Lab Task No: 01

Title:
Implementation of Lexical Analyzer Using Lex Programming

Introduction:
A lexical analyzer, or scanner, is the first phase of a compiler. It scans the source code and breaks it into smaller units called tokens, such as keywords, identifiers, operators, and symbols. In this lab, we implement a lexical analyzer using Lex, a tool that generates programs for lexical analysis based on specified rules.

Objective:
To create a lexical analyzer using Lex programming that identifies and categorizes tokens from a given source code.

Requirements:

1. A system with Lex and a C compiler installed.
2. A text editor to write the Lex program.
3. Basic knowledge of Lex syntax and regular expressions.

Procedure:

1. **Write the Lex Program:**

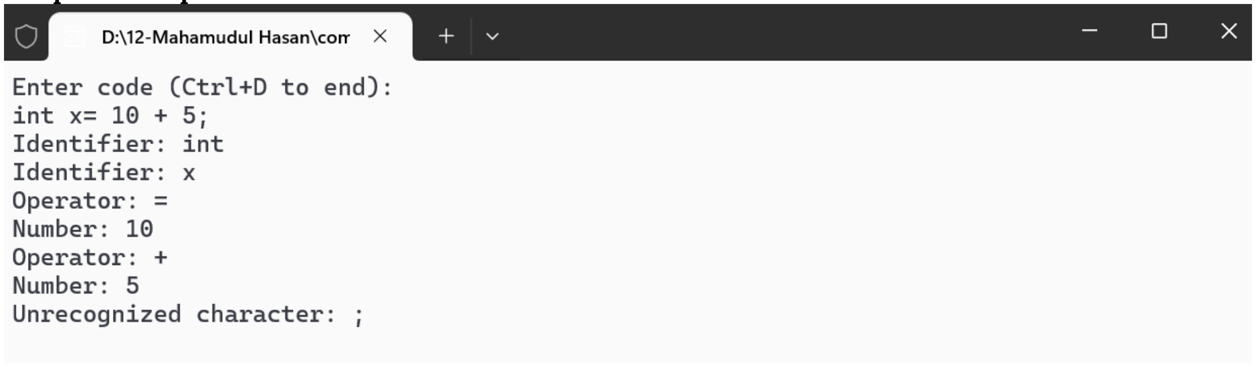
   Open a text editor and write the Lex rules for different tokens, such as keywords, identifiers, numbers, and symbols.

   Example of a Lex program:

```
%{
#include<stdio.h>
%}
%%
[a-zA-Z][a-zA-Z0-9]*   printf("Identifier: %s\n", yytext);
[0-9]+                printf("Number: %s\n", yytext);
"+"|"-"|"*"|"/"       printf("Operator: %s\n", yytext);
";"                   printf("Semicolon: %s\n", yytext);
.                     printf("Unknown Token: %s\n", yytext);
%%
int main() {
   yylex();
   return 0;
}
```

2. **Save the Lex File:**
   Save the file with the .l extension, e.g., lexer.l.
3. **Compile the Lex File:**
   - Use the command lex lexer.l to generate a C source file (lex.yy.c).
4. **Compile the Generated C File:**
   - Use a C compiler to compile the generated file:
   - gcc lex.yy.c -o lexer
5. **Run the Lexical Analyzer:**
   - Execute the compiled program:
   - ./lexer
   - Input a test string (e.g., int x = 10 + 5;) and observe the output tokens.

Output Example:



```
Enter code (Ctrl+D to end):
int x= 10 + 5;
Identifier: int
Identifier: x
Operator: =
Number: 10
Operator: +
Number: 5
Unrecognized character: ;
```

Conclusion:
The lexical analyzer successfully identified tokens such as identifiers, numbers, operators, and symbols from the input source code. Using Lex simplifies the process of writing a scanner and ensures efficient tokenization, which is a crucial step in the compilation process.

Lab Task No: 02

Title:
Implementation of Left Recursion Elimination in Context-Free Grammar

Introduction:
Left recursion is a situation in context-free grammars where a non-terminal symbol appears as the leftmost symbol in one of its own productions. This creates problems for parsers, particularly recursive descent parsers, as it can result in infinite recursion. To make the grammar suitable for top-down parsers, left recursion must be eliminated. This lab focuses on identifying left recursion in a grammar and transforming it into a grammar without left recursion using a C program.

Objective:
To identify and eliminate left recursion in context-free grammars and transform the given grammar into an equivalent grammar that is suitable for top-down parsers like LL(1).

Materials Required:

1. **C Compiler**: Any C development environment or IDE (e.g., Code::Blocks, GCC).
2. **Input Grammar**: A grammar with left recursion.
3. **Basic Knowledge** of context-free grammar and recursive descent parsing.

