

Experiment - 1

Aim:-

Implement a NFA to recognize keyword ‘while’ in lexical analysis

Procedure:-

In lexical analysis, a **lexical analyzer (lexer)** is responsible for scanning the source code and identifying different tokens such as keywords, identifiers, operators, and constants. This program is designed to simulate a **finite automaton (FA)** that checks whether a given input string matches the specific keyword “**while**” in the C programming language.

By processing each character of the input one by one and moving through defined states, the program determines if the sequence of characters exactly matches the keyword “**while**.” If all transitions are valid and the final state corresponds to the acceptance state, the input is recognized as a **valid keyword**. Otherwise, it is rejected as an **invalid keyword**.

This approach demonstrates how **lexical analyzers** identify reserved words in programming languages during the **Tokenization Phase** of a compiler.

- 1) **Start** the program.
- 2) **Declare** a function **isWhileKeyword(char keyword[])** to check whether the input string matches the keyword “**while**”.
- 3) **Initialize** a state variable **s = 0** and an index **i = 0**.
- 4) **Read** the input string and calculate its length **n**.
- 5) **Iterate** through each character of the string using a while loop (**i < n**).
- 6) **Use a switch-case structure** to simulate state transitions:
 - **State 0:** If the character is '**w**', move to **State 1**; otherwise, reject (return 0).
 - **State 1:** If the character is '**h**', move to **State 2**; otherwise, reject.
 - **State 2:** If the character is '**i**', move to **State 3**; otherwise, reject.
 - **State 3:** If the character is '**T**', move to **State 4**; otherwise, reject.
 - **State 4:** If the character is '**e**', move to **State 5**; otherwise, reject.
 - **State 5:** End of valid keyword; no further input should be processed.
- 7) After processing all characters, **accept** the string if the final state **s == 5**, otherwise **reject**.
- 8) In the **main()** function:
 - **Input** a string from the user.
 - **Call** **isWhileKeyword(keyword)** to check validity.
 - **Display output:**
 - If valid → Print "**Valid Keyword**" and token **<Keyword, while>**.
 - If invalid → Print "**Invalid Keyword**".
- 9) **Stop** the program.

Program:-

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

int isWhileKeyword(char keyword[]){
    int s = 0;
    int i = 0;
    int n = strlen(keyword);

    while(i < n){
        char ch = keyword[i];
        switch(s){
            case 0:
                if(ch == 'w'){
                    s = 1;
                } else {
                    return 0;
                }
                break;

            case 1:
                if(ch == 'h'){
                    s = 2;
                } else {
                    return 0;
                }
                break;

            case 2:
                if(ch == 'i'){
                    s = 3;
                } else {
                    return 0;
                }
                break;

            case 3:
                if(ch == 'l'){
                    s = 4;
                } else {
                    return 0;
                }
                break;

            case 4:
                if(ch == 'e'){
                    s = 5;
                } else {
                    return 0;
                }
                break;
        }
    }
    if(s == 5)
        return 1;
    else
        return 0;
}
```

```

        case 3:
            if(ch == 'l'){
                s = 4;
            } else {
                return 0;
            }
            break;
        case 4:
            if(ch == 'e'){
                s = 5;
            } else {
                return 0;
            }
            break;
        case 5:
            return 0;
    }

    i++;
}
return s == 5;
}

int main() {
    char keyword[100];

    printf("Enter a Keyword: ");
    scanf("%s", keyword);

    if (isWhileKeyword(keyword)) {
        printf("Valid Keyword\n");
        printf("Token Generated is <Keyword, %s>", keyword);
    } else {
        printf("Invalid Keyword\n");
    }

