Experiment - 9

Aim:-

Write a program to construct the LL(1) parsing table for the given grammar.

Procedure:-

In compiler design, **predictive parsing** is a **top-down parsing technique** that uses lookahead symbols to predict which production rule to apply.

A grammar is said to be **LL(1)** if it can be parsed from **Left-to-right**, producing a **Leftmost derivation** using **1 lookahead symbol.**

To build a **predictive parser**, we first need to compute:

- FIRST sets
- FOLLOW sets
- And then use them to construct the **LL(1) parsing table**

1. FIRST Set
   - The **FIRST set** of a non-terminal is the set of terminals that begin the strings derivable from that non-terminal.
   - If a non-terminal can derive ε (epsilon), then ε is included in its FIRST set.
   - Example:

A → aB | #
FIRST(A) = { a, # }

2. FOLLOW Set

- The **FOLLOW set** of a non-terminal is the set of terminals that can appear **immediately to the right** of that non-terminal in some derivation.
- The symbol **$** (end marker) is always in the FOLLOW of the start symbol.
- Example

S → AB
A → a | #
B → b

FOLLOW(A) = { b }
FOLLOW(S) = { $ }

3. LL(1) Parsing Table Construction Rules

For each production A → α:

- For every terminal **a in FIRST(α)**, place **A → α in M[A, a].**
- If **Epsilon ∈ FIRST(α)**, for every terminal b in FOLLOW(A), place **A → α in M[A, b].**
- If **$ ∈ FOLLOW(A),** place **A → α in M[A, $].**

Algorithm for FIRST Set:

1. Start with all FIRST sets empty.
2. For every production of the form A → a:
   - For each symbol X in a (from left to right):
     - If X is a terminal, add X to FIRST(A) and stop.
     - If X is a non-terminal, add all symbols of FIRST(X) except Epsilon to FIRST(A).
     - If FIRST(X) contains epsilon, continue to the next symbol.
   - If all symbols in a can derive epsilon, add epsilon to FIRST(A).
3. Repeat until no more additions can be made to any FIRST set.

Algorithm for FOLLOW Set :

1. Initialize all FOLLOW sets to empty.
2. Add $ to FOLLOW(Start symbol).
3. For each production A → αBβ:
   - Add **FIRST(β) – {ε}** to FOLLOW(B).
   - If β can derive ε or β is empty, add FOLLOW(A) to FOLLOW(B).

4. Repeat steps 2–3 until no more additions occur.

Algorithm: LL(1) Parse Table Construction

1) For every production **A → α:**
   - For each terminal a in **FIRST(α),**
   - insert the production in M[A, a].
   - If **ε ∈ FIRST(α),**
     insert the production in all entries **M[A, b] where b ∈ FOLLOW(A).**

2) Print the constructed parsing table showing non-terminals vs terminals.

- Input the grammar rules (e.g., **E->E+T|T**).
- Compute **FIRST** and **FOLLOW** sets for each non-terminal.
- Store all terminals and non-terminals found.
- Construct the **LL(1) parsing table** using FIRST and FOLLOW.
- **Stop** the program.

Program:-

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

#define MAX_RULES 20
#define MAX_LEN 20
#define MAX_SYMBOLS 26

char grammar[MAX_RULES][MAX_LEN];
int numRules = 0;
char first[MAX_SYMBOLS][MAX_LEN];
char follow[MAX_SYMBOLS][MAX_LEN];
bool firstComputed[MAX_SYMBOLS];
bool followComputed[MAX_SYMBOLS];
char startSymbol;

int countTerminals = 0;
int countNonTerminals = 0;

char terminals[MAX_SYMBOLS];
char nonTerminals[MAX_SYMBOLS];
char parseTable[MAX_SYMBOLS][MAX_SYMBOLS][MAX_LEN];

void computeFirstOfString(const char *str, char *result);

int getNTIndex(char nt) {
    for (int i = 0; i < countNonTerminals; i++) {
        if (nonTerminals[i] == nt)
            return i;
    }
    return -1;
}

int getTIndex(char t) {
    for (int i = 0; i < countTerminals; i++) {
        if (terminals[i] == t)
            return i;
    }
    return -1;

void getNonTerminals(char nt) {
    if(nt == '#') {
        return;
    }

    for(int i=0; i<26; i++){
        if(nonTerminals[i] == nt){
            return;
        }
    }

    nonTerminals[countNonTerminals] = nt;
    countNonTerminals++;
}
```

```
    void getTerminals(char t){

        for(int i=0; i<26; i++){
            if(terminals[i] == t){
                return;
            }
        }

        terminals[countTerminals] = t;
        countTerminals++;
    }


void printParseTable() {
    for (int i = 0; i < MAX_SYMBOLS; i++)
        for (int j = 0; j < MAX_SYMBOLS; j++)
            parseTable[i][j][0] = '\0';

    for (int i = 0; i < numRules; i++) {
        char lhs = grammar[i][0];
        const char *rhs = grammar[i] + 3;

        char firstSet[MAX_LEN] = "";
        computeFirstOfString(rhs, firstSet);

        for (int j = 0; firstSet[j] != '\0'; j++) {
            if (firstSet[j] != '#') {
                int ntIndex = getNTIndex(lhs);
                int tIndex = getTIndex(firstSet[j]);
                if (ntIndex != -1 && tIndex != -1)
                    strcpy(parseTable[ntIndex][tIndex], grammar[i]);
            }
        }

        for (int j = 0; firstSet[j] != '\0'; j++) {
            if (firstSet[j] == '#') {
                int ntIndex = getNTIndex(lhs);
                for (int k = 0; follow[lhs - 'A'][k] != '\0'; k++) {
                    int tIndex = getTIndex(follow[lhs - 'A'][k]);
                    if (ntIndex != -1 && tIndex != -1)
                        strcpy(parseTable[ntIndex][tIndex], grammar[i]);
                }
            }
        }
    }

    printf("\n\n=== LL(1) Parse Table ===\n\n");
    printf("%-10s", " ");
    for (int j = 0; j < countTerminals; j++) {
        printf("%-10c", terminals[j]);
    }
    printf("\n");

    for (int i = 0; i < countNonTerminals; i++) {
        printf("%-10c", nonTerminals[i]);
        for (int j = 0; j < countTerminals; j++) {
            if (strlen(parseTable[i][j]) > 0)
                printf("%-10s", parseTable[i][j]);
            else
                printf("%-10s", "-");
        }
        printf("\n");
    }
}
 void addToSet(char *set, char symbol) {
    if (symbol == '\0') return;
    for (int i = 0; set[i] != '\0'; i++) {
        if (set[i] == symbol) return;
    }
    int len = strlen(set);
    set[len] = symbol;
    set[len + 1] = '\0';
}
```