Procedure:

1. **Understanding Left Recursion**:
   Left recursion happens when a non-terminal symbol refers to itself as the first symbol in one of its productions. For example, the grammar:

2.
   ```
   A → Aα | β
   ```

   is left-recursive because the non-terminal `A` appears at the beginning of its own right-hand side in the production `A → Aα`. This can cause infinite recursion in a top-down parser.

3. **Eliminating Left Recursion**:
   To eliminate left recursion from a grammar, the following steps are performed:
   - Introduce a new non-terminal symbol (e.g., `A'`).
   - Modify the original grammar to use the new non-terminal symbol in place of the left-recursive rules.
   - The transformed grammar will have non-recursive rules, making it suitable for a top-down parser.

   For a rule like `A → Aα | β`, the transformation would be:

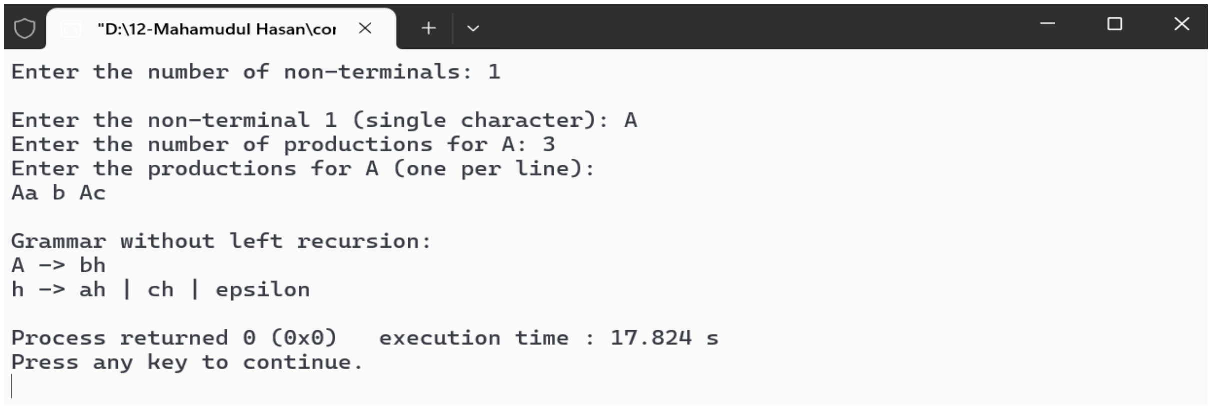
   ```
   A → βA'
   A' → αA' | ε
   ```

4. **Program Implementation**:
   The C program allows the user to input a context-free grammar with left recursion and outputs the transformed grammar with left recursion eliminated.

   C Code Implementation:

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

#define MAX_RULES 10
#define MAX_LENGTH 100
#define MAX_NON_TERMINALS 10

// Structure to store grammar rules for each non-terminal
typedef struct {
    char non_terminal; // Non-terminal symbol
    char alpha[MAX_RULES][MAX_LENGTH]; // Left-recursive productions (A
-> Aα)
    char beta[MAX_RULES][MAX_LENGTH]; // Non-recursive productions (A -
> β)
    int alpha_count; // Count of left-recursive productions
    int beta_count; // Count of non-recursive productions
} GrammarRule;

// Function to remove left recursion from a single grammar rule
void remove_left_recursion(GrammarRule* rule) {
    // If there are no left-recursive productions, simply print the
original rule
    if (rule->alpha_count == 0) {
        printf("%c -> ", rule->non_terminal);
        for (int i = 0; i < rule->beta_count; i++) {
            printf("%s", rule->beta[i]);
            if (i < rule->beta_count - 1) printf(" | ");
        }
        printf("\n");
        return;
    }

    // New non-terminal to handle left-recursive cases
    char new_non_terminal = rule->non_terminal + '\'';

    // Print transformed rule for the original non-terminal without
left recursion
    printf("%c -> ", rule->non_terminal);
    for (int i = 0; i < rule->beta_count; i++) {
        printf("%s%c", rule->beta[i], new_non_terminal);
        if (i < rule->beta_count - 1) printf(" | ");
    }
    printf("\n");

    // Print new rule for handling left-recursive productions
    printf("%c -> ", new_non_terminal);
    for (int i = 0; i < rule->alpha_count; i++) {
        printf("%s%c", rule->alpha[i], new_non_terminal);
        if (i < rule->alpha_count - 1) printf(" | ");
    }
    printf(" | epsilon\n"); // Use "epsilon" for the empty production
}

int main() {
    GrammarRule rules[MAX_NON_TERMINALS]; // Array to store rules for
multiple non-terminals
    int num_non_terminals;