    return 0;
}

```

Output:-

- ankitgupta@Ankits-MacBook-Pro compiler design % gcc Experiment1.c -o experiment1
- ankitgupta@Ankits-MacBook-Pro compiler design % ./experiment1

Enter a Keyword: while

Valid Keyword

Token Generated is <Keyword, while>
- ankitgupta@Ankits-MacBook-Pro compiler design % gcc Experiment1.c -o experiment1
- ankitgupta@Ankits-MacBook-Pro compiler design % ./experiment1

Enter a Keyword: int

Invalid Keyword

Experiment - 2

Aim:-

Implement NFAs to recognize variables and numbers in lexical analysis.

Procedure:-

In programming languages like C, an **identifier** is the name given to variables, functions, or other user-defined entities. An identifier must follow specific lexical rules:

- It **must begin** with a letter (**A-Z or a-z**) or an underscore (**_**).
- Subsequent characters can include **letters, digits (0-9), or underscores**.
- It cannot start with a digit and cannot contain special characters such as **@, #, or -**.

This program uses the concept of a **finite automaton (FA)** to verify whether a given string satisfies the rules of a valid identifier.

By scanning each character one by one and transitioning through defined states, the program simulates how a **lexical analyzer** identifies valid identifiers during the **tokenization** process in a compiler.

- 1) **Start** the program.
- 2) **Define** a function **isIdentifier(char id[])** that checks whether the input string is a valid identifier.
- 3) **Initialize** state **s = 0** and index **i = 0**.
- 4) **Find** the length of the string **n = strlen(id)**.
- 5) **Use a loop** to process each character until **i < n**.
- 6) **Switch based on state s:**
 - State 0:
 - If the first character is a letter (**A-Z / a-z**) or an underscore (**_**), move to **State 1**.
 - Otherwise, reject (return 0).
 - State 1:
 - For remaining characters:
 - Accept letters, digits, or underscores and stay in State 1.
 - If any other character is found, reject (return 0).
- 7) After processing all characters, accept the string if **s == 1**.
- 8) In the **main()** function:
 - **Input** a string from the user.
 - **Call** **isIdentifier(id)** to check validity.
 - **Display:**
 - If valid → Print "**Valid Identifier**" and the token **<id, identifier>**.
 - If invalid → Print "**Invalid Identifier**".
- 9) **Stop** the program.

Program:-

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

int isIdentifier(char id[]) {
    int s = 0;
    int i = 0;
    int n = strlen(id);

    while (i < n) {
        char ch = id[i];
        switch (s) {
            case 0:
                if ((ch >= 'a' && ch <= 'z') ||
                    (ch >= 'A' && ch <= 'Z') ||
                    ch == '_') {
                    s = 1;
                    break;
                } else {
                    return 0;
                }
        }
        i++;
    }
    if (s == 1)
        return 1;
    else
        return 0;
}
```

```
        }
        i++;
    }

    return s == 1;
}

int main() {
    char id[100];

    printf("Enter an identifier: ");
    scanf("%s", id);

    if (isIdentifier(id)) {
        printf("Valid Identifier\n");
        printf("Token Generated is <id, %s>\n", id);
    } else {
        printf("Invalid Identifier\n");
    }

    return 0;
}
```

Output:-

- ankitgupta@Ankits-MacBook-Pro compiler design % gcc Experiment2.c -o experiment2
- ankitgupta@Ankits-MacBook-Pro compiler design % ./experiment2
Enter an identifier: ab123
Valid Identifier
Token Generated is <id, ab123>
- ankitgupta@Ankits-MacBook-Pro compiler design % ./experiment2
Enter an identifier: ab12@
Invalid Identifier

Experiment - 3

Aim:-

Implement NFAs to recognize relational operators, arithmetic operators, parenthesis, and white space in lexical analysis.

Procedure:-

In the process of **compilation**, the first phase is **lexical analysis**, where the source code is broken down into a sequence of tokens.

Tokens are the smallest meaningful units in a program, such as keywords, identifiers, constants, operators, and delimiters.

This program implements a **simple lexical analyzer** that reads an input string and categorizes each word or symbol into one of the following token types:

1. **Keyword** – Recognizes the keyword "while".
2. **Identifier** – Checks if a word follows the rules of a valid identifier (starts with a letter or underscore, followed by letters, digits, or underscores).
3. **Arithmetic Operator** – Detects operators like +, -, *, /, and =.

