Practical 1

- **Write a C program to remove all the comments from the program.**

Code:

```
#include <stdio.h>

int main() {
    FILE *inputFile, *outputFile;
    char inputFile_name[100], outputFile_name[100];
    char c, next;

    printf("Enter the name of the input file: ");
    scanf("%s", inputFile_name);

    printf("Enter the name of the output file: ");
    scanf("%s", outputFile_name);

    inputFile = fopen(inputFile_name, "r");
    if (inputFile == NULL) {
        printf("Unable to open input file.\n");
        return 1;
    }

    outputFile = fopen(outputFile_name, "w");
    if (outputFile == NULL) {
        printf("Unable to open output file.\n");
        fclose(inputFile);
        return 1;
    }

    int insideComment = 0;

    while ((c = fgetc(inputFile)) != EOF) {
        if (insideComment) {
            if (c == '*' && (next = fgetc(inputFile)) == '/') {
                insideComment = 0;
            }
        } else {
            if (c == '/') {
                next = fgetc(inputFile);
                if (next == '/') {
                    // Single-line comment, skip until the end of the line
                    while ((c = fgetc(inputFile)) != '\n' && c != EOF) {
                        continue;
```

```
                }
            } else if (next == '*') {
                // Multi-line comment, set insideComment flag
                insideComment = 1;
            } else {
                // Not a comment, write '/' and the next character
                fputc(c, outputFile);
                fputc(next, outputFile);
            }
        } else {
            // Not a comment, write the character
            fputc(c, outputFile);
        }
    }
}

fclose(inputFile);
fclose(outputFile);

printf("Comments removed successfully.\n");

return 0;
}
```

Output:

```
PS D:\adit\Sem7\CD> cd "d:\adit\Sem7\CD\" ; if ($?) { g++ practical1.cpp -o practical1 } ; if ($?) { .\practical1 }
Enter the name of the input file: example.txt
Enter the name of the output file: example2.txt
Comments removed successfully.
PS D:\adit\Sem7\CD>
```

example.txt

```
1   Hello
2   I am Gautam
3   /* This is comment */
4   // This is also a comment
```

example2.txt

```
1   Hello
2   I am Gautam
```

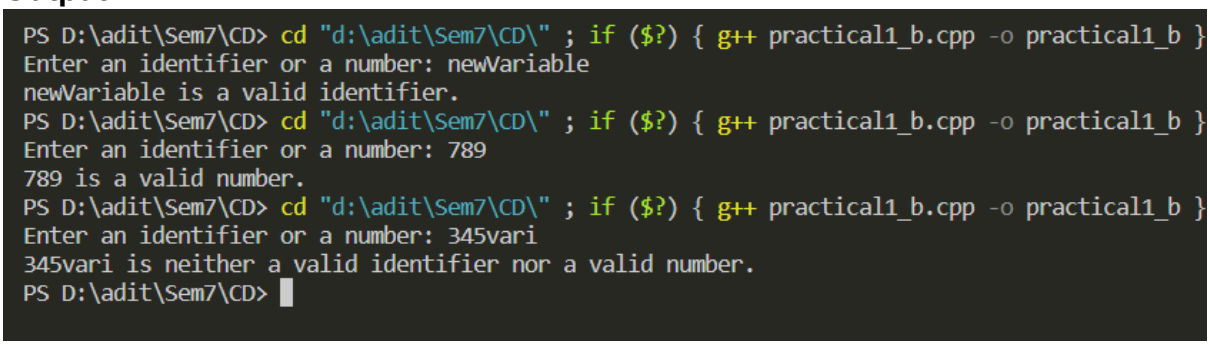
- **Write a C program to recognize identifiers and numbers.**

Code:

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

bool isIdentifierChar(char ch) {
    return isalnum(ch) || ch == '_';
}

bool isIdentifier(const char *str) {
    if (!isalpha(str[0]) && str[0] != '_')
        return false;

    for (int i = 1; i < strlen(str); i++) {
        if (!isIdentifierChar(str[i]))
            return false;
    }

    return true;
}

bool isNumber(const char *str) {
    int len = strlen(str);
    bool hasDecimal = false;

    for (int i = 0; i < len; i++) {
        if (i == 0 && (str[i] == '-' || str[i] == '+')) {
            continue;
        } else if (isdigit(str[i])) {
            continue;
        } else if (str[i] == '.') {
            if (hasDecimal)
                return false;
            hasDecimal = true;
        } else {
            return false;
        }
    }

    return hasDecimal || len > 1;
}
```

```
int main() {
    char input[100];

    printf("Enter an identifier or a number: ");
    scanf("%s", input);

    if (isIdentifier(input)) {
        printf("%s is a valid identifier.\n", input);
    } else if (isNumber(input)) {
        printf("%s is a valid number.\n", input);
    } else {
        printf("%s is neither a valid identifier nor a valid number.\n", input);
    }

    return 0;
}
```

Output:



Practical 2

- **Write a C program to generate tokens for a C program.**

Code:

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

typedef enum {
    TOKEN_UNKNOWN,
    TOKEN_IDENTIFIER,
    TOKEN_NUMBER,
    TOKEN_KEYWORD,
    TOKEN_OPERATOR,
    TOKEN_SEPARATOR,
} TokenType;

typedef struct {
    TokenType type;
    char value[100]; // Assuming a maximum token length of 100 characters
} Token;

int isKeyword(const char *str) {
    const char *keywords[] = {"int", "if", "else", "while", "return"};
    int numKeywords = sizeof(keywords) / sizeof(keywords[0]);

    for (int i = 0; i < numKeywords; i++) {
        if (strcmp(str, keywords[i]) == 0) {
            return 1;
        }
    }

    return 0;
}

void tokenizeCProgram(const char *program) {
    int programLength = strlen(program);
    int currentIndex = 0;

    while (currentIndex < programLength) {
        Token token;
        token.type = TOKEN_UNKNOWN;
        memset(token.value, 0, sizeof(token.value));
```

```
        while (isspace(program[currentIndex])) {
            currentIndex++;
        }

        if (isalpha(program[currentIndex]) || program[currentIndex] == '_') {
            int tokenLength = 0;
            while (isalnum(program[currentIndex]) || program[currentIndex] == '_')
{
                token.value[tokenLength++] = program[currentIndex++];
            }

            token.value[tokenLength] = '\0';

            if (isKeyword(token.value)) {
                token.type = TOKEN_KEYWORD;
            } else {
                token.type = TOKEN_IDENTIFIER;
            }
        }

        else if (isdigit(program[currentIndex]) || program[currentIndex] == '-' ||
program[currentIndex] == '+') {
            int tokenLength = 0;
            while (isdigit(program[currentIndex]) || program[currentIndex] == '.' ||
program[currentIndex] == 'e' || program[currentIndex] == 'E' ||
program[currentIndex] == '-' || program[currentIndex] == '+') {
                token.value[tokenLength++] = program[currentIndex++];
            }

            token.value[tokenLength] = '\0';
            token.type = TOKEN_NUMBER;
        }

        else {
            token.value[0] = program[currentIndex++];
            token.value[1] = '\0';

            if (strchr("+-*/%=&|<>!(){}", token.value[0]) != NULL) {
                token.type = TOKEN_OPERATOR;
            } else {
                token.type = TOKEN_SEPARATOR;
            }
        }

        printf("Type: %d, Value: %s\n", token.type, token.value);
```

```
    }
}

int main() {
    const char *cProgram = "int main() { int x = 5; return x; }";

    tokenizeCProgram(cProgram);

    return 0;
}
```

Output:

```
PS D:\adit\Sem7\CD> cd "d:\adit\Sem7\CD\" ; if ($?) { g++ practical2.cpp -o practical2 }
Type: 3, Value: int
Type: 1, Value: main
Type: 4, Value: (
Type: 4, Value: )
Type: 4, Value: {
Type: 3, Value: int
Type: 1, Value: x
Type: 4, Value: =
Type: 2, Value: 5
Type: 5, Value: ;
Type: 3, Value: return
Type: 1, Value: x
Type: 5, Value: ;
Type: 4, Value: }
PS D:\adit\Sem7\CD>
```