```
void computeFirst(char symbol) {
    int symIndex = symbol - 'A';
    if (firstComputed[symIndex]) return;
    firstComputed[symIndex] = true;

    for (int i = 0; i < numRules; i++) {
        if (grammar[i][0] != symbol) continue;

        const char *rhs = grammar[i] + 3;

        if (rhs[0] == '#' && rhs[1] == '\0') {
            addToSet(first[symIndex], '#');
            continue;
        }

        int nullable = 1;
        for (int j = 0; rhs[j] != '\0' && nullable; j++) {
            nullable = 0;

            if (rhs[j] >= 'a' && rhs[j] <= 'z') {
                addToSet(first[symIndex], rhs[j]);
            } else if (rhs[j] >= 'A' && rhs[j] <= 'Z') {
                computeFirst(rhs[j]);
                int index = rhs[j] - 'A';

                for (int k = 0; first[index][k] != '\0'; k++) {
                    if (first[index][k] != '#') {
                        addToSet(first[symIndex], first[index][k]);
                    }
                }

                for (int k = 0; first[index][k] != '\0'; k++) {
                    if (first[index][k] == '#') {
                        nullable = 1;
                        break;
                    }
                }
            }

            if (!nullable) break;
        }

        if (nullable) {
            addToSet(first[symIndex], '#');
        }
    }
}

void computeFirstOfString(const char *str, char *result) {
    if (str[0] == '\0') return;

    int i = 0;
    int nullable = 1;
    while (str[i] != '\0' && nullable) {
        nullable = 0;

        if (str[i] >= 'a' && str[i] <= 'z') {
            addToSet(result, str[i]);
        } else if (str[i] >= 'A' && str[i] <= 'Z') {
            computeFirst(str[i]);
            int index = str[i] - 'A';
            for (int j = 0; first[index][j] != '\0'; j++) {
                if (first[index][j] != '#') {
                    addToSet(result, first[index][j]);
                }
            }

            for (int j = 0; first[index][j] != '\0'; j++) {
                if (first[index][j] == '#') {
                    nullable = 1;
                    break;
                }
            }
        }

        if (!nullable) return;
        i++;
    }

    if (nullable) {
        addToSet(result, '#');
    }
}

int isTerminal(char input){
    if((input >= 'a' && input <= 'z') || input == '+' || input == '-' || input == ')' || input == '(' || input == '*' || input == '$'){
        return 1;
    }

    return 0;
}
```

```
void computeFollow(char symbol) {
    if (followComputed[symbol - 'A']) return;
    followComputed[symbol - 'A'] = true;

    if (symbol == startSymbol) {
        addToSet(follow[symbol - 'A'], '$');
    }

    for (int i = 0; i < numRules; i++) {
        const char *rhs = grammar[i] + 3;
        int len = strlen(rhs);
        for (int j = 0; j < len; j++) {
            if (rhs[j] == symbol) {
                if (j + 1 < len) {
                    if (rhs[j + 1] >= 'a' && rhs[j + 1] <= 'z') {
                        addToSet(follow[symbol - 'A'], rhs[j + 1]);
                    } else if (rhs[j + 1] >= 'A' && rhs[j + 1] <= 'Z') {
                        computeFirst(rhs[j + 1]);
                        int idx = rhs[j + 1] - 'A';
                        for (int k = 0; first[idx][k] != '\0'; k++) {
                            if (first[idx][k] != '#') {
                                addToSet(follow[symbol - 'A'], first[idx][k]);
                            }
                        }
                    }
                }

                if (j + 1 >= len) {
                    if (grammar[i][0] != symbol) {
                        computeFollow(grammar[i][0]);
                        int lhsIdx = grammar[i][0] - 'A';
                        for (int k = 0; follow[lhsIdx][k] != '\0'; k++) {
                            addToSet(follow[symbol - 'A'], follow[lhsIdx][k]);
                        }
                    }
                }
            }
        }
    }
}
```

```
void parseProduction(char *input) {
   char lhs = input[0];
    int i = 0;

    while (input[i] != '\0') {
        if (input[i] == '-' && input[i + 1] == '>') {
            i += 2;
            break;
        }
        i++;
    }

    int start = i;
    while (input[i] != '\0') {
        if (input[i] == '|' || input[i + 1] == '\0') {
            int end = (input[i] == '|') ? i : i + 1;


            grammar[numRules][0] = lhs;
            grammar[numRules][1] = '-';
            grammar[numRules][2] = '>';

            getNonTerminals(lhs);


            int k = 3;
            for (int j = start; j < end; j++) {
                grammar[numRules][k++] = input[j];
                if(isTerminal(input[j])) {
                    getTerminals(input[j]);
                } else {
                    getNonTerminals(input[j]);
                }
            }
            grammar[numRules][k] = '\0';

            numRules++;

            start = i + 1;
        }
        i++;
}
}
```

```c
int main() {
    int productionCount;
    char buffer[MAX_LEN];

    printf("Enter the number of productions: ");
    scanf("%d", &productionCount);
    getchar();

    printf("Enter the productions (e.g., A->aB|c):\n");
    for (int i = 0; i < productionCount; i++) {
        printf("Rule %d: ", i + 1);
        fgets(buffer, sizeof(buffer), stdin);
        buffer[strcspn(buffer, "\n")] = '\0';
        parseProduction(buffer);
    }