It uses the **finite automata (FA)** approach for keyword and identifier validation, and maintains a symbol table that stores identifiers with unique token numbers. The output is a sequence of tokens that represent the lexical structure of the input — similar to the output of a compiler's lexical analyzer

- 1) **Start** the program.
- 2) **Define** functions:
 - **isWhileKeyword()** – Checks whether a string matches the keyword "while".
 - **isIdentifier()** – Verifies if a string is a valid identifier using FA rules.
 - **isArithmeticOp()** – Checks if a character is an arithmetic operator.
 - **isAlreadyFound()** – Searches if an identifier already exists in the symbol table.
- 3) **Initialize** arrays **symbol[][]** and **symbolNumber[]** to maintain a symbol table and unique ID numbers.
- 4) **Read** the input string from the user.
- 5) **Scan** the string character by character:
 - **Ignore** spaces.
 - If an **arithmetic operator** is found, print its token <ARTH, operator>.
 - If an **alphabetic or underscore character** is found, extract the entire word until a space or symbol appears.
- 6) **Classify** the extracted word:
 - If it matches "while" → print <while, KEYWORD>.
 - Else if it satisfies identifier rules →
 - Check if it already exists in the symbol table.
 - If found → print <word, idN> using the existing ID.
 - Else → assign a new ID, print <word, idN>, and add it to the symbol table.
- 7) **Continue** scanning until the end of the input string.
- 8) **Stop** the program.

Program:-

```
#include <stdio.h>
#include <string.h>
#define maxTokenCount 100

char symbol[maxTokenCount][100];
int symbolNumber[maxTokenCount];
int symbolCount = 0;

int isWhileKeyword(char keyword[]){
    int s = 0;
    int i = 0;
    int n = strlen(keyword);

    while(i < n){
        char ch = keyword[i];
        switch(s){
            case 0:
                if(ch == 'w'){
                    s = 1;
                } else {
                    return 0;
                }
        }
        break;
    }
}
```

```

        case 1:
            if(ch == 'h'){
                s = 2;
            } else {
                return 0;
            }
            break;
        case 2:
            if(ch == 'i'){
                s = 3;
            } else {
                return 0;
            }
            break;
        case 3:
            if(ch == 'l'){
                s = 4;
            } else {
                return 0;
            }
            break;
        case 4:
            if(ch == 'e'){
                s = 5;
            } else {
                return 0;
            }
            break;
        case 5:
            return 0;
    }

    i++;
}
return s == 5;
}

int isIdentifier(char id[]) {
    int s = 0;
    int i = 0;
    int n = strlen(id);

    while (i < n) {
        char ch = id[i];
        switch (s) {
            case 0:
                if ((ch >= 'a' && ch <= 'z') ||
                    (ch >= 'A' && ch <= 'Z') ||
                    ch == '_') {
                    s = 1;
                    break;
                } else {
                    return 0;
                }
            case 1:
                if ((ch >= 'a' && ch <= 'z') ||
                    (ch >= 'A' && ch <= 'Z') ||
                    (ch >= '0' && ch <= '9') ||
                    ch == '_') {
                    s = 1;
                    break;
                } else {
                    return 0;
                }
        }
        i++;
    }

    return s == 1;
}
int isArithmeticOp(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '=');
}

int isAlreadyFound(char word[]){
    for(int i=0; i<100; i++){
        int isFound = 1;
        for(int j=0; j<strlen(word); j++){
            if(word[j] != symbol[i][j]){
                isFound = 0;
                break;
            }
        }

        if(isFound == 1){
            return symbolNumber[i];
        }
    }
    return -1;
}

```

```

int main() {
    char input[100];
    printf("Enter input: ");
    fgets(input, sizeof(input), stdin);

    int i = 0, tokenCount = 1;
    while (input[i] != '\0' && input[i] != '\n') {
        char word[100];
        int j = 0;

        while (input[i] == ' ') i++;

        if (isArithmeticOp(input[i])) {
            printf("<ARTH, %c>\n", input[i]);
            i++;
            continue;
        }

        while ((input[i] >= 'a' && input[i] <= 'z') ||
               (input[i] >= 'A' && input[i] <= 'Z') ||
               (input[i] >= '0' && input[i] <= '9') || input[i] == '_') {
            word[j++] = input[i++];
        }
        word[j] = '\0';

        if (strlen(word) > 0) {
            if (isWhileKeyword(word)) {
                printf("<%s, KEYWORD>\n", word);
            } else if (isIdentifier(word)) {
                int existingCount = isAlreadyFound(word);
                if(existingCount != -1){
                    printf("<%s, id%d>\n", word, existingCount);
                } else {
                    printf("<%s, id%d>\n", word, tokenCount);
                    strcpy(symbol[symbolCount],word);
                    symbolNumber[symbolCount] = tokenCount;
                    tokenCount++;
                    symbolCount++;
                }
            }
        }
    }
    return 0;
}