Practical 3

- **To Study about Lexical Analyzer Generator (LEX).**

  Lexical Analyzer Generator, often referred to as Lex, is a tool for generating lexical analyzers, which are an essential component of a compiler or interpreter. Lex helps convert a source code text into a stream of tokens for further processing by a parser. Lex is typically used in combination with another tool called Yacc (Yet Another Compiler Compiler) for creating the complete front end of a compiler.
  Here are the key concepts and steps to understand Lex and its role in compiler construction:

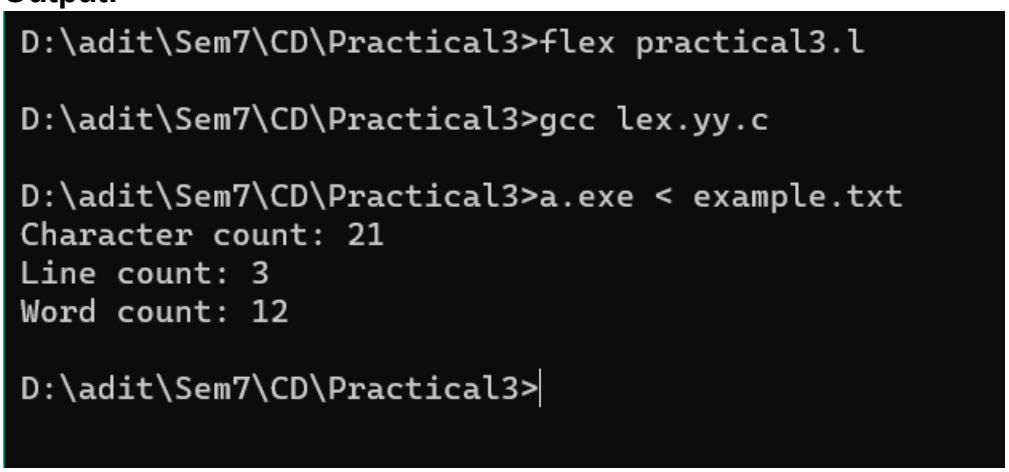
  1. **Lexical Analysis (Scanning):** Lexical analysis is the first phase of the compiler, where the source code is read character by character and converted into a stream of tokens. Tokens are the smallest units of meaning in a programming language, such as keywords, identifiers, constants, and operators. Lex helps automate the process of recognizing and categorizing tokens.
  2. **Regular Expressions:** Lex uses regular expressions to describe the patterns of tokens in the source code. Regular expressions are a powerful way to define matching patterns for strings. Lex allows you to define regular expressions that correspond to the tokens you want to recognize.
  3. **Lexical Rules:** In Lex, you write a set of rules that consist of regular expressions and associated actions. These rules specify how to recognize tokens and what to do when a match is found. When Lex reads the input source code, it applies these rules to find the longest possible match for each token.
  4. **Token Actions:** For each rule, you can specify an action to be performed when a token is recognized. These actions can include updating symbol tables, generating code, or simply recording information about the token.
  5. **Lexical Analyzer Generation:** Once you have defined your lexical rules in a Lex program, you run Lex to generate a C source code file for the lexical analyzer. Lex produces a C program that can scan the input source code and produce tokens based on your rules.
  6. **Integration with Yacc:** Lexical analyzers generated by Lex are often used in conjunction with parsers generated by Yacc (or Bison). Yacc is responsible for parsing the syntax and structure of the source code, while Lex handles the recognition of individual tokens.
  7. **Compilation:** After generating the lexical analyzer using Lex, you compile the resulting C code along with the parser generated by Yacc to create the complete compiler or interpreter.

