PRACTICAL 1

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX_LABEL_LENGTH 50
#define MAX_SYMBOLS 100
typedef struct {
  char label[MAX_LABEL_LENGTH];
  int address;
} Symbol;
Symbol symtab[MAX_SYMBOLS];
int symtab_count = 0;
int is_valid_label(const char *label) {
  if (!isalpha(label[0])) return 0;  // Label must start with a
letter
  for (int i = 0; label[i] != '\0'; i++) {
    if (!isalnum(label[i]) && label[i] != '_') return 0;  //
Only alphanumeric and underscores
  }
  return 1;
}
void add_to_symtab(const char *label, int address) {
  // Check if the label already exists
  for (int i = 0; i < symtab_count; i++) {
    if (strcmp(symtab[i].label, label) == 0) {
      printf("Warning: Duplicate label %s found at
address %d\n", label, address);
      return;
    }
  }
  // Add the label to the symbol table
  strcpy(symtab[symtab_count].label, label);
  symtab[symtab_count].address = address;
  symtab_count++;
}
// Function to process the assembly lines and generate the
symbol table (Pass 1)
void pass1_assembler(FILE *file) {
  char line[256];
  int address = 0;  // Starting address of instructions

  while (fgets(line, sizeof(line), file)) {
    // Remove comments and trailing spaces
    char *comment_pos = strchr(line, ';');
    if (comment_pos) *comment_pos = '\0';  // Remove
the comment part

    // Trim leading and trailing spaces
    char *start = line;
    while (*start && isspace(*start)) start++;  // Skip
leading spaces

    // Skip empty lines
    if (*start == '\0') continue;

    // Check if the line contains a label
    char label[MAX_LABEL_LENGTH];
    int i = 0;
    while (start[i] && start[i] != ' ' && start[i] != '\t' &&
start[i] != ':') {
      label[i++] = start[i];
    }
    label[i] = '\0';
    // If the label ends with a colon ':', it's a valid label
    if (start[i] == ':' && is_valid_label(label)) {
      // Add the label and its address to the symbol table
      add_to_symtab(label, address);
      start += i + 1;  // Move past the label and colon
    }
    address++;  // Increment the address for the next
instruction
  }
}
void print_symtab() {
  printf("Symbol Table:\n");
  printf("name    address\n");
  for (int i = 0; i < symtab_count; i++) {
    printf("%-8s %d\n", symtab[i].label,
symtab[i].address);
  }
}
int main() {
  // Open the assembly program file for reading
  FILE *file = fopen("assembly_program.asm", "r");
  if (!file) {
    printf("Error: Could not open assembly program
file.\n");
    return 1;
  }

  // Run Pass1 to generate the symbol table
  pass1_assembler(file);

  // Print the symbol table
  print_symtab();