```

Output :-

- ankitgupta@Ankits-MacBook-Pro compiler design % gcc Experiment3.c -o experiment3
 - ankitgupta@Ankits-MacBook-Pro compiler design % ./experiment3
- Enter input: ab1=
- <ab1, id1>
- <ARTH, =>

Experiment - 4

Aim:-

Write a program to generate tokens for the high-level source code: while (ab>=a1+b1)

Procedure:

In the **compilation process**, lexical analysis is the first phase of a compiler. It converts a sequence of characters from the source code into a sequence of tokens, which are the smallest meaningful units of the language. Each token represents a specific category such as keyword, identifier, operator, constant, or delimiter.

The **lexical analyzer** (lexer) scans the input source code character by character and groups them into tokens using predefined lexical rules.

For example:

- "while" → Keyword
- "x", "sum_1" → Identifiers
- "10", "45.2" → Numbers
- "+", "==" → Operators
- "()", ")" → Parentheses
- " " → Whitespace

This program demonstrates how lexical analysis works using finite automata (FA) to detect valid tokens and how a symbol table is maintained for identifiers.

- 1) **Start** the program.
- 2) **Define** functions:
 - **isWhileKeyword()** – Checks whether a string matches the keyword "while".
 - **isIdentifier()** – Verifies if a string is a valid identifier using FA rules.
 - **isNumber()** – Checks whether the input is a valid integer or floating-point number.
 - **isArithmeticOp()** – Checks if a character is an arithmetic operator.
 - **isRelationalOp()** – Detects relational operators like <, >, <=, >=, ==, !=.
 - **isParenthesis()** – Detects parentheses (and).
 - **isAlreadyFound()** – Searches if an identifier already exists in the symbol table.
- 3) **Initialize** arrays **symbol[][]** and **symbolNumber[]** to maintain a symbol table and unique ID numbers.
- 4) **Read** the input string from the user.
- 5) **Scan** the string character by character:
 - **Ignore** spaces.
 - If an **arithmetic operator** is found, print its token <ARTH, operator>.
 - If an **alphabetic or underscore character** is found, extract the entire word until a space or symbol appears.
- 6) **Classify** the extracted word:
 - If it matches "while" → print <while, KEYWORD>.
 - Else if it satisfies identifier rules →
 - Check if it already exists in the symbol table.
 - If found → print <word, idN> using the existing ID.
 - Else → assign a new ID, print <word, idN>, and add it to the symbol table.
- 7) **Continue** scanning until the end of the input string.
- 8) **Stop** the program.

Program:-

```
#include <stdio.h>
#include <string.h>
#define maxTokenCount 100

char symbol[maxTokenCount][100];
int symbolNumber[maxTokenCount];
int symbolCount = 0;
```

```

int isWhileKeyword(char keyword[]) {
    int s = 0, i = 0, n = strlen(keyword);
    while (i < n) {
        char ch = keyword[i];
        switch (s) {
            case 0: if (ch == 'w') s = 1; else return 0; break;
            case 1: if (ch == 'h') s = 2; else return 0; break;
            case 2: if (ch == 'i') s = 3; else return 0; break;
            case 3: if (ch == 'l') s = 4; else return 0; break;
            case 4: if (ch == 'e') s = 5; else return 0; break;
            case 5: return 0;
        }
        i++;
    }
    return s == 5;
}

int isNumber(char constant[]) {
    int i = 0, dotCount = 0;
    int n = strlen(constant);
    if (n == 0) return 0;

    if (!(constant[i] >= '0' && constant[i] <= '9'))
        return 0;

    for (i = 0; i < n; i++) {
        char ch = constant[i];
        if (ch == '.') {
            dotCount++;
            if (dotCount > 1) return 0;
        } else if (!(ch >= '0' && ch <= '9')) {
            return 0;
        }
    }
    return 1;
}

int isIdentifier(char id[]) {
    int i = 0, n = strlen(id);
    if (n == 0) return 0;

    if (!((id[0] >= 'a' && id[0] <= 'z') ||
           (id[0] >= 'A' && id[0] <= 'Z') ||
           id[0] == '_')) {
        return 0;
    }

    for (i = 1; i < n; i++) {
        char ch = id[i];
        if (!((ch >= 'a' && ch <= 'z') ||
               (ch >= 'A' && ch <= 'Z') ||
               (ch >= '0' && ch <= '9') ||
               ch == '_')) {
            return 0;
        }
    }
    return 1;
}

int isArithmeticOp(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '=');
}

int isRelationalOp(char ch) {
    return (ch == '<' || ch == '>' || ch == '!' || ch == '=');
}

int isParenthesis(char ch) {
    return (ch == '(' || ch == ')');
}

int isAlreadyFound(char word[]) {
    for (int i = 0; i < symbolCount; i++) {
        if (strcmp(word, symbol[i]) == 0)
            return symbolNumber[i];
    }
    return -1;
}