- **Create a Lex program to take input from text file and count no of characters, no. of lines & no. of words.**

Code:

```
%{
#include <stdio.h>
int charCount = 0;
int lineCount = 0;
int wordCount = 0;
%}

%%
[a-zA-Z]+   { wordCount++; }
\n          { lineCount++; charCount++; }
.           { charCount++; }
%%

int main() {
    yylex();
    printf("Character count: %d\n", charCount);
    printf("Line count: %d\n", lineCount);
    printf("Word count: %d\n", wordCount);
    return 0;
}
```

Output:



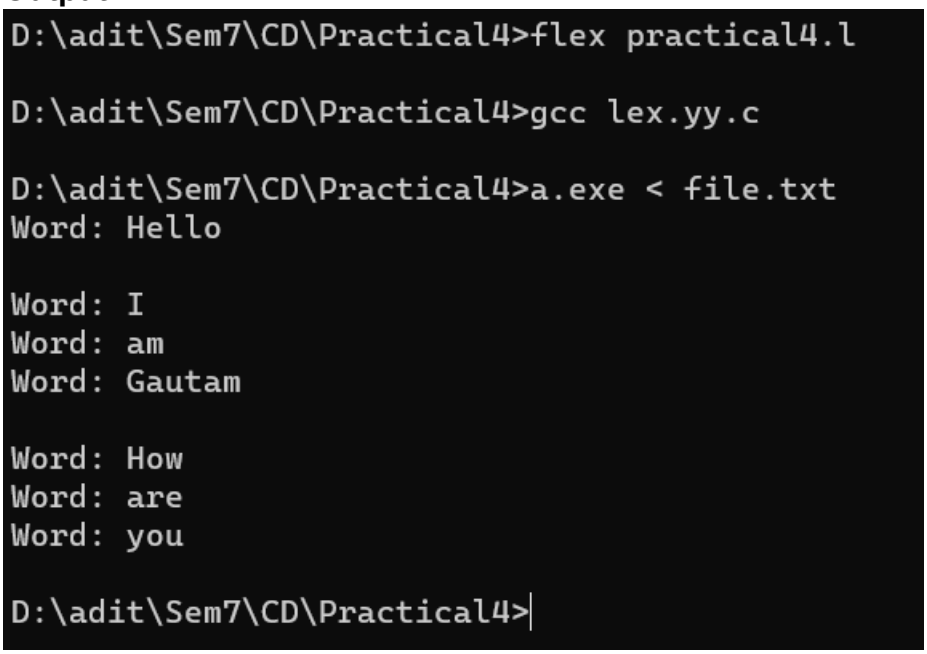
Practical 4

- **WAP to implement yytext method in a LEX program.**

Lex Code:

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

%%
[a-zA-Z]+   { printf("Word: %s\n", yytext); }
[0-9]+      { printf("Number: %s\n", yytext); }
.           { /* Ignore other characters */ }
%%
int yywrap(){
    return 1;
}
int main() {
    yylex();
    return 0;
}
```

Output:



- **WAP to implement ECHO, REJECT functions provided in Lex.**

Lex Code:

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

%%
[a-zA-Z]+   {
              printf("Word: %s\n", yytext);
              ECHO;
            }

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

.           { ECHO; }
%%

int yywrap(){
    return 1;
}

int main() {
    yylex();
    return 0;
}
```

Output:

```
D:\adit\Sem7\CD>flex practical4_b.l

D:\adit\Sem7\CD>gcc -o practical4_b lex.yy.c

D:\adit\Sem7\CD>practical4_b.exe < file.txt
Word: Hello
Hello
Word: I
I Word: am
am Word: Gautam
Gautam
Word: How
How Word: are
are Word: you
you
D:\adit\Sem7\CD>
```

- **WAP to implement BEGIN directive in a LEX program.**

Lex Code:

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

%option noyywrap

%x STATE1
%x STATE2

%%
"BEGIN1"    { BEGIN STATE1; }
"BEGIN2"    { BEGIN STATE2; }
<STATE1>.   { printf("In State 1: %s\n", yytext); }
<STATE2>.   { printf("In State 2: %s\n", yytext); }
.           { printf("Default State: %s\n", yytext); }
\n          { /* ignore newlines */ }