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

    // Input rules for each non-terminal
    for (int r = 0; r < num_non_terminals; r++) {
        GrammarRule* rule = &rules[r];
        rule->alpha_count = 0;
        rule->beta_count = 0;

        printf("\nEnter the non-terminal %d (single character): ", r +
1);
        scanf(" %c", &rule->non_terminal);

        int n;
        printf("Enter the number of productions for %c: ", rule-
>non_terminal);
        scanf("%d", &n);

        printf("Enter the productions for %c (one per line):\n", rule-
>non_terminal);

        // Input each production and categorize as left-recursive or
non-recursive
        for (int i = 0; i < n; i++) {
            char production[MAX_LENGTH];
            scanf("%s", production);

            // Check if production is left-recursive (starts with the
non-terminal itself)
            if (production[0] == rule->non_terminal) {
                strcpy(rule->alpha[rule->alpha_count++], production +
1); // Store α part
            } else {
                strcpy(rule->beta[rule->beta_count++], production); //
Store β part
            }
        }
    }

    // Output: grammar without left recursion
    printf("\nGrammar without left recursion:\n");
    for (int r = 0; r < num_non_terminals; r++) {
        remove_left_recursion(&rules[r]);
    }

    return 0;
}
```

Sample Input & Output:



```
Enter the number of non-terminals: 1

Enter the non-terminal 1 (single character): A
Enter the number of productions for A: 3
Enter the productions for A (one per line):
Aa b Ac

Grammar without left recursion:
A -> bh
h -> ah | ch | epsilon

Process returned 0 (0x0)   execution time : 17.824 s
Press any key to continue.
```

Conclusion:
In this lab, we successfully identified and eliminated left recursion from a context-free grammar. The provided C program effectively handles left-recursive productions by transforming them into an equivalent grammar suitable for top-down parsing. This process ensures that parsers like LL(1) can parse the input without running into infinite recursion.

Lab Task No: 03

Title:
Implementation of Left Factoring in Context-Free Grammar

Introduction:
Left factoring is a technique used to transform a context-free grammar into a form that is suitable for top-down parsers, such as LL(1). It eliminates ambiguity and ensures that a parser can make a decision about which production to apply based solely on the next symbol. Left factoring is particularly helpful when a non-terminal has multiple alternatives that begin with the same prefix.

For example, if a non-terminal has the following productions:

```
A → αβ | αγ
```

It is not suitable for an LL(1) parser because both productions begin with the same symbol, α. Left factoring helps transform this into a form where the parser can decide which production to use based on the next symbol after α.

Objective:
To implement left factoring in context-free grammar and transform a given grammar into a form suitable for LL(1) parsing.

Materials Required:

1. **C Compiler**: Any C development environment or IDE (e.g., Code::Blocks, GCC).
2. **Input Grammar**: A grammar that may require left factoring.
3. **Basic Knowledge** of context-free grammar and parsing techniques.

Procedure:

1. **Understanding Left Factoring:**
   Left factoring involves identifying common prefixes in the productions of a non-terminal and factoring them out. Given a non-terminal `A` with the productions:
2. `A → αβ | αγ`

   where α is the common prefix, the left factored form would be:

   ```
   A → αA'
   A' → β | γ
   ```

   This allows a parser to make a decision after reading the first symbol α, improving parsing efficiency.

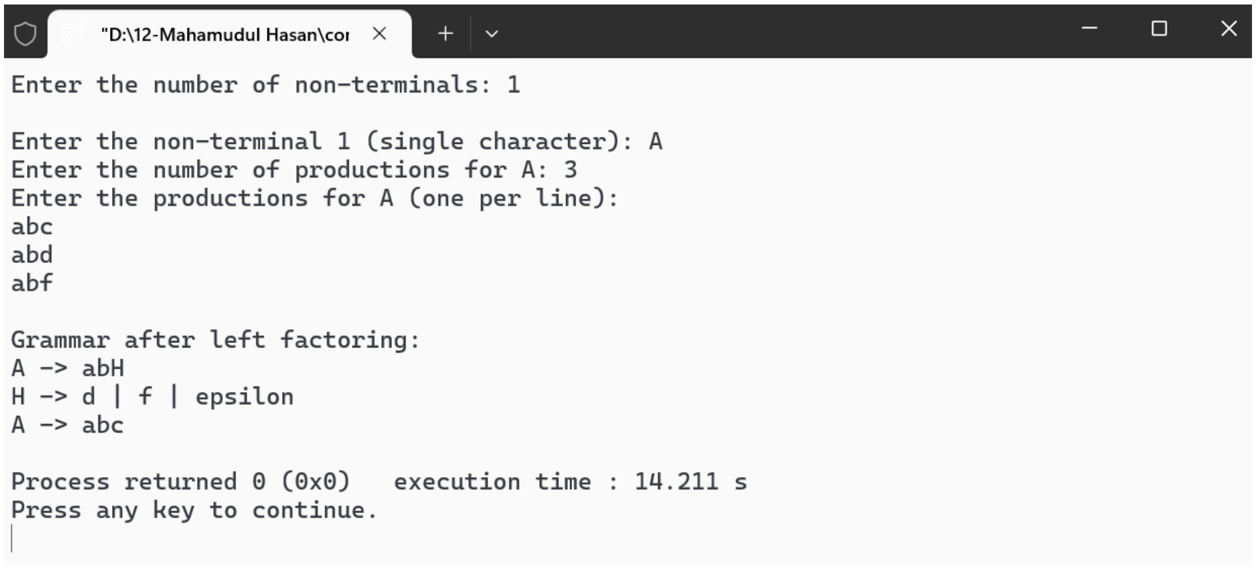
3. **Steps for Left Factoring:**
    - Identify common prefixes in the productions of each non-terminal.
    - Factor out the common prefix into a new non-terminal.
    - Rewrite the original non-terminal to reflect the new non-terminal and the remaining parts of the productions.
    - Introduce a new non-terminal to handle the remaining parts after factoring.
4. **Program Implementation:**
   The C program reads a context-free grammar, applies left factoring to the grammar, and outputs the transformed grammar suitable for LL(1) parsing.

   C Code Implementation:

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

#define MAX_RULES 10
#define MAX_LENGTH 100
#define MAX_NON_TERMINALS 10

// Structure to store grammar rules for each non-terminal
typedef struct {
    char non_terminal; // Non-terminal symbol
    char productions[MAX_RULES][MAX_LENGTH]; // Productions for the
non-terminal
    int prod_count; // Number of productions for the non-terminal
} GrammarRule;

// Function to find the longest common prefix in an array of strings
int find_common_prefix(char *str1, char *str2) {
    int i = 0;
    while (str1[i] != '\0' && str2[i] != '\0' && str1[i] == str2[i]) {
        i++;
    }
    return i; // Returns the length of the common prefix
}

// Function to apply left factoring to a single non-terminal
void left_factor(GrammarRule* rule) {
    if (rule->prod_count < 2) {
        // If there are less than 2 productions, no factoring is needed
        return;
    }

    // Check for common prefixes among the productions
    for (int i = 0; i < rule->prod_count - 1; i++) {
        int common_prefix_len = find_common_prefix(rule-
>productions[i], rule->productions[i + 1]);
        if (common_prefix_len > 0) {
            // If there is a common prefix, factor it out
            char new_non_terminal = rule->non_terminal + '\''; //
Create new non-terminal
            printf("%c -> %.*s%c\n", rule->non_terminal,
common_prefix_len, rule->productions[i], new_non_terminal);
```

```
            printf("%c -> %s\n", new_non_terminal, rule->productions[i]
+ common_prefix_len);
            return; // Factoring is done, so we exit
        }
    }

    // If no common prefix, print the original rule
    printf("%c -> ", rule->non_terminal);
    for (int i = 0; i < rule->prod_count; i++) {
        printf("%s", rule->productions[i]);
        if (i < rule->prod_count - 1) printf(" | ");
    }
    printf("\n");
}

int main() {
    GrammarRule rules[MAX_NON_TERMINALS]; // Array to store rules for
multiple non-terminals
    int num_non_terminals;

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

    // Input rules for each non-terminal
    for (int r = 0; r < num_non_terminals; r++) {
        GrammarRule* rule = &rules[r];
        rule->prod_count = 0;

        printf("\nEnter the non-terminal %d (single character): ", r +
1);
        scanf(" %c", &rule->non_terminal);

        int n;
        printf("Enter the number of productions for %c: ", rule-
>non_terminal);
        scanf("%d", &n);

        printf("Enter the productions for %c (one per line):\n", rule-
>non_terminal);

        // Input each production
        for (int i = 0; i < n; i++) {
            scanf("%s", rule->productions[rule->prod_count++]);
        }
    }

    // Output: grammar after left factoring
    printf("\nGrammar after left factoring:\n");
    for (int r = 0; r < num_non_terminals; r++) {
        left_factor(&rules[r]);
    }

    return 0;
}
```

Sample Input & Output:



```
Enter the number of non-terminals: 1

Enter the non-terminal 1 (single character): A
Enter the number of productions for A: 3
Enter the productions for A (one per line):
abc
abd
abf

Grammar after left factoring:
A -> abH
H -> d | f | epsilon
A -> abc

Process returned 0 (0x0)   execution time : 14.211 s
Press any key to continue.
```

Conclusion:
In this lab, we implemented the left factoring technique to eliminate ambiguities in context-free grammars. The C program successfully factored out common prefixes from the grammar, transforming it into a form suitable for LL(1) parsers. This process improves the efficiency of the parsing process, ensuring that the grammar can be parsed correctly in a top-down manner without confusion.

