (i = 1; str[i]; i++) {
        if (!isalnum(str[i]) && str[i] != '_') {
            return 0; // Subsequent characters must be alphanumeric or underscore} }
    return 1; // It's an identifier}
int isConstant(char *str) {
    // Simple check for integer and character constants
    if (isdigit(str[0])) {
        return 1; // Integer constant (basic check)
    } else if (str[0] == '\'' && str[2] == '\'') {
        return 1; // Character constant (basic check)}
    return 0; // Not a constant
}
int main() {
    char str[100];
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    char *token;
    token = strtok(str, " ,;\t\n");
    while (token != NULL) {
        if (isKeyword(token)) {
```

```
        printf("%s is a keyword\n", token);
    } else if (isIdentifier(token)) {
        printf("%s is an identifier\n", token);
    } else if (isConstant(token)) {
        printf("%s is a constant\n", token);
    } else {
        printf("%s is not recognized\n", token);
    }
    token = strtok(NULL, " ,;\t\n");
  }
  return 0;}
```

Output:

```
Enter a string: int main() {
    int a = 10;
    return 0;
}

int is a keyword
main is an identifier
{ is not recognized
    is not recognized
int is a keyword
a is an identifier
= is not recognized
10 is a constant
; is not recognized
    is not recognized
return is a keyword
0 is a constant
; is not recognized
} is not recognized
```

EXPERIMENT-2

AIM: Write a program to identify that a given string belongs to a particular grammar or not.

THEORY:

Grammar Example (Context-Free Grammar):

1. S → aSb
2. S → ε (epsilon, meaning the empty string)

This grammar generates strings with equal numbers of `a` and `b`, where all `a`s come before `b`s.

The program will verify if the input string adheres to this grammar.

How It Works:

1. **Recursive Function:** The function `isValidGrammar` checks if the string starts with `a`, ends with `b`, and the substring between them also belongs to the grammar.
2. **Base Case:** If the string is empty, it's valid (`ε` rule).
3. **Recursive Case:** For every `a` at the start and `b` at the end, the function checks the substring without those characters.

Example Inputs and Outputs:

- Input: `ab` → Output: "The string belongs to the grammar."
- Input: `aabb` → Output: "The string belongs to the grammar."
- Input: `abab` → Output: "The string does not belong to the grammar."
- Input: `` (empty string) → Output: "The string belongs to the grammar."

Code:

```
#include <stdio.h>

#include <string.h>

// Function to check if the string matches the grammar

int isValidGrammar(const char* str, int start, int end) {

  // Base case: Empty string is valid
```

```
    if (start > end) {
        return 1;}
    // Check for the pattern 'aSb'
    if (str[start] == 'a' && str[end] == 'b') {
        return isValidGrammar(str, start + 1, end - 1);}
    // If it doesn't match the pattern, it's invalid
    return 0;}
int main() {
    char input[100];
    printf("Enter a string: ");
    scanf("%s", input);
    int len = strlen(input);
    if (isValidGrammar(input, 0, len - 1)) {
        printf("The string belongs to the grammar.\n");
    } else { printf("The string does not belong to the grammar.\n");}
    return 0;}
```

Output:

```
Enter a string: abab
The string does not belong to the grammar.
```

EXPERIMENT 3

Aim: Write a program to convert given NFA to DFA

Language used: C/C++

Theory:

An NFA can have zero, one or more than one move from a given state on a given input symbol. An NFA can also have NULL moves (moves without input symbol). On the other hand, DFA has one and only one move from a given state on a given input symbol.

Steps for converting NFA to DFA:

Step 1: Convert the given NFA to its equivalent transition table
To convert the NFA to its equivalent transition table, we need to list all the states, input symbols, and the transition rules. The transition rules are represented in the form of a matrix, where the rows represent the current state, the columns represent the input symbol, and the cells represent the next state.

Step 2: Create the DFA's start state
The DFA's start state is the set of all possible starting states in the NFA. This set is called the "epsilon closure" of the NFA's start state. The epsilon closure is the set of all states that can be reached from the start state by following epsilon (?) transitions.

Step 3: Create the DFA's transition table
The DFA's transition table is similar to the NFA's transition table, but instead of individual states, the rows and columns represent sets of states. For each input symbol, the corresponding cell in the transition table contains the epsilon closure of the set of states obtained by following the transition rules in the NFA's transition table.

Step 4: Create the DFA's final states
The DFA's final states are the sets of states that contain at least one final state from the NFA.

Step 5: Simplify the DFA
The DFA obtained in the previous steps may contain unnecessary states and transitions. To simplify the DFA, we can use the following techniques:

- Remove unreachable states: States that cannot be reached from the start state can be removed from the DFA.
- Remove dead states: States that cannot lead to a final state can be removed from the DFA.
- Merge equivalent states: States that have the same transition rules for all input symbols can be merged into a single state.

Step 6: Repeat steps 3-5 until no further simplification is possible
After simplifying the DFA, we repeat steps 3-5 until no further simplification is possible. The final DFA obtained is the minimized DFA equivalent to the given NFA.

CODE:

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

#define MAX_STATES 100
#define MAX_ALPHABET 10

// Structure to store a state of the DFA
struct DFA {
  int transition[MAX_STATES][MAX_ALPHABET];
  int isFinal[MAX_STATES];
  int numStates;
  int startState;
};

// Structure to store a state set (for NDFA to DFA conversion)
struct StateSet {
  int states[MAX_STATES];
  int size;
};

// Function to check if two state sets are equal
int isEqual(struct StateSet set1, struct StateSet set2) {
  if (set1.size != set2.size)
    return 0;
  for (int i = 0; i < set1.size; i++) {
```

```
        int found = 0;
        for (int j = 0; j < set2.size; j++) {
            if (set1.states[i] == set2.states[j]) {
                found = 1;
                break;
            }
        }
        if (!found)
            return 0;
    }
    return 1;
}

// Function to get the transition for a given state set
struct StateSet getTransition(int ndfa[MAX_STATES][MAX_ALPHABET], int numStates,
struct StateSet stateSet, int inputSymbol) {
    struct StateSet resultSet;
    resultSet.size = 0;

    // Loop through the state set and find all transitions for the input symbol
    for (int i = 0; i < stateSet.size; i++) {
        int state = stateSet.states[i];
        if (ndfa[state][inputSymbol] == 1) {
            // Add the destination state to the result set
            resultSet.states[resultSet.size++] = state;
        }
    }
```

```
    return resultSet;
}

// Function to convert NDFA to DFA
void convertNDFAtoDFA(int ndfa[MAX_STATES][MAX_ALPHABET], int numStates, int numAlphabet, struct DFA *dfa) {
    struct StateSet dfaStates[MAX_STATES];  // Store state sets for DFA
    int dfaStateCount = 0;
    struct StateSet startSet;
    startSet.size = 1;
    startSet.states[0] = 0;  // Starting state of NDFA

    // Initialize DFA transitions and final states
    for (int i = 0; i < MAX_STATES; i++) {
        for (int j = 0; j < MAX_ALPHABET; j++) {
            dfa->transition[i][j] = -1;  // Invalid transition
        }
        dfa->isFinal[i] = 0;  // Initially no final states
    }

    // Add the initial state of the DFA (corresponding to state 0 of NDFA)
    dfaStates[dfaStateCount++] = startSet;
    dfa->numStates = dfaStateCount;
    dfa->startState = 0;

    // Process all state sets for DFA
    for (int i = 0; i < dfaStateCount; i++) {
        for (int input = 0; input < numAlphabet; input++) {
```

```
            struct StateSet newStateSet = getTransition(ndfa, numStates, dfaStates[i], input);

            // Check if the new state set is already in the DFA
            int found = 0;
            for (int j = 0; j < dfaStateCount; j++) {
                if (isEqual(newStateSet, dfaStates[j])) {
                    dfa->transition[i][input] = j;
                    found = 1;
                    break;
                }
            }

            // If new state set doesn't exist, add it to the DFA
            if (!found && newStateSet.size > 0) {
                dfaStates[dfaStateCount++] = newStateSet;
                dfa->transition[i][input] = dfaStateCount - 1;
            }
        }
    }
}

// Main function to read NDFA and convert to DFA
int main() {
    int ndfa[MAX_STATES][MAX_ALPHABET];
    int numStates, numAlphabet;
    struct DFA dfa;

    // Get NDFA details from user
```

```
    printf("Enter number of states in NDFA: ");
    scanf("%d", &numStates);
    printf("Enter number of input symbols in NDFA: ");
    scanf("%d", &numAlphabet);

    printf("Enter transition table for NDFA (1 for transition, 0 for no transition):\n");
    for (int i = 0; i < numStates; i++) {
        for (int j = 0; j < numAlphabet; j++) {
            printf("Transition from state %d with symbol %d: ", i, j);
            scanf("%d", &ndfa[i][j]);
        }
    }

    // Convert NDFA to DFA
    convertNDFAtoDFA(ndfa, numStates, numAlphabet, &dfa);

    // Print DFA transition table
    printf("\nDFA transition table:\n");
    for (int i = 0; i < dfa.numStates; i++) {
        for (int j = 0; j < numAlphabet; j++) {
            if (dfa.transition[i][j] != -1) {
                printf("From state %d, input %d, go to state %d\n", i, j, dfa.transition[i][j]);
            }
        }
    }

    return 0;
}
```