```

```

int main() {
    char input[200];
    printf("Enter input: ");
    fgets(input, sizeof(input), stdin);

    int i = 0, tokenCount = 1;
    while (input[i] != '\0' && input[i] != '\n') {
        char word[100];
        int j = 0;

        while (input[i] == ' ') {
            printf("<WS,>\n");
            i++;
        }

        if (isArithmeticOp(input[i])) {
            printf("<ARTH, %c>\n", input[i]);
            i++;
            continue;
        }

        if (isRelationalOp(input[i])) {
            if (input[i + 1] == '=') {
                printf("<REL, %c=%c>\n", input[i], input[i + 1]);
                i += 2;
            } else {
                printf("<REL, %c>\n", input[i]);
                i++;
            }
            continue;
        }

        if (isParenthesis(input[i])) {
            printf("<PAR, %c>\n", input[i]);
            i++;
            continue;
        }
    }

    while ((input[i] >= 'a' && input[i] <= 'z') ||
           (input[i] >= 'A' && input[i] <= 'Z') ||
           (input[i] >= '0' && input[i] <= '9') ||
           input[i] == '_') {
        word[j++] = input[i++];
    }
    word[j] = '\0';

    if (strlen(word) > 0) {
        if (isWhileKeyword(word)) {
            printf("<%s, KEYWORD>\n", word);
        } else if (isIdentifier(word)) {
            int existingCount = isAlreadyFound(word);
            if (existingCount != -1) {
                printf("<%s, id%d>\n", word, existingCount);
            } else {
                printf("<%s, id%d>\n", word, tokenCount);
                strcpy(symbol[symbolCount], word);
                symbolNumber[symbolCount] = tokenCount;
                tokenCount++;
                symbolCount++;
            }
        } else if (isNumber(word)) {
            printf("<NUM, %s>\n", word);
        } else {
            printf("<INVALID, %s>\n", word);
        }
    }
}

return 0;
}

```

Output: -

```
● ankitgupta@Ankits-MacBook-Pro compiler design % gcc Experiment4.c -o experiment4
● ankitgupta@Ankits-MacBook-Pro compiler design % ./experiment4
Enter input: while(ab1>=a1+b1)
<while, KEYWORD>
<PAR, (>
<ab1, id1>
<REL, >==>
<a1, id2>
<ARTH, +>
<b1, id3>
<PAR, )>
```

Experiment - 5

Aim:-

Write a program to perform left recursion the given grammar

Procedure:

In **compiler design**, grammars are used to define the syntax of a programming language. A grammar is said to have **left recursion** if a non-terminal on the left-hand side of a production rule appears again as the **first symbol** on the right-hand side.

$$A \rightarrow A\alpha | \beta$$

Here, the grammar is left recursive because the first alternative starts with the same non-terminal **A**. Left recursion causes problems for **top-down parsers**, such as **recursive descent parsers**, because it can lead to **infinite recursion** during parsing. Hence, it must be **eliminated** before parser construction.

