Experiment No :2

```
#include <stdio.h>
#include <string.h>
#define MAX 100
typedef struct {
        char label[20];
        int addr;
} Symbol;
Symbol symtab[MAX];
int count = 0;
void add_symbol(const char *label, int addr)
{
        for (int i = 0; i < count; i++) {
        if (strcmp(symtab[i].label, label) ==
0) return;
        }
        strcpy(symtab[count].label, label);
        symtab[count++].addr = addr;
}
void show_table() {
        printf("\nSymbol Table:\n%-10s |
%s\n", "Symbol", "Address");
        for (int i = 0; i < count; i++)
        printf("%-10s | %d\n",
symtab[i].label, symtab[i].addr);
}
int main() {
        const char *code[7][3] = {
        {"", "START", "200"},
        {"LOOP", "LDA", "A"},
        {"", "ADD", "ONE"},
        {"", "STA", "A"},
        {"A", "DB", "5"},
        {"ONE", "DB", "1"},
        {"", "END", ""}
        };

        for (int loc = 199, i = 0; i < 7; i++,
loc++) {
        if (strlen(code[i][0]) > 0)
        add_symbol(code[i][0], loc);
        }

        show_table();
        return 0;
}
```

Output:

Symbol Table:

Symbol	Address
LOOP	200
A	203
ONE	204

Experiment No:03

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

#define MAX 100

typedef struct { char symbol[20]; int address; } Symbol;
typedef struct { char literal[20]; int address; } Literal;
typedef struct { char mnemonic[20]; int opcode; } Mnemonic;

char* code[] = {
        "START 100", "X LDA 200", "STA Z",
        "ADD -20", "MUL -30", "END"
};

Mnemonic mTab[] = {
        {"START", 0}, {"LDA", 1}, {"STA", 2}, {"ADD", 3},
        {"SUB", 4}, {"MUL", 5}, {"DIV", 6}, {"END", 7}
};

Symbol sym[MAX]; Literal lit[MAX];
int symCount = 0, litCount = 0, locTable[MAX], locIndex = 0;

int isMnemonic(char* token) {
        for (int i = 0; i < 8; i++)
        if (!strcmp(mTab[i].mnemonic, token)) return 1;
        return 0;
}

void firstPass() {
        int loc = 0;
        for (int i = 0; i < 6; i++) {
        char w1[20] = "", w2[20] = "", w3[20] = "";
        int n = sscanf(code[i], "%s %s %s", w1, w2, w3);

        if (!strcmp(w1, "START") || !strcmp(w2, "START")) {
        sscanf(n == 2 ? w2 : w3, "%d", &loc);
        continue;
        }

        if (!isMnemonic(w1)) {
        strcpy(sym[symCount].symbol, w1);
        sym[symCount++].address = loc;
```

```
        }

        char* operand = (n == 2) ? w2 : w3;
        if (operand[0] == '-') {
        int exists = 0;
        for (int j = 0; j < litCount; j++)
                if (!strcmp(lit[j].literal, operand)) exists = 1;
        if (!exists) {
                strcpy(lit[litCount].literal, operand);
                lit[litCount++].address = -1;
        }
        }

        locTable[locIndex++] = loc++;
        }
}

void secondPass() {
        for (int i = 0, litAddr = locTable[locIndex - 1] + 1; i < litCount; i++)
        if (lit[i].address == -1) lit[i].address = litAddr++;
}

void printTables() {
        printf("\nSymbol Table:\n");
        for (int i = 0; i < symCount; i++)
        printf("%s : %d\n", sym[i].symbol, sym[i].address);

        printf("\nLiteral Table:\n");
        for (int i = 0; i < litCount; i++)
        printf("%s : %d\n", lit[i].literal, lit[i].address);

        printf("\nLocation Table:\n");
        for (int i = 0; i < locIndex; i++)
        printf("Line %d : %d\n", i + 1, locTable[i]);

        printf("\nMnemonic Table:\n");
        for (int i = 0; i < 8; i++)
        printf("%s : %d\n", mTab[i].mnemonic, mTab[i].opcode);
}

int main() {
        firstPass();
        secondPass();
        printTables();
```

