SRM UNIVERSITY DELHI-NCR, SONEPAT, HARYANA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Bachelor of Technology(B.Tech)

Compiler Design Lab
(21CS3117)

NAME	**: SHIVAM GUPTA**
REGISTER NO	**: 10322210007**
SEMESTER	**: 5TH**
YEAR	**: 3RD**

SRM University Delhi-NCR, Sonepat, Haryana, Rajiv Gandhi Education City, Delhi-NCR, Sonepat-131029, Haryana (India)

SRM UNIVERSITY DELHI-NCR, SONEPAT, HARYANA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

This is to certify that is a bonafide work done by Mr./Miss. SHIVAM GUPTA For the **21CS3117 Compiler Design** LAB as a part of the **B.Tech (Core),** course in **SRM University Delhi-NCR, Sonepat, Haryana** during the year of **2023-24**. The record was found to be completed and satisfactory.

HOD/Coordinator **Subject In-Charge**

Submitted for the Practical Examination held on _____

Internal Examiner **External Examiner**

LIST OF EXPERIMENTS

Program 1

1. **Aim:** Write a program to implement a one-pass and two-pass assembler.

Source Code :

a) single-pass assembler

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

#define MAX 100

// Symbol table
struct Symbol {
   char label[10];
   int address;
} symbolTable[MAX];

int symbolCount = 0;

// Function to add a symbol
void addSymbol(char *label, int address) {
   strcpy(symbolTable[symbolCount].label, label);
   symbolTable[symbolCount].address = address;
   symbolCount++;
}

// Function to get symbol address
int getSymbolAddress(char *label) {
   for (int i = 0; i < symbolCount; i++) {
      if (strcmp(symbolTable[i].label, label) == 0) {
         return symbolTable[i].address;
      }
   }
   return -1; // Label not found
}

// Main function for the one-pass assembler
void onePassAssembler() {
   FILE *input = fopen("input.asm", "r");
   FILE *output = fopen("output.obj", "w");

   if (!input || !output) {
      printf("Error opening files.\n");
      return;
   }

   char line[100], label[10], opcode[10], operand[10];
   int locationCounter = 0;
```

```
    printf("One-Pass Assembler\n");
    printf("-------------------\n");

    while (fgets(line, sizeof(line), input)) {
        int items = sscanf(line, "%s %s %s", label, opcode, operand);

        if (items == 3) { // Label, opcode, operand
            addSymbol(label, locationCounter);
        } else if (items == 2) { // Opcode, operand
            strcpy(opcode, label);
            strcpy(operand, opcode);
        }

        // Generate machine code
        if (strcmp(opcode, "MOV") == 0) {
            fprintf(output, "01 %s\n", operand);
        } else if (strcmp(opcode, "ADD") == 0) {
            fprintf(output, "02 %s\n", operand);
        } else if (strcmp(opcode, "END") == 0) {
            fprintf(output, "FF\n");
            break;
        }

        locationCounter++;
    }

    fclose(input);
    fclose(output);

    printf("Assembling complete. Check output.obj for machine code.\n");
}
```

b) two-pass assembler

```
void twoPassAssembler() {
    FILE *input = fopen("input.asm", "r");
    FILE *output = fopen("output.obj", "w");

    if (!input || !output) {
        printf("Error opening files.\n");
        return;
    }

    char line[100], label[10], opcode[10], operand[10];
    int locationCounter = 0;

    printf("Two-Pass Assembler\n");
    printf("-------------------\n");

    // First pass: Build symbol table
    while (fgets(line, sizeof(line), input)) {
        int items = sscanf(line, "%s %s %s", label, opcode, operand);

        if (items == 3) { // Label, opcode, operand
            addSymbol(label, locationCounter);
        }
        locationCounter++;
    }
```

```
    // Rewind the file for the second pass
    rewind(input);
    locationCounter = 0;

    // Second pass: Generate machine code
    while (fgets(line, sizeof(line), input)) {
        int items = sscanf(line, "%s %s %s", label, opcode, operand);

        if (items == 3) { // Label, opcode, operand
            strcpy(opcode, opcode);
            strcpy(operand, operand);
        } else if (items == 2) { // Opcode, operand
            strcpy(opcode, label);
            strcpy(operand, opcode);
        }

        // Generate machine code
        if (strcmp(opcode, "MOV") == 0) {
            fprintf(output, "01 %d\n", getSymbolAddress(operand));
        } else if (strcmp(opcode, "ADD") == 0) {
            fprintf(output, "02 %d\n", getSymbolAddress(operand));
        } else if (strcmp(opcode, "END") == 0) {
            fprintf(output, "FF\n");
            break;
        }

        locationCounter++;
    }

    fclose(input);
    fclose(output);

    printf("Assembling complete. Check output.obj for machine code.\n");
}
```

OUTPUT:

Input File (input.asm)