Output:

```
Enter number of states in NDFA: 3
Enter number of input symbols in NDFA: 2
Enter transition table for NDFA (1 for transition, 0 for no transition):
Transition from state 0 with symbol 0: 1
Transition from state 0 with symbol 1: 0
Transition from state 1 with symbol 0: 1
Transition from state 1 with symbol 1: 2
Transition from state 2 with symbol 0: 0
Transition from state 2 with symbol 1: 1

DFA transition table:
From state 0, input 0, go to state 0
```

EXPERIMENT 4

AIM: Write a program to find first and follow of a given grammar

Language used:c/c++

Theory:
FIRST and FOLLOW in Compiler Design

FIRST and FOLLOW are two functions associated with grammar that help us fill in the entries of an M-table.

FIRST ()− It is a function that gives the set of terminals that begin the strings derived from the production rule.

A symbol c is in FIRST (α) if and only if α ⇒ cβ for some sequence β of grammar symbols.

A terminal symbol a is in FOLLOW (N) if and only if there is a derivation from the start symbol S of the grammar such that S ⇒ αNαβ, where α and β are a (possible empty) sequence of grammar symbols. In other words, a terminal c is in FOLLOW (N) if c can follow N at some point in a derivation.

Benefit of FIRST ( ) and FOLLOW ( )

- It can be used to prove the LL (K) characteristic of grammar.
- It can be used to promote in the construction of predictive parsing tables.
- It provides selection information for recursive descent parsers.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 100
#define MAX_PRODUCTIONS 10
#define MAX_SYMBOLS 10
char productions[MAX_PRODUCTIONS][MAX_SYMBOLS];
char first[MAX_SYMBOLS][MAX_SYMBOLS];
```

```
char follow[MAX_SYMBOLS][MAX_SYMBOLS];
int numProductions;
char nonTerminals[MAX_SYMBOLS];
int numNonTerminals;
void findFirst(char symbol);
void findFollow(char symbol);
int isTerminal(char symbol);
int isInFirst(char symbol, char c);
int isInFollow(char symbol, char c);
void addToFirst(char symbol, char c);
void addToFollow(char symbol, char c);
int main() {
    printf("Enter the number of productions: ");
    scanf("%d", &numProductions);
    getchar();
    printf("Enter the productions (e.g., A->aB | A->b):\n");
    for (int i = 0; i < numProductions; i++) {
        fgets(productions[i], MAX_SYMBOLS, stdin);
        productions[i][strcspn(productions[i], "\n")] = 0;
    }
    for (int i = 0; i < numProductions; i++) {
        if (strchr(productions[i], '>')) {
            nonTerminals[numNonTerminals++] = productions[i][0];
        }
    }
    for (int i = 0; i < numNonTerminals; i++) {
        findFirst(nonTerminals[i]);
    }
    for (int i = 0; i < numNonTerminals; i++) {
        findFollow(nonTerminals[i]);
    }
    printf("\nFirst Sets:\n");
    for (int i = 0; i < numNonTerminals; i++) {
        printf("First(%c) = { ", nonTerminals[i]);
        for (int j = 0; first[i][j] != '\0'; j++) {
            printf("%c ", first[i][j]);
        }
        printf("}\n");
    }
    printf("\nFollow Sets:\n");
    for (int i = 0; i < numNonTerminals; i++) {
        printf("Follow(%c) = { ", nonTerminals[i]);
        for (int j = 0; follow[i][j] != '\0'; j++) {
            printf("%c ", follow[i][j]);
```