    printf("Enter the start symbol: ");
    scanf(" %c", &startSymbol);

    for (int i = 0; i < MAX_SYMBOLS; i++) {
        first[i][0] = '\0';
        follow[i][0] = '\0';
        firstComputed[i] = false;
        followComputed[i] = false;
    }

    for (int i = 0; i < numRules; i++) {
        computeFirst(grammar[i][0]);
    }

    for (int i = 0; i < numRules; i++) {
        computeFollow(grammar[i][0]);
    }

    printf("\nFIRST sets:\n");
    for (int i = 0; i < numRules; i++) {
        char nt = grammar[i][0];
        if (i == 0 || grammar[i][0] != grammar[i - 1][0]) {
            printf("FIRST(%c) = { ", nt);
            for (int j = 0; first[nt - 'A'][j] != '\0'; j++) {
                printf("%c ", first[nt - 'A'][j]);
            }
            printf("}\n");
        }
    }

    printf("\nFOLLOW sets:\n");
    for (int i = 0; i < numRules; i++) {
        char nt = grammar[i][0];
        if (i == 0 || grammar[i][0] != grammar[i - 1][0]) {
            printf("FOLLOW(%c) = { ", nt);
            for (int j = 0; follow[nt - 'A'][j] != '\0'; j++) {
                printf("%c ", follow[nt - 'A'][j]);
            }
            printf("}\n");
        }
    }

    terminals[countTerminals] = '$';
    countTerminals++;

    printParseTable();

    return 0;
}
```

Output:-

```
ankitgupta@Ankits-MacBook-Pro compiler design % gcc Experiment_9.c -o experiment9
ankitgupta@Ankits-MacBook-Pro compiler design % ./experiment9
Enter the number of productions: 2
Enter the productions (e.g., A->aB|c):
Rule 1: A->aB|c
Rule 2: B->b
Enter the start symbol: A