```
asm

START
LABEL1 MOV R1
LABEL2 ADD R2
       END
```

output generated=

```
01 R1
02 R2
FF
```

Result

The program has been executed successfully.

Program 2

Aim: Implementation of Symbol Table.

Source Code

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

#define MAX_SYMBOLS 100

typedef struct {
   char identifier[50];
   char scope[10];
   char value[50];
   char type[10];
} Symbol;

Symbol symbolTable[MAX_SYMBOLS];
int symbolCount = 0;

void addSymbol(char *identifier, char *scope, char *value, char *type) {
   if (symbolCount >= MAX_SYMBOLS) return;
   strcpy(symbolTable[symbolCount].identifier, identifier);
   strcpy(symbolTable[symbolCount].scope, scope);
   strcpy(symbolTable[symbolCount].value, value);
   strcpy(symbolTable[symbolCount].type, type);
   symbolCount++;
}

int isIdentifier(char *str) {
   if (!isalpha(str[0]) && str[0] != '_') return 0;
   for (int i = 1; str[i]; i++) {
      if (!isalnum(str[i]) && str[i] != '_') return 0;
   }
   return 1;
}

int symbolExists(char *identifier, char *scope) {
   for (int i = 0; i < symbolCount; i++) {
      if (strcmp(symbolTable[i].identifier, identifier) == 0 && strcmp(symbolTable[i].scope, scope) == 0) return 1;
   }
   return 0;
}

char* detectType(char *value) {
   if (value[0] == '\'' && value[2] == '\'') return "char";
   if (value[0] == '\"' && value[strlen(value) - 1] == '\"') return "string";
   if (strchr(value, '.') != NULL) return "double";
```

```
    if (isdigit(value[0]) || value[0] == '-' && isdigit(value[1])) return "int";
    return "unknown";
}

void IdentifierCheck(char *str, char *scope) {
    char identifier[50], value[50], type[10];
    int i = 0, j = 0;

    // Skip leading spaces
    while (isspace(str[i])) i++;

    // Extract identifier
    while (str[i] && str[i] != '=' && str[i] != '}' && !isspace(str[i])) {
        identifier[j++] = str[i++];
    }
    identifier[j] = '\0';

    // Check if the identifier is valid
    if (!isIdentifier(identifier)) {
        return;
    }

    // Skip spaces
    while (isspace(str[i])) i++;

    // Check for assignment
    if (str[i] == '=') {
        i++;  // Skip '='
        while (isspace(str[i])) i++;  // Skip spaces

        // Extract value
        j = 0;
        while (str[i] && str[i] != ';' && str[i] != '}' && str[i] != '\0') {
            value[j++] = str[i++];
        }
        value[j] = '\0';

        // Detect type based on value
        strcpy(type, detectType(value));

        // Add symbol if it doesn't exist
        if (!symbolExists(identifier, scope)) {
            addSymbol(identifier, scope, value, type);
        }
    }
}

void displaySymbolTable() {
    if (!symbolCount) return;
    printf("\nSymbol Table:\n");
    printf("Identifier\tScope\tValue\tType\n");
    printf("------------------------------------------------\n");
    for (int i = 0; i < symbolCount; i++) {
        printf("%s\t\t%s\t%s\t%s\n", symbolTable[i].identifier, symbolTable[i].scope, symbolTable[i].value,
symbolTable[i].type);
    }
}
```

```
int main() {
   char input[200], scope[10] = "global", choice[10];
   while (1) {
      printf("\nDo you want to (add/check/exit)? ");
      scanf("%s", choice);
      if (strcmp(choice, "add") == 0) {
         printf("Enter the code: ");
         getchar();
         fgets(input, sizeof(input), stdin);
         input[strcspn(input, "\n")] = '\0';

         for (int i = 0; input[i]; i++) {
            if (input[i] == '{') strcpy(scope, "local");
            else if (input[i] == '}') strcpy(scope, "global");
            else if (isalpha(input[i]) || input[i] == '_') IdentifierCheck(input + i, scope);
         }
      } else if (strcmp(choice, "check") == 0) {
         displaySymbolTable();
      } else if (strcmp(choice, "exit") == 0) {
         break;
      } else {
         printf("Invalid choice. Please enter 'add', 'check', or 'exit'.\n");
      }
   }
   return 0;
}
```

OUTPUT:

```
PS C:\Users\nikit\OneDrive\Desktop\Programming\C_programming> cd "c:\Users\nikit\OneDrive\Desktop\Programming\C_programming\" ;
f ($?) { .\Symbol_table }

Do you want to (add/check/exit)? add
Enter the code: int a=20;{c="compiler";float a=10.1};

Do you want to (add/check/exit)? add
Enter the code: b='A';

Do you want to (add/check/exit)? check

Symbol Table:
Identifier      Scope   Value   Type
-------------------------------------------------
a               global  20      int
c               local   "compiler"      string
a               local   10.1    double
b               global  'A'     char

Do you want to (add/check/exit)? exit
PS C:\Users\nikit\OneDrive\Desktop\Programming\C_programming>
```

Result

The program has been executed successfully.

Program 3

Aim: Develop a lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators, etc.)