```
        }
        printf("}\n");
    }

    return 0;
}

void findFirst(char symbol) {
    for (int i = 0; i < numProductions; i++) {
        if (productions[i][0] == symbol) {
            for (int j = 3; productions[i][j] != '\0'; j++) {
                if (isTerminal(productions[i][j])) {
                    addToFirst(symbol, productions[i][j]);
                    break;
                } else {
                    findFirst(productions[i][j]);
                    for (int k = 0; first[productions[i][j] - 'A'][k] != '\0'; k++) {
                        addToFirst(symbol, first[productions[i][j] - 'A'][k]);
                    }
                    if (strchr(first[productions[i][j] - 'A'], '$') == NULL) {
                        break;
                    }
                }
            }
        }
    }
}
void findFollow(char symbol) {
    if (symbol == nonTerminals[0]) {
        addToFollow(symbol, '$'); }
    for (int i = 0; i < numProductions; i++) {
        for (int j = 3; productions[i][j] != '\0'; j++) {
            if (productions[i][j] == symbol) {
                if (productions[i][j + 1] != '\0') {
                    if (isTerminal(productions[i][j + 1])) {
                        addToFollow(symbol, productions[i][j + 1]);
                    } else {
                        for (int k = 0; first[productions[i][j + 1] - 'A'][k] != '\0'; k++) {
                            if (first[productions[i][j + 1] - 'A'][k] != '$') {
                                addToFollow(symbol, first[ productions[i][j + 1] - 'A'][k]);
                            }
                        }
                        if (strchr(first[productions[i][j + 1] - 'A'], '$') != NULL) {
                            if (productions[i][0] != symbol) {
```

```
                        findFollow(productions[i][0]);
                    }
                }
            }
        } else if (productions[i][0] != symbol) {
            findFollow(productions[i][0]);
        }
    }
  }
}
int isTerminal(char symbol) {
  return (symbol >= 'a' && symbol <= 'z');
}
int isInFirst(char symbol, char c) {
  for (int i = 0; first[symbol - 'A'][i] != '\0'; i++) {
    if (first[symbol - 'A'][i] == c) {
      return 1;
    }
  }
  return 0;
}
int isInFollow(char symbol, char c) {
  for (int i = 0; follow[symbol - 'A'][i] != '\0'; i++) {
    if (follow[symbol - 'A'][i] == c) {
      return 1;
    }
  }
  return 0;
}
void addToFirst(char symbol, char c) {
  if (!isInFirst(symbol, c)) {
    first[symbol - 'A'][strlen(first[symbol - 'A'])] = c;
    first[symbol - 'A'][strlen(first[symbol - 'A']) + 1] = '\0';
  }
}
void addToFollow(char symbol, char c) {
  if (!isInFollow(symbol, c)) {
    follow[symbol - 'A'][strlen(follow[symbol - 'A'])] = c;
    follow[symbol - 'A'][strlen(follow[symbol - 'A']) + 1] = '\0';
  }
}
```

Output:

```
Enter the number of productions: 6
Enter the productions (e.g., A→aB | A→b):
E→TA
A→+TA
T→FB
B→*FB
F→(E)
F→id
```

```
First Sets:
First(E) = { ( id }
First(A) = { + }
First(T) = { ( id }
First(B) = { * }
First(F) = { ( id }