The general technique for eliminating left recursion:

If we have:

$$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | \beta_1 | \beta_2 | \dots$$

We replace it with:

$$\begin{aligned} A &\rightarrow \beta_1 A' | \beta_2 A' \\ A' &\rightarrow \alpha_1 A' | \alpha_2 A' | \epsilon \end{aligned}$$

Where:

- **A** is the original non-terminal,
- **α** represents the left recursive part,
- **β** represents the non-left-recursive alternatives,
- and ϵ denotes epsilon (empty string).

- 1) **Start** the program.
- 2) Input the number of production rules **n**.
- 3) For each production:
 1. Read the production in the form **A-> $\alpha_1|\alpha_2|...$**
 2. Extract the **non-terminal A**.
 3. Split the right-hand side into multiple alternatives separated by '|'.
 4. For each alternative:
 - If it starts with the same non-terminal A, store it as α (left recursive part).
 - Otherwise, store it as β (non-left-recursive part).
 5. If **no α exists** → print "**No left recursion**" for that non-terminal.
 6. Otherwise:
 - Generate new productions:
 - **A-> $\beta A'$**
 - **A'-> $\alpha A' | \epsilon$**
 - 4) **Display** the transformed grammar after left recursion elimination.
 - 5) **Stop** the program.

Program: -

```
#include <stdio.h>

int stringLength(char s[]) {
    int i = 0;
    while (s[i] != '\0') i++;
    return i;
}
```

```

int main() {
    int n;
    printf("Enter number of production rules: ");
    scanf("%d", &n);

    for (int p = 0; p < n; p++) {
        char input[200], nonTerminal;
        char prods[10][100], alpha[10][100], beta[10][100];
        int altCount = 0, alphaCount = 0, betaCount = 0;

        printf("\nEnter production rule: ");
        scanf("%s", input);

        nonTerminal = input[0];

        int i = 0, j;
        while (1) {
            j = 0;
            while (input[i] != '|' && input[i] != '\0')
                prods[altCount][j++] = input[i++];
            prods[altCount][j] = '\0';
            altCount++;
            if (input[i] == '\0') break;
            else i++;
        }

        for (i = 0; i < altCount; i++) {
            if (prods[i][0] == nonTerminal) {

                int k = 0;
                for (j = 1; prods[i][j] != '\0'; j++)
                    alpha[alphaCount][k++] = prods[i][j];
                alpha[alphaCount][k] = '\0';
                alphaCount++;

            } else {

                int k = 0;
                for (j = 0; prods[i][j] != '\0'; j++)
                    beta[betaCount][k++] = prods[i][j];
                beta[betaCount][k] = '\0';
                betaCount++;

            }
        }

        if (alphaCount == 0) {
            printf("\nNo left recursion in %c\n", nonTerminal);
            printf("%c->", nonTerminal);
            for (i = 0; i < altCount; i++) {
                printf("%s", prods[i]);
                if (i != altCount - 1) printf(" |");
            }
            printf("\n");
        } else {
            printf("\nLeft recursion eliminated for %c\n", nonTerminal);

            printf("%c->", nonTerminal);
            for (i = 0; i < betaCount; i++) {
                printf("%s%c", beta[i], nonTerminal);
                if (i != betaCount - 1) printf(" |");
            }
            printf("\n");

            printf("%c'->", nonTerminal);
            for (i = 0; i < alphaCount; i++) {
                printf("%s%c", alpha[i], nonTerminal);
                if (i != alphaCount - 1) printf(" |");
            }
            printf("|Epsilon\n");
        }
    }
}

return 0;
}

```

Output: -

```
▶ ankitgupta@Ankits-MacBook-Pro compiler design % gcc Experiment5_LeftRecursion.c -o experiment5_Left
▶ ankitgupta@Ankits-MacBook-Pro compiler design % ./experiment5_Left
Enter number of production rules: 1
```

```
Enter production rule: A->Aa|b
```

```
Left recursion eliminated for A
```

```
A->bA'
```

```
A'->aA'|Epsilon
```

Experiment - 6

Aim:-

Write a program to perform left factoring on the given grammar

Procedure:

In **compiler design**, **left factoring** is a grammar transformation technique used to **remove ambiguity** and make a grammar suitable for **top-down parsing**, particularly **LL(1)** parsers.

