Token sepreationi:-

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

#define MAX_LINES 100
#define MAX_LENGTH 256

// Structure to hold variable information
typedef struct {
    char name[20];
    char value[20];
    int used; // Flag to indicate if the variable is used
} Variable;

// Global variables
Variable variables[MAX_LINES];
int var_count = 0;

// Function to find a variable in the list
int findVariable(char *name) {
    for (int i = 0; i < var_count; i++) {
        if (strcmp(variables[i].name, name) == 0) {
            return i;
        }
    }
    return -1;
}

// Function to add or update a variable
void addOrUpdateVariable(char *name, char *value) {
    int index = findVariable(name);
    if (index != -1) {
        strcpy(variables[index].value, value);
        variables[index].used = 1; // Mark as used
    } else {
        strcpy(variables[var_count].name, name);
        strcpy(variables[var_count].value, value);
```

```
        variables[var_count].used = 1; // Mark as used
        var_count++;
    }
}

// Function to eliminate dead code
void deadCodeElimination() {
    printf("\nDead Code Elimination:\n");
    for (int i = 0; i < var_count; i++) {
        if (!variables[i].used) {
            printf("Removing unused variable: %s\n", variables[i].name);
        } else {
            printf("%s = %s\n", variables[i].name, variables[i].value);
        }
    }
}

// Function to perform common subexpression elimination
void commonSubexpressionElimination() {
    printf("\nCommon Subexpression Elimination:\n");
    for (int i = 0; i < var_count; i++) {
        for (int j = i + 1; j < var_count; j++) {
            if (strcmp(variables[i].value, variables[j].value) == 0) {
                printf("Replacing %s with %s\n", variables[j].name,
variables[i].name);
                strcpy(variables[j].value, variables[i].name); // Replace
with common expression
            }
        }
    }
}

// Function to perform strength reduction
void strengthReduction() {
    printf("\nStrength Reduction:\n");
    for (int i = 0; i < var_count; i++) {
        if (strstr(variables[i].value, "* 2") != NULL) { // Example of
strength reduction
            printf("Reducing: %s = %s\n", variables[i].name,
variables[i].value);
```

```
            strcpy(variables[i].value, "shift_left"); // Replace
multiplication by 2 with left shift
            printf("Strength Reduced: %s = %s\n", variables[i].name,
variables[i].value);
        }
    }
}

int main() {
    char line[MAX_LENGTH];

    printf("Enter a simple C-like program (type 'END' to finish):\n");

    while (1) {
        fgets(line, sizeof(line), stdin);

        if (strcmp(line, "END\n") == 0) break;

        char var_name[20], op[3], value[20];

        // Example input: "a = b + c"
        if (sscanf(line, "%s %s %s", var_name, op, value) == 3 &&
strcmp(op, "=") == 0) {
            addOrUpdateVariable(var_name, value); // Add or update
variable with its value
        }
    }

    // Perform optimizations
    commonSubexpressionElimination();
    deadCodeElimination();
    strengthReduction();

    return 0;
}
```

Symboltable:-

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

```c
#include <ctype.h>

#define MAX_SYMBOLS 100
#define MAX_LINE_LENGTH 256

// Define the structure for a symbol table entry
typedef struct Symbol {
    char name[50];       // Name of the identifier
    int address;         // Memory address of the identifier
    char dataType[20];   // Data type of the identifier (e.g., int, float)
    int index;           // Index for array-like structures
    char value[50];      // Current value of the identifier
    char nextUse[50];    // Next use of the identifier
    char status[10];     // Status (alive or dead)
    int size;            // Size of the identifier
} Symbol;

// Declare the symbol table and a counter for the number of symbols
Symbol symbolTable[MAX_SYMBOLS];
int symbolCount = 0;

// Function to initialize the symbol table
void initializeSymbolTable() {
    symbolCount = 0;
}

// Function to insert a new symbol into the symbol table
int insertSymbol(char *name, char *dataType, int address, int index, char
*status, int size) {
    if (symbolCount >= MAX_SYMBOLS) {
        printf("Error: Symbol table is full.\n");
        return -1;
    }

    // Check for duplicate symbols
    for (int i = 0; i < symbolCount; i++) {
        if (strcmp(symbolTable[i].name, name) == 0) {
            return 0; // Do not insert duplicate
        }
    }
```

```
    // Add new symbol
    strcpy(symbolTable[symbolCount].name, name);
    symbolTable[symbolCount].address = address;
    strcpy(symbolTable[symbolCount].dataType, dataType);
    symbolTable[symbolCount].index = index;
    strcpy(symbolTable[symbolCount].value, "undefined");
    strcpy(symbolTable[symbolCount].nextUse, "none");
    strcpy(symbolTable[symbolCount].status, status);
    symbolTable[symbolCount].size = size;

    symbolCount++;
    return 0;
}

// Function to display the symbol table
void displaySymbolTable() {
    printf("\nSymbol Table:\n");

printf("Name\t\tAddress\tDataType\tIndex\tValue\tNextUse\tStatus\tSize\n")
;
    for (int i = 0; i < symbolCount; i++) {
        printf("%s\t%d\t%s\t%d\t%s\t%s\t%s\t%d\n",
               symbolTable[i].name,
               symbolTable[i].address,
               symbolTable[i].dataType,
               symbolTable[i].index,
               symbolTable[i].value,
               symbolTable[i].nextUse,
               symbolTable[i].status,
               symbolTable[i].size);
    }
}