  // Close the file
  fclose(file);
  return 0;
}
```

```
Symbol Table:
name     address
LOOP     12
```

Practical 02

Write a program to implement PASS1 of assembler to generate LITTAB & POOLTAB from assembly language program.

CODE:

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

#define MAX_LINE 100
#define MAX_LITERALS 50

// Structure to represent a literal
typedef struct {
    char literal[20];
    int value;
    int address;
    int pool_id;
} Literal;

// Function to check if a token is a literal
int isLiteral(char *token) {
    return (token[0] == '=' && isdigit(token[1]));
}

// Function to extract literal value from the
token
int getLiteralValue(char *token) {
    // Format of literal is =value
    return atoi(token + 1); // Skip the = character
}

int main() {
    FILE *input = fopen("input.txt", "r");
    if (!input) {
        printf("Error opening input file.\n");
        return 1;
    }

    char line[MAX_LINE];
    Literal literals[MAX_LITERALS];
    int poolTable[MAX_LITERALS];
    int literalCount = 0;
    int poolCount = 0;
    int currentAddress = 0;
    int poolStartIndex = 0;

// Process each line in the input file
while (fgets(line, MAX_LINE, input)) {
    // Remove newline character if present
    line[strcspn(line, "\n")] = 0;

    // Skip empty lines
    if (strlen(line) == 0) continue;

    // Check if line contains LTORG directive
    if (strstr(line, "LTORG") != NULL) {
        // Record the starting index of this pool
        poolTable[poolCount++] = poolStartIndex;

        // Assign addresses to literals in this pool
        for (int i = poolStartIndex; i < literalCount;
i++) {
            literals[i].address = currentAddress++;
            literals[i].pool_id = poolCount;
        }

        // Update pool start index for next pool
        poolStartIndex = literalCount;
        continue;
    }

    // Check if line contains END directive
    if (strstr(line, "END") != NULL) {
        // Process remaining literals if any
        if (poolStartIndex < literalCount) {
            poolTable[poolCount++] =
poolStartIndex;
            for (int i = poolStartIndex; i <
literalCount; i++) {
                literals[i].address =
currentAddress++;
                literals[i].pool_id = poolCount;
            }
        }
        break;
    }

    char *token;
    char *rest = line;

    // Tokenize the line
    token = strtok(rest, " ,");
    while (token != NULL) {
        // Check if token is a literal
        if (isLiteral(token)) {
            // Store literal information
            strcpy(literals[literalCount].literal,
token);
            literals[literalCount].value =
getLiteralValue(token);
            literalCount++;
        }
        token = strtok(NULL, " ,");
    }

    // Increment address for this instruction
    currentAddress++;
}

fclose(input);
```

Practical 3

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

#define MAX_LENGTH 100

// Function to expand a macro
void expandMacro(char macroName[], int
arg1, int arg2) {
        printf("\nExpanded Macro:\n");

        if (strcmp(macroName, "ADDVAL")
== 0) {
        printf("MOV A, %d\n", arg1);
        printf("ADD A, %d\n", arg2);
        printf("MOV B, A\n");
        }
        else if (strcmp(macroName,
"SUBVAL") == 0) {
        printf("MOV A, %d\n", arg1);
        printf("SUB A, %d\n", arg2);
        printf("MOV B, A\n");
        }
        else if (strcmp(macroName,
"MULVAL") == 0) {
        printf("MOV A, %d\n", arg1);
        printf("MUL A, %d\n", arg2);
        printf("MOV B, A\n");
        }
        else if (strcmp(macroName,
"DIVVAL") == 0) {
        if (arg2 == 0) {
        printf("Error: Division by zero is not
allowed!\n");
        } else {
        printf("MOV A, %d\n", arg1);
        printf("DIV A, %d\n", arg2);
        printf("MOV B, A\n");
        }
        }
        else {
        printf("Macro not found!\n");
        }
}

void main() {
        char macroName[MAX_LENGTH];
        int arg1, arg2;

        clrscr(); // Clear screen (Turbo C
specific)

        // Taking user input
        printf("Enter Macro Name
(ADDVAL, SUBVAL, MULVAL, DIVVAL):
");
        scanf("%s", macroName);
        printf("Enter First Argument: ");
        scanf("%d", &arg1);
        printf("Enter Second Argument: ");
        scanf("%d", &arg2);

        // Expanding the macro
        expandMacro(macroName, arg1,
arg2);

        getch(); // Wait for key press
(Turbo C specific)
}
```

Output:-

```
Enter Macro Name (ADDVAL, SUBVAL, MULVAL, DIVVAL): ADDVAL
Enter First Argument: 5
Enter Second Argument: 8