```
        return 0;
}
```

Output :

Symbol Table:
X : 100

Literal Table:

Location Counter Table:
Line 1 : 100
Line 2 : 101
Line 3 : 102
Line 4 : 103
Line 5 : 104

Mnemonic Table:
START : 0
LDA : 1
STA : 2
ADD : 3
SUB : 4
MUL : 5
DIV : 6
END : 7

Experiment No : 04

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

char macroBody[10][100];
int macroLines = 0;

int main() {
        char input[][100] = {
        "MACRO", "INCR &ARG1", "ADD &ARG1", "MEND",
        "START 100", "INCR A", "END"
        };
        int inMacro = 0, lines = sizeof(input) / sizeof(input[0]);

        puts("Expanded Output:");
        for (int i = 0; i < lines; i++) {
        if (!strcmp(input[i], "MACRO")) { inMacro = 1; continue; }
        if (!strcmp(input[i], "MEND"))  { inMacro = 0; continue; }

        if (inMacro) { strcpy(macroBody[macroLines++], input[i]); continue; }

        char cmd[20], arg[20];
        if (sscanf(input[i], "%s %s", cmd, arg) == 2 && !strcmp(cmd, "INCR")) {
        for (int j = 0; j < macroLines; j++) {
                char *p = strstr(macroBody[j], "&ARG1");
                if (p) {
                *p = '\0';
                printf("%s%s%s\n", macroBody[j], arg, p + 6);
                } else puts(macroBody[j]);
        }
        } else puts(input[i]);
        }
        return 0;
}
```

Output :

```
Expanded Output:
START 100
INCR A
ADD A
END
```

Experiment No : 06

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

int isValid(char expr[]) {
	int n = strlen(expr);

	if (n % 2 == 0) return 0; // Length must be odd

	for (int i = 0; i < n; i++) {
	// Check for operands (a, b, c...)
	if (i % 2 == 0) {
	if (!isalpha(expr[i])) return 0; // If not a letter, invalid
	}
	// Check for operators (+, -, *, /)
	else {
	if (expr[i] != '+' && expr[i] != '-' && expr[i] != '*' && expr[i] != '/') return 0; // Invalid operator
	}
	}

	return 1; // Valid expression
}

int main() {
	char expr[100];

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

	if (isValid(expr))
	printf("Valid syntax\n");
	else
	printf("Invalid syntax\n");

	return 0;
}
```

Output:

```
Enter expression: a+b*c
Valid syntax

Enter expression: bc
Invalid syntax
```

Experiment No : 07

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

void eliminateLeftRecursion(char nonTerminal, char alpha[], char beta[]) {
        printf("Original Grammar:\n");
        printf("%c -> %c%s | %s\n", nonTerminal, nonTerminal, alpha, beta);

        // Grammar after eliminating left recursion
        printf("\nGrammar after eliminating left recursion:\n");
        printf("%c -> %s%c'\n", nonTerminal, beta, nonTerminal);
        printf("%c' -> %s%c' | (empty)\n", nonTerminal, alpha, nonTerminal);
}

int main() {
        char nonTerminal = 'A';
        char alpha[] = "a";
        char beta[] = "b";

        eliminateLeftRecursion(nonTerminal, alpha, beta);

        return 0;
}
```

Output:

```
Original Grammar:
A -> Aa | b

Grammar after eliminating left recursion:
A -> bA'
A' -> aA' | (empty)
```