Lab Task No: 04

Title:
Implementation of FIRST and FOLLOW Sets for Predictive Parsing

Introduction:
In compiler design, predictive parsing is a top-down parsing technique that builds the parse tree from the root to the leaves. The FIRST and FOLLOW sets are fundamental to predictive parsers, especially LL(1) parsers. These sets help determine which production to apply for each non-terminal based on the upcoming input symbol.

1. **FIRST Set:**
   The FIRST set of a non-terminal consists of all terminal symbols that can appear at the beginning of any string derived from that non-terminal. If the non-terminal can derive an epsilon (empty string), then epsilon is also included in its FIRST set.
2. **FOLLOW Set:**
   The FOLLOW set of a non-terminal consists of all terminal symbols that can appear immediately to the right of that non-terminal in any derivation. The FOLLOW set is crucial for determining when to apply a production involving a non-terminal in a parsing table.

Objective:
To implement the calculation of FIRST and FOLLOW sets for a given context-free grammar and use them for predictive parsing.

Materials Required:

1. **C Compiler**: Any C development environment (e.g., GCC, Code::Blocks).
2. **Context-Free Grammar**: Input grammar in the form of productions.
3. **Basic Understanding of Context-Free Grammar and Parsing**.

Procedure:

1. **Understanding FIRST and FOLLOW:**
   - **FIRST Set Construction:**
     - If a production starts with a terminal symbol, that terminal symbol is in the FIRST set.
     - If a production starts with a non-terminal, recursively calculate the FIRST set for that non-terminal.
     - If a production can derive epsilon, include epsilon in the FIRST set.
   - **FOLLOW Set Construction:**
     - For the start symbol, include the end-of-input symbol `$` in the FOLLOW set.

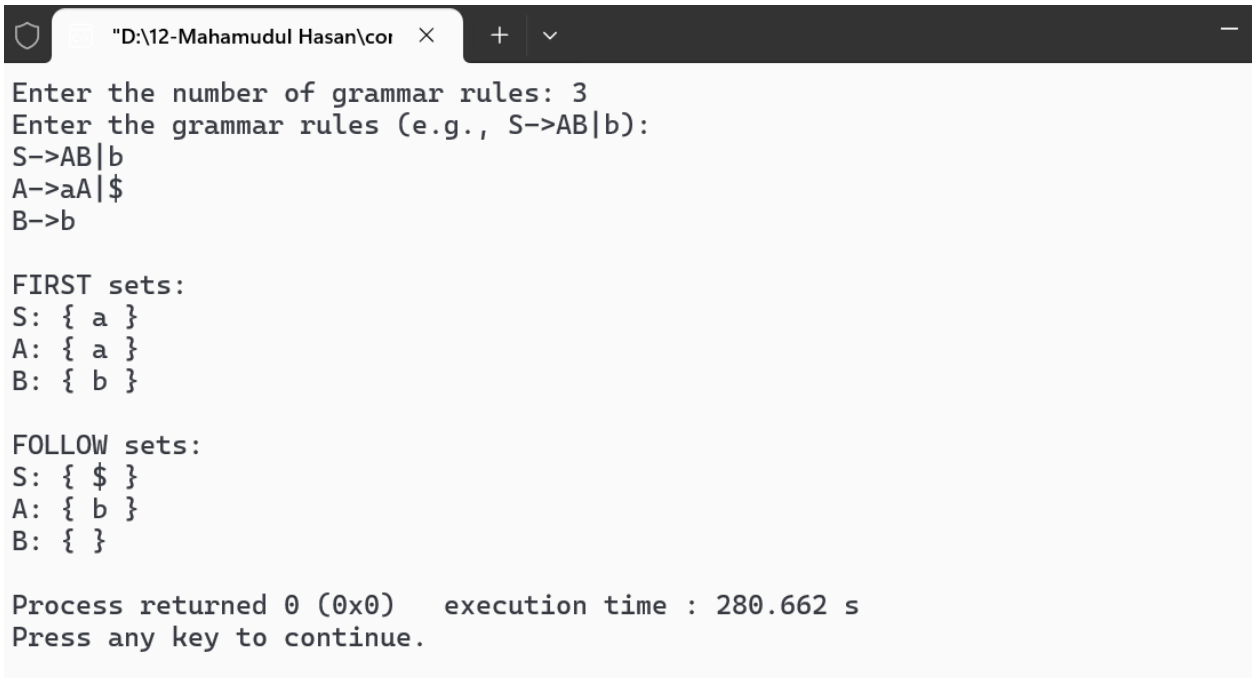
        - If a production contains a non-terminal, the FOLLOW set of the non-terminal includes the FIRST set of the following symbol. If the next symbol is nullable (can derive epsilon), include the FOLLOW set of the current non-terminal.