// Function to parse the input C program and populate the symbol table
void parseCProgram(char *program) {
    char *line = strtok(program, "\n");
    int address = 0;

    while (line != NULL) {
```

```
        char dataType[20];
        char name[50];
        int index = -1;
        char status[10] = "alive";
        int size = 0;

        // Ignore preprocessor directives
        if (line[0] == '#') {
            line = strtok(NULL, "\n");
            continue;
        }

        // Check for variable declarations
        if (strstr(line, "int") != NULL || strstr(line, "float") != NULL
|| strstr(line, "char") != NULL) {
            // Extract the data type
            sscanf(line, "%s %s", dataType, name);

            // Check for array declaration
            if (strstr(name, "[") != NULL) {
                // Example: arr[3]
                char *arrayName = strtok(name, "[");
                char *sizeStr = strtok(NULL, "]");
                if (sizeStr != NULL) {
                    index = atoi(sizeStr); // Get the size of the array
                    insertSymbol(arrayName, dataType, address++, index,
"alive", index * sizeof(int)); // Assuming int size
                    line = strtok(NULL, "\n"); // Move to the next line
                    continue;
                }
            }
            // Insert the symbol into the table for non-array variables
            insertSymbol(name, dataType, address++, index, "alive",
sizeof(int)); // Assuming int size for simplicity
        }

        // Move to the next line
        line = strtok(NULL, "\n");
    }
}
```

```c
int main() {
    initializeSymbolTable();

    // Read C program input from the user
    char program[1024] = ""; // Initialize to empty string
    printf("Enter a C program (end with 'END'):\n");

    // Read lines until the user enters "END"
    char line[MAX_LINE_LENGTH];
    while (1) {
        fgets(line, sizeof(line), stdin);
        if (strcmp(line, "END\n") == 0) {
            break;
        }
        strcat(program, line);
    }

    // Parse the input C program and populate the symbol table
    parseCProgram(program);

    // Display the symbol table
    displaySymbolTable();

    return 0;
}
```

Sr parse:-

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

#define MAX 100

char stack[MAX];
char input[MAX];
int top = -1;
int inputIndex = 0;
```

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

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

void printStack() {
    printf("22bce0174\n");
    printf("$");
    for (int i = 0; i <= top; i++) {
        printf("%c", stack[i]);
    }
    printf("\t");
}

void shift() {
    printf("22bce0174\n");
    printf("SHIFT\t");
    push(input[inputIndex++]);
}

void reduce(char *rule) {
    printf("22bce0174\n");
    printf("REDUCE TO %s\t", rule);

    if (strcmp(rule, "E -> E + T") == 0) {
        pop(); // pop T
        pop(); // pop +
        pop(); // pop E
        push('E'); // push E
    } else if (strcmp(rule, "E -> T") == 0) {
        pop(); // pop T
        push('E'); // push E
```

```
    } else if (strcmp(rule, "T -> T * F") == 0) {
        pop(); // pop F
        pop(); // pop *
        pop(); // pop T
        push('T'); // push T
    } else if (strcmp(rule, "T -> F") == 0) {
        pop(); // pop F
        push('T'); // push T
    } else if (strcmp(rule, "F -> id") == 0) {
        pop(); // pop id
        push('F'); // push F
    }
}

int parse() {
    while (1) {
        printStack();

        if (inputIndex < strlen(input)) {
            shift();
        } else if (top >= 2 && stack[top] == 'T' && stack[top - 1] == '+'
&& stack[top - 2] == 'E') {
            reduce("E -> E + T");
        } else if (top >= 1 && stack[top] == 'T') {
            reduce("E -> T");
        } else if (top >= 2 && stack[top] == 'F' && stack[top - 1] == '*'
&& stack[top - 2] == 'T') {
            reduce("T -> T * F");
        } else if (top >= 0 && stack[top] == 'F') {
            reduce("T -> F");
        } else if (top >= 0 && stack[top] == 'a') { // Assuming 'id' is
represented as 'a'
            reduce("F -> id");
        } else if (top >= 0 && stack[top] == 'b') { // Assuming 'id' is
represented as 'b'
            reduce("F -> id");
        } else if (top == 0 && stack[top] == 'E' && inputIndex >=
strlen(input)) {
            printStack();
            printf("22bce0174\nACCEPT\n");
```

```
            return 1;
        } else {
            printStack();
            printf("22bce0174\nERROR\n");
            return 0;
        }
    }
}

int main() {
    printf("Enter the input string: ");
    fgets(input, sizeof(input), stdin);

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

    parse();