%%

int yywrap(){
    return 1;
}

int main() {
    yylex();
    return 0;
}
```

Output:

```
D:\adit\Sem7\CD>flex prac4_c.l

D:\adit\Sem7\CD>gcc -o prac4_c lex.yy.c

D:\adit\Sem7\CD>prac4_c.exe < file.txt
Default State: H
Default State: e
Default State: l
Default State: l
Default State: o
Default State:
Default State: I
Default State:
Default State: a
Default State: m
Default State:
Default State: G
Default State: a
Default State: u
Default State: t
Default State: a
Default State: m
Default State: H
Default State: o
Default State: w
Default State:
Default State: a
Default State: r
Default State: e
Default State:
Default State: y
Default State: o
Default State: u
```

```
Default State: ,
Default State:
Default State: y
Default State: o
Default State: u
Default State:
Default State: g
Default State: o
Default State: o
Default State: d
Default State: ?
```

Practical 5

- **Write a Lex program to count number of vowels and consonants in a given input string.**

Code:

```
%{
#include <stdio.h>
int vowelCount = 0;
int consonantCount = 0;
%}

%%
[aAeEiIoOuU]   { vowelCount++; }
[a-zA-Z]       { consonantCount++; }
.            { /* Ignore other characters */ }
%%

int yywrap(){
    return 1;
}

int main() {
    yylex();
    printf("Vowel count: %d\n", vowelCount);
    printf("Consonant count: %d\n", consonantCount);
    return 0;
}
```

Output:

```
D:\adit\Sem7\CD>flex prac5_a.l

D:\adit\Sem7\CD>gcc -o prac5_a lex.yy.c

D:\adit\Sem7\CD>prac5_a.exe < file.txt


Vowel count: 16
Consonant count: 14

D:\adit\Sem7\CD>
```

- **Write a Lex program to print out all numbers from the given file.**

Code:

```
%{
```

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

%%
[0-9]+        { printf("Number: %s\n", yytext); }
.             { /* Ignore other characters */ }
%%

int yywrap(){
    return 1;
}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    FILE* inputFile = fopen(argv[1], "r");
    if (!inputFile) {
        printf("Unable to open input file.\n");
        return 1;
    }
    yyin = inputFile; // Set Lex input to the provided file
    yylex(); // Start Lexical Analysis
    fclose(inputFile);
    return 0;
}
```

Output:

```
D:\adit\Sem7\CD>flex prac5_b.l

D:\adit\Sem7\CD>gcc -o prac5_b lex.yy.c

D:\adit\Sem7\CD>prac5_b number.txt
Number: 1

Number: 2

Number: 3

Number: 4

Number: 6

Number: 7

Number: 9

Number: 10
```

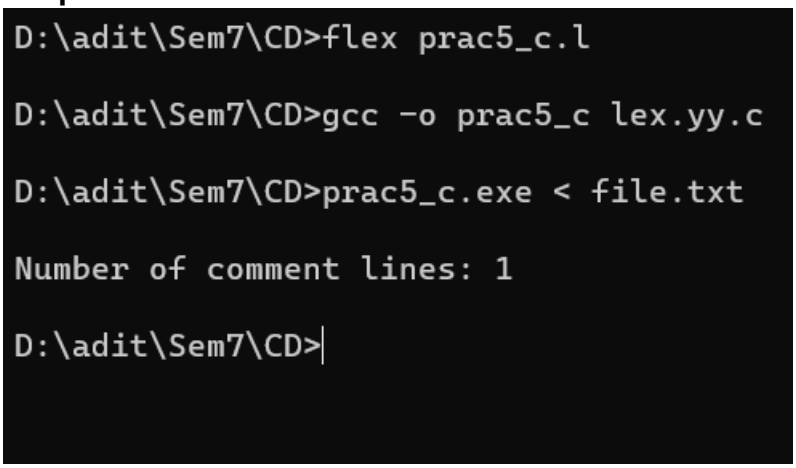
- **Write a Lex program to count the number of comment lines in a given C program.**