Experiment No : 08

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

#define MAX 10

struct Rule { char left[10], right[10]; };

int main() {
        int n, i = 0, j;
        char input[50], stack[50] = "", tmp[20], *sub;
        struct Rule rules[MAX];

        printf("Enter number of rules: ");
        scanf("%d", &n);
        getchar(); // consume newline after scanf

        printf("Enter rules in format X->Y (no spaces):\n");
        for (j = 0; j < n; j++) {
        fgets(tmp, sizeof(tmp), stdin);
        tmp[strcspn(tmp, "\n")] = 0;  // remove newline

        char *l = strtok(tmp, "->");
        char *r = strtok(NULL, "->");

        if (l && r) {
        strcpy(rules[j].left, l);
        strcpy(rules[j].right, r);
        } else {
        printf("Invalid rule format. Please use format like A->aB\n");
        j--; // repeat this iteration
        }
        }

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

        while (i <= strlen(input)) {
        // SHIFT
        if (i < strlen(input)) {
        int len = strlen(stack);
        stack[len] = input[i];
        stack[len + 1] = '\0';
        printf("%s\t%s\tShift %c\n", stack, &input[i + 1], input[i]);
```

```
        i++;
        }

        // REDUCE
        int reduced = 0;
        for (j = 0; j < n; j++) {
        sub = strstr(stack, rules[j].right);
        if (sub != NULL) {
                int pos = sub - stack;
                stack[pos] = '\0';
                strcat(stack, rules[j].left);
                printf("%s\t%s\tReduce %s->%s\n", stack, &input[i], rules[j].left, rules[j].right);
                reduced = 1;
                break;
        }
        }
        if (reduced) continue;

        // FINAL CHECK
        if (i == strlen(input)) {
        if (strcmp(stack, rules[0].left) == 0)
                printf("\nAccepted\n");
        else
                printf("\nNot Accepted\n");
        break;
        }
        }
        return 0;
}
```

Output:

```
Enter number of production rules: 2
Enter the production rules (in the form 'left->right'):
E->E+E
E->E*E
Enter the input string: E+E*E
E       +E*E    Shift E
E+      E*E     Shift +
E+E     *E      Shift E
E       *E      Reduce E->E+E
E*      E       Shift *
E*E     Shift E
E       Reduce E->E*E
Accepted
```

Experiment No : 09

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

int precedence(char op) {
	return (op == '+' || op == '-') ? 1 : (op == '*' || op == '/') ? 2 : 0;
}

void infixToPostfix(char *infix, char *postfix) {
	char stack[100];
	int top = -1, k = 0;

	for (int i = 0; infix[i]; i++) {
	char ch = infix[i];
	if (isalnum(ch)) postfix[k++] = ch;
	else if (ch == '(') stack[++top] = ch;
	else if (ch == ')') {
	while (top >= 0 && stack[top] != '(') postfix[k++] = stack[top--];
	top--; // pop '('
	} else {
	while (top >= 0 && precedence(stack[top]) >= precedence(ch)) postfix[k++] =
stack[top--];
	stack[++top] = ch;
	}
	}
	while (top >= 0) postfix[k++] = stack[top--];
	postfix[k] = '\0';
}

void generateTAC(char *postfix) {
	char stack[100][10];
	int top = -1, temp = 1;

	for (int i = 0; postfix[i]; i++) {
	char ch = postfix[i];
	if (isalnum(ch)) {
	char op[2] = {ch, '\0'};
	strcpy(stack[++top], op);
	} else {
	char arg2[10], arg1[10], res[10];
	strcpy(arg2, stack[top--]);
	strcpy(arg1, stack[top--]);
	sprintf(res, "t%d", temp++);
```

```
            printf("%s = %s %c %s\n", res, arg1, ch, arg2);
            strcpy(stack[++top], res);
            }
            }
}

int main() {
            char expr[100], postfix[100];
            printf("Enter an expression: ");
            scanf("%s", expr);

            infixToPostfix(expr, postfix);
            printf("\nThree Address Code:\n");
            generateTAC(postfix);

            return 0;
}
```

Output:
Enter an expression: (a+b)*c-d

Three Address Code:
t1 = a + b
t2 = t1 * c
t3 = t2 - d