    return 0;
}
```

Left recursion:-

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

#define MAX_PRODUCTIONS 10
#define MAX_LENGTH 100

typedef struct {
    char non_terminal;
    char productions[MAX_PRODUCTIONS][MAX_LENGTH];
    int production_count;
} Grammar;

// Function to remove left recursion
void remove_left_recursion(Grammar *grammar) {
    printf("Removing left recursion for grammar:\n");
    char new_non_terminal = grammar->non_terminal + '\'';
    int found_recursion = 0;
```

```
    for (int i = 0; i < grammar->production_count; i++) {
        if (grammar->productions[i][0] == grammar->non_terminal) {
            found_recursion = 1;
            break;
        }
    }

    if (found_recursion) {
        printf("Productions for %c:\n", grammar->non_terminal);
        for (int i = 0; i < grammar->production_count; i++) {
            if (grammar->productions[i][0] != grammar->non_terminal) {
                printf("%s%c\n", grammar->productions[i],
new_non_terminal);
            }
        }
        printf("%c -> ", new_non_terminal);
        for (int i = 0; i < grammar->production_count; i++) {
            if (grammar->productions[i][0] == grammar->non_terminal) {
                printf("%s ", grammar->productions[i] + 1); // Skip the
left non-terminal
            }
        }
        printf("| ε\n");
    } else {
        printf("No left recursion found for %c.\n", grammar-
>non_terminal);
    }
}

int main() {
    Grammar grammar;

    // User input for non-terminal
    printf("Enter the non-terminal symbol: ");
    scanf(" %c", &grammar.non_terminal);

    // User input for productions
    printf("Enter the number of productions for %c: ",
grammar.non_terminal);
    scanf("%d", &grammar.production_count);
```

```
    // Input each production
    printf("Enter the productions (one per line):\n");
    for (int i = 0; i < grammar.production_count; i++) {
        scanf("%s", grammar.productions[i]);
    }

    // Remove left recursion
    remove_left_recursion(&grammar);

    return 0;
}
```

Left factoring:-

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

#define MAX_PRODUCTIONS 10
#define MAX_LENGTH 100

typedef struct {
    char non_terminal;
    char productions[MAX_PRODUCTIONS][MAX_LENGTH];
    int production_count;
} Grammar;

// Function to perform left factoring
void left_factoring(Grammar *grammar) {
    printf("Performing left factoring for grammar:\n");
    char new_non_terminal = grammar->non_terminal + '\'';

    int found_factoring = 0;

    for (int i = 0; i < grammar->production_count; i++) {
        for (int j = i + 1; j < grammar->production_count; j++) {
            if (grammar->productions[i][0] == grammar->productions[j][0])
{
                found_factoring = 1;
```

```
                printf("%c -> %c%c\n", grammar->non_terminal, grammar-
>productions[i][0], new_non_terminal);
                printf("%c -> ", new_non_terminal);
                printf("%s | %s\n", grammar->productions[i] + 1, grammar-
>productions[j] + 1);
                return; // Exit after factoring the first pair found
            }
        }
    }

    if (!found_factoring) {
        printf("No left factoring needed for %c.\n", grammar-
>non_terminal);
    }
}

int main() {
    Grammar grammar;

    // User input for non-terminal
    printf("Enter the non-terminal symbol: ");
    scanf(" %c", &grammar.non_terminal);

    // User input for productions
    printf("Enter the number of productions for %c: ",
grammar.non_terminal);
    scanf("%d", &grammar.production_count);

    // Input each production
    printf("Enter the productions (one per line):\n");
    for (int i = 0; i < grammar.production_count; i++) {
        scanf("%s", grammar.productions[i]);
    }

    // Perform left factoring
    left_factoring(&grammar);