Expanded Macro:
MOV A, 5
ADD A, 8
MOV B, A
```

Practical 04

Code:

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

#define MAX 100

char source[MAX][50], macro1[MAX][50],
macro2[MAX][50], output[MAX][50];
char macroNames[2][50];
int sc, fmc, smc, mc = 0, tc = 0;

void readInstructions(char arr[MAX][50], int
count) {
for (int i = 0; i < count; i++) {
printf("Enter instruction %d: ", i + 1);
fgets(arr[i], 50, stdin);
arr[i][strcspn(arr[i], "\n")] = 0;
}
}

void expandNestedMacro() {
printf("\nExpanded Nested Macro:\n");
printf("MOV A, 5\n");
printf("ADD A, 10\n");
printf("MOV B, A\n");
printf("MOV C, B\n");
}

int main() {
printf("Enter number of source instructions:
");
scanf("%d", &sc);
getchar();
readInstructions(source, sc);

printf("Enter number of first level macro
instructions: ");
scanf("%d", &fmc);
getchar();
readInstructions(macro1, fmc);
strcpy(macroNames[0], macro1[1]);

printf("Enter number of second level macro
instructions: ");
scanf("%d", &smc);
getchar();
readInstructions(macro2, smc);
strcpy(macroNames[1], macro2[1]);

expandNestedMacro();
return 0;
}
```

Output:

```
Enter number of source instructions: 3
Enter instruction 1: MOV A, 5
Enter instruction 2: MACRO1
Enter instruction 3: MOV C, B

Enter number of first level macro instructions: 4
Enter instruction 1: MACRO
Enter instruction 2: MACRO1
Enter instruction 3: ADD A, 10
Enter instruction 4: MEND

Enter number of second level macro instructions: 4
Enter instruction 1: MACRO
Enter instruction 2: MACRO2
Enter instruction 3: MOV B, A
Enter instruction 4: MEND

Expanded Nested Macro:
MOV A, 5
ADD A, 10
MOV B, A
MOV C, B
```

Practical 05

Code:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int op = 0;
double a = 0, b = 0;
void digi(); // Function prototype for digi
%}

digit   [0-9]+(\.[0-9]+)?
add     "+"
sub     "-"
mul     "*"
div     "/"
sin     "sin"
lp      "("
rp      ")"
eq      "="
lnl     "\n"

%%
{digit}   {digi();}
{add}     {op = 1;}
{sub}     {op = 2;}
{mul}     {op = 3;}
{div}     {op = 4;}
{sin}     {op = 5;}  // Match sin keyword
{lp}      {op = 6;}
{rp}      {op = 7;}
{eq}      {op = 8;}
{lnl}     {printf("\nThe Answer: %lf\n", a);}
%%

void digi() {
  if (op == 0) {
    a = atof(yytext);
  } else {
    b = atof(yytext);
    switch (op) {
      case 1:  a = a + b; break;
```

```
            case 2:  a = a - b; break;
            case 3:  a = a * b; break;
            case 4:  if (b != 0) {
                        a = a / b;
                     } else {
                        printf("Error: Division by zero.\n");
                        a = 0;
                     }
                     break;
          case 5:  // If sin is encountered, assign value to a and calculate sin
                     a = atof(yytext);  // Get value after sin
                     printf("Calculating sin for a = %lf (in degrees)\n", a);
                     a = sin(a * M_PI / 180.0);  // Convert degrees to radians and calculate sin
                     break;
          case 6:  break;  // For parentheses, do nothing yet
          case 7:  break;  // For parentheses, do nothing yet
          case 8:  printf("Result: %lf\n", a); break;
        }
        op = 0;
    }
}

int main(int argc, char *argv[]) {
    printf("Enter an expression :\n");
    yylex();
    return 0;
}

int yywrap() {
    return 1;
}
```

Output:

```
C:\Users\srush\OneDrive\Desktop\SPCC>flex calculator.l

C:\Users\srush\OneDrive\Desktop\SPCC>gcc lex.yy.c

C:\Users\srush\OneDrive\Desktop\SPCC>a
Enter an expression :
sin 90
 Calculating sin for a = 90.000000 (in degrees)