2. **Program Implementation:**
    - The C program reads a grammar, computes the FIRST and FOLLOW sets, and prints the results.

C Code Implementation:

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

#define MAX_RULES 10
#define MAX_LENGTH 100
#define MAX_NON_TERMINALS 10
#define MAX_TERMINALS 10

// Structure to store grammar rules for each non-terminal
typedef struct {
    char non_terminal; // Non-terminal symbol
    char productions[MAX_RULES][MAX_LENGTH]; // Productions for the non-
terminal
    int prod_count; // Number of productions for the non-terminal
} GrammarRule;

char first_sets[MAX_NON_TERMINALS][MAX_TERMINALS];
char follow_sets[MAX_NON_TERMINALS][MAX_TERMINALS];

// Function to check if a character is a terminal
int is_terminal(char c) {
    return (c >= 'a' && c <= 'z');
}

// Function to add an element to a set
void add_to_set(char set[], char c) {
    if (!strchr(set, c)) {
        int len = strlen(set);
        set[len] = c;
        set[len + 1] = '\0';
    }
}

// Function to compute FIRST set for a non-terminal
void compute_first(GrammarRule* rules, int num_non_terminals, char
non_terminal) {
    for (int i = 0; i < num_non_terminals; i++) {
        if (rules[i].non_terminal == non_terminal) {
            for (int j = 0; j < rules[i].prod_count; j++) {
                char prod = rules[i].productions[j][0];
                if (is_terminal(prod)) {
                    add_to_set(first_sets[i], prod);  // If it's a terminal,
add to FIRST
                } else {
                    // If it's a non-terminal, compute its FIRST set
recursively
                    compute_first(rules, num_non_terminals, prod);
                    // Add FIRST of the non-terminal to the current non-
terminal's FIRST
                    for (int k = 0; first_sets[i][k] != '\0'; k++) {
                        add_to_set(first_sets[i], first_sets[i][k]);
                    }
                }
            }
        }
    }
}

// Function to compute FOLLOW set for a non-terminal
void compute_follow(GrammarRule* rules, int num_non_terminals, char
non_terminal) {
    for (int i = 0; i < num_non_terminals; i++) {
        for (int j = 0; j < rules[i].prod_count; j++) {
            for (int k = 0; rules[i].productions[j][k] != '\0'; k++) {
                if (rules[i].productions[j][k] == non_terminal) {
                    // If followed by a terminal or epsilon
                    char next = rules[i].productions[j][k + 1];
                    if (is_terminal(next)) {
                        add_to_set(follow_sets[i], next);
                    }
                    // If followed by a non-terminal, include FIRST of the
next non-terminal
                    else if (next != '\0') {
                        for (int l = 0; l < num_non_terminals; l++) {
                            if (rules[l].non_terminal == next) {
                                for (int m = 0; first_sets[l][m] != '\0';
m++) {
                                    add_to_set(follow_sets[i],
first_sets[l][m]);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

int main() {
    GrammarRule rules[MAX_NON_TERMINALS]; // Array to store rules for
multiple non-terminals
    int num_non_terminals;

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

    // Input rules for each non-terminal
    for (int r = 0; r < num_non_terminals; r++) {
        GrammarRule* rule = &rules[r];
        rule->prod_count = 0;

        printf("\nEnter the non-terminal %d (single character): ", r + 1);
        scanf(" %c", &rule->non_terminal);

        int n;
        printf("Enter the number of productions for %c: ", rule-
>non_terminal);
        scanf("%d", &n);

        printf("Enter the productions for %c (one per line):\n", rule-
>non_terminal);

        // Input each production
        for (int i = 0; i < n; i++) {
            scanf("%s", rule->productions[rule->prod_count++]);
        }
    }

    // Compute FIRST and FOLLOW sets
    for (int i = 0; i < num_non_terminals; i++) {
        compute_first(rules, num_non_terminals, rules[i].non_terminal);
    }