    return 0;
}
```

Parse tree:-

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

#define MAX 100

// Structure for a tree node
typedef struct Node {
    char data;
    struct Node* left;
    struct Node* right;
} Node;

// Function to create a new tree node
Node* createNode(char data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to print the parse tree
void printTree(Node* root, int space) {
    if (root == NULL)
        return;

    space += 10; // Increase distance between levels

    printTree(root->right, space); // Process right child first

    printf("\n");
    for (int i = 10; i < space; i++)
        printf(" "); // Print spaces for formatting
    printf("%c\n", root->data); // Print current node's data

    printTree(root->left, space); // Process left child
}
```

```
// Function to check if the character is an operator
int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

// Function to construct a parse tree from infix expression
Node* constructInfixTree(char* expr) {
    // Stack for operators
    char opStack[MAX];
    int opTop = -1;

    // Stack for nodes
    Node* nodeStack[MAX];
    int nodeTop = -1;

    for (int i = 0; expr[i]; i++) {
        if (isspace(expr[i])) continue; // Ignore spaces

        if (isalnum(expr[i])) { // If operand, create a node and push to
node stack
            Node* newNode = createNode(expr[i]);
            nodeStack[++nodeTop] = newNode;
        } else if (expr[i] == '(') { // Push '(' to operator stack
            opStack[++opTop] = expr[i];
        } else if (expr[i] == ')') { // Pop until '(' and create nodes
            while (opTop != -1 && opStack[opTop] != '(') {
                char op = opStack[opTop--];
                Node* right = nodeStack[nodeTop--];
                Node* left = nodeStack[nodeTop--];
                Node* newNode = createNode(op);
                newNode->left = left;
                newNode->right = right;
                nodeStack[++nodeTop] = newNode;
            }
            opTop--; // Pop '('
        } else if (isOperator(expr[i])) { // Operator encountered
            while (opTop != -1 && isOperator(opStack[opTop])) {
                char op = opStack[opTop--];
                Node* right = nodeStack[nodeTop--];
                Node* left = nodeStack[nodeTop--];
```

```
                Node* newNode = createNode(op);
                newNode->left = left;
                newNode->right = right;
                nodeStack[++nodeTop] = newNode;
            }
            opStack[++opTop] = expr[i]; // Push current operator
        }
    }

    while (opTop != -1) { // Pop remaining operators
        char op = opStack[opTop--];
        Node* right = nodeStack[nodeTop--];
        Node* left = nodeStack[nodeTop--];
        Node* newNode = createNode(op);
        newNode->left = left;
        newNode->right = right;
        nodeStack[++nodeTop] = newNode;
    }

    return nodeStack[nodeTop]; // Return root of the parse tree
}

// Function to construct a parse tree from prefix expression
Node* constructPrefixTree(char* expr) {
    Node* stack[MAX];
    int top = -1;

    for (int i = strlen(expr) - 1; i >= 0; i--) {
        if (isspace(expr[i])) continue; // Ignore spaces

        if (isalnum(expr[i])) { // If operand, create a node and push to
stack
            stack[++top] = createNode(expr[i]);
        } else if (isOperator(expr[i])) { // Operator encountered
            Node* newNode = createNode(expr[i]);
            newNode->left = stack[top--]; // Pop two operands for the
operator
            newNode->right = stack[top--];
            stack[++top] = newNode; // Push back the subtree
        }
```

```
    }

    return stack[top]; // Return root of the parse tree
}

// Function to construct a parse tree from postfix expression
Node* constructPostfixTree(char* expr) {
    Node* stack[MAX];
    int top = -1;

    for (int i = 0; expr[i]; i++) {
        if (isspace(expr[i])) continue; // Ignore spaces

        if (isalnum(expr[i])) { // If operand, create a node and push to
stack
            stack[++top] = createNode(expr[i]);
        } else if (isOperator(expr[i])) { // Operator encountered
            Node* newNode = createNode(expr[i]);
            newNode->right = stack[top--]; // Pop two operands for the
operator
            newNode->left = stack[top--];
            stack[++top] = newNode; // Push back the subtree
        }
    }

    return stack[top]; // Return root of the parse tree
}

int main() {
    char infixExpr[MAX], prefixExpr[MAX], postfixExpr[MAX];

    printf("Enter infix expression: ");
    fgets(infixExpr, sizeof(infixExpr), stdin);

    printf("Enter prefix expression: ");
    fgets(prefixExpr, sizeof(prefixExpr), stdin);

    printf("Enter postfix expression: ");
    fgets(postfixExpr, sizeof(postfixExpr), stdin);
```

```
    printf("\nConstructing Parse Tree from Infix Expression:\n");
    Node* infixTreeRoot = constructInfixTree(infixExpr);
    printTree(infixTreeRoot, 0);

    printf("\nConstructing Parse Tree from Prefix Expression:\n");
    Node* prefixTreeRoot = constructPrefixTree(prefixExpr);
    printTree(prefixTreeRoot, 0);

    printf("\nConstructing Parse Tree from Postfix Expression:\n");
    Node* postfixTreeRoot = constructPostfixTree(postfixExpr);
    printTree(postfixTreeRoot, 0);

    return 0;
}
```

Recursive decent:-

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

#define SUCCESS 1
#define FAILED 0

int E(), E_prime(), T(), T_prime(), F();

const char *cursor;
char string[64];

int main()
{
    puts("Enter the string: ");
    scanf("%s", string);
    cursor = string;
    puts("");
    puts("Input         Action");
    puts("--------------------------------");

    if (E() && *cursor == '\0') {
        puts("--------------------------------");
        puts("String is successfully parsed");
        return 0;
    } else {
```

```
        puts("--------------------------------");
        puts("Error in parsing String");
        return 1;
    }
}

// E -> T E'
int E()
{
    printf("%-16s E -> T E'\n", cursor);
    if (T()) {
        return E_prime();  // Explicitly call E' after parsing T
    } else
        return FAILED;
}

// E' -> + T E' | ε
int E_prime()
{
    if (*cursor == '+') {
        printf("%-16s E' -> + T E'\n", cursor);
        cursor++;  // Move past '+'
        if (T()) {
            return E_prime();  // Recursively handle additional '+'
operators
        } else
            return FAILED;
    }
    // E' -> ε (do nothing and return SUCCESS)
    return SUCCESS;
}

// T -> F T'
int T()
{
    printf("%-16s T -> F T'\n", cursor);
    if (F()) {
        return T_prime();  // Explicitly call T' after parsing F
    } else
        return FAILED;
```

```
}

// T' -> * F T' | ε
int T_prime()
{
    if (*cursor == '*') {  // Corrected condition to check for '*'
        printf("%-16s T' -> * F T'\n", cursor);
        cursor++;  // Move past '*'
        if (F()) {
            return T_prime();  // Recursively handle additional '*'
operators
        } else
            return FAILED;
    }
    // T' -> ε (do nothing and return SUCCESS)
    return SUCCESS;
}