FIRST sets:
FIRST(A) = { a c }
FIRST(B) = { b }

FOLLOW sets:
FOLLOW(A) = { $ }
FOLLOW(B) = { $ }


=== LL(1) Parse Table ===

          a         c         b         $
A         A->aB     A->c      -         -
B         -         -         B->b      -
```

Experiment - 10

Aim:-

Write a program to parse the string id =id + id generating a sequence of moves table based on the LL(1) parsing table.

Procedure:-

In compiler design, **predictive parsing** is a **top-down parsing technique** that uses lookahead symbols to predict which production rule to apply.

A grammar is said to be **LL(1)** if it can be parsed from **Left-to-right**, producing a **Leftmost derivation** using **1 lookahead symbol.**

To build a **predictive parser**, we first need to compute:

- FIRST sets
- FOLLOW sets
- And then use them to construct the **LL(1) parsing table**

1. FIRST Set
   - The **FIRST set** of a non-terminal is the set of terminals that begin the strings derivable from that non-terminal.
   - If a non-terminal can derive ε (epsilon), then ε is included in its FIRST set.
   - Example:

A → aB | #
FIRST(A) = { a, # }

2. FOLLOW Set

- The **FOLLOW set** of a non-terminal is the set of terminals that can appear **immediately to the right** of that non-terminal in some derivation.
- The symbol **$** (end marker) is always in the FOLLOW of the start symbol.
- Example

S → AB
A → a | #
B → b

FOLLOW(A) = { b }
FOLLOW(S) = { $ }

3. LL(1) Parsing Table Construction Rules

For each production A → α:

- For every terminal **a in FIRST(α)**, place **A → α in M[A, a].**
- If **Epsilon ∈ FIRST(α)**, for every terminal b in FOLLOW(A), place **A → α in M[A, b].**
- If **$ ∈ FOLLOW(A),** place **A → α in M[A, $].**

4. Parsing Process
   - Use a **stack** initialized with the start symbol and $.
   - Repeatedly compare the **top of stack (X)** and **current input symbol (a):**
     1. If both are terminals and **X == a**, pop and advance input.
     2. If **X** is a non-terminal, consult the parsing table **M[X, a]**:
        - Replace **X** with the RHS of the selected production.
     3. If **X** is **$** and **a is $**, accept the string.
     4. Otherwise, report a **parsing error**.

Algorithm for FIRST Set:

1. Start with all FIRST sets empty.
2. For every production of the form A → a:
   - For each symbol X in a (from left to right):
     - If X is a terminal, add X to FIRST(A) and stop.
     - If X is a non-terminal, add all symbols of FIRST(X) except Epsilon to FIRST(A).
     - If FIRST(X) contains epsilon, continue to the next symbol.
   - If all symbols in a can derive epsilon, add epsilon to FIRST(A).
3. Repeat until no more additions can be made to any FIRST set.

Algorithm for FOLLOW Set :

1. Initialize all FOLLOW sets to empty.
2. Add $ to FOLLOW(Start symbol).
3. For each production A → αBβ:
   - Add **FIRST(β) – {ε}** to FOLLOW(B).
   - If β can derive ε or β is empty, add FOLLOW(A) to FOLLOW(B).
4. Repeat steps 2–3 until no more additions occur.

Algorithm: LL(1) Parse Table Construction
1) For every production **A → α:**
   - For each terminal a in **FIRST(α),**
   - insert the production in M[A, a].
   - If **ε ∈ FIRST(α),**
     insert the production in all entries **M[A, b] where b ∈ FOLLOW(A).**
2) Print the constructed parsing table showing non-terminals vs terminals.

Parsing Process

- Use a **stack** initialized with the start symbol and **$**.
- Repeatedly compare the **top of stack (X)** and **current input symbol (a)**:
  1. If both are terminals and **X == a**, pop and advance input.
  2. If X is a non-terminal, consult the parsing table **M[X, a]**:
     - Replace **X** with the RHS of the selected production.
  3. If **X** is **$** and **a** is **$**, accept the string.
  4. Otherwise, report a **parsing error**.

Program:-

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

#define MAX_RULES 20
#define MAX_LEN 20
#define MAX_SYMBOLS 26

char grammar[MAX_RULES][MAX_LEN];
int numRules = 0;
char first[MAX_SYMBOLS][MAX_LEN];
char follow[MAX_SYMBOLS][MAX_LEN];
bool firstComputed[MAX_SYMBOLS];
bool followComputed[MAX_SYMBOLS];
char startSymbol;

int countTerminals = 0;
int countNonTerminals = 0;

char terminals[MAX_SYMBOLS];
char nonTerminals[MAX_SYMBOLS];
char parseTable[MAX_SYMBOLS][MAX_SYMBOLS][MAX_LEN];

void computeFirstOfString(const char *str, char *result);

int getNTIndex(char nt) {
    for (int i = 0; i < countNonTerminals; i++) {
        if (nonTerminals[i] == nt)
            return i;
    }
    return -1;
}

int getTIndex(char t) {
    for (int i = 0; i < countTerminals; i++) {
        if (terminals[i] == t)
            return i;
    }
    return -1;
}
```

```
void printParseTable() {
    for (int i = 0; i < MAX_SYMBOLS; i++)
        for (int j = 0; j < MAX_SYMBOLS; j++)
            parseTable[i][j][0] = '\0';

    for (int i = 0; i < numRules; i++) {
        char lhs = grammar[i][0];
        const char *rhs = grammar[i] + 3;

        char firstSet[MAX_LEN] = "";
        computeFirstOfString(rhs, firstSet);

        for (int j = 0; firstSet[j] != '\0'; j++) {
            if (firstSet[j] != '#') {
                int ntIndex = getNTIndex(lhs);
                int tIndex = getTIndex(firstSet[j]);
                if (ntIndex != -1 && tIndex != -1)
                    strcpy(parseTable[ntIndex][tIndex], grammar[i]);
            }
        }

        for (int j = 0; firstSet[j] != '\0'; j++) {
            if (firstSet[j] == '#') {
                int ntIndex = getNTIndex(lhs);
                for (int k = 0; follow[lhs - 'A'][k] != '\0'; k++) {
                    int tIndex = getTIndex(follow[lhs - 'A'][k]);
                    if (ntIndex != -1 && tIndex != -1)
                        strcpy(parseTable[ntIndex][tIndex], grammar[i]);
                }
            }
        }
    }

    printf("\n\n=== LL(1) Parse Table ===\n\n");
    printf("%-10s", " ");
    for (int j = 0; j < countTerminals; j++) {
        printf("%-10c", terminals[j]);
    }
    printf("\n");

    for (int i = 0; i < countNonTerminals; i++) {
        printf("%-10c", nonTerminals[i]);
        for (int j = 0; j < countTerminals; j++) {
            if (strlen(parseTable[i][j]) > 0)
                printf("%-10s", parseTable[i][j]);
            else
                printf("%-10s", "-");
        }
        printf("\n");
    }
}

void getNonTerminals(char nt) {
    if (nt == '#' || nt < 'A' || nt > 'Z') return;
    for (int i = 0; i < countNonTerminals; i++) {
        if (nonTerminals[i] == nt)
            return;
    }
    nonTerminals[countNonTerminals++] = nt;
}

void getTerminals(char t) {
    if (!((t >= 'a' && t <= 'z') || strchr("+-*/()=$#", t)))
        return;
    for (int i = 0; i < countTerminals; i++) {
        if (terminals[i] == t)
            return;
    }
    terminals[countTerminals++] = t;
}
```

```c
int addToSet(char *set, char symbol) {
    if (symbol == '\0') return 0;
    for (int i = 0; set[i] != '\0'; i++) {
        if (set[i] == symbol) return 0;
    }
    int len = strlen(set);
    if (len >= MAX_LEN - 1) return 0;
    set[len] = symbol;
    set[len + 1] = '\0';
    return 1;
}


void computeFirst(char symbol) {
    if (symbol < 'A' || symbol > 'Z') return;

    bool changed = true;
    while (changed) {
        changed = false;

        for (int r = 0; r < numRules; r++) {
            if (grammar[r][0] != symbol) continue;

            const char *rhs = grammar[r] + 3;

            /* if RHS is exactly epsilon '#' */
            if (rhs[0] == '#' && rhs[1] == '\0') {
                if (addToSet(first[symbol - 'A'], '#')) changed = true;
                continue;
            }

            int nullable = 1;
            for (int j = 0; rhs[j] != '\0' && nullable; j++) {
                nullable = 0;
                char c = rhs[j];

                if (!(c >= 'A' && c <= 'Z')) {
                    if (addToSet(first[symbol - 'A'], c)) changed = true;
                } else {
                    int idx = c - 'A';

                    for (int k = 0; first[idx][k] != '\0'; k++) {
                        if (first[idx][k] != '#') {
                            if (addToSet(first[symbol - 'A'], first[idx][k])) changed = true;
                        }
                    }

                    for (int k = 0; first[idx][k] != '\0'; k++) {
                        if (first[idx][k] == '#') {
                            nullable = 1;
                            break;
                        }
                    }
                }

                if (!nullable) break;
            }

            if (nullable) {
                if (addToSet(first[symbol - 'A'], '#')) changed = true;
            }
        }
    }
}

int isTerminal(char input) {
    if ((input >= 'a' && input <= 'z') || strchr("+-*/()=$#", input))
        return 1;
    return 0;
}
```

```c
void computeFirstOfString(const char *str, char *result) {
    result[0] = '\0';
    if (str == NULL || str[0] == '\0') return;

    int i = 0;
    int nullable = 1;
    while (str[i] != '\0' && nullable) {
        nullable = 0;
        char c = str[i];

        if (!(c >= 'A' && c <= 'Z')) {
            addToSet(result, c);
        } else {

            int idx = c - 'A';

            for (int j = 0; first[idx][j] != '\0'; j++) {
                if (first[idx][j] != '#') addToSet(result, first[idx][j]);
            }
            for (int j = 0; first[idx][j] != '\0'; j++) {
                if (first[idx][j] == '#') { nullable = 1; break; }
            }
        }

        if (!nullable) break;
        i++;
    }

    if (nullable) addToSet(result, '#');
}
void computeFollow(char symbol) {
    if (followComputed[symbol - 'A']) return;
    followComputed[symbol - 'A'] = true;

    if (symbol == startSymbol) {
        addToSet(follow[symbol - 'A'], '$');
    }

    for (int i = 0; i < numRules; i++) {
        const char *rhs = grammar[i] + 3;
        int len = strlen(rhs);
        for (int j = 0; j < len; j++) {
            if (rhs[j] == symbol) {
                if (j + 1 < len) {
                    if (rhs[j + 1] >= 'a' && rhs[j + 1] <= 'z') {
                        addToSet(follow[symbol - 'A'], rhs[j + 1]);
                    } else if (rhs[j + 1] >= 'A' && rhs[j + 1] <= 'Z') {
                        computeFirst(rhs[j + 1]);
                        int idx = rhs[j + 1] - 'A';
                        for (int k = 0; first[idx][k] != '\0'; k++) {
                            if (first[idx][k] != '#') {
                                addToSet(follow[symbol - 'A'], first[idx][k]);
                            }
                        }
                    }
                } else {
                    if (grammar[i][0] != symbol) {
                        computeFollow(grammar[i][0]);
                        int lhsIdx = grammar[i][0] - 'A';
                        for (int k = 0; follow[lhsIdx][k] != '\0'; k++) {
                            addToSet(follow[symbol - 'A'], follow[lhsIdx][k]);
                        }
                    }
                }
            }
        }
    }
}
```

```c
void parseProduction(char *input) {
    char lhs = input[0];
    int i = 0;

    while (input[i] != '\0') {
        if (input[i] == '-' && input[i + 1] == '>') {
            i += 2;
            break;
        }
        i++;
    }

    int start = i;
    while (input[i] != '\0') {
        if (input[i] == '|' || input[i + 1] == '\0') {
            int end = (input[i] == '|') ? i : i + 1;

            grammar[numRules][0] = lhs;
            grammar[numRules][1] = '-';
            grammar[numRules][2] = '>';

            getNonTerminals(lhs);

            int k = 3;
            for (int j = start; j < end; j++) {
                grammar[numRules][k++] = input[j];
                if (isTerminal(input[j])) {
                    getTerminals(input[j]);
                } else {
                    getNonTerminals(input[j]);
                }
            }
            grammar[numRules][k] = '\0';
            numRules++;
            start = i + 1;
        }
        i++;
    }
}
```

```c
void parseString(const char *input) {
    char stack[100];
    int top = 0;
    stack[top] = '$';
    stack[++top] = startSymbol;
    stack[top + 1] = '\0';

    int ip = 0;
    printf("\n%-20s%-20s%-20s\n", "Stack", "Input", "Action");
    printf("-------------------------------------------------------------\n");

    while (1) {
        printf("%-20s%-20s", stack, input + ip);

        char X = stack[top];
        char a = input[ip];

        if (X == '$' && a == '$') {
            printf("Accept\n");
            break;
        } else if (isTerminal(X) || X == '$') {
            if (X == a) {
                top--;
                ip++;
                stack[top + 1] = '\0';
                printf("Match %c\n", a);
            } else {
                printf("Error\n");
                break;
            }
        } else {
            int nIdx = getNTIndex(X);
            int tIdx = getTIndex(a);
            if (nIdx < 0 || tIdx < 0 || parseTable[nIdx][tIdx][0] == '\0') {
                printf("Error\n");
                break;
            }
            printf("%s\n", parseTable[nIdx][tIdx]);
            const char *rhs = parseTable[nIdx][tIdx] + 3;
            top--;
            for (int i = strlen(rhs) - 1; i >= 0; i--) {
                if (rhs[i] != '#')
                    stack[++top] = rhs[i];
```

```
            }
            stack[top + 1] = '\0';
        }
    }
}

int main() {
    int productionCount;
    char buffer[MAX_LEN];

    printf("Enter the number of productions: ");
    scanf("%d", &productionCount);
    getchar();

    printf("Enter the productions (e.g., E->E+T|T):\n");
    for (int i = 0; i < productionCount; i++) {
        printf("Rule %d: ", i + 1);
        fgets(buffer, sizeof(buffer), stdin);
        buffer[strcspn(buffer, "\n")] = '\0';
        parseProduction(buffer);
    }

    printf("Enter the start symbol: ");
    scanf(" %c", &startSymbol);

    for (int i = 0; i < MAX_SYMBOLS; i++) {
        first[i][0] = '\0';
        follow[i][0] = '\0';
        firstComputed[i] = false;
        followComputed[i] = false;
    }

    for (int i = 0; i < numRules; i++) {
        computeFirst(grammar[i][0]);
    }

    for (int i = 0; i < numRules; i++) {
        computeFollow(grammar[i][0]);
    }

    printf("\nFIRST sets:\n");
    for (int i = 0; i < numRules; i++) {
        char nt = grammar[i][0];
        if (i == 0 || grammar[i][0] != grammar[i - 1][0]) {
            printf("FIRST(%c) = { ", nt);
            for (int j = 0; first[nt - 'A'][j] != '\0'; j++) {
                printf("%c ", first[nt - 'A'][j]);
            }
            printf("}\n");
        }
    }

    printf("\nFOLLOW sets:\n");
    for (int i = 0; i < numRules; i++) {
        char nt = grammar[i][0];
        if (i == 0 || grammar[i][0] != grammar[i - 1][0]) {
            printf("FOLLOW(%c) = { ", nt);
            for (int j = 0; follow[nt - 'A'][j] != '\0'; j++) {
                printf("%c ", follow[nt - 'A'][j]);
            }
            printf("}\n");
        }
    }

    terminals[countTerminals++] = '$';

    printParseTable();

    printf("\nEnter input string to parse (e.g., id=id+id): ");
    char input[50];
    scanf("%s", input);
    strcat(input, "$");

    parseString(input);

    return 0;
}
```