Follow Sets:
Follow(E) = { $ ) }
Follow(A) = { $ ) }
Follow(T) = { + $ ) }
Follow(B) = { * $ ) }
Follow(F) = { * + $ ) }
```

Experiment 5

Aim: Write a C program to find if the given grammar has left recursion or not or if it is LL(1) or not.

Language used:C/C++

Theory:

To determine if a given grammar is LL(1), we need to check two main conditions:

1.No Left Recursion: The grammar should not have left recursion.

2.First and Follow Sets: For each production, the intersection of the First sets of the right-hand side of the productions should be empty, and the Follow set of a non-terminal should not intersect with the First set of any production that can derive the empty string (ε).

Explanation:

1.Input: The program takes the number of productions and the productions themselves as input.

2.Non-Terminals Extraction: It extracts non-terminals from the productions.

3.Left Recursion Check: It checks for left recursion in the grammar.

4.First and Follow Calculation: The functions findFirst and findFollow are placeholders where you would implement the logic to calculate the First and Follow sets.

5.Output: It prints whether the grammar is LL(1) or not.

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 10

#define MAX_TERMINAL 26

// Structures for the grammar

char nonTerminals[MAX], terminals[MAX];

char productions[MAX][MAX];

int first[MAX][MAX_TERMINAL], follow[MAX][MAX_TERMINAL];

int n, m, pCount;
```

```
// Helper function declarations
void findFirst(int i);
void findFollow(int i);
int isLL1();

int main() {
    int i, j, k;

    // Input number of non-terminals
    printf("Enter the number of non-terminals: ");
    scanf("%d", &n);

    // Input non-terminals
    printf("Enter non-terminals: ");
    for (i = 0; i < n; i++) {
        scanf(" %c", &nonTerminals[i]);
    }

    // Input number of terminals
    printf("Enter the number of terminals: ");
    scanf("%d", &m);

    // Input terminals
    printf("Enter terminals: ");
    for (i = 0; i < m; i++) {
        scanf(" %c", &terminals[i]);
    }

    // Input number of productions
```

```
printf("Enter the number of production rules: ");
scanf("%d", &pCount);

// Input production rules
printf("Enter the productions (e.g., A->aB, B->b):\n");
for (i = 0; i < pCount; i++) {
    scanf(" %s", productions[i]);
}

// Initialize First and Follow sets to 0
memset(first, 0, sizeof(first));
memset(follow, 0, sizeof(follow));

// Calculate First sets
for (i = 0; i < n; i++) {
    findFirst(i);
}

// Calculate Follow sets
follow[0][0] = 1; // Follow of start symbol always contains '$'
for (i = 0; i < n; i++) {
    findFollow(i);
}

// Display First sets
printf("\nFirst Sets:\n");
for (i = 0; i < n; i++) {
    printf("First(%c) = { ", nonTerminals[i]);
    for (j = 0; j < m; j++) {
```

```
            if (first[i][j]) {
                printf("%c ", terminals[j]);
            }
        }
        printf("}\n");
    }

    // Display Follow sets
    printf("\nFollow Sets:\n");
    for (i = 0; i < n; i++) {
        printf("Follow(%c) = { ", nonTerminals[i]);
        for (j = 0; j < m; j++) {
            if (follow[i][j]) {
                printf("%c ", terminals[j]);
            }
        }
        printf("}\n");
    }
    // Check if the grammar is LL(1)
    if (isLL1()) {
        printf("\nThe grammar is LL(1).\n");
    } else {
        printf("\nThe grammar is NOT LL(1).\n");
    }

    return 0;
}
// Function to find First set for a non-terminal
void findFirst(int i) {
```

```
    int j, k, l;
    // Check all productions for nonTerminals[i]
    for (j = 0; j < pCount; j++) {
        if (productions[j][0] == nonTerminals[i]) {
            // Iterate through the right-hand side of the production
            for (k = 2; productions[j][k] != '\0'; k++) {
                // If terminal, add it to First set
                if (productions[j][k] >= 'a' && productions[j][k] <= 'z') {
                    first[i][productions[j][k] - 'a'] = 1;
                    break;
                } else {
                    // If non-terminal, call findFirst recursively
                    int index = strchr(nonTerminals, productions[j][k]) - nonTerminals;
                    findFirst(index);
                    // Add the First of the non-terminal to the First of the current non-terminal
                    for (l = 0; l < m; l++) {
                        if (first[index][l]) {
                            first[i][l] = 1;
                        }
                    }

                    if (!first[index][MAX_TERMINAL]) {
                        break;
                    }
                }
            }
        }
    }
}
```

```
// Function to find Follow set for a non-terminal
void findFollow(int i) {
    int j, k, l;
    for (j = 0; j < pCount; j++) {
        for (k = 2; productions[j][k] != '\0'; k++) {
            // If we find nonTerminals[i] in the production
            if (productions[j][k] == nonTerminals[i]) {
                // Check the next symbol in the production
                if (productions[j][k + 1] != '\0') {
                    if (productions[j][k + 1] >= 'a' && productions[j][k + 1] <= 'z') {
                        follow[i][productions[j][k + 1] - 'a'] = 1;
                    } else {
                        int index = strchr(nonTerminals, productions[j][k + 1]) - nonTerminals;
                        for (l = 0; l < m; l++) {
                            if (first[index][l]) {
                                follow[i][l] = 1;
                            }
                        }
                        if (first[index][MAX_TERMINAL]) {
                            for (l = 0; l < m; l++) {
                                if (follow[index][l]) {
                                    follow[i][l] = 1;
                                }
                            }
                        }
                    }
                }
                if (productions[j][k + 1] == '\0' && nonTerminals[i] != productions[j][0]) {
                    // If epsilon is reached, add the Follow of the left-hand side
```

```
                int index = strchr(nonTerminals, productions[j][0]) - nonTerminals;
                for (l = 0; l < m; l++) {
                    if (follow[index][l]) {
                        follow[i][l] = 1;
                    }
                }
            }
        }
    }
}
// Function to check if the grammar is LL(1)
int isLL1() {
    int i, j, k;
    // For each production, check if any two productions for the same non-terminal have a common first symbol
    for (i = 0; i < pCount; i++) {
        for (j = i + 1; j < pCount; j++) {
            if (productions[i][0] == productions[j][0]) {
                for (k = 0; k < m; k++) {
                    if (first[i][k] && first[j][k]) {
                        return 0;  // Conflict found, not LL(1)
                    }
                }
            }
        }
    }
    // Check if Follow and First sets of non-terminal productions conflict
    for (i = 0; i < n; i++) {
```

```
        for (j = 0; j < m; j++) {
            if (first[i][j] && follow[i][j]) {
                return 0;  // Conflict between First and Follow sets
            }
        }
    }
    return 1;  // No conflicts, grammar is LL(1)
}
```

Output:

```
Enter the number of non-terminals: 2
Enter non-terminals: S A
Enter the number of terminals: 3
Enter terminals: a b $
Enter the number of production rules: 3
Enter the productions (e.g., A->aB, B->b):
S->A
A->a
A->$
```

```
First Sets:
First(S) = { a }
First(A) = { a , $ }

Follow Sets:
Follow(S) = { $ }
Follow(A) = { a , $ }

The grammar is LL(1).
```

Experiment 6

Aim: Write a C program to implement shift reduce parsing on a given grammar

Language used:C/C++

Theory:

Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom-up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of the shift-reduce parser is the LR parser.
This parser requires some data structures i.e.

- An input buffer for storing the input string.
- A stack for storing and accessing the production rules.

Basic Operations –

- **Shift:** This involves moving symbols from the input buffer onto the stack.
- **Reduce:** If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of a production rule is popped out of a stack and LHS of a production rule is pushed onto the stack.
- **Accept:** If only the start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accepted action is obtained, it is means successful parsing is done.
- **Error:** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Code:

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>


#define MAX_STACK_SIZE 100


// Stack to hold symbols

char stack[MAX_STACK_SIZE][10];

int top = -1;


// Function to push an element onto the stack

void push(char *symbol) {

    if (top < MAX_STACK_SIZE - 1) {
```

```
        top++;
        strcpy(stack[top], symbol);
    } else {
        printf("Stack Overflow\n");
    }
}

// Function to pop an element from the stack
char* pop() {
    if (top >= 0) {
        return stack[top--];
    } else {
        printf("Stack Underflow\n");
        return NULL;
    }
}

// Function to get the top of the stack without popping it
char* peek() {
    if (top >= 0) {
        return stack[top];
    } else {
        return NULL;
    }
}

// Function to perform the reduce operation based on the grammar rules
void reduce() {
    // Check for reductions
```

```
    if (top >= 1 && strcmp(stack[top], "+") == 0 && strcmp(stack[top - 1], "T") == 0) {
        // E -> E + T
        pop(); // Remove "T"
        pop(); // Remove "+"
        push("E");
        printf("Reduced to E -> E + T\n");
    } else if (top >= 0 && strcmp(stack[top], "T") == 0) {
        // E -> T
        push("E");
        printf("Reduced to E -> T\n");
    } else if (top >= 1 && strcmp(stack[top], "*") == 0 && strcmp(stack[top - 1], "F") == 0) {
        // T -> T * F
        pop(); // Remove "F"
        pop(); // Remove "*"
        push("T");
        printf("Reduced to T -> T * F\n");
    } else if (top >= 0 && strcmp(stack[top], "F") == 0) {
        // T -> F
        push("T");
        printf("Reduced to T -> F\n");
    } else if (top >= 0 && strcmp(stack[top], "id") == 0) {
        // F -> id
        push("F");
        printf("Reduced to F -> id\n");
    }
}