// F -> ( E ) | i
int F()
{
    if (*cursor == '(') {
        printf("%-16s F -> ( E )\n", cursor);
        cursor++;  // Move past '('
        if (E()) {
            if (*cursor == ')') {
                cursor++;  // Move past ')'
                return SUCCESS;
            } else
                return FAILED;
        } else
            return FAILED;
    } else if (*cursor == 'i') {  // Identifier
        printf("%-16s F -> i\n", cursor);
        cursor++;  // Move past 'i'
        return SUCCESS;
    } else
        return FAILED;
}
```

ddfaone:-

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

#define MAX_SYMBOLS 10
#define MAX_STRING_LENGTH 100

// Define the grammar
char *productions[] = {
    "S -> A B",
    "A -> a",
    "B -> b"
};

char first[MAX_SYMBOLS][MAX_STRING_LENGTH];
char last[MAX_SYMBOLS][MAX_STRING_LENGTH];
char follow[MAX_SYMBOLS][MAX_STRING_LENGTH];

int numSymbols = 3; // Number of symbols

// Function to convert character to array index
int charToIndex(char symbol) {
    switch (symbol) {
        case 'S': return 0;
        case 'A': return 1;
        case 'B': return 2;
        default: return -1; // Invalid symbol
    }
}

// Function to add a symbol to a set if it is not already present
void addToSet(char *set, char symbol) {
    if (strchr(set, symbol) == NULL) {
        size_t len = strlen(set);
        if (len + 1 < MAX_STRING_LENGTH) {
            set[len] = symbol;
            set[len + 1] = '\0';
        }
    }
}
```

```
// Function to find the First set for a given symbol
void findFirst(char symbol) {
    int index = charToIndex(symbol);
    if (index == -1) return;

    // Clear previous contents
    first[index][0] = '\0';

    for (int i = 0; i < sizeof(productions) / sizeof(productions[0]); i++)
{
        if (productions[i][0] == symbol) {
            char *prod = strchr(productions[i], '>') + 2; // Get the
production part
            if (prod[0] != '\0') {
                // Add the first symbol of the production to the First set
                addToSet(first[index], prod[0]);
            }
        }
    }
}

// Function to find the Last set for a given symbol
void findLast(char symbol) {
    int index = charToIndex(symbol);
    if (index == -1) return;

    // Clear previous contents
    last[index][0] = '\0';

    for (int i = 0; i < sizeof(productions) / sizeof(productions[0]); i++)
{
        if (productions[i][0] == symbol) {
            char *prod = strchr(productions[i], '>') + 2; // Get the
production part
            int len = strlen(prod);
            if (len > 0) {
                // Add the last symbol of the production to the Last set
                addToSet(last[index], prod[len-1]);
            }
```

```
        }
    }
}

// Function to find the Follow set for a given symbol
void findFollow(char symbol) {
    int index = charToIndex(symbol);
    if (index == -1) return;

    // Clear previous contents
    follow[index][0] = '\0';

    if (symbol == 'S') {
        addToSet(follow[index], '$');
    }

    for (int i = 0; i < sizeof(productions) / sizeof(productions[0]); i++)
{
        char *prod = strchr(productions[i], '>') + 2; // Get the
production part
        char *pos = strchr(prod, symbol);

        while (pos != NULL) {
            // Check if symbol is not the last symbol in production
            if (*(pos + 1) != '\0') {
                addToSet(follow[index], *(pos + 1));
            } else {
                // Add Follow set of the non-terminal
                if (productions[i][0] != symbol) {
                    int nonTerminalIndex = charToIndex(productions[i][0]);
                    if (nonTerminalIndex != -1) {
                        strcat(follow[index], follow[nonTerminalIndex]);
                    }
                }
            }
            // Continue searching for more occurrences
            pos = strchr(pos + 1, symbol);
        }
    }
}
```

```c
int main() {
    // Initialize First, Last, and Follow sets
    for (int i = 0; i < MAX_SYMBOLS; i++) {
        first[i][0] = '\0';
        last[i][0] = '\0';
        follow[i][0] = '\0';
    }

    // Calculate First sets
    findFirst('S');
    findFirst('A');
    findFirst('B');

    // Calculate Last sets
    findLast('S');
    findLast('A');
    findLast('B');

    // Calculate Follow sets
    findFollow('S');
    findFollow('A');
    findFollow('B');

    // Display the results
    printf("First Sets:\n");
    printf("First(S) = { %s }\n", first[charToIndex('S')]);
    printf("First(A) = { %s }\n", first[charToIndex('A')]);
    printf("First(B) = { %s }\n", first[charToIndex('B')]);

    printf("\nLast Sets:\n");
    printf("Last(S) = { %s }\n", last[charToIndex('S')]);
    printf("Last(A) = { %s }\n", last[charToIndex('A')]);
    printf("Last(B) = { %s }\n", last[charToIndex('B')]);

    printf("\nFollow Sets:\n");
    printf("Follow(S) = { %s }\n", follow[charToIndex('S')]);
    printf("Follow(A) = { %s }\n", follow[charToIndex('A')]);
    printf("Follow(B) = { %s }\n", follow[charToIndex('B')]);
```

```
    return 0;
}
```

Ddfatwo:-

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

#define MAX_STATES 10
#define MAX_SYMBOLS 2
#define MAX_PRODUCTIONS 10

// Production rules
char productions[MAX_PRODUCTIONS][10] = {
    "S->aA",
    "A->aA",
    "A->b",
    "B->b"
};

// Function to find the next state
int getNextState(char currentState, char inputSymbol) {
    for (int i = 0; i < MAX_PRODUCTIONS; i++) {
        char leftSide[10];
        char rightSide[10];
        sscanf(productions[i], "%[^->]->%s", leftSide, rightSide);
        if (leftSide[0] == currentState) {
            if (rightSide[0] == inputSymbol) {
                if (strlen(rightSide) > 1) {
                    return rightSide[1];
                } else {
                    return rightSide[0];
                }
            }
        }
    }
    return -1; // Error: no transition found
}