Output:-

```
ankitgupta@Ankits-MacBook-Pro compiler design % ./experiment10
Enter the number of productions: 5
Enter the productions (e.g., E->E+T|T):
Rule 1: E->TA
Rule 2: A->+TA|#
Rule 3: T->FB
Rule 4: B->*FB|#
Rule 5: F->i|(E)
Enter the start symbol: E

FIRST sets:
FIRST(E) = { i ( }
FIRST(T) = { i ( }
FIRST(A) = { + # }
FIRST(F) = { i ( }
FIRST(B) = { * # }

FOLLOW sets:
FOLLOW(E) = { $ ) }
FOLLOW(T) = { + $ ) }
FOLLOW(A) = { $ ) }
FOLLOW(F) = { * + $ ) }
FOLLOW(B) = { + $ ) }


=== LL(1) Parse Table ===

          +         #         *         i         (         )         $
E         -         -         -         E->TA     E->TA     -         -
T         -         -         -         T->FB     T->FB     -         -
A         A->+TA    -         -         -         -         A->#      A->#
F         -         -         -         F->i      F->(E)    -         -
B         B->#      -         B->*FB    -         -         B->#      B->#

Enter input string to parse (e.g., id+id): i+i

Stack               Input               Action
--------------------------------------------------------------
$E                  i+i$                E->TA
$AT                 i+i$                T->FB
$ABF                i+i$                F->i
$ABi                i+i$                Match i
$AB                 +i$                 B->#
$A                  +i$                 A->+TA
$AT+                +i$                 Match +
$AT                 i$                  T->FB
$ABF                i$                  F->i
$ABi                i$                  Match i
$AB                 $                   B->#
$A                  $                   A->#
$                   $                   Accept
```