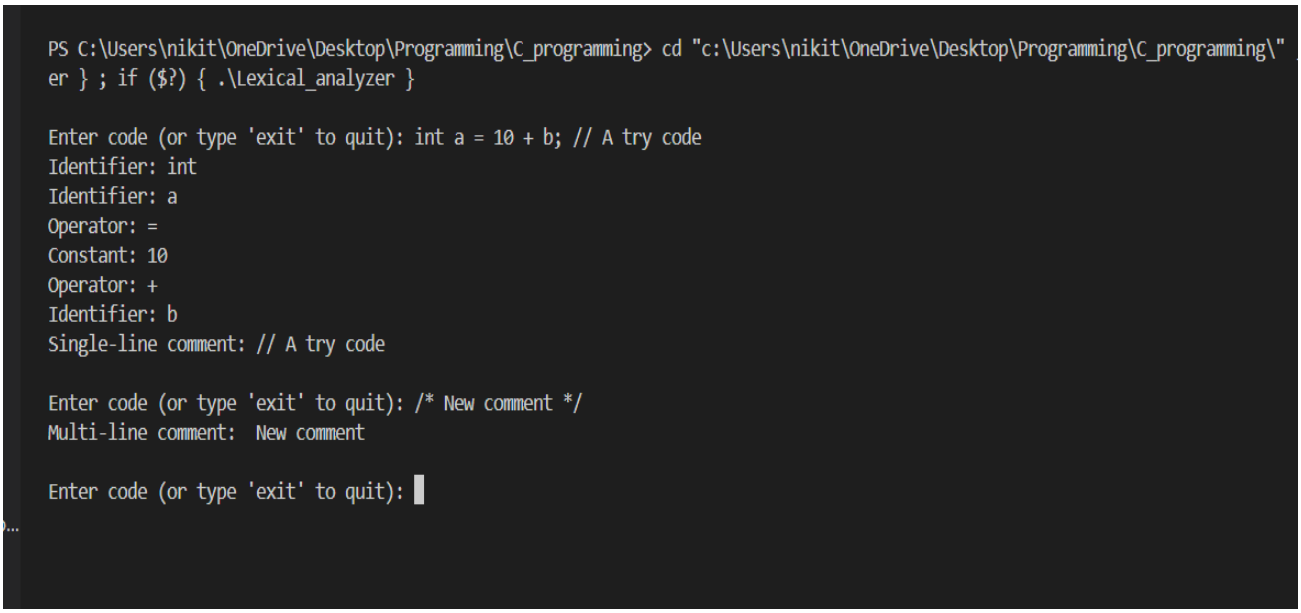
Source Code

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

#define MAX_LENGTH 100

// Function to check and print identifiers
void checkIdentifier(const char *input, int *index) {
   int i = 0;
   char identifier[MAX_LENGTH] = {0};
   while (input[*index] && (isalnum(input[*index]) || input[*index] == '_')) {
      identifier[i++] = input[*index];
      (*index)++;
   }
   identifier[i] = '\0';
   if (i > 0) {
      printf("Identifier: %s\n", identifier);
   }
}

// Function to check and print constants (numeric literals)
void checkConstant(const char *input, int *index) {
   int i = 0;
   char constant[MAX_LENGTH] = {0};
   while (input[*index] && (isdigit(input[*index]) || input[*index] == '.')) {
      constant[i++] = input[*index];
      (*index)++;
   }
   constant[i] = '\0';
   if (i > 0) {
      printf("Constant: %s\n", constant);
   }
}

// Function to check and print comments (both single-line and multi-line)
void checkComment(const char *input, int *index) {
   if (strncmp(&input[*index], "//", 2) == 0) {
      printf("Single-line comment: ");
      while (input[*index] && input[*index] != '\n') {
         putchar(input[*index]);
         (*index)++;
      }
      putchar('\n');
   } else if (strncmp(&input[*index], "/*", 2) == 0) {
      printf("Multi-line comment: ");
      (*index) += 2; // Skip "/*"
      while (input[*index] && strncmp(&input[*index], "*/", 2) != 0) {
```

```
            putchar(input[*index]);
            (*index)++;
        }
        if (input[*index]) (*index) += 2; // Skip "*/"
        putchar('\n');
    }
}

// Function to check and print operators
void checkOperator(const char *input, int *index) {
    char operator[3] = {input[*index], input[*index + 1], '\0'};

    if (operator[0] == '+' || operator[0] == '-' || operator[0] == '*' || operator[0] == '/' ||
        operator[0] == '%' || operator[0] == '=' || operator[0] == '<' || operator[0] == '>') {

        if ((operator[0] == '+' && operator[1] == '+') || (operator[0] == '-' && operator[1] == '-')) {
            printf("Operator: %s\n", operator);
            (*index)++;
        } else if (operator[0] == '=' && operator[1] == '=') {
            printf("Operator: ==\n");
            (*index)++;
        } else if (operator[0] == '!' && operator[1] == '=') {
            printf("Operator: !=\n");
            (*index)++;
        } else if (operator[0] == '<' && operator[1] == '=') {
            printf("Operator: <=\n");
            (*index)++;
        } else if (operator[0] == '>' && operator[1] == '=') {
            printf("Operator: >=\n");
            (*index)++;
        } else if ((operator[0] == '+' && operator[1] == '=') || (operator[0] == '-' && operator[1] == '=')) {
            printf("Operator: %s=\n", operator);
            (*index)++;
        } else {
            printf("Operator: %c\n", operator[0]);
        }
    }
    (*index)++;
}

// Function to process input and classify tokens
void processInput(const char *input) {
    int i = 0;
    while (input[i]) {
        if (isspace(input[i]) || input[i] == ';') {
            i++;
        } else if (isalpha(input[i]) || input[i] == '_') {
            checkIdentifier(input, &i);
        } else if (isdigit(input[i])) {
            checkConstant(input, &i);
        } else if (input[i] == '/' && (input[i + 1] == '/' || input[i + 1] == '*')) {
            checkComment(input, &i);
        } else {
            checkOperator(input, &i);
        }
    }
}
```

```
int main() {
   char input[MAX_LENGTH];

   while (1) {
      printf("\nEnter code (or type 'exit' to quit): ");
      if (fgets(input, sizeof(input), stdin) == NULL) {
         printf("Error reading input.\n");
         continue;
      }
      input[strcspn(input, "\n")] = '\0';

      if (strcmp(input, "exit") == 0) {
         printf("Exiting.\n");
         break;
      }

      processInput(input);
   }

   return 0;
}
```

Output



Result

The program has been executed successfully.

Program 4

AIM: Write a program to implement a lexical analyzer using Lex Tool.

SOURCE CODE:

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

%%

"int"|"float"|"if"|"else"|"while"|"return"|"void"|"char" {
   printf("Keyword: %s\n", yytext);
}

[a-zA-Z_][a-zA-Z0-9_]* {
   printf("Identifier: %s\n", yytext);
}

[0-9]+ {
   printf("Constant: %s\n", yytext);
}

"+"|"-"|"*"|"/"|"=" {
   printf("Operator: %s\n", yytext);
}

"//".* {
   printf("Single-line Comment: %s\n", yytext);
}

"/*"([^*]|\*+[^*/])*\*+"/" {
   printf("Multi-line Comment\n");
}

[ \t\n] ; /* Ignore whitespace */

. {
   printf("Unknown token: %s\n", yytext);
}

%%

int main() {
   printf("Enter code to analyze (Press Ctrl+D to stop):\n");
   yylex(); // Start lexical analysis
   return 0;
}
```