// Function to construct the DFA transition table
```

```
void constructTransitionTable(int transitionTable[][MAX_SYMBOLS], char
states[]) {
    int numStates = strlen(states);
    for (int i = 0; i < numStates; i++) {
        for (int j = 0; j < MAX_SYMBOLS; j++) {
            char inputSymbol = (j == 0) ? 'a' : 'b';
            int nextState = getNextState(states[i], inputSymbol);
            if (nextState != -1) {
                for (int k = 0; k < numStates; k++) {
                    if (states[k] == nextState) {
                        transitionTable[i][j] = k;
                        break;
                    }
                }
            } else {
                transitionTable[i][j] = -1; // Error: no transition found
            }
        }
    }
}

// Function to print the DFA transition table
void printTransitionTable(int transitionTable[][MAX_SYMBOLS], char
states[]) {
    int numStates = strlen(states);
    printf("DFA Transition Table:\n");
    printf("   | a | b\n");
    for (int i = 0; i < numStates; i++) {
        printf("q%d |", i);
        for (int j = 0; j < MAX_SYMBOLS; j++) {
            if (transitionTable[i][j] != -1) {
                printf(" q%d |", transitionTable[i][j]);
            } else {
                printf(" - |");
            }
        }
        printf("\n");
    }
}
```

```
int main() {
    char states[] = "SAB"; // Define the states
    int transitionTable[MAX_STATES][MAX_SYMBOLS];
    constructTransitionTable(transitionTable, states);
    printTransitionTable(transitionTable, states);
    return 0;
}
```

Operator precedence:-

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

#define MAX 100

char stack[MAX];
char input[MAX];
int top = -1;
int i = 0; // Input index

// Precedence table
// Format: precedence[stack_symbol][input_symbol]
char precedence[8][8] = {
    //    +   -   *   /   ^   i   (   )
    /* + */ {'>', '>', '<', '<', '<', '>', '<', '>'},
    /* - */ {'>', '>', '<', '<', '<', '>', '<', '>'},
    /* * */ {'>', '>', '>', '>', '<', '>', '<', '>'},
    /* / */ {'>', '>', '>', '>', '<', '>', '<', '>'},
    /* ^ */ {'>', '>', '>', '>', '>', '>', '<', '>'},
    /* i */ {'<', '<', '<', '<', '<', '=', '<', '>'},
    /* ( */ {'<', '<', '<', '<', '<', '<', '=', 'e'},
    /* ) */ {'>', '>', '>', '>', '>', 'e', 'e', '>'}
};

// Function to get index of terminal symbols
int getIndex(char c) {
    switch (c) {
        case '+': return 0;
        case '-': return 1;
        case '*': return 2;
        case '/': return 3;
```

```
        case '^': return 4;
        case 'i': return 5; // Identifier
        case '(' : return 6;
        case ')' : return 7;
        default: return -1; // Invalid character
    }
}

// Function to shift operation
void shift() {
    stack[++top] = input[i++];
}

// Function to reduce operation
int reduce() {
    // Simple reduction logic for demonstration purposes
    if (top >= 2 && stack[top] == ')' && stack[top - 1] == '(') {
        top -= 2; // Pop ( and )
        stack[++top] = 'E'; // Replace with non-terminal E
        return 1; // Successful reduction
    }

    // Add more reduction rules as needed based on grammar
    return 0; // No reduction possible
}

// Function to display current state of stack and input
void displayState() {
    printf("Stack: ");
    for (int j = 0; j <= top; j++) {
        printf("%c ", stack[j]);
    }

    printf("\tInput: ");
    for (int j = i; j < strlen(input); j++) {
        printf("%c ", input[j]);
    }

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

```
int main() {
    printf("Enter an expression (use identifiers as 'i' and operators +, -
, *, /, ^): ");
    scanf("%s", input);

    strcat(input, "$"); // Append end marker

    stack[top++] = '$'; // Initialize stack with end marker

    printf("\nSTACK\t\tINPUT\t\tACTION\n");

    while (i <= strlen(input)) {
        displayState(); // Display current state

        // Shift operation
        shift();
        printf("Shift\n");

        while (1) { // Check for reductions
            int indexStack = getIndex(stack[top - 1]);
            int indexInput = getIndex(input[i]);

            if (indexStack == -1 || indexInput == -1) {
                break; // Invalid character, exit loop
            }

            if (precedence[indexStack][indexInput] == '>') {
                while (reduce()) { // Perform reduction if possible
                    printf("Reduce\n");
                    displayState(); // Display current state after
reduction
                }
            } else if (precedence[indexStack][indexInput] == '=') {
                break; // Acceptable state, exit loop
            } else if (precedence[indexStack][indexInput] == '<') {
                break; // Shift or wait for further action
            } else {
                printf("Error: Unexpected symbol.\n");
                return 1; // Exit with error
```

```
            }
        }

        if (strcmp(stack, "$E$") == 0) { // Check for acceptance condition
            printf("Accepted\n");
            break;
        }

        if (i > strlen(input)) {
            printf("Not Accepted\n");
            break;
        }
    }