Experiment - 11

Aim:-

WAP to generate three address code for the given source code.

Procedure:-

In **compiler design**, the **Intermediate Code Generation (ICG)** phase translates high-level source code into a form that is easy to optimize and later converted into target machine code.

One of the most common intermediate representations is the **Three Address Code (TAC)**.
Each TAC instruction contains **at most three operands** — typically two operands and one result.

1. Three Address Code (TAC)

- It is a sequence of statements of the general form:

x = y op z

- where **x, y,** and **z** are names, constants, or temporary variables, and **op is an operator.**
- TAC helps in simplifying complex expressions into smaller steps using **temporary variables**.

Example:

A = B + C * D

Can be broken as:

t1 = C * D
t2 = B + t1
A = t2

2. Conversion to Postfix (Reverse Polish Notation)

Before generating TAC, the infix expression is first converted to **postfix notation** because:

- Postfix eliminates the need for parentheses.
- It allows easier evaluation using a stack-based method.
- Example

Infix: A = B + C * D
Postfix: B C D * +

3. Generation of TAC from Postfix

Using the postfix expression, TAC is generated by:

- Scanning each symbol from left to right.
- Pushing operands (variables) on the stack.
- When an operator is encountered:
  - Pop two operands.
  - Generate a TAC instruction using a temporary variable.
  - Push this temporary variable back on the stack.

Algorithm: Infix to Postfix Conversion

1. Initialize an empty stack for operators.
2. Scan the infix expression from left to right.
3. If the scanned symbol is:
   - **Operand** → Append to output.
   - **‘(’** → Push onto stack.
   - **‘)’** → Pop and append operators until '(' is found.
   - **Operator** → Pop all operators with higher or equal precedence, then push the current operator.
4. Pop all remaining operators from the stack and append to output.

Algorithm: Generate Three Address Code (TAC)

1. I**nitialize** an empty stack for operands.
2. Scan each s**ymbol of the postfix expression**.
3. If the symbol is an operand:
   - Push it onto the operand stack.
4. If the symbol is an operator:
   - Pop two operands, say **op1 and op2.**
   - Create a temporary variable **t = op1 operator op2**.
   - Print the TAC statement and push **t** back on the stack.
5. Repeat until the postfix expression is completely scanned.

Program:-

```c
#include <stdio.h>
#define MAX 100
char stack[MAX];
int top = -1;
int tempCount = 1;

void push(char c) {
    top++;
    stack[top] = c;
}

char pop() {
    if (top == -1)
        return '\0';
    char c = stack[top];
    top--;
    return c;
}

int precedence(char c) {
    if (c == '+' || c == '-') return 1;
    if (c == '*' || c == '/') return 2;
    return 0;
}

void infixToPostfix(char infix[], char postfix[]) {
    int i = 0, k = 0;
    char c;

    while ((c = infix[i]) != '\0') {
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')) {
            postfix[k] = c;
            k++;
        } else if (c == '(') {
            push(c);
        } else if (c == ')') {
            while (top != -1 && stack[top] != '(') {
                postfix[k] = pop();
                k++;
            }
            pop();
        } else if (c == '+' || c == '-' || c == '*' || c == '/') {
            while (top != -1 && precedence(stack[top]) >= precedence(c)) {
                postfix[k] = pop();
                k++;
            }
            push(c);
        }
        i++;
    }

    while (top != -1) {
        postfix[k] = pop();
        k++;
    }

    postfix[k] = '\0';
}
```

```c
void generateTAC(char postfix[]) {
    char operandStack[MAX][5];
    int opTop = -1;
    int i = 0, len = 0;
    char c;

    while ((c = postfix[i]) != '\0') {
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')) {
            operandStack[++opTop][0] = c;
            operandStack[opTop][1] = '\0';
        } else {
            char op2[5], op1[5];
            int k;


            for (k = 0; operandStack[opTop][k] != '\0'; k++)
                op2[k] = operandStack[opTop][k];
            op2[k] = '\0';
            opTop--;


            for (k = 0; operandStack[opTop][k] != '\0'; k++)
                op1[k] = operandStack[opTop][k];
            op1[k] = '\0';
            opTop--;

            printf("t%d = %s %c %s\n", tempCount, op1, c, op2);

            operandStack[++opTop][0] = 't';
            operandStack[opTop][1] = (char)('0' + tempCount);
            operandStack[opTop][2] = '\0';

            tempCount++;
        }
        i++;
    }
}

int main() {
    char expr[MAX], postfix[MAX], rhs[MAX];
    char lhs;
    int i = 0, j = 0;

    printf("Enter an expression: ");
    scanf("%s", expr);

    lhs = expr[0];

    for (i = 2; expr[i] != '\0'; i++) {
        rhs[j] = expr[i];
        j++;
    }
    rhs[j] = '\0';

    infixToPostfix(rhs, postfix);

    printf("\nPostfix Expression: %s\n", postfix);
    printf("\n=== Three Address Code ===\n");
    generateTAC(postfix);

    printf("%c = t%d\n", lhs, tempCount - 1);

    return 0;
}
```