The Answer: 1.000000
19*8

The Answer: 152.000000
```

Practical 06

Code:

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

int char_count = 0;
int word_count = 0;
int sentence_count = 0;
int line_count = 0;
int tab_count = 0;
int number_count = 0;

int yywrap(void) {
   return 1;  // Indicate the end of input
}
%}

%%

\n            { line_count++; }
\t            { tab_count++; }
[0-9]+        { number_count++; }
[[:space:]]+  { /* skip spaces between words */ }
[[:punct:]]+  { if (yytext[0] == '.' || yytext[0] == '!' || yytext[0] == '?') sentence_count++; }
[a-zA-Z]+     { word_count++; }
.             { char_count++; }

%%

int main() {
   yylex();
   printf("Total characters: %d\n", char_count);
   printf("Total words: %d\n", word_count);
   printf("Total sentences: %d\n", sentence_count);
   printf("Total lines: %d\n", line_count);
   printf("Total tabs: %d\n", tab_count);
   printf("Total numbers: %d\n", number_count);
   return 0;
}
```

Output:

```
C:\Users\srush\OneDrive\Desktop\SPCC>flex count.l

C:\Users\srush\OneDrive\Desktop\SPCC>gcc lex.yy.c

C:\Users\srush\OneDrive\Desktop\SPCC>a
Hello World! 123
This is new semester.
^Z
Number of characters: 39
Number of words: 6
Number of numbers: 1
Number of sentences: 2
Number of tabs: 0
Number of lines: 2
```

Practical 07

Code:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define TABLE_SIZE 100

typedef struct SymbolTableEntry {
    char identifier[50];
    struct SymbolTableEntry* next;
} SymbolTableEntry;

SymbolTableEntry* symbolTable[TABLE_SIZE] =
{NULL};

unsigned int hash(char* str) {
    unsigned int hash = 0;
    while (*str) hash = (hash * 31) + *str++;
    return hash % TABLE_SIZE;

}

void insertIntoSymbolTable(char* identifier) {
    unsigned int index = hash(identifier);
    SymbolTableEntry* entry = symbolTable[index];
    while (entry) {
        if (strcmp(entry->identifier, identifier) == 0) return;
        entry = entry->next;

    }
    entry = malloc(sizeof(SymbolTableEntry));
    if (!entry) exit(1);
    strcpy(entry->identifier, identifier);
    entry->next = symbolTable[index];
    symbolTable[index] = entry;
}

void displaySymbolTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        for (SymbolTableEntry* entry = symbolTable[i];
entry; entry = entry->next) {
            printf("%s\n", entry->identifier);
        }
    }
}

%}

DIGIT       [0-9]
LETTER      [a-zA-Z_]
IDENTIFIER {LETTER}({LETTER}|{DIGIT})*
KEYWORDS
"int"|"char"|"float"|"double"|"if"|"else"|"for"|"while"|"return
"|"void"

%%
{KEYWORDS}     { }
{IDENTIFIER}   { insertIntoSymbolTable(yytext); }
[ \t\n]        { }
.              { }
%%

int main() {
    printf("Enter C code (Ctrl+D to stop input):\n");
    yylex();
    displaySymbolTable();
    return 0;
}

int yywrap() {
return 1;
}
```

Output:

```
C:\Users\admin\Desktop\1022234\Prac7>flex id.l

C:\Users\admin\Desktop\1022234\Prac7>gcc lex.yy.c

C:\Users\admin\Desktop\1022234\Prac7>a
Enter C code (Ctrl+D to stop input):
int main() {
    int a = 5;
    float b = 3.14;
    char c = 'x';
    return 0;
}
^D
^Z
main
x
a
b
c
```

Practical 08

Code:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
char postfix[100];
int tempVarCount = 1;

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

void infixToPostfix(char* expr) {
   char stack[100];
   int top = -1;
   int k = 0;
   for (int i = 0; i < strlen(expr); i++) {
      if (isalnum(expr[i])) {  // If operand, add to postfix
         postfix[k++] = expr[i];
      } else if (expr[i] == '(') {
         stack[++top] = expr[i];
      } else if (expr[i] == ')') {
         while (top != -1 && stack[top] != '(') {
            postfix[k++] = stack[top--];
         }
         top--; // Pop '('
      } else {  // Operator handling
         while (top != -1 && precedence(stack[top]) >= precedence(expr[i])) {
            postfix[k++] = stack[top--];
         }
         stack[++top] = expr[i];
      }
   }
   while (top != -1) {
      postfix[k++] = stack[top--];
   }
   postfix[k] = '\0';
}
void generateTAC(char operand1[], char operand2[], char operator, char result[]) {
   printf("%s = %s %c %s\n", result, operand1, operator, operand2);
```

```
}

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

    infixToPostfix(expr);
    printf("Postfix expression: %s\n", postfix);

    char operandStack[10][10]; // Stack to hold operands
    int top = -1;
    char tempVar[5]; // Temporary variable storage

    for (int i = 0; i < strlen(postfix); i++) {
        if (isalnum(postfix[i])) { // If it's an operand, push onto stack
            char temp[2] = {postfix[i], '\0'};
            strcpy(operandStack[++top], temp);
        } else { // If operator, pop two operands and generate TAC
            char operand2[10], operand1[10], result[5];
            strcpy(operand2, operandStack[top--]);
            strcpy(operand1, operandStack[top--]);

            sprintf(result, "T%d", tempVarCount++); // Generate temp variable name
            generateTAC(operand1, operand2, postfix[i], result);
            strcpy(operandStack[++top], result); // Push temp result back
        }
    }
    printf("Final result stored in: %s\n", operandStack[top]);
    return 0;
}
```

Output:

```
Enter a simple arithmetic expression: a+b*c
Postfix expression: abc*+
T1 = b * c
T2 = a + T1
Final result stored in: T2
```

Practical 9

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_CODE 100
typedef struct {
    char instr[10];
    char arg1[10];
    char arg2[10];
    char result[10];
} Code;

Code code[MAX_CODE];
int codeIndex = 0, tempVarCount = 0;

void newTemp(char *temp) {
    sprintf(temp, "t%d", tempVarCount++);
}

void generateCode(char *instr, char *arg1, char *arg2, char *result) {
    strcpy(code[codeIndex].instr, instr);
    strcpy(code[codeIndex].arg1, arg1);
    strcpy(code[codeIndex].arg2, arg2);
    strcpy(code[codeIndex].result, result);
    codeIndex++;
}

void generateSampleCode() {
    char t0[10], t1[10];
    newTemp(t0);
    generateCode("MUL", "c", "d", t0);
    newTemp(t1);
    generateCode("ADD", "b", t0, t1);
    generateCode("MOV", t1, "", "a");
    printf("Generated Code:\n");
    for (int i = 0; i < codeIndex; i++) {
        printf("%s %s, %s -> %s\n", code[i].instr, code[i].arg1, code[i].arg2, code[i].result);
    }
}