Code:

```
%{
int commentLines = 0; // Initialize the comment line count to zero
%}

%%
"/*"    { BEGIN(comment); } // Start of a multiline comment
"//"    { BEGIN(singleline_comment); } // Start of a single-line comment

<comment>.|\n   { /* Ignore characters inside multiline comments */ }

<comment>"*/"   { BEGIN(INITIAL); } // End of multiline comment

<singleline_comment>[^\n]*   { /* Ignore characters inside single-line
comments */ }
<singleline_comment>\n       { BEGIN(INITIAL); commentLines++; } // End of
single-line comment

.      { /* Ignore other characters */ }
%%

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

```
int main() {
    yylex(); // Start Lexical Analysis

    printf("Number of comment lines: %d\n", commentLines);

    return 0;
}
```

Output:



Practical 6

- **WAP to implement unput and input.**

Code:

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

%%
[a-zA-Z]+   {
              printf("Word: %s\n", yytext);
              unput(' '); // Push a space back into the input stream
            }

.           {
              char c = input(); // Get the next character from the input stream
              if (c != EOF) {
                  printf("Character: %c\n", c);
              }
            }
%%

int yywrap(){
    return 1;
}

int main() {
    yylex();
    return 0;
}
```

Output:

```
D:\adit\Sem7\CD>flex prac6_a.l

D:\adit\Sem7\CD>gcc -o prac6_a lex.yy.c

D:\adit\Sem7\CD>prac6_a < file.txt
Word: Hello
Character:

Word: I
Character:
Word: am
Character:
Word: Gautam
Character:

Word: How
Character:
Word: are
Character:
Word: you
Character: ,
Character: y
Word: ou
Character:
Word: good
Character: ?
```

- **WAP to implement yyterminate, yy_flush_bufferin LEX program.**

Code:

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

%option noinput   // Disable default input buffer

%%
[a-zA-Z]+   { printf("Word: %s\n", yytext); }

"exit"      { printf("Terminating lexer\n"); yyterminate(); }

\n          { /* Handle newline */ }

.           { /* Ignore other characters */ }
```

```
%%

int yywrap(){
    return 1;
}

int main() {
    while (true) {
        int c = getchar();
        if (c == EOF) {
            break;
        }
        unput(c);
        yylex();
    }
    return 0;
}
```

Output:

```
D:\adit\Sem7\CD>flex prac6_b.l

D:\adit\Sem7\CD>gcc -o prac6_b lex.yy.c

D:\adit\Sem7\CD>prac6_b < file.txt


Terminating the Lexer.
```

- **WAP to implement yywrap in LEX program.**

Code:

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

%%
[a-zA-Z]+   { printf("Word: %s\n", yytext); }
.           { /* Ignore other characters */ }
%%

int yywrap() {
    // Custom logic to switch to a new input source or terminate the lexer
    return 1; // Returning 1 indicates that there are no more input sources
}
```

```
int main() {
    yylex();
    return 0;
}
```

Output:

```
D:\adit\Sem7\CD>flex prac6_c.l

D:\adit\Sem7\CD>gcc -o prac6_c lex.yy.c

D:\adit\Sem7\CD>prac6_c < file.txt
Word: Hello

Word: I
Word: am
Word: Gautam

Word: How
Word: are
Word: you
Word: you
Word: good
```

- **WAP to implement yymore and yyless in LEX program.**

Code:

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

%%
[a-zA-Z]+   {
              printf("Original Token: %s\n", yytext);
              yymore();  // Append text to the current token
              printf("Appended Token: %s\n", yytext);
            }
[0-9]+      {
              printf("Original Token: %s\n", yytext);
              yyless(2); // Shorten the current token by 2 characters
              printf("Shortened Token: %s\n", yytext);
            }
.           { /* Ignore other characters */ }
%%

int main() {
```

```
    yylex();
    return 0;
}
```

Output:

```
D:\adit\Sem7\CD>flex prac6_d.l

D:\adit\Sem7\CD>gcc -o prac6_d lex.yy.c

D:\adit\Sem7\CD>prac6_d < file.txt
Original Token: Hello
Appended Token: Hello

Original Token: I
Appended Token: I
Original Token: am
Appended Token: am
Original Token: Gautam
Appended Token: Gautam

