Set-9

1. The main function of the Intermediate code generation is producing three address code statements for a given input expression. The three address codes help in determining the sequence in which operations are actioned by the compiler. The key work of Intermediate code generators is to simplify the process of Code Generator. Write a C Program to Generate the Three address code representation for the given input statement.

Code:

```
#include <stdio.h>

#include <string.h>


int tempVarCount = 1;


// Function to generate a new temporary variable

void newTemp(char *temp) {

  sprintf(temp, "t%d", tempVarCount++);

}


// Function to generate three-address code for an arithmetic expression

void generateTAC(char *expr) {

  char op1[10], op2[10], op[2], result[10];

  int i = 0, j = 0;


  while (expr[i] != '\0') {

    if (expr[i] == '+' || expr[i] == '-' || expr[i] == '*' || expr[i] == '/') {

      op1[j] = '\0';

      j = 0;

      op[0] = expr[i];

      op[1] = '\0';
```

```
            i++;
            while (expr[i] == ' ') i++;
            while (expr[i] != ' ' && expr[i] != '\0') {
                op2[j++] = expr[i++];
            }
            op2[j] = '\0';

            newTemp(result);
            printf("%s = %s %s %s\n", result, op1, op, op2);
            strcpy(op1, result);
            j = 0;
        } else if (expr[i] != ' ') {
            op1[j++] = expr[i];
        }
        i++;
    }
}

int main() {
    char expr[100];
    printf("Enter an arithmetic expression: ");
    scanf("%[^"]s", expr);
    printf("Three Address Code:\n");
    generateTAC(expr);
    return 0;
}
```

2.Write a C Program for code optimization to eliminate common subexpression.

Code:

```
#include <stdio.h>

#include <string.h>


#define MAX 100


typedef struct {

   char expr[50];

   char temp[10];

} Expression;


Expression expressions[MAX];

int exprCount = 0, tempVarCount = 1;


// Function to generate a new temporary variable

void newTemp(char *temp) {

   sprintf(temp, "t%d", tempVarCount++);

}


// Function to check if an expression already exists

int findExpression(char *expr) {

   for (int i = 0; i < exprCount; i++) {

      if (strcmp(expressions[i].expr, expr) == 0) {

         return i;

      }

   }

   return -1;

}
```

```
// Function to optimize and print expressions
void optimizeExpression(char *expr) {
    char temp[10];
    int index = findExpression(expr);

    if (index == -1) {  // New expression, store it
        newTemp(temp);
        strcpy(expressions[exprCount].expr, expr);
        strcpy(expressions[exprCount].temp, temp);
        exprCount++;
        printf("%s = %s\n", temp, expr);
    } else {  // Use existing temporary variable
        printf("Using existing: %s = %s\n", expressions[index].temp, expr);
    }
}

int main() {
    int n;
    char expr[50];

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

    printf("Enter expressions:\n");
    for (int i = 0; i < n; i++) {
        fgets(expr, 50, stdin);
        expr[strcspn(expr, "\n")] = 0; // Remove newline
        optimizeExpression(expr);
    }


    return 0;
}
```

3.In a class, an English teacher was teaching the vowels and consonants to the students. She says "Vowel sounds allow the air to flow freely, causing the chin to drop noticeably, whilst consonant sounds are produced by restricting the air flow". As a class activity the students are asked to identify the vowels and consonants in the given word/sentence and count the number of elements in each. Write an algorithm to help the student to count the number of vowels and consonants in the given sentence.

Code:

```
#include <stdio.h>

#include <ctype.h>

#include <string.h>


// Function to count vowels and consonants

void countVowelsConsonants(char *str, int *vowelCount, int *consonantCount) {

    *vowelCount = 0;

    *consonantCount = 0;

    char vowels[] = "AEIOUaeiou";


    for (int i = 0; str[i] != '\0'; i++) {

        if (isalpha(str[i])) { // Check if character is a letter

            if (strchr(vowels, str[i])) {

                (*vowelCount)++;

            } else {

                (*consonantCount)++;

            }
        }
    }
}

int main() {
    char sentence[200];
    int vowels = 0, consonants = 0;

    printf("Enter a sentence: ");
    fgets(sentence, sizeof(sentence), stdin);

    countVowelsConsonants(sentence, &vowels, &consonants);

    printf("Number of vowels: %d\n", vowels);
    printf("Number of consonants: %d\n", consonants);

    return 0;
}
```

4. Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. In general there are 32 keywords. The prime function of Lexical Analyser is token Generation. Among the 6 types of tokens, differentiating Keyword and Identifier is a challenging issue. Thus write a LEX program to separate keywords and identifiers.

Code:

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

// List of C keywords
const char *keywords[] = {
    "auto", "break", "case", "char", "const", "continue", "default", "do", "double", "else",
"enum", "extern",
    "float", "for", "goto", "if", "int", "long", "register", "return", "short", "signed", "sizeof",
"static", "struct", "switch",
    "typedef", "union", "unsigned", "void", "volatile", "while"
};

#define NUM_KEYWORDS (sizeof(keywords) / sizeof(keywords[0]))

// Function to check if a word is a keyword
int isKeyword(char *word) {
    for (int i = 0; i < NUM_KEYWORDS; i++) {
        if (strcmp(word, keywords[i]) == 0) {
            return 1;
        }
    }
    return 0;
}
%}

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

```
[ \t\n]+ ; // Ignore whitespace

%%

int main() {
    printf("Enter a C program:\n");
    yylex();
    return 0;
}
```