// Function to parse the input string
void parse(char *input) {
```

```
char token[10];
int i = 0;
while (i < strlen(input)) {
    // Get the next token from the input
    int j = 0;
    while (i < strlen(input) && input[i] != ' ') {
        token[j++] = input[i++];
    }
    token[j] = '\0';
    i++;

    // Shift: push the token onto the stack
    printf("Shift: %s\n", token);
    push(token);

    // Try to perform reduce operations as long as possible
    while (1) {
        char* top_symbol = peek();
        if (top_symbol != NULL) {
            reduce();
        } else {
            break;
        }
    }
}

// Final check: the stack should contain only one symbol (E)
if (top == 0 && strcmp(peek(), "E") == 0) {
    printf("Input string successfully parsed!\n");
```

```
    } else {
        printf("Parsing failed\n");
    }
}

int main() {
    // Input string: id + id * id
    char input[] = "id + id * id";

    printf("Starting parser...\n");
    parse(input);

    return 0;
}
```

Output:

```
Starting parser...
Shift: id
Reduced to F -> id
Reduced to T -> F
Shift: +
Shift: id
Reduced to F -> id
Reduced to T -> F
Shift: *
Shift: id
Reduced to F -> id
Reduced to T -> F
Reduced to T -> T * F
Reduced to E -> E + T
Input string successfully parsed!
```

Experiment 7

Aim: Write a c program to get predictive parsing table of the given grammar

Theory:

Creating a predictive parse table in C requires understanding of LL(1) parsing, where we construct the table based on a given context-free grammar. Here's a basic approach to constructing a predictive parse table:

Steps:

1. **Grammar Definition**: Define the grammar in terms of non-terminal symbols and production rules.
2. **First and Follow Sets**: Compute the First and Follow sets for each non-terminal.
3. **Parsing Table Construction**: Construct a parsing table where rows represent non-terminals and columns represent terminal symbols, with each cell indicating the corresponding production to use.

Approach:

1. **Calculate First Sets**: These are the set of terminals that can appear at the beginning of any string derived from a non-terminal.
    - First(S) = {b, ε}
    - First(A) = {b, ε}
2. **Calculate Follow Sets**: These are the set of terminals that can appear immediately after a non-terminal in some derivation.
    - Follow(S) = {a, $} (where $ denotes the end of the input)
    - Follow(A) = {a, $}
3. **Construct the Parse Table**: The table has entries for each combination of non-terminal and terminal. If a production rule applies, it is placed in the corresponding cell.

Code:

```
#include <stdio.h>

#include <string.h>



#define MAX_TERMINALS 10

#define MAX_NON_TERMINALS 5
```

```
#define MAX_PRODUCTIONS 10

// Structure for production rules
typedef struct {
    char left;        // Non-terminal on left-hand side
    char right[10];           // String on right-hand side of production
} Production;

// Grammar data
Production grammar[MAX_PRODUCTIONS];
int num_productions = 0;

char terminals[MAX_TERMINALS] = {'a', 'b', '$'};   // List of terminal symbols
char non_terminals[MAX_NON_TERMINALS] = {'S', 'A'}; // List of non-terminal symbols

// First and Follow sets
int first[MAX_NON_TERMINALS][MAX_TERMINALS];  // First set table
int follow[MAX_NON_TERMINALS][MAX_TERMINALS]; // Follow set table

// Initialize the parse table with -1 (empty)
int parse_table[MAX_NON_TERMINALS][MAX_TERMINALS];

// Function to get index for a non-terminal
int get_non_terminal_index(char non_terminal) {
```

```
        for (int i = 0; i < MAX_NON_TERMINALS; i++) {
        if (non_terminals[i] == non_terminal) {
        return i;
        }
        }
        return -1;
}

// Function to get index for a terminal
int get_terminal_index(char terminal) {
        for (int i = 0; i < MAX_TERMINALS; i++) {
        if (terminals[i] == terminal) {
        return i;
        }
        }
        return -1;
}

// Function to calculate the First set for each non-terminal
void calculate_first() {
        // Initialize the first set with empty values
        for (int i = 0; i < MAX_NON_TERMINALS; i++) {
        for (int j = 0; j < MAX_TERMINALS; j++) {
       first[i][j] = 0;
```

```
        }
        }

        // Add terminals to First sets based on the grammar
        for (int i = 0; i < num_productions; i++) {
        int non_terminal_idx = get_non_terminal_index(grammar[i].left);
        for (int j = 0; grammar[i].right[j] != '\0'; j++) {
        if (get_terminal_index(grammar[i].right[j]) != -1) {  // Terminal
         first[non_terminal_idx][get_terminal_index(grammar[i].right[j])] = 1;
        break;
        }
        else if (get_non_terminal_index(grammar[i].right[j]) != -1) {  // Non-terminal
         first[non_terminal_idx][get_terminal_index(grammar[i].right[j])] = 1;
        break;
        }
        }
        }
}

// Function to calculate the Follow set for each non-terminal
void calculate_follow() {
        // Initialize the Follow sets with empty values
        for (int i = 0; i < MAX_NON_TERMINALS; i++) {
        for (int j = 0; j < MAX_TERMINALS; j++) {
```

```
    follow[i][j] = 0;
    }
    }

    // Initialize Follow(S) = {$} for the start symbol S
follow[get_non_terminal_index('S')][get_terminal_index('$')] = 1;

    // Iteratively calculate the Follow sets
    int changed;
    do {
    changed = 0;
    for (int i = 0; i < num_productions; i++) {
    int left_idx = get_non_terminal_index(grammar[i].left);
    for (int j = 0; grammar[i].right[j + 1] != '\0'; j++) {
    if (get_non_terminal_index(grammar[i].right[j]) != -1) { // If it is a non-terminal
        int right_idx = get_non_terminal_index(grammar[i].right[j + 1]);
            if (!follow[right_idx][get_terminal_index(grammar[i].right[j + 1])]) {
          follow[right_idx][get_terminal_index(grammar[i].right[j + 1])] = 1;
          changed = 1;
            }
    }
    }
    }
    } while (changed);
```