Original Token: How
Appended Token: How
Original Token: are
Appended Token: are
Original Token: you
Appended Token: you
Original Token: you
Appended Token: you
Original Token: good
Appended Token: good
```

Practical 7

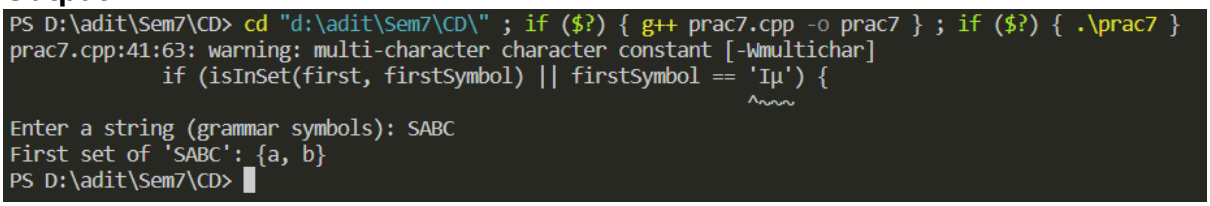
- **WAP to find the “First” set**
  Input: The string consists of grammar symbols.
  Output: The First set for a given string.
  Explanation: The student has to assume a typical grammar. The program when run will ask for the string to be entered. The program will find the First set of the given string.

Code:

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

typedef struct {
  char symbol;
  char* first_set;
} FirstSet;

FirstSet* find_first_set(char* string) {
  FirstSet* first_sets = malloc(sizeof(FirstSet) * strlen(string));
  for (int i = 0; i < strlen(string); i++) {
    first_sets[i].symbol = string[i];
    first_sets[i].first_set = NULL;
  }

  // Recursively calculate the First set for each symbol.
  for (int i = 0; i < strlen(string); i++) {
    if (string[i] == 'S' || string[i] == 'A' || string[i] == 'B') {
      first_sets[i].first_set = malloc(sizeof(char) * 2);
      first_sets[i].first_set[0] = string[i];
      first_sets[i].first_set[1] = '\0'; // Null-terminate the string.
    } else if (string[i] == 'C') {
      first_sets[i].first_set = malloc(sizeof(char) * 3);
      strcpy(first_sets[i].first_set, "ab");
    }
  }

  return first_sets;
}

int main() {
  char string[] = "SABC";
```

```
  // Find the First set of the string.
  FirstSet* first_sets = find_first_set(string);

  // Print the First set.
  for (int i = 0; i < strlen(string); i++) {
    printf("%c: %s\n", first_sets[i].symbol, first_sets[i].first_set);
  }

  // Free the memory allocated for the First set.
  for (int i = 0; i < strlen(string); i++) {
    free(first_sets[i].first_set);
  }
  free(first_sets);

  return 0;
}
```

Output:



Practical 8

- **WAP to find the “Follow” set.**
  Input: The string consists of grammar symbols.
  Output: The Follow set for a given string.
  Explanation: The student has to assume a typical grammar. The program when run will ask for the string to be entered. The program will find the Follow set of the given string.

Code:

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

#define MAX_RULES 100
#define MAX_SYMBOLS 10

typedef struct {
    char symbols[MAX_SYMBOLS];
    int count;
} SymbolSet;

typedef struct {
    char nonTerminal;
    char production[MAX_SYMBOLS];
} ProductionRule;

bool isTerminal(char symbol) {
    return !(symbol >= 'A' && symbol <= 'Z');
}

bool isNonTerminal(char symbol) {
    return (symbol >= 'A' && symbol <= 'Z');
}

void addToSet(SymbolSet *set, char symbol) {
    if (set->count < MAX_SYMBOLS) {
        set->symbols[set->count++] = symbol;
    }
}

bool addToSetUnique(SymbolSet *set, char symbol) {
    for (int i = 0; i < set->count; i++) {
        if (set->symbols[i] == symbol) {
```