Left factoring is applied when two or more productions of a non-terminal begin with the **same prefix**. This causes ambiguity because the parser cannot decide which production to use based on the next input symbol. Example:

$$A \rightarrow abC \mid abD$$

Here, both alternatives start with the common prefix **ab**.

This grammar can be **left factored** as:

$$\begin{aligned} A &\rightarrow abA' \\ A' &\rightarrow C \mid D \end{aligned}$$

This transformation helps the parser to defer the decision until it has enough input symbols to choose the correct rule — thus removing ambiguity.

1) **Start** the program.

2) Input the number of production rules **n**.

3) For each production:

1. Read the production in the form **A->a1|a2|....**
2. Split the right-hand side into multiple alternatives separated by '|'.
3. Compare all alternatives to **find the common prefix**.
4. If no common prefix exists:
 - Print "**No Left Factoring Needed.**"
5. If a common prefix exists:
 - Factor out the common part and rewrite the production in the form:

$$\begin{aligned} A &\rightarrow \text{common_prefix } A' \\ A' &\rightarrow \text{remaining_parts} \mid \epsilon \end{aligned}$$

4) **Print** the transformed grammar after left factoring.

5) **Stop** the program.

Program:-

```
#include <stdio.h>

int stringLength(char s[]){
    int i = 0;

    while(s[i] != '\0'){
        i++;
    }

    return i;
}

void copyString(char src[], char dest[]){
    int i = 0;
    while(src[i] != '\0'){
        dest[i] = src[i];
        i++;
    }

    dest[i] = '\0';
}
```

```

void commonPrefix(char s1[], char s2[], char prefix[]){
    int i = 0;

    while(s1[i] != '\0' && s2[i] != '\0' && s1[i] == s2[i]){
        prefix[i] = s1[i];
        i++;
    }

    prefix[i] = '\0';
}

int main()
{
    int n;

    printf("Enter the number of production rule: \n");
    scanf("%d", &n);

    for(int p=0; p<n; p++){
        char input[100], nonTerminal;
        char prods[10][100], prefix[100], temp[100];

        int i, j, k = 0, altCount = 0;

        printf("\nEnter the production rule : ");
        scanf("%s", input);

        nonTerminal = input[0];

        i = 3;

        while(1){
            j = 0;
            while(input[i] != '|' && input[i] != '\0'){
                prods[altCount][j++] = input[i++];
            }
            prods[altCount][j] = '\0';
            altCount++;

            if(input[i] == '\0'){
                break;
            } else {
                i++;
            }
        }

        copyString(prods[0], prefix);

        int found = 0;
        for (i = 0; i < altCount; i++) {
            for (j = i + 1; j < altCount; j++) {
                commonPrefix(prods[i], prods[j], temp);
                if (temp[0] != '\0') {
                    copyString(temp, prefix);
                    found = 1;
                    break;
                }
            }
            if (found) break;
        }

        if(prefix[0] == '\0'){
            printf("No left Factoring Needed the Production Rules are as Follow \n");
            printf("%c->", nonTerminal);
            for (i = 0; i < altCount; i++) {
                for (j = 0; prods[i][j] != '\0'; j++)
                    printf("%c", prods[i][j]);
                if (i != altCount - 1)
                    printf("|");
            }
            printf("\n");
        }
    }
}

```

```

} else {
    printf("Left Factoring Needed the Production Rules are as Follow \n");
    printf("%c->", nonTerminal);

    for (i = 0; prefix[i] != '\0'; i++){
        printf("%c", prefix[i]);
    }
    printf("%c'", nonTerminal);

    int first_other = 1;
    for (i = 0; i < altCount; i++) {
        int m = 0;
        while (prods[i][m] == prefix[m] && prefix[m] != '\0'){
            m++;
        }
        if (m == 0) {
            if (first_other) printf("|");
            first_other = 0;
            for (j = 0; prods[i][j] != '\0'; j++)
                printf("%c", prods[i][j]);
        }
    }
    printf("\n");

    printf("%c'->", nonTerminal);
    int first = 1;
    for (i = 0; i < altCount; i++) {
        int prelen = 0;
        while (prefix[prelen] != '\0') prelen++;

        int m = 0;
        while (prods[i][m] == prefix[m] && prefix[m] != '\0')
            m++;

        if (m != 0) {
            if (!first) printf("|");
            first = 0;
            if (prods[i][prelen] == '\0')
                printf("epsilon");
            else {
                for (j = prelen; prods[i][j] != '\0'; j++)
                    printf("%c", prods[i][j]);
            }
        }
        printf("\n");
    }
}

return 0;
}