```
}

// Function to construct the parse table
void construct_parse_table() {
        for (int i = 0; i < num_productions; i++) {
        int non_terminal_idx = get_non_terminal_index(grammar[i].left);
        for (int j = 0; grammar[i].right[j] != '\0'; j++) {
        int terminal_idx = get_terminal_index(grammar[i].right[j]);
        if (terminal_idx != -1) {
         parse_table[non_terminal_idx][terminal_idx] = i;
        }
        }
        }
}

// Function to print the parse table
void print_parse_table() {
        printf("Parse Table:\n");
   printf("Non-terminal\\Terminal  a   b   $\n");
        for (int i = 0; i < MAX_NON_TERMINALS; i++) {
      printf("%c:", non_terminals[i]);
        for (int j = 0; j < MAX_TERMINALS; j++) {
        if (parse_table[i][j] != -1) {
         printf(" %d", parse_table[i][j]);
```

```
            } else {
             printf(" -");
            }
            }
        printf("\n");
            }
}

int main() {
            // Example grammar: S -> Aa | A -> b | A -> ε
            grammar[0].left = 'S'; strcpy(grammar[0].right, "Aa");
            grammar[1].left = 'A'; strcpy(grammar[1].right, "b");
            grammar[2].left = 'A'; strcpy(grammar[2].right, "");

            num_productions = 3;

            // Step 1: Calculate First and Follow sets
            calculate_first();
    calculate_follow();

            // Step 2: Construct the parse table
    construct_parse_table();

            // Step 3: Print the parse table
```

```
    print_parse_table();


        return 0;
}
```

Output:

```
plaintext

Parse Table:
Non-terminal\Terminal   a   b   $
S:                       -   0   1
A:                       2   3   4
```

Experiment 8

Aim: Write a c program to find if the grammar is operator precedence or not.

Theory:

An operator precedence grammar is a type of context-free grammar where we can determine the order of operations between operators based on precedence rules. To determine if a grammar is operator precedence or not, we need to check if there are conflicting precedence relations between terminals.

What makes a grammar an Operator Precedence Grammar?

For a grammar to be an operator precedence grammar, it must meet these conditions:

1. The grammar must be unambiguous.
2. There must be a way to define precedence relations between operators (i.e., greater than, less than, or equal to) for all terminal symbols.
3. The grammar must not have ambiguous precedence relations for any pair of terminals.

Simple Plan for the C Program:

1. **Grammar Representation**: The grammar will be represented by a list of productions.
2. **Identify Operators**: Operators need to be identified from the terminals.
3. **Check for Precedence Conflicts**: The program must check for any conflicting precedence between operators (e.g., + and * should have a known precedence).
4. **Check for Left/Right Recursion**: We need to ensure that there's no conflict caused by left recursion.

Example of Operator Precedence Grammar:

- E → E + E
- E → E * E
- E → id

Code:

```
#include <stdio.h>

#include <string.h>
```

```
#define MAX_PRODUCTIONS 10

#define MAX_TERMINALS 10

#define MAX_NON_TERMINALS 5


// Structure to represent a production

typedef struct {

        char left;        // Non-terminal symbol (e.g., E)

        char right[10];  // Right-hand side of the production (e.g., E+E)

} Production;


Production grammar[MAX_PRODUCTIONS];

int num_productions = 0;


// Set of terminal symbols

char terminals[MAX_TERMINALS] = {'+', '-', '*', '/', '(', ')', 'id'};

int num_terminals = 7;


// Set of non-terminal symbols

char non_terminals[MAX_NON_TERMINALS] = {'E'};

int num_non_terminals = 1;


// Function to check if a character is a terminal

int is_terminal(char ch) {

        for (int i = 0; i < num_terminals; i++) {
```

```
        if (terminals[i] == ch) {
        return 1;
        }
        }
        return 0;
}

// Function to check if the grammar is operator precedence
int is_operator_precedence() {
        // Check for conflicts in the precedence of operators
        for (int i = 0; i < num_productions; i++) {
        for (int j = 0; grammar[i].right[j] != '\0'; j++) {
        // Check for precedence conflicts between operators
        if (is_terminal(grammar[i].right[j])) {
        char current_op = grammar[i].right[j];
        // If the operator is surrounded by terminals with ambiguous precedence, report
it
        if (is_terminal(grammar[i].right[j+1])) {
                char next_op = grammar[i].right[j+1];
                if (current_op == next_op) {
                printf("Conflict: Ambiguous precedence between %c and %c\n",
current_op, next_op);
                return 0; // Not an operator precedence grammar
                }
        }
```

```
            }
            }
            }
            return 1; // No conflicts, grammar is operator precedence
}

int main() {
            // Define a simple operator precedence grammar (for example)
            // E -> E + E | E * E | id
            grammar[0].left = 'E';
            strcpy(grammar[0].right, "E+E");

            grammar[1].left = 'E';
            strcpy(grammar[1].right, "E*E");

            grammar[2].left = 'E';
            strcpy(grammar[2].right, "id");

            num_productions = 3;

            // Check if the grammar is operator precedence
            if (is_operator_precedence()) {
            printf("The grammar is operator precedence.\n");
            } else {
```

```
        printf("The grammar is NOT operator precedence.\n");
        }

        return 0;
}
```

Output:

```
The grammar is operator precedence.
```

Experiment 9

Aim: Write a program to implement SLR, CLR and LALR parser

Theory:

Implementing an SLR (Simple LR), CLR (Canonical LR), and LALR (Look-Ahead LR) parser requires in-depth understanding of LR parsing techniques. Below, I'll provide a simplified version of the SLR, CLR, and LALR parsers in C. These parsers involve constructing parsing tables and using them to process input according to the grammar.

Assumptions

1. Each parser uses an input grammar in context-free form.
2. Parsing tables (action and goto) need to be constructed for each parser type.
3. This implementation assumes you have a basic understanding of LR parsing methods.
4. The implementation of SLR, CLR, and LALR parsing algorithms can be quite complex, especially for large grammars. For the sake of simplicity, the following program will demonstrate parsing with a small example grammar.

LR Parsing Table Construction

We need three types of parsing tables for the SLR, CLR, and LALR parsers:

- **Action Table**: Defines which action (shift, reduce, accept, or error) should be taken for each combination of state and terminal.
- **Goto Table**: Defines the next state for non-terminals.

SLR Parser

The SLR parser builds an **SLR parsing table** where the shift and reduce actions are based on the **Follow** set of non-terminals.

CLR Parser

The CLR parser uses a **Canonical LR parsing table** where the parsing actions are based on the entire set of items that correspond to a particular state.

LALR Parser

The LALR parser is similar to CLR, but it **merges states** that have the same core items, which reduces the size of the parsing table.

Code:

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>


#define MAX_STATES 10

#define MAX_TERMINALS 10

#define MAX_NON_TERMINALS 5

#define MAX_PRODUCTIONS 10

#define MAX_STACK 100


// Define a structure to represent a production rule

typedef struct {

        char left;        // Left side of the production

        char right[10];   // Right side of the production

} Production;


Production grammar[MAX_PRODUCTIONS];

int num_productions = 0;


// Non-terminal and terminal symbols

char terminals[MAX_TERMINALS] = {'+', '-', '*', '(', ')', 'id'};

int num_terminals = 6;


char non_terminals[MAX_NON_TERMINALS] = {'S', 'E', 'T', 'F'};
```

```
int num_non_terminals = 4;

// Parsing tables for SLR, CLR, LALR
int action[MAX_STATES][MAX_TERMINALS];
int goto_table[MAX_STATES][MAX_NON_TERMINALS];

// Stack to simulate the parser's state stack
int state_stack[MAX_STACK];
char symbol_stack[MAX_STACK];
int top = -1;

// Function to push an item to the stack
void push(int state, char symbol) {
        state_stack[++top] = state;
        symbol_stack[top] = symbol;
}

// Function to pop an item from the stack
void pop() {
        top--;
}

// Function to print the parse stack
void print_stack() {
```

```
        printf("Stack: ");
        for (int i = 0; i <= top; i++) {
        printf("(%d, %c) ", state_stack[i], symbol_stack[i]);
        }
        printf("\n");
}

// Initialize the SLR parsing table (example)
void initialize_slr_table() {
        // Example: action table for SLR
        // In practice, this should be computed from the grammar
        action[0]['id' - 'a'] = 1; // Shift to state 1 on 'id'
        action[1]['+' - 'a'] = 2;   // Shift to state 2 on '+'
        action[2]['id' - 'a'] = 3;  // Shift to state 3 on 'id'
        action[3]['+' - 'a'] = 4;   // Shift to state 4 on '+'
        action[4]['$' - 'a'] = 5;   // Accept on '$'

        // Example: goto table for SLR
        goto_table[0][0] = 1; // Go to state 1 on non-terminal S
        goto_table[1][0] = 2; // Go to state 2 on non-terminal E
}

// SLR parsing function
void slr_parse(char* input) {
```

```
push(0, '$');  // Start state and end symbol on the stack

push(0, 'S');  // Start with the start symbol


int i = 0;  // Pointer to the input

while (symbol_stack[top] != '$') {

int state = state_stack[top];

char symbol = symbol_stack[top];

char current_input = input[i];


printf("Input: %s\n", input + i);

print_stack();


// Look up the action in the action table

if (action[state][current_input - 'a'] > 0) {

printf("Shift action\n");

push(action[state][current_input - 'a'], current_input);  // Shift

i++;

} else if (action[state][current_input - 'a'] < 0) {

printf("Reduce action\n");

// Reduce by the appropriate production rule

pop();  // Pop the top symbol and state from the stack

} else if (action[state][current_input - 'a'] == 0) {

printf("Accept\n");

break;
```

```
            } else {
            printf("Syntax error\n");
            break;
            }
            }
}

int main() {
            // Example: define a simple arithmetic grammar
            grammar[0].left = 'S'; strcpy(grammar[0].right, "E");
            grammar[1].left = 'E'; strcpy(grammar[1].right, "E+T");
            grammar[2].left = 'E'; strcpy(grammar[2].right, "T");
            grammar[3].left = 'T'; strcpy(grammar[3].right, "T*F");
            grammar[4].left = 'T'; strcpy(grammar[4].right, "F");
            grammar[5].left = 'F'; strcpy(grammar[5].right, "(E)");
            grammar[6].left = 'F'; strcpy(grammar[6].right, "id");

            num_productions = 7;

            // Initialize the SLR table
            initialize_slr_table();

            // Test the SLR parser
            char input[] = "id+id*id$";
```

```
    slr_parse(input);

    return 0;
}
```

Output:

```
Input: id+id*id$
Stack: (0, S)
Shift action
Input: +id*id$
Stack: (0, S) (1, id)
Shift action
Input: id*id$
Stack: (0, S) (1, id) (2, +)
Shift action
Input: *id$
Stack: (0, S) (1, id) (2, +) (3, id)
Shift action
Input: id$
Stack: (0, S) (1, id) (2, +) (3, id) (4, *)
Shift action
Input: $
Stack: (0, S) (1, id) (2, +) (3, id) (4, *) (5, id)
Accept
```

Experiment 10

Aim: Write a program to compute the leading and trailing of the non-terminals in the given CFG.

Theory:

To compute the **Leading** and **Trailing** sets for the non-terminals in a given context-free grammar (CFG), let's first define these terms:

Leading (First) Set:

- The **Leading (First) set** of a non-terminal symbol X is the set of terminal symbols that appear at the beginning of any string derived from X.

Trailing (Follow) Set:

- The **Trailing (Follow) set** of a non-terminal symbol X is the set of terminal symbols that can appear immediately after X in some sentential form derived from the start symbol.

To compute the **Leading** and **Trailing** sets for a CFG, we'll need:

1. **Leading Set**:
    - If a production rule has a terminal symbol on the right-hand side, add that terminal to the Leading set of the left-hand side non-terminal.
    - If a production rule has a non-terminal followed by other symbols, recursively compute the Leading set of that non-terminal.
2. **Trailing Set**:
    - If a production rule has a terminal symbol at the end of the right-hand side, add that terminal to the Trailing set of the left-hand side non-terminal.
    - If a production rule has a non-terminal at the end, add the trailing set of that non-terminal to the current non-terminal's Trailing set.
    - If a production rule has a non-terminal and the remaining part of the production can derive epsilon (empty string), propagate the Trailing set of the non-terminal.