    return 0;
}
```

Treeadress:-

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

#define MAX 100

// Structure to hold a three-address code entry
typedef struct {
    char op[10];      // Operator
    char arg1[10];    // First argument
    char arg2[10];    // Second argument
    char result[10];  // Result variable
} TAC;

// Global variables
TAC tac[MAX];
int tac_count = 0;

// Function to generate three-address code
void generateTAC(char *op, char *arg1, char *arg2, char *result) {
    strcpy(tac[tac_count].op, op);
    strcpy(tac[tac_count].arg1, arg1);
```

```
    strcpy(tac[tac_count].arg2, arg2);
    strcpy(tac[tac_count].result, result);
    tac_count++;
}

// Function to print all three-address codes
void printTAC() {
    printf("\nThree Address Code (TAC):\n");
    for (int i = 0; i < tac_count; i++) {
        printf("%s %s %s -> %s\n", tac[i].op, tac[i].arg1, tac[i].arg2,
tac[i].result);
    }
}

// Function to generate quadruples
void printQuadruples() {
    printf("\nQuadruples:\n");
    for (int i = 0; i < tac_count; i++) {
        printf("(%s, %s, %s, %s)\n", tac[i].op, tac[i].arg1, tac[i].arg2,
tac[i].result);
    }
}

// Function to generate triples
void printTriples() {
    printf("\nTriples:\n");
    for (int i = 0; i < tac_count; i++) {
        printf("(%d, %s, %s)\n", i + 1, tac[i].op, tac[i].arg1);
        if (strlen(tac[i].arg2) > 0) {
            printf("(%d, %s)\n", i + 1, tac[i].arg2);
        }
        printf("-> %s\n", tac[i].result);
    }
}

// Function to parse an expression and generate TAC
void parseExpression(char *expression) {
    char stack[MAX][10]; // Stack for operators and operands
    int top = -1;
```

```
    char temp_var[10];
    int temp_count = 1;

    for (int i = 0; expression[i] != '\0'; i++) {
        if (isspace(expression[i])) continue;

        if (isalnum(expression[i])) { // If operand (variable or number)
            stack[++top][0] = expression[i];
            stack[top][1] = '\0';
        } else { // Operator encountered
            char arg2[10], arg1[10];
            strcpy(arg2, stack[top--]); // Get second operand
            strcpy(arg1, stack[top--]); // Get first operand

            sprintf(temp_var, "t%d", temp_count++); // Create a new
temporary variable

            generateTAC(&expression[i], arg1, arg2, temp_var); // Generate
TAC

            // Push the result back onto the stack
            strcpy(stack[++top], temp_var);
        }
    }
}

int main() {
    char expression[MAX];

    printf("Enter an arithmetic expression: ");
    fgets(expression, sizeof(expression), stdin);

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

    parseExpression(expression); // Parse the input expression
```

```
    printTAC();      // Print Three Address Code
    printQuadruples(); // Print Quadruples
    printTriples();   // Print Triples

    return 0;
}
```

Common:-

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

#define MAX_LINES 100
#define MAX_LENGTH 256

// Structure to hold variable information
typedef struct {
    char name[20];
    char value[20];
    int used; // Flag to indicate if the variable is used
} Variable;

// Global variables
Variable variables[MAX_LINES];
int var_count = 0;

// Function to find a variable in the list
int findVariable(char *name) {
    for (int i = 0; i < var_count; i++) {
        if (strcmp(variables[i].name, name) == 0) {
            return i;
        }
    }
    return -1;
}

// Function to add or update a variable
void addOrUpdateVariable(char *name, char *value) {
    int index = findVariable(name);
    if (index != -1) {
        strcpy(variables[index].value, value);
```

```
        variables[index].used = 1; // Mark as used
    } else {
        strcpy(variables[var_count].name, name);
        strcpy(variables[var_count].value, value);
        variables[var_count].used = 1; // Mark as used
        var_count++;
    }
}

// Function to eliminate dead code
void deadCodeElimination() {
    printf("\nDead Code Elimination:\n");
    for (int i = 0; i < var_count; i++) {
        if (!variables[i].used) {
            printf("Removing unused variable: %s\n", variables[i].name);
        } else {
            printf("%s = %s\n", variables[i].name, variables[i].value);
        }
    }
}

// Function to perform common subexpression elimination
void commonSubexpressionElimination() {
    printf("\nCommon Subexpression Elimination:\n");
    for (int i = 0; i < var_count; i++) {
        for (int j = i + 1; j < var_count; j++) {
            if (strcmp(variables[i].value, variables[j].value) == 0) {
                printf("Replacing %s with %s\n", variables[j].name,
variables[i].name);
                strcpy(variables[j].value, variables[i].name); // Replace
with common expression
            }
        }
    }
}

// Function to perform strength reduction
void strengthReduction() {
    printf("\nStrength Reduction:\n");
    for (int i = 0; i < var_count; i++) {
```

```
        if (strstr(variables[i].value, "* 2") != NULL) { // Example of
strength reduction
            printf("Reducing: %s = %s\n", variables[i].name,
variables[i].value);
            strcpy(variables[i].value, "shift_left"); // Replace
multiplication by 2 with left shift
            printf("Strength Reduced: %s = %s\n", variables[i].name,
variables[i].value);
        }
    }
}

int main() {
    char line[MAX_LENGTH];

    printf("Enter a simple C-like program (type 'END' to finish):\n");

    while (1) {
        fgets(line, sizeof(line), stdin);

        if (strcmp(line, "END\n") == 0) break;

        char var_name[20], op[3], value[20];

        // Example input: "a = b + c"
        if (sscanf(line, "%s %s %s", var_name, op, value) == 3 &&
strcmp(op, "=") == 0) {
            addOrUpdateVariable(var_name, value); // Add or update
variable with its value
        }
    }