Output:-

```
ankitgupta@Ankits-MacBook-Pro compiler design % gcc Experiment_11.c -o experiment11
ankitgupta@Ankits-MacBook-Pro compiler design % ./experiment11
Enter an expression: a=a+(b+c*(d+e))

Postfix Expression: abcde+*++

=== Three Address Code ===
t1 = d + e
t2 = c * t1
t3 = b + t2
t4 = a + t3
a = t4
```

Experiment - 12

Aim:-

WAP to implement a shift reduce parse/LR parser.

Procedure:-

A **Shift–Reduce Parser** is a **bottom-up parser** that reduces an input string to the start symbol of a grammar using **shift**and **reduce** operations.
It works by **shifting input symbols** onto a stack until a substring on the top of the stack matches the **right-hand side (RHS)** of a grammar production, which is then **reduced** to its corresponding **left-hand side (LHS)** non-terminal.
This technique follows the **LR parsing principle**, but without building parsing tables explicitly — it simulates the process manually by applying shift and reduce actions in the correct order

1. Shift

- Move (shift) the next input symbol from the input buffer onto the stack.

Example:

Stack: $
Input: i+i*i$
Action: Shift i
t1 = C * D
t2 = B + t1
A = t2

2. Reduce

- If the symbols on top of the stack form the RHS of a production rule, replace them with the corresponding LHS (non-terminal).
- Example

Production: E → E+T
Stack before: $E+T
Stack after: $E

3. Accept

- If the stack contains only the start symbol and $ and the input is fully consumed, the string is accepted.

4**. Error**

- If no valid shift or reduction is possible, the parser reports an error — meaning the input string doesn't conform to the grammar.

Algorithm: Shift–Reduce Parsing

1. Initialize:
   Push **$** onto the stack.
   Read the input string and append **$** at the end.
2. Repeat until input is empty or accepted:
   Step 1: Shift
   Move the next input symbol to the top of the stack.
   Print the action "Shift → symbol".
   Step 2: Reduce
   Compare the top of the stack with each RHS of the grammar.
   If a match is found:
   Replace the matched substring by the LHS of that production.
   Print the action "Reduce: RHS → LHS".
   Repeat this step as long as further reductions are possible.
   Step 3: Accept
   If the stack contains **$StartSymbol** and the input is **$**, accept.
   Step 4: Error
   If no shift or reduce operation is possible, reject the input.
3. End

Program:-

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

#define MAX 512
#define MAX_PROD 64
#define MAX_RHS 64

typedef struct {
    char lhs;
    char rhs[MAX_RHS];
    int rhs_len;
} Prod;

static Prod prods[MAX_PROD];
static int prod_count = 0;

static char stack_arr[MAX];
static int top = -1;

void push(char c) {
    if (top < MAX - 1)
        stack_arr[++top] = c;
}

char pop(void) {
    if (top >= 0)
        return stack_arr[top--];
    return '\0';
}

void print_state(const char *action, const char *input, int ipos) {
    char stack_str[MAX];
    int sidx = 0;

    for (int i = 0; i <= top; ++i)
        stack_str[sidx++] = stack_arr[i];
    stack_str[sidx] = '\0';

    printf("%-20s | %-20s | ", action, stack_str);
    for (int i = ipos; input[i] != '\0'; i++)
        putchar(input[i]);
    printf("\n");
}
int match_rhs_at_top(const char *rhs, int rhs_len) {
    if (rhs_len == 0) return 0;
    if (top + 1 < rhs_len) return 0;

    int start = top - rhs_len + 1;
    for (int i = 0; i < rhs_len; ++i)
        if (stack_arr[start + i] != rhs[i])
            return 0;
    return 1;
}

int try_reduce(void) {
    int best = -1, best_len = -1;
    for (int i = 0; i < prod_count; ++i) {
        int rlen = prods[i].rhs_len;
        if (rlen == 0) continue;
        if (match_rhs_at_top(prods[i].rhs, rlen)) {
            if (rlen > best_len) { best_len = rlen; best = i; }
        }
    }

    if (best >= 0) {
        for (int k = 0; k < prods[best].rhs_len; ++k)
            pop();
        push(prods[best].lhs);

        char action_str[64];
        int idx = 0;
        idx += sprintf(action_str, "Reduce: ");
        for (int k = 0; k < prods[best].rhs_len; ++k)
            idx += sprintf(action_str + idx, "%c", prods[best].rhs[k]);
        sprintf(action_str + idx, "->%c", prods[best].lhs);

        print_state(action_str, "", 0);
        return 1;
    }
    return 0;
}
```

```c
int trim_newline(char *s) {
    int i = 0;
    while (s[i] != '\0') ++i;
    if (i > 0 && s[i - 1] == '\n') {
        s[i - 1] = '\0';
        --i;
    }
    return i;
}

void read_productions(void) {
    char line[MAX];
    printf("Enter productions (one per line), e.g. E->E+T. Blank line to finish:\n");

    while (1) {
        if (fgets(line, sizeof(line), stdin) == NULL) break;
        trim_newline(line);

        int p = 0;
        while (line[p] == ' ' || line[p] == '\t') ++p;
        if (line[p] == '\0') break;

        char lhs = line[p];
        int arrow_pos = -1;
        for (int i = p + 1; line[i] != '\0'; ++i)
            if (line[i] == '-' && line[i + 1] == '>') {
                arrow_pos = i;
                break;
            }