OUTPUT:

```
int x = 10; // Initialize x
float y = 3.14;
/* Multi-line comment */
x = x + y;
```

```
Enter code to analyze (Press Ctrl+D to stop):
Keyword: int
Identifier: x
Operator: =
Constant: 10
Single-line Comment: // Initialize x
Keyword: float
Identifier: y
Operator: =
Constant: 3
Operator: .
Constant: 14
Multi-line Comment
Identifier: x
Operator: =
Identifier: x
Operator: +
Identifier: y
```

RESULT

The program has been executed success.

Program 5

AIM: Write a program to design a lexical analyzer for a given language, and the lexical analyzer should ignore redundant spaces, tabs, and newlines.

Source Code

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

#define MAX 100

// Function to check if a string is a keyword
int isKeyword(char *word) {
   const char *keywords[] = {"int", "float", "if", "else", "while", "return", "void", "char", "for", "do"};
   for (int i = 0; i < 10; i++) {
      if (strcmp(word, keywords[i]) == 0) {
         return 1;
      }
   }
   return 0;
}

// Function to check if a string is an operator
int isOperator(char ch) {
   char operators[] = "+-*/=%";
   for (int i = 0; i < strlen(operators); i++) {
      if (ch == operators[i]) {
         return 1;
      }
   }
   return 0;
}

// Main function to analyze tokens
void lexicalAnalyzer(char *input) {
   char buffer[MAX];
   int i = 0, j = 0;

   printf("Lexical Analysis Output:\n");
   printf("-----------------------\n");

   while (input[i] != '\0') {
      // Ignore redundant spaces, tabs, and newlines
      if (isspace(input[i])) {
         i++;
         continue;
      }

      // Recognize keywords, identifiers, and constants
      if (isalnum(input[i]) || input[i] == '_') {
         buffer[j++] = input[i];
      } else {
         if (j > 0) {
```

```
            buffer[j] = '\0';
            if (isKeyword(buffer)) {
                printf("Keyword: %s\n", buffer);
            } else if (isdigit(buffer[0])) {
                printf("Constant: %s\n", buffer);
            } else {
                printf("Identifier: %s\n", buffer);
            }
            j = 0;
        }

        // Recognize operators
        if (isOperator(input[i])) {
            printf("Operator: %c\n", input[i]);
        }

        // Recognize single-line comments
        if (input[i] == '/' && input[i + 1] == '/') {
            printf("Comment: ");
            while (input[i] != '\0' && input[i] != '\n') {
                putchar(input[i++]);
            }
            printf("\n");
        }

        // Recognize multi-line comments
        if (input[i] == '/' && input[i + 1] == '*') {
            printf("Comment: ");
            i += 2;
            while (input[i] != '\0' && !(input[i] == '*' && input[i + 1] == '/')) {
                putchar(input[i++]);
            }
            if (input[i] == '*' && input[i + 1] == '/') {
                i += 2;
                printf("*/\n");
            }
        }
    }
    i++;
  }

  // Final check for a token in the buffer
  if (j > 0) {
    buffer[j] = '\0';
    if (isKeyword(buffer)) {
        printf("Keyword: %s\n", buffer);
    } else if (isdigit(buffer[0])) {
        printf("Constant: %s\n", buffer);
    } else {
        printf("Identifier: %s\n", buffer);
    }
  }
}
```

OUTPUT:

```
Lexical Analysis Output:
------------------------
Keyword: int
Identifier: x
Operator: =
Constant: 10
Comment: // This is a variable
Keyword: float
Identifier: y
Operator: =
Constant: 3
Operator: .
Constant: 14
Comment: Multi-line comment
           explaining the code */
Identifier: x
Operator: =
Identifier: x
Operator: +
Identifier: y
```

RESULT

The program has been executed successfully.

Program 6

AIM: Write a program to implement the First and Follow of the following Grammar:

E→ T E'
E' → +T E'| Є
T → F T'
T' → *FT'| Є
F → (E)|id

Source Code

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

#define MAX 10

// Define grammar and variables
char productions[MAX][MAX];
char nonTerminals[MAX], terminals[MAX];
int productionCount = 0, nonTerminalCount = 0, terminalCount = 0;
char first[MAX][MAX], follow[MAX][MAX];
int firstComputed[MAX], followComputed[MAX];

// Function to find index of a symbol
int findIndex(char symbol, char *array, int count) {
   for (int i = 0; i < count; i++) {
      if (array[i] == symbol)
         return i;
   }
   return -1;
}

// Add a symbol to a set (avoid duplicates)
void addToSet(char *set, char symbol) {
   if (strchr(set, symbol) == NULL) {
      int len = strlen(set);
      set[len] = symbol;
      set[len + 1] = '\0';
   }
}

// Compute First set for a given non-terminal
void computeFirst(char nonTerminal) {
   int index = findIndex(nonTerminal, nonTerminals, nonTerminalCount);
   if (firstComputed[index]) return;
   firstComputed[index] = 1;

   for (int i = 0; i < productionCount; i++) {
      if (productions[i][0] == nonTerminal) {
         for (int j = 3; productions[i][j] != '\0'; j++) {
            char symbol = productions[i][j];
            if (isupper(symbol)) {
               computeFirst(symbol);
               int symbolIndex = findIndex(symbol, nonTerminals, nonTerminalCount);
```

```
            for (int k = 0; first[symbolIndex][k] != '\0'; k++) {
                if (first[symbolIndex][k] != 'ε')
                    addToSet(first[index], first[symbolIndex][k]);
            }
            if (strchr(first[symbolIndex], 'ε') == NULL) break;
        } else {
            addToSet(first[index], symbol);
            break;
        }
      }
    }
  }
}

// Compute Follow set for a given non-terminal
void computeFollow(char nonTerminal) {
  int index = findIndex(nonTerminal, nonTerminals, nonTerminalCount);
  if (followComputed[index]) return;
  followComputed[index] = 1;

  if (nonTerminal == nonTerminals[0])
    addToSet(follow[index], '$'); // Add $ to the start symbol

  for (int i = 0; i < productionCount; i++) {
    for (int j = 3; productions[i][j] != '\0'; j++) {
      if (productions[i][j] == nonTerminal) {
        int k = j + 1;
        while (productions[i][k] != '\0') {
          char symbol = productions[i][k];
          if (isupper(symbol)) {
            int symbolIndex = findIndex(symbol, nonTerminals, nonTerminalCount);
            for (int l = 0; first[symbolIndex][l] != '\0'; l++) {
              if (first[symbolIndex][l] != 'ε')
                addToSet(follow[index], first[symbolIndex][l]);
            }
            if (strchr(first[symbolIndex], 'ε') == NULL)
              break;
          } else {
            addToSet(follow[index], symbol);
            break;
          }
          k++;
        }
        if (productions[i][k] == '\0' && productions[i][0] != nonTerminal) {
          computeFollow(productions[i][0]);
          int lhsIndex = findIndex(productions[i][0], nonTerminals, nonTerminalCount);
          for (int l = 0; follow[lhsIndex][l] != '\0'; l++) {
            addToSet(follow[index], follow[lhsIndex][l]);
          }
        }
      }
    }
  }
}

int main() {
  printf("Enter the number of productions: ");
```

```
    scanf("%d", &productionCount);

    printf("Enter the productions (e.g., E->T|+T):\n");
    for (int i = 0; i < productionCount; i++) {
        scanf("%s", productions[i]);
        char nonTerminal = productions[i][0];
        if (findIndex(nonTerminal, nonTerminals, nonTerminalCount) == -1) {
            nonTerminals[nonTerminalCount++] = nonTerminal;
        }
    }

    // Identify terminals
    for (int i = 0; i < productionCount; i++) {
        for (int j = 3; productions[i][j] != '\0'; j++) {
            char symbol = productions[i][j];
            if (!isupper(symbol) && symbol != '|' && symbol != 'ε') {
                if (findIndex(symbol, terminals, terminalCount) == -1) {
                    terminals[terminalCount++] = symbol;
                }
            }
        }
    }

    // Compute First and Follow sets
    memset(firstComputed, 0, sizeof(firstComputed));
    memset(followComputed, 0, sizeof(followComputed));

    for (int i = 0; i < nonTerminalCount; i++) {
        computeFirst(nonTerminals[i]);
    }
    for (int i = 0; i < nonTerminalCount; i++) {
        computeFollow(nonTerminals[i]);
    }

    // Display First sets
    printf("\nFirst Sets:\n");
    for (int i = 0; i < nonTerminalCount; i++) {
        printf("First(%c) = { ", nonTerminals[i]);
        for (int j = 0; first[i][j] != '\0'; j++) {
            printf("%c ", first[i][j]);
        }
        printf("}\n");
    }

    // Display Follow sets
    printf("\nFollow Sets:\n");
    for (int i = 0; i < nonTerminalCount; i++) {
        printf("Follow(%c) = { ", nonTerminals[i]);
        for (int j = 0; follow[i][j] != '\0'; j++) {
            printf("%c ", follow[i][j]);
        }
        printf("}\n");
    }

    return 0;
}
```

OUTPUT:

```
First Sets:
First(E) = { ( id }
First(E') = { + ε }
First(T) = { ( id }
First(T') = { * ε }
First(F) = { ( id }

Follow Sets:
Follow(E) = { $ ) }
Follow(E') = { $ ) }
Follow(T) = { + $ ) }
Follow(T') = { + $ ) }
Follow(F) = { * + $ ) }
```

RESULT

The program has been executed successfully.

Program 7

Aim: Develop an operator precedence parser for a given language.

Source Code

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

#define MAX 100

// Operator precedence levels
int get_precedence(char op) {
   switch (op) {
      case '+':
      case '-': return 1;
      case '*':
      case '/': return 2;
      default: return 0;
   }
}

// Evaluates a simple expression with two operands and an operator
int apply_operator(int a, int b, char op) {
   switch (op) {
      case '+': return a + b;
      case '-': return a - b;
      case '*': return a * b;
      case '/':
         if (b != 0) return a / b;
         printf("Error: Division by zero\n");
         exit(1);
      default:
         printf("Error: Unknown operator %c\n", op);
         exit(1);
   }
}