    // Perform optimizations
    commonSubexpressionElimination();
    deadCodeElimination();
    strengthReduction();

    return 0;
}
```

Predictive:-

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

#define MAX_PRODUCTIONS 10
#define MAX_LENGTH 100
#define MAX_TERMINALS 10

// Structure to hold grammar information
typedef struct {
    char non_terminal;
    char productions[MAX_PRODUCTIONS][MAX_LENGTH];
    int production_count;
} Grammar;

// Global variables for parsing table
char parsingTable[MAX_PRODUCTIONS][MAX_TERMINALS][MAX_LENGTH];
char terminals[MAX_TERMINALS];
int terminal_count = 0;

// Function to initialize the parsing table
void initializeParsingTable() {
    for (int i = 0; i < MAX_PRODUCTIONS; i++) {
        for (int j = 0; j < MAX_TERMINALS; j++) {
            strcpy(parsingTable[i][j], ""); // Initialize with empty
strings
        }
    }
}

// Function to find the index of a terminal in the terminals array
int findTerminalIndex(char terminal) {
    for (int i = 0; i < terminal_count; i++) {
        if (terminals[i] == terminal) {
            return i;
        }
    }
    return -1;
}
```

```c
// Function to compute FIRST set for a given production
void computeFirst(Grammar *grammar) {
    // Simple implementation of FIRST set computation
    for (int i = 0; i < grammar->production_count; i++) {
        char *production = grammar->productions[i];
        if (isalpha(production[0])) { // If it starts with a terminal
            strncat(first[i], &production[0], 1);
        } else if (isupper(production[0])) { // If it starts with a non-
terminal
            // For simplicity, let's assume it directly maps to the first
character.
            strncat(first[i], &production[0], 1);
        }
    }
}

// Function to construct the parsing table using FIRST sets
void constructParsingTable(Grammar *grammar) {
    for (int i = 0; i < grammar->production_count; i++) {
        char *production = grammar->productions[i];
        int firstIndex = findTerminalIndex(production[0]); // Assume first
character is terminal

        if (firstIndex != -1) {
            strcpy(parsingTable[i][firstIndex], production);
        }

        // Handle epsilon productions and follow sets here...
    }
}

// Function to print the parsing table
void printParsingTable() {
    printf("Parsing Table:\n");
    printf("Non-Terminal | ");
    for (int j = 0; j < terminal_count; j++) {
        printf("%-10c ", terminals[j]);
    }
    printf("\n");
```

```
    for (int i = 0; i < MAX_PRODUCTIONS; i++) {
        printf("%c            | ", 'A' + i); // Assuming non-terminals are
A, B, C...
        for (int j = 0; j < terminal_count; j++) {
            printf("%-10s ", parsingTable[i][j]);
        }
        printf("\n");
    }
}

// Function to parse input string using the constructed parsing table
void parseInputString(char *input, Grammar *grammar) {
    char stack[MAX_LENGTH];
    int top = -1;

    stack[++top] = '$'; // End marker
    stack[++top] = grammar->non_terminal; // Start symbol

    int index = 0;

    while (top != -1) {
        char topSymbol = stack[top--];

        if (topSymbol == input[index]) { // Match terminal
            index++;
            continue;
        } else if (isupper(topSymbol)) { // Non-terminal
            int ruleIndex = topSymbol - 'A'; // Assuming A=0, B=1,...
            int terminalIndex = findTerminalIndex(input[index]);

            if (terminalIndex != -1 &&
strcmp(parsingTable[ruleIndex][terminalIndex], "") != 0) {
                // Push production onto stack in reverse order
                char *production = parsingTable[ruleIndex][terminalIndex];
                for (int j = strlen(production) - 1; j >= 0; j--) {
                    stack[++top] = production[j]; // Push each symbol onto
stack
                }
            } else {
```

```
                printf("Error: No matching production for %c\n",
topSymbol);
                return;
            }
        } else { // Error case: unexpected symbol
            printf("Error: Unexpected symbol %c\n", topSymbol);
            return;
        }

        // Print current state of the stack and input index
        printf("Current Stack: ");
        for (int k = top; k >= 0; k--) {
            printf("%c ", stack[k]);
        }
        printf(" | Input Index: %d\n", index);
    }

    if (input[index] == '$') { // Successfully parsed input string
        printf("Input string successfully parsed.\n");
    } else {
        printf("Error: Input string not fully consumed.\n");
    }
}

int main() {
    Grammar grammar;

    // Define a sample grammar directly in code.
    grammar.non_terminal = 'E'; // Starting non-terminal

    // Sample productions for E -> T E' | ε, E' -> + T E' | ε, T -> id | (
E )

    grammar.production_count = 5;

    strcpy(grammar.productions[0], "E->T E'");
    strcpy(grammar.productions[1], "E'->+ T E'");
    strcpy(grammar.productions[2], "E'->ε");
    strcpy(grammar.productions[3], "T->id");
    strcpy(grammar.productions[4], "T->( E )");
```

```
    // Extract terminals from productions and fill terminals array
    terminals[terminal_count++] = 'id';   // Assuming 'id' is treated as a
single terminal.
    terminals[terminal_count++] = '+' ;
    terminals[terminal_count++] = '(' ;
    terminals[terminal_count++] = ')' ;

   initializeParsingTable(); // Initialize parsing table

   computeFirst(&grammar);   // Compute FIRST sets.

   constructParsingTable(&grammar); // Construct the parsing table.

   printParsingTable();       // Print the constructed parsing table.

   char input[MAX_LENGTH];

   printf("Enter input string to parse: ");
   scanf("%s", input);

   parseInputString(input, &grammar); // Parse the input string.

   return 0;
}
```