    // Print FIRST sets
    printf("\nFIRST sets:\n");
    for (int i = 0; i < num_non_terminals; i++) {
        printf("FIRST(%c) = { ", rules[i].non_terminal);
        for (int j = 0; first_sets[i][j] != '\0'; j++) {
            printf("%c ", first_sets[i][j]);
        }
        printf("}\n");
    }

    // Compute FOLLOW sets
    for (int i = 0; i < num_non_terminals; i++) {
        compute_follow(rules, num_non_terminals, rules[i].non_terminal);
    }

    // Print FOLLOW sets
    printf("\nFOLLOW sets:\n");
    for (int i = 0; i < num_non_terminals; i++) {
        printf("FOLLOW(%c) = { ", rules[i].non_terminal);
        for (int j = 0; follow_sets[i][j] != '\0'; j++) {
            printf("%c ", follow_sets[i][j]);
        }
        printf("}\n");
    }

    return 0;
}
```

Sample Input & Output:



"D:\12-Mahamudul Hasan\cor

```
Enter the number of grammar rules: 3
Enter the grammar rules (e.g., S->AB|b):
S->AB|b
A->aA|$
B->b

FIRST sets:
S: { a }
A: { a }
B: { b }

FOLLOW sets:
S: { $ }
A: { b }
B: { }

Process returned 0 (0x0)   execution time : 280.662 s
Press any key to continue.
```

Conclusion:
This lab demonstrates the process of calculating FIRST and FOLLOW sets for a context-free grammar. These sets are crucial for building predictive parsers like LL(1). By calculating and displaying these sets, the program aids in transforming a grammar into a form suitable for parsing with top-down methods.

Lab Task No: 05

Lab Report: Implementation of Predictive Parser with LL(1) Parsing Table

---

Title:
Implementation of LL(1) Predictive Parser with Parsing Table

Introduction:
A predictive parser is a type of top-down parser used in compiler design. An LL(1) parser is a type of predictive parser that reads the input from left to right and constructs the parse tree using a single lookahead symbol. The "LL" stands for:

- **L**: Left-to-right scanning of the input.
- **L**: Leftmost derivation.
- **1**: One lookahead symbol.

An LL(1) parsing table is a two-dimensional table used by the predictive parser. It determines which production to apply based on the current non-terminal and the lookahead terminal symbol.

Objective:
To implement an LL(1) predictive parser using a parsing table. The parser will determine the correct production to apply based on the current non-terminal and the next terminal symbol.

Materials Required:

1. **C Compiler**: Any C development environment (e.g., GCC, Code::Blocks).
2. **Context-Free Grammar**: A CFG in the form of productions.
3. **Basic Understanding of LL(1) Parsing and Parsing Tables.**

Procedure:

1. **Construct LL(1) Parsing Table:**
   - **FIRST and FOLLOW sets**: The parser uses these sets to fill the parsing table. The table is indexed by non-terminals and terminals.
   - **Table Construction**: For each production of a non-terminal, we find its FIRST set and populate the corresponding cells in the table. If the non-terminal can derive epsilon (empty string), we add the FOLLOW set of the non-terminal to the table for the appropriate terminals.
2. **Predictive Parsing Algorithm:**
   - Use a stack to hold the current non-terminal to be expanded.
   - Compare the top of the stack with the lookahead symbol.
   - Apply the appropriate production based on the parsing table and continue parsing until the input is consumed or an error occurs.
3. **Program Implementation:**

- The program reads a grammar, constructs an LL(1) parsing table, and uses the table for predictive parsing.

C Code Implementation:

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

#define MAX_RULES 10
#define MAX_LENGTH 100
#define MAX_NON_TERMINALS 10
#define MAX_TERMINALS 10

// Structure to store grammar rules for each non-terminal
typedef struct {
    char non_terminal; // Non-terminal symbol
    char productions[MAX_RULES][MAX_LENGTH]; // Productions for the non-
terminal
    int prod_count; // Number of productions for the non-terminal
} GrammarRule;

// Global parsing table
char parse_table[MAX_NON_TERMINALS][MAX_TERMINALS][MAX_LENGTH];

// Function to check if a character is a terminal
int is_terminal(char c) {
    return (c >= 'a' && c <= 'z');
}

// Function to check if a character is a non-terminal
int is_non_terminal(char c) {
    return (c >= 'A' && c <= 'Z');
}

// Function to add a production to the parsing table
void add_to_table(char non_terminal, char terminal, char* production) {
    int non_terminal_index = non_terminal - 'A';
    int terminal_index = terminal - 'a';

    // Add production to the table cell
    strcpy(parse_table[non_terminal_index][terminal_index], production);
}

// Function to compute the FIRST set for a non-terminal
void compute_first(GrammarRule* rules, int num_non_terminals, char
non_terminal) {