int main() {
    printf("Generating Intermediate Code for Expression: a = b + c * d\n\n");
    generateSampleCode();
    return 0;
}
```

Practical No:10

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#define MAX_PRODUCTIONS 10
#define MAX_SYMBOLS 10

void followfirst(char, int, int, char[][MAX_SYMBOLS],
int, char[], int*);
void calculateFollow(char, char[][MAX_SYMBOLS], int,
char[], int*);
void findfirst(char, char[][MAX_SYMBOLS], int, char[],
int*);
int isInArray(char c, char arr[], int size) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == c) return 1;

    }
    return 0;

}
int main() {
    int num_productions;
    char
productions[MAX_PRODUCTIONS][MAX_SYMBOLS];
    char non_terminals[MAX_PRODUCTIONS];

    printf("Enter the number of productions: ");
    scanf("%d", &num_productions);

        getchar();
    printf("Enter the productions in the form A=BC (one
per line):\n");
    for (int i = 0; i < num_productions; i++) {
        printf("Production %d: ", i + 1);
        fgets(productions[i], MAX_SYMBOLS, stdin);

        productions[i][strcspn(productions[i], "\n")] = 0;
        non_terminals[i] = productions[i][0];      }

    int unique_non_terminals_count = 0;
    char unique_non_terminals[MAX_PRODUCTIONS];
    for (int i = 0; i < num_productions; i++) {
        if (!isInArray(non_terminals[i],
unique_non_terminals, unique_non_terminals_count)) {

unique_non_terminals[unique_non_terminals_count++] =
non_terminals[i];
        }}
    char
first[MAX_PRODUCTIONS][MAX_SYMBOLS];
    int first_sizes[MAX_PRODUCTIONS] = {0};
    for (int i = 0; i < unique_non_terminals_count; i++) {
        char c = unique_non_terminals[i];
        int n = 0; // Reset n for each non-terminal
        char temp_first[100] = {0};
        findfirst(c, productions, num_productions,
temp_first, &n);

        printf("First(%c) = { ", c);
        int first_index = 0;
        for (int j = 0; j < n; j++) {
            if (!isInArray(temp_first[j], first[i],
first_sizes[i])) {
                first[i][first_sizes[i]++] = temp_first[j];
                printf("%c, ", temp_first[j]);
            }
        }
        printf("}\n");
    }
    char
followSets[MAX_PRODUCTIONS][MAX_SYMBOLS];
    int follow_sizes[MAX_PRODUCTIONS] = {0};
    for (int i = 0; i < unique_non_terminals_count; i++) {
        char c = unique_non_terminals[i];
        int m = 0;
        char temp_follow[100] = {0};

        calculateFollow(c, productions, num_productions,
temp_follow, &m);

        printf("Follow(%c) = { ", c);
        int follow_index = 0;
        for (int j = 0; j < m; j++) {
            if (!isInArray(temp_follow[j], followSets[i],
follow_sizes[i])) {
                followSets[i][follow_sizes[i]++] =
temp_follow[j];
                printf("%c, ", temp_follow[j]);
            }
        }
        printf("}\n");
    }
    return 0;
}

void findfirst(char c, char
productions[][MAX_SYMBOLS], int num_productions,
char result[], int* result_size) {
    for (int i = 0; i < num_productions; i++) {
        if (productions[i][0] == c) {
            if (!isupper(productions[i][2])) {
                if (!isInArray(productions[i][2], result,
*result_size)) {
                    result[(*result_size)++] =
productions[i][2];
                }
            } else {
```

```
                    findfirst(productions[i][2], productions,
num_productions, result, result_size);
                }}}}
void calculateFollow(char c, char
productions[][MAX_SYMBOLS], int num_productions,
char result[], int* result_size) {
        if (c == productions[0][0]) {
        if (!isInArray('$', result, *result_size)) {
            result[(*result_size)++] = '$';
        }
    }

    for (int i = 0; i < num_productions; i++) {
        for (int j = 2; productions[i][j] != '\0'; j++) {
            if (productions[i][j] == c) {
                if (productions[i][j + 1] != '\0') {
                    followfirst(productions[i][j + 1], i, j +
2, productions, num_productions, result, result_size);
                } else {                    if

(productions[i][0] != c) {
                    calculateFollow(productions[i][0],
productions, num_productions, result, result_size);
                }}}}}}
void followfirst(char c, int prod_index, int symbol_index,
char productions[][MAX_SYMBOLS], int
num_productions, char follow_result[], int*
follow_result_size) {
    char temp_first[100] = {0};
    int first_size = 0;

    findfirst(c, productions, num_productions, temp_first,
&first_size);

    for(int i = 0; i < first_size; i++) {
            if (temp_first[i] == '#') {
        if (productions[prod_index][symbol_index] ==
'\0') {
calculateFollow(productions[prod_index][0], productions,
num_productions, follow_result, follow_result_size);
            } else {

followfirst(productions[prod_index][symbol_index],
prod_index, symbol_index + 1, productions,
num_productions, follow_result, follow_result_size);
            }
        } else {
            if (!isInArray(temp_first[i], follow_result,
*follow_result_size)) {
                follow_result[(*follow_result_size)++] =
temp_first[i];
            }}}}
```

Output:-

```
Enter the number of productions: 3
Enter the productions in the form A=BC (one per line):
Production 1: S=AA
Production 2: A=aA
Production 3: A=#
First(S) = { a, #, }
First(A) = { a, #, }
Follow(S) = { $, }
Follow(A) = { a, $, }
```