        if (arrow_pos < 0) {
            printf("Invalid: %s\n", line + p);
            continue;
        }

        int r = arrow_pos + 2;
        while (line[r] == ' ' || line[r] == '\t') ++r;

        int ridx = 0;
        while (line[r] != '\0' && ridx < MAX_RHS - 1) {
            if (line[r] != ' ' && line[r] != '\t')
                prods[prod_count].rhs[ridx++] = line[r];
            ++r;
        }
        prods[prod_count].rhs[ridx] = '\0';
        prods[prod_count].rhs_len = ridx;
        prods[prod_count].lhs = lhs;
        ++prod_count;
    }

    if (prod_count == 0) {
        fprintf(stderr, "No productions entered.\n");
        exit(1);
    }
}

int main(void) {
    char input[MAX];
    read_productions();

    printf("Enter input expression (use single-char tokens, e.g. i+i*i):\n");
    if (fgets(input, sizeof(input), stdin) == NULL) return 1;

    int len = trim_newline(input);
    input[len] = '$';
    input[len + 1] = '\0';

    top = -1;
    push('$');

    int ipos = 0;
    printf("\n%-20s | %-20s | %s\n", "Action", "Stack", "Input");
    printf("-------------------------------------------------------------------\n");
    print_state("Start", input, ipos);

    for (;;) {
        if (top == 1 && stack_arr[0] == '$' && stack_arr[1] == prods[0].lhs && input[ipos] == '$') {
            printf("-------------------------------------------------------------------\n");
            printf("\nInput accepted by the grammar (start symbol %c).\n", prods[0].lhs);
            return 0;
        }

        printf("-------------------------------------------------------------------\n");
        printf("\nError: No valid shift or reduce possible.\n");
        return 1;
    }

    char cur = input[ipos];
    char act[32];
    sprintf(act, "Shift -> %c", cur);
    push(cur);
    ++ipos;
    print_state(act, input, ipos);

    while (try_reduce());
  }
}
```

Output:-

```
ankitgupta@Ankits-MacBook-Pro compiler design % ./experiment12
Enter productions (one per line), e.g. E->E+T. Blank line to finish:
A->aB
B->b

Enter input expression (use single-char tokens, e.g. i+i*i):
ab

Action               | Stack               | Input
---------------------------------------------------------------------
Start                | $                   | ab$
Shift -> a           | $a                  | b$
Shift -> b           | $ab                 | $
Reduce: b->B         | $aB                 |
Reduce: aB->A        | $A                  |
---------------------------------------------------------------------

Input accepted by the grammar (start symbol A). _
```

Experiment-13

Aim:-

Implement a Lexical Analyzer to tokenize the input using Lex

Procedure:-

A **lexical analyzer** (or *lexer*) is the first phase of a compiler.
It scans the source code and converts sequences of characters into **tokens**, which are meaningful units for the parser.
Each token type **(like IDENTIFIER, NUMBER, or KEYWORD)** is matched using regular expressions.
Lex (or Flex — Fast Lexical Analyzer) automates this process. You provide a set of patterns (regexes) and actions (C code to execute when a pattern matches).

Structure of a Lex Program

A Lex file has three main sections:

%{
Declarations (C code)
%}

%%
Pattern Action
Pattern Action
%%

User-defined C code (like main function)

1) Declarations:

%{ #include <stdio.h> %}

includes header files or C declarations that can be used in actions.

2) Patterns and Actions:

- "int"|"float"|"char" { printf("KEYWORD(%s)\n", yytext); }
- [a-zA-Z_][a-zA-Z0-9_]* { printf("IDENTIFIER(%s)\n", yytext); }
- [0-9]+ { printf("NUMBER(%s)\n", yytext); }
- "=" { printf("ASSIGN(%s)\n", yytext); }
- "+" { printf("PLUS(%s)\n", yytext); }

3) User Code

```
int main() {
   yylex();
   return 0;
}

int yywrap() {
   return 1;
}
```

yylex() → The main Lex function that reads input and applies the above patterns.
yywrap() → Informs Lex that input is finished.

Program:-

```
%{
#include <stdio.h>
%}

%%
[ \t\n]+                ; /* ignore whitespace */
"int"|"float"|"char"    { printf("KEYWORD(%s)\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]*  { printf("IDENTIFIER(%s)\n", yytext); }
[0-9]+                  { printf("NUMBER(%s)\n", yytext); }
"="                     { printf("ASSIGN(%s)\n", yytext); }
"+"                     { printf("PLUS(%s)\n", yytext); }
"-"                     { printf("MINUS(%s)\n", yytext); }
"*"                     { printf("MULT(%s)\n", yytext); }
"/"                     { printf("DIV(%s)\n", yytext); }
";"                     { printf("SEMICOLON(%s)\n", yytext); }
"("|")"|"["|"]"|"{"|"}" { printf("Parenthesis(%s)\n", yytext);}
.                       { printf("UNKNOWN(%s)\n", yytext); }
%%

int main(void) {
    yylex();
    return 0;
}

int yywrap(void) {
    return 1;
}
```

Output:-

```
ankitgupta@Ankits-MacBook-Pro ~ % cd desktop
ankitgupta@Ankits-MacBook-Pro desktop % cd compiler design
cd: string not in pwd: compiler
ankitgupta@Ankits-MacBook-Pro desktop % cd "compiler design"
ankitgupta@Ankits-MacBook-Pro compiler design % flex lexical_analyzer.l
ankitgupta@Ankits-MacBook-Pro compiler design % gcc lex.yy.c -o lexical_analyzer
^[[A%
ankitgupta@Ankits-MacBook-Pro compiler design % ./lexical_analyzer

int a = b * (c + d * (e + f))
KEYWORD(int)
IDENTIFIER(a)
ASSIGN(=)
IDENTIFIER(b)
MULT(*)
Parenthesis(()
IDENTIFIER(c)
PLUS(+)
IDENTIFIER(d)
MULT(*)
Parenthesis(()
IDENTIFIER(e)
PLUS(+)
IDENTIFIER(f)
Parenthesis())
Parenthesis())
```