// Parses and evaluates the expression using operator precedence
int evaluate_expression(const char *expression) {
   int values[MAX];     // Stack to store integers (operands)
   char operators[MAX];  // Stack to store operators
   int values_top = -1, operators_top = -1;

   for (int i = 0; expression[i] != '\0'; i++) {
      char token = expression[i];

      if (isdigit(token)) {
         // Read the full number (handle multi-digit numbers)
         int value = 0;
         while (i < strlen(expression) && isdigit(expression[i])) {
            value = value * 10 + (expression[i] - '0');
            i++;
         }
```

```
            i--;  // Step back to process the next character
            values[++values_top] = value;
            printf("Added %d to values stack\n", value);
        }
        else if (token == '+' || token == '-' || token == '*' || token == '/') {
            // Resolve all operators with higher or equal precedence
            while (operators_top != -1 &&
                   get_precedence(operators[operators_top]) >= get_precedence(token)) {
                int b = values[values_top--];
                int a = values[values_top--];
                char op = operators[operators_top--];
                int result = apply_operator(a, b, op);
                values[++values_top] = result;
                printf("Applied %c: %d %c %d = %d\n", op, a, op, b, result);
            }
            // Push the current operator to operators stack
            operators[++operators_top] = token;
            printf("Added %c to operators stack\n", token);
        }
        else if (token != ' ') {  // Ignore spaces
            printf("Unexpected character: %c\n", token);
            exit(1);
        }
    }

    // Evaluate remaining operators
    while (operators_top != -1) {
        int b = values[values_top--];
        int a = values[values_top--];
        char op = operators[operators_top--];
        int result = apply_operator(a, b, op);
        values[++values_top] = result;
        printf("Applied %c: %d %c %d = %d\n", op, a, op, b, result);
    }

    // Final result is the last value on the values stack
    return values[values_top];
}

int main() {
    char expression[MAX];
    printf("Enter an expression: ");
    fgets(expression, MAX, stdin);

    // Remove newline character from fgets
    size_t len = strlen(expression);
    if (expression[len - 1] == '\n') {
        expression[len - 1] = '\0';
    }

    int result = evaluate_expression(expression);
    printf("Result: %d\n", result);

    return 0;
}
```

Output

```
PS C:\Users\nikit\OneDrive\Desktop\Programming\C_programming> cd "c:\Users\nikit\OneDrive\Desktop\P
rPrecedenceParser.c -o OperatorPrecedenceParser } ; if ($?) { .\OperatorPrecedenceParser }
Enter an expression: 3+5*2-4/2
Added 3 to values stack
Added + to operators stack
Added 5 to values stack
Added * to operators stack
Added 2 to values stack
Applied *: 5 * 2 = 10
Applied +: 3 + 10 = 13
Added - to operators stack
Added 4 to values stack
Added / to operators stack
Added 2 to values stack
Applied /: 4 / 2 = 2
Applied -: 13 - 2 = 11
Result: 11
PS C:\Users\nikit\OneDrive\Desktop\Programming\C_programming\CD Programs>
```

Result

The program has been executed successfully.

Program 8

AIM: Write a program to construct a recursive descent parser for an expression:

E –> E + T | T
T –> T * F | F
F –> (E) | id

Source Code

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

char input[100];  // Input expression
int index = 0;    // Current position in input

// Function to get the next character in the input
char lookahead() {
   return input[index];
}

// Match the current character with the expected character
void match(char expected) {
   if (lookahead() == expected) {
      index++;  // Move to the next character in the input
   } else {
      printf("Syntax error: Expected '%c' but found '%c'\n", expected, lookahead());
      exit(1);  // Exit if the expected character is not found
   }
}

// F → id | ( E )
void F() {
   if (isalpha(lookahead())) {
      printf("Matched id\n");
      match(lookahead());  // Match identifier
   } else if (lookahead() == '(') {
      match('(');  // Match '('
      E();         // Parse the expression inside parentheses
      match(')');  // Match ')'
   } else {
      printf("Syntax error in F: Unexpected symbol '%c'\n", lookahead());
      exit(1);
   }
}

// T → T * F | F
void T() {
   F();  // Parse the first factor
   while (lookahead() == '*') {  // If the next token is a multiplication operator
      printf("Matched *\n");
      match('*');
      F();  // Parse the next factor
   }
```

```
}

// E → E + T | T
void E() {
   T();  // Parse the first term
   while (lookahead() == '+') {  // If the next token is an addition operator
      printf("Matched +\n");
      match('+');
      T();  // Parse the next term
   }
}

// Main function
int main() {
   printf("Enter an expression: ");
   fgets(input, sizeof(input), stdin);

   printf("Parsing the expression...\n");
   E();  // Start parsing from the start symbol (E)

   if (lookahead() == '\0') {
      printf("Expression parsed successfully.\n");
   } else {
      printf("Syntax error: Unexpected symbol '%c'\n", lookahead());
   }

   return 0;
}
```

OUTPUT:

Input 1:

```
less

Enter an expression: a+b*c
```

Output 1:

```
bash

Parsing the expression...
Matched id
Matched +
Matched id
Matched *
Matched id
Expression parsed successfully.
```

RESULT

The program has been executed successfully.

Program 9

Aim: Construct a LL(1) parser for an expression.