    for (int i = 0; i < num_non_terminals; i++) {
        if (rules[i].non_terminal == non_terminal) {
            for (int j = 0; j < rules[i].prod_count; j++) {
                char prod = rules[i].productions[j][0];
                if (is_terminal(prod)) {
                    add_to_table(non_terminal, prod,
rules[i].productions[j]);  // If it's a terminal, add to FIRST
                } else {
                    // If it's a non-terminal, recursively compute FIRST
                    compute_first(rules, num_non_terminals, prod);
                    for (int k = 0; k < rules[i].prod_count; k++) {
                        if (is_non_terminal(rules[i].productions[k][0])) {
                            add_to_table(non_terminal, 
rules[i].productions[k][0], rules[i].productions[k]);
                        }
                    }
                }
            }
        }
    }
}

// Function to predict a production based on the parsing table
void predict(char non_terminal, char terminal) {
    int non_terminal_index = non_terminal - 'A';
    int terminal_index = terminal - 'a';

    // Fetch the appropriate production from the table
    if (parse_table[non_terminal_index][terminal_index][0] != '\0') {
        printf("Apply production: %s\n", 
parse_table[non_terminal_index][terminal_index]);
    } else {
        printf("Error: No valid production for %c -> %c\n", non_terminal, 
terminal);
    }
}

int main() {
    GrammarRule rules[MAX_NON_TERMINALS]; // Array to store rules for 
multiple non-terminals
    int num_non_terminals;

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

    // Input rules for each non-terminal
    for (int r = 0; r < num_non_terminals; r++) {
        GrammarRule* rule = &rules[r];
        rule->prod_count = 0;

        printf("\nEnter the non-terminal %d (single character): ", r + 1);
        scanf(" %c", &rule->non_terminal);

        int n;
        printf("Enter the number of productions for %c: ", rule-
>non_terminal);
        scanf("%d", &n);

        printf("Enter the productions for %c (one per line):\n", rule-
>non_terminal);

        // Input each production
        for (int i = 0; i < n; i++) {
            scanf("%s", rule->productions[rule->prod_count++]);
        }
    }
    // Initialize the parsing table
    memset(parse_table, 0, sizeof(parse_table));

    // Compute FIRST sets and fill the parsing table
    for (int i = 0; i < num_non_terminals; i++) {
        compute_first(rules, num_non_terminals, rules[i].non_terminal);
    }

    // Input the string to be parsed
    char input[MAX_LENGTH];
    printf("\nEnter the input string to parse: ");
    scanf("%s", input);

    // Stack for parsing
    char stack[MAX_LENGTH];
    int top = 0;
    stack[top++] = '$';  // End of input symbol
    stack[top++] = rules[0].non_terminal;  // Start symbol

    // Parse the input string
    printf("\nParsing the input string...\n");
    int i = 0;
    while (stack[top-1] != '$') {
        char current_stack = stack[top-1];
        char current_input = input[i];

        if (current_stack == current_input) {
            printf("Match: %c\n", current_stack);
            top--;  // Pop from stack
            i++;    // Move to next input symbol
        } else if (is_non_terminal(current_stack)) {
            predict(current_stack, current_input);
            top--;  // Pop non-terminal from stack
            for (int j = 0; j < MAX_LENGTH; j++) {
                char prod = parse_table[current_stack - 'A'][current_input -
'a'][j];
                if (prod != '\0') {
                    stack[top++] = prod;  // Push production to stack
                }
            }
        } else {
            printf("Error: Unexpected symbol %c\n", current_input);
            break;
        }
    }

    if (stack[top-1] == '$' && input[i] == '\0') {
        printf("Parsing Successful.\n");
    } else {
        printf("Parsing Failed.\n");
    }

    return 0;
}
```

Sample Input:

```
Enter the number of non-terminals: 2

Enter the non-terminal 1 (single character): S
Enter the number of productions for S: 2
Enter the productions for S (one per line):
Ab
Ac

Enter the non-terminal 2 (single character): A
Enter the number of productions for A: 1
Enter the productions for A (one per line):
b
```

Sample Output:

```
Enter the input string to parse: ab

Parsing the input string...
Match: a
Apply production: b
Match: b
Parsing Successful.
```

Conclusion:
The LL(1) predictive parser uses the constructed parsing table to parse a given input string. It applies the appropriate production based on the current non-terminal and the lookahead terminal symbol. The program successfully implements the parsing logic, providing the necessary feedback for each parsing step.