```

Output:-

- ankitgupta@Ankits-MacBook-Pro compiler design % ./experiment5
Enter the number of production rules: 1

Enter the production rule: A->abC|abD
Left Factoring Applied. The production rules are as follows:
A->abA'
A'->C|D

Experiment - 7

Aim:-

Write a program to find the First set of given grammar

Procedure:-

In the process of **syntax analysis** (parsing) in compiler design, the **FIRST** and **FOLLOW** sets are essential for constructing predictive parsers such as **LL(1) parsers**.

FIRST Set: The FIRST set of a non-terminal symbol consists of all terminal symbols that can appear at the beginning of strings derived from that non-terminal.

- If a non-terminal can derive epsilon, then epsilon is also included in its FIRST set.
- Example:

$$A \rightarrow aB \mid \#$$

$$\text{FIRST}(A) = \{ a, \# \}$$

This transformation helps the parser to defer the decision until it has enough input symbols to choose the correct rule — thus removing ambiguity.

1. Start with all FIRST sets empty.
2. For every production of the form $A \rightarrow a$:
 - 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.
4. Input the number of productions and grammar rules (e.g., $A \rightarrow aB|c$).
5. Identify all non-terminals and store productions.
6. Recursively compute FIRST for each non-terminal using the above rules.
7. Display the FIRST.
8. Stop the program.

Program:

```
#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;

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, '#');
    }
}

```

```

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] = '>';

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

            numRules++;

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

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]);
                        }
                    }
                }
            }
        }
    }
}

```

```

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");
        }
    }

    return 0;
}

```

Output:-

```

Enter the number of productions: 2
Enter the productions (e.g., A->aB|c):
Rule 1: A->aB|C
Rule 2: B->b|c
Enter the start symbol: A

```

```

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

```

Experiment - 8

Aim:-

Write a program to find the Follow set of given grammar

Procedure:-

In the process of **syntax analysis** (parsing) in compiler design, the **FIRST** and **FOLLOW** sets are essential for constructing predictive parsers such as **LL(1)** **parsers**.

FOLLOW Set: The FOLLOW set of a non-terminal symbol consists of all terminal symbols that can appear **immediately to the right** of that non-terminal in some derivation.

- The end-of-input marker $\$$ is always in FOLLOW of the start symbol.
- If a non-terminal can be followed by another non-terminal that can derive Epsilon, then FOLLOW of the first also includes FOLLOW of the second.
- Example:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow a \mid \# \\ B &\rightarrow b \mid \# \end{aligned}$$

$$\begin{aligned} \text{FOLLOW}(A) &= \{ b, \$ \} \\ \text{FOLLOW}(B) &= \{ \$ \} \end{aligned}$$

1. Initialize all FOLLOW sets to empty.
2. Add $\$$ to FOLLOW(Start symbol).
3. For each production $A \rightarrow \alpha B \beta$:
 - Add $\text{FIRST}(\beta) - \{\epsilon\}$ 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.
5. Input the number of productions and grammar rules (e.g., $A \rightarrow aB|c$).
6. Identify all non-terminals and store productions.
7. Recursively compute FOLLOW for each non-terminal using the above rules.
8. Display the FOLLOW.
9. Stop the program.

Program:-

```
#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;

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, '#');
    }
}
}

```

```

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] = '>';

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

            numRules++;

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

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]);
                        }
                    }
                }
            }
        }
    }
}

```

```

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");
        }
    }

    return 0;
}

```

Output:-

```

Enter the number of productions: 2
Enter the productions (e.g., A->aB|c):
Rule 1: A->aB|C
Rule 2: B->b|c
Enter the start symbol: A

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

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

```