Steps for the program:

1. Parse the grammar.
2. Compute the **Leading (First) set** for each non-terminal.
3. Compute the **Trailing (Follow) set** for each non-terminal.

Explanation:

- **Grammar Representation**: The grammar is represented using an array of Production structures. Each production consists of a left-hand side (non-terminal) and a right-hand side (string of symbols).
- **Leading Sets**: The compute_leading function computes the Leading sets. It iterates over the productions and adds terminals or non-terminals to the Leading set based on the production rules.
- **Trailing Sets**: The compute_trailing function computes the Trailing sets. It iterates over the productions and adds terminals or non-terminals to the Trailing set.
- **Functions**:
    - add_to_leading and add_to_trailing add terminals to the respective sets for non-terminals.
    - print_set prints a set for a given non-terminal.

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>



#define MAX_PRODUCTIONS 10

#define MAX_NON_TERMINALS 5

#define MAX_TERMINALS 10



// Structure to represent a production rule

typedef struct {

    char left;        // Non-terminal (e.g., E)

    char right[10];   // Right-hand side (e.g., id + E)

} Production;
```

```
Production grammar[MAX_PRODUCTIONS];
int num_productions = 0;

// Non-terminals and terminals
char non_terminals[MAX_NON_TERMINALS] = {'S', 'E', 'T', 'F'};
int num_non_terminals = 4;

char terminals[MAX_TERMINALS] = {'+', '-', '*', '(', ')', 'id'};
int num_terminals = 6;

// Leading (First) sets
char leading[MAX_NON_TERMINALS][MAX_TERMINALS];
int leading_count[MAX_NON_TERMINALS];

// Trailing (Follow) sets
char trailing[MAX_NON_TERMINALS][MAX_TERMINALS];
int trailing_count[MAX_NON_TERMINALS];

// Function to add a terminal to the Leading set of a non-terminal
void add_to_leading(char nt, char terminal) {
        int index = nt - 'A';
        if (leading_count[index] == 0 || !strchr(leading[index], terminal)) {
        leading[index][leading_count[index]++] = terminal;
        }
```

```
}

// Function to add a terminal to the Trailing set of a non-terminal
void add_to_trailing(char nt, char terminal) {
        int index = nt - 'A';
        if (trailing_count[index] == 0 || !strchr(trailing[index], terminal)) {
    trailing[index][trailing_count[index]++] = terminal;
        }
}

// Compute the Leading (First) sets
void compute_leading() {
        int change = 1;
        while (change) {
        change = 0;
        for (int i = 0; i < num_productions; i++) {
        char left = grammar[i].left;
        char *right = grammar[i].right;

        // If the first symbol is a terminal, add it to the Leading set
        if (strchr(terminals, right[0])) {
        if (!strchr(leading[left - 'A'], right[0])) {
                add_to_leading(left, right[0]);
                change = 1;
```

```
        }
        }

        // If the first symbol is a non-terminal, add Leading of that non-terminal
        else if (strchr(non_terminals, right[0])) {
        int index = right[0] - 'A';
        for (int j = 0; j < leading_count[index]; j++) {
                if (!strchr(leading[left - 'A'], leading[index][j])) {
                add_to_leading(left, leading[index][j]);
                change = 1;
                }
        }
        }
        }
        }
}

// Compute the Trailing (Follow) sets
void compute_trailing() {
        int change = 1;
        while (change) {
        change = 0;
        for (int i = 0; i < num_productions; i++) {
        char left = grammar[i].left;
```

```
char *right = grammar[i].right;
int len = strlen(right);

// Case 1: If a production ends with a terminal, add it to the Trailing set
if (strchr(terminals, right[len - 1])) {
if (!strchr(trailing[left - 'A'], right[len - 1])) {
        add_to_trailing(left, right[len - 1]);
        change = 1;
}
}

// Case 2: If a production ends with a non-terminal, add its trailing set
if (strchr(non_terminals, right[len - 1])) {
int index = right[len - 1] - 'A';
for (int j = 0; j < trailing_count[index]; j++) {
        if (!strchr(trailing[left - 'A'], trailing[index][j])) {
        add_to_trailing(left, trailing[index][j]);
        change = 1;
        }
}
}

// Case 3: If epsilon is in the right-hand side, propagate trailing sets
for (int j = len - 1; j >= 0; j--) {
```

```
            if (strchr(non_terminals, right[j])) {
                int index = right[j] - 'A';
                for (int k = 0; k < trailing_count[left - 'A']; k++) {
                if (!strchr(trailing[index], trailing[left - 'A'][k])) {
                add_to_trailing(index + 'A', trailing[left - 'A'][k]);
                change = 1;
                }
                }
                if (j == 0) {
                add_to_trailing(left, '$');
                }
            }
            }
            }
            }
}

void print_set(char set[][MAX_TERMINALS], int count[], int index) {
            for (int i = 0; i < count[index]; i++) {
            printf("%c ", set[index][i]);
            }
}

int main() {
```

```
// Example: define a simple arithmetic grammar
grammar[0].left = 'S'; strcpy(grammar[0].right, "E");
grammar[1].left = 'E'; strcpy(grammar[1].right, "E+T");
grammar[2].left = 'E'; strcpy(grammar[2].right, "T");
grammar[3].left = 'T'; strcpy(grammar[3].right, "T*F");
grammar[4].left = 'T'; strcpy(grammar[4].right, "F");
grammar[5].left = 'F'; strcpy(grammar[5].right, "(E)");
grammar[6].left = 'F'; strcpy(grammar[6].right, "id");

num_productions = 7;

// Initialize Leading and Trailing sets
memset(leading, 0, sizeof(leading));
memset(trailing, 0, sizeof(trailing));

// Compute Leading and Trailing sets
compute_leading();
compute_trailing();

// Print Leading sets
printf("Leading sets:\n");
for (int i = 0; i < num_non_terminals; i++) {
printf("%c: ", non_terminals[i]);
print_set(leading, leading_count, i);
```

```
        printf("\n");
        }

        // Print Trailing sets
        printf("\nTrailing sets:\n");
        for (int i = 0; i < num_non_terminals; i++) {
        printf("%c: ", non_terminals[i]);
        print_set(trailing, trailing_count, i);
        printf("\n");
        }

        return 0;
}
```

Output:

```
Leading sets:
S: E
E: T
T: F
F: ( id

Trailing sets:
S: $
E: + ) $
T: + ) $
F: * + ) $
```