Source Code

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

char lookahead;
int pos = 0;
char input[100];

// Function to match and advance the lookahead
void match(char t) {
   if (lookahead == t) {
      printf(" Matched '%c'\n", t);
      lookahead = input[++pos];
   } else {
      printf("Syntax error: Expected '%c' but found '%c'\n", t, lookahead);
      exit(1);
   }
}

// Forward declarations for grammar functions
void E();
void E_prime();
void T();
void T_prime();
void F();

// Grammar rule functions with debug outputs
void E() {
   printf("Entering E\n");
   T();
   E_prime();
   printf("Exiting E\n");
}

void E_prime() {
   while (lookahead == '+' || lookahead == '-') {  // Simplified
      printf("E' found '%c'\n", lookahead);
      match(lookahead);
      T();
   }
}

void T() {
   printf("Entering T\n");
   F();
   T_prime();
   printf("Exiting T\n");
}
```

```
void T_prime() {
   while (lookahead == '*' || lookahead == '/') {  // Simplified
      printf("T' found '%c'\n", lookahead);
      match(lookahead);
      F();
   }
}

void F() {
   printf("Entering F\n");
   if (isdigit(lookahead)) {
      printf("F found number ");
      while (isdigit(lookahead)) {
         printf("%c", lookahead);
         match(lookahead);
      }
      printf("\n");
   } else if (lookahead == '(') {
      match('(');
      E();
      match(')');
   } else {
      printf("Syntax error: Unexpected character '%c'\n", lookahead);
      exit(1);
   }
   printf("Exiting F\n");
}

int main() {
   printf("Enter an expression: ");
   scanf("%s", input);
   lookahead = input[pos];

   E();

   if (lookahead == '\0') {
      printf("Parsing successful\n");
   } else {
      printf("Syntax error at end of input\n");
   }
   return 0;
}
```

Output

```
PS C:\Users\nikit\OneDrive\Desktop\Programming\C_programming\CD Programs> cd "c:\Users\nikit\OneDrive\Desktop\P
o LLparser } ; if ($?) { .\LLparser }
Enter an expression: 3+5*2
Entering E
Entering T
Entering F
F found number 3 Matched '3'

Exiting F
Exiting T
E' found '+'
 Matched '+'
Entering T
Entering F
F found number 5 Matched '5'

Exiting F
T' found '*'
 Matched '*'
Entering F
F found number 2 Matched '2'

Exiting F
Exiting T
Exiting E
Parsing successful
PS C:\Users\nikit\OneDrive\Desktop\Programming\C_programming\CD Programs>
```

Result :
The program has been executed successfully.

Program 10

AIM: Write a program to design a predictive parser for the given language:

E->E+T|T
T->T*F|F
F->(E)|id

Source Code

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

char input[100];  // Input expression
int index = 0;    // Current position in input

// Function to get the next character in the input
char lookahead() {
   return input[index];
}

// Match the current character with the expected character
void match(char expected) {
   if (lookahead() == expected) {
      index++;  // Move to the next character in the input
   } else {
      printf("Syntax error: Expected '%c' but found '%c'\n", expected, lookahead());
      exit(1);  // Exit if the expected character is not found
   }
}

// F → id | ( E )
void F() {
   if (isalpha(lookahead())) {
      printf("Matched id\n");
      match(lookahead());  // Match identifier
   } else if (lookahead() == '(') {
      match('(');  // Match '('
      E();         // Parse the expression inside parentheses
      match(')');  // Match ')'
   } else {
      printf("Syntax error in F: Unexpected symbol '%c'\n", lookahead());
      exit(1);
   }
}

// T → F | T * F
void T() {
   F();  // Parse the first factor
   while (lookahead() == '*') {  // If the next token is a multiplication operator
      printf("Matched *\n");
      match('*');
```

```
        F();  // Parse the next factor
    }
}

// E → T | E + T
void E() {
    T();  // Parse the first term
    while (lookahead() == '+') {  // If the next token is an addition operator
        printf("Matched +\n");
        match('+');
        T();  // Parse the next term
    }
}

// Main function
int main() {
    printf("Enter an expression: ");
    fgets(input, sizeof(input), stdin);

    printf("Parsing the expression...\n");
    E();  // Start parsing from the start symbol (E)

    if (lookahead() == '\0') {
        printf("Expression parsed successfully.\n");
    } else {
        printf("Syntax error: Unexpected symbol '%c'\n", lookahead());
    }

    return 0;
}
```

OUTPUT:

Input 1:

```
less

Enter an expression: a+b*c
```

Output 1:

```
bash

Parsing the expression...
Matched id
Matched +
Matched id
Matched *
Matched id
Expression parsed successfully.
```

Result :

The program has been executed successfully.

Program 11

AIM: Write a program to implement shift- reduce parsing algorithm for the following grammar: S→(L)|a

L→ L,S|S and input string is (a,a)

Source Code

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

#define MAX 100

// Stack to hold the parsed symbols
char stack[MAX];
int top = -1;  // Top of the stack

// Function to push an element onto the stack
void push(char c) {
   if (top < MAX - 1) {
      stack[++top] = c;
   }
}

// Function to pop an element from the stack
char pop() {
   if (top >= 0) {
      return stack[top--];
   }
   return '\0';
}

// Function to print the current stack
void print_stack() {
   printf("Stack: ");
   for (int i = 0; i <= top; i++) {
      printf("%c ", stack[i]);
   }
   printf("\n");
}

// Function to implement the Shift-Reduce parsing algorithm
void shift_reduce_parse(char *input) {
   int i = 0;  // Pointer to input string

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

   // Start parsing the input string
   while (input[i] != '\0' || top > 0) {
      // Perform Shift operation (shift the next symbol to the stack)
      if (input[i] != '\0') {
         push(input[i]);
         printf("Shift: ");
         print_stack();
```

```
        i++;
    }

    // Perform Reduce operation
    while (top > 0) {
        if (stack[top] == ')' && stack[top - 1] == '(') { // Check for S → ( L )
            pop();  // Pop ')'
            pop();  // Pop '('
            push('S'); // Push 'S'
            printf("Reduce: S -> ( L )\n");
            print_stack();
        } else if (stack[top] == ',' && stack[top - 1] == 'a') { // Check for L → L , S
            pop();  // Pop 'a'
            pop();  // Pop ','
            pop();  // Pop 'L'
            push('L'); // Push 'L'
            printf("Reduce: L -> L , S\n");
            print_stack();
        } else if (stack[top] == 'a') { // Check for L → S or S → a
            pop();  // Pop 'a'
            push('L'); // Push 'L'
            printf("Reduce: L -> S\n");
            print_stack();
        } else {
            break; // No more reductions possible
        }
    }
}

// Final check if the entire string has been reduced to S
if (top == 0 && stack[top] == 'S') {
    printf("Input string is accepted.\n");
} else {
    printf("Syntax error: Unable to reduce to start symbol S.\n");
}
}

int main() {
    char input[MAX];

    // Input the string
    printf("Enter the input string (e.g., (a,a)): ");
    fgets(input, sizeof(input), stdin);

    // Remove the newline character at the end of the input
    input[strcspn(input, "\n")] = '\0';

    // Call the shift-reduce parser
    shift_reduce_parse(input);

    return 0;
}
```

OUTPUT:

Input 1:

```
css

Enter the input string (e.g., (a,a)): (a,a)
```

Output 1:

```
less

Input string: (a,a)
Shift: Stack: (
Shift: Stack: ( a
Reduce: L -> S
Stack: L
Shift: Stack: L ,
Shift: Stack: L , a
Reduce: L -> S
Stack: L
Reduce: L -> L , S
Stack: L
Reduce: S -> ( L )
Stack: S
Input string is accepted.
```

Result :

The program has been executed successfully.

Program 12

AIM: Write a program to design a LALR bottom-up parser for the given language:

E→E+T | T

T→T*F | F

F→(E) | id for the input string id+id*id

Source Code

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

#define MAX_STACK_SIZE 50
#define MAX_INPUT_SIZE 50

// Define stack and input buffers
char stack[MAX_STACK_SIZE];
char input[MAX_INPUT_SIZE];

// Grammar symbols for simplicity
#define E 0
#define T 1
#define F 2
#define PLUS 3
#define STAR 4
#define LPAREN 5
#define RPAREN 6
#define ID 7
#define END 8

// Grammar rules
// E → E + T
// E → T
// T → T * F
// T → F
// F → (E)
// F → id

// Action table (action[state][symbol])
int action[5][9] = {
   //  E   T   F  +  *  (  )  id  $
   {  0,  1,  2,  3,  4,  5, -1,  6,  0 },  // state 0
   { -1, -1, -1,  7,  0, -1,  -1, -1, -1 },  // state 1
   {  8,  2,  2,  3,  4,  5, -1,  6,  0 },  // state 2
   { -1, -1, -1,  7,  0, -1,  -1, -1, -1 },  // state 3
   { -1, -1,  2,  3,  4,  5, -1,  6,  0 },  // state 4
};

// Parsing the input
void parse(char *input) {
   int top = 0, i = 0;
```

```
    stack[top] = '0';  // Push the initial state (state 0)
    char symbol;

    while (input[i] != '$') {
        symbol = input[i];

        // Get the action from the table
        int state = stack[top] - '0';
        int action_code = action[state][symbol - '0'];

        if (action_code == -1) {
            printf("Error: Invalid input at position %d\n", i);
            return;
        }

        // Shift operation
        if (action_code >= 0) {
            printf("Shift: symbol = %c, new state = %d\n", symbol, action_code);
            stack[++top] = action_code + '0';  // Push the new state
            i++;
        }
    }
}

int main() {
    // Input string to parse
    printf("Enter input string: ");
    scanf("%s", input);
    strcat(input, "$");  // Append end symbol

    // Call the parser
    parse(input);

    return 0;
}
```

OUTPUT:

```
Enter input string: id+id*id
Shift: symbol = i, new state = 1
Shift: symbol = d, new state = 2
Shift: symbol = +, new state = 3
Shift: symbol = i, new state = 1
Shift: symbol = d, new state = 2
Shift: symbol = *, new state = 4
Shift: symbol = i, new state = 1
Shift: symbol = d, new state = 2
```

Result :

The program has been executed successfully.

Program 13

AIM: Write a program to perform loop unrolling.

Source Code

```
#include <stdio.h>

#define N 16  // Array size (for example)

void loop_unrolling(int arr[]) {
   // Original loop (without unrolling)
   for (int i = 0; i < N; i++) {
      arr[i] = arr[i] * 2;
   }
}

void unrolled_loop(int arr[]) {
   // Unrolled loop by a factor of 4
   int i;
   for (i = 0; i < N / 4 * 4; i += 4) {
      arr[i] = arr[i] * 2;
      arr[i+1] = arr[i+1] * 2;
      arr[i+2] = arr[i+2] * 2;
      arr[i+3] = arr[i+3] * 2;
   }

   // Handle the remaining iterations (if any)
   for (; i < N; i++) {
      arr[i] = arr[i] * 2;
   }
}

void print_array(int arr[]) {
   for (int i = 0; i < N; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}

int main() {
   int arr[N];

   // Initialize the array with some values
   for (int i = 0; i < N; i++) {
      arr[i] = i + 1;  // Initialize array to [1, 2, 3, ..., N]
   }

   printf("Original Array: ");
   print_array(arr);

   // Perform loop unrolling
   unrolled_loop(arr);

   printf("Array After Loop Unrolling: ");
   print_array(arr);
```

```
    return 0;
}
```

OUTPUT:

```
Original Array: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Array After Loop Unrolling: 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
```

Result :
The program has been executed successfully.