```
            return false; // Symbol is already in the set
        }
    }
    addToSet(set, symbol);
    return true;
}

bool containsEpsilon(char production[MAX_SYMBOLS], int len) {
    for (int i = 0; i < len; i++) {
        if (production[i] != 'e') {
            return false;
        }
    }
    return true;
}

void calculateFollow(ProductionRule rules[MAX_RULES], int numRules, char
startSymbol, char symbol, SymbolSet *followSet) {
    for (int i = 0; i < numRules; i++) {
        ProductionRule rule = rules[i];
        int len = strlen(rule.production);
        for (int j = 0; j < len; j++) {
            if (rule.production[j] == symbol) {
                if (j < len - 1) {
                    // Case 1: The symbol has non-epsilon symbols following it
                    for (int k = j + 1; k < len; k++) {
                        if (isTerminal(rule.production[k])) {
                            addToSet(followSet, rule.production[k]);
                            break;
                        } else {
                            // For non-terminals, add First set of the non-terminal's
production
                            bool hasEpsilon = false;
                            for (int l = k; l < len; l++) {
                                if (isTerminal(rule.production[l])) {
                                    addToSet(followSet, rule.production[l]);
                                    break;
                                } else {
                                    if (addToSetUnique(followSet, rule.production[l])) {
                                        if (!containsEpsilon(rules[l].production,
strlen(rules[l].production))) {
                                            break;
                                        }
                                        hasEpsilon = true;
                                    }
                                }
```

```
                    }
                    if (!hasEpsilon) {
                        break;
                    }
                }
            }
        }
        if (j == len - 1 || containsEpsilon(rule.production + j + 1, len - j - 1)) {
            // Case 2: The symbol is at the end or followed by epsilon
            if (rule.nonTerminal != symbol) {
                calculateFollow(rules, numRules, startSymbol,
rule.nonTerminal, followSet);
            }
        }
    }
  }
}

int main() {
    int numRules;
    printf("Enter the number of production rules: ");
    scanf("%d", &numRules);

    ProductionRule rules[MAX_RULES];
    printf("Enter the production rules in the format 'NonTerminal ->
Production':\n");
    for (int i = 0; i < numRules; i++) {
        scanf(" %c -> %[^\n]s", &rules[i].nonTerminal, rules[i].production);
        getchar(); // Consume the newline character
    }

    char startSymbol;
    printf("Enter the start symbol: ");
    scanf(" %c", &startSymbol);

    char symbol;
    printf("Enter a symbol to find its Follow set: ");
    scanf(" %c", &symbol);

    SymbolSet followSet;
    followSet.count = 0;

    // Initialize Follow set with '$' if the start symbol is being followed
    if (symbol == startSymbol) {
        addToSet(&followSet, '$');
```

```
    }

    calculateFollow(rules, numRules, startSymbol, symbol, &followSet);

    printf("Follow(%c) = {", symbol);
    for (int i = 0; i < followSet.count; i++) {
        printf(" %c", followSet.symbols[i]);
    }
    printf(" }\n");

    return 0;
}
```

Output:

```
PS D:\adit\Sem7\CD> cd "d:\adit\Sem7\CD\" ; if ($?) { g++ prac8.cpp -o prac8
Enter the number of production rules: 4
Enter the production rules in the format 'NonTerminal -> Production':
S -> AB
A -> aA | b
B -> bB | c
C -> aC | d
Enter the start symbol: S
Enter a symbol to find its Follow set: S
Follow(S) = { $ }
```

Practical 9

- **Construct a recursive descent parser for a given grammar.**

Code:

Grammar:
E -> T + E | T - E | T
T -> F * T | F / T | F
F -> (E) | num

Code:

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

#define MAX_EXPRESSION_SIZE 100

// Function prototypes
bool parseE(char input[], int* index);
bool parseT(char input[], int* index);
bool parseF(char input[], int* index);

bool parseE(char input[], int* index) {
    if (parseT(input, index)) {
        while (input[*index] == '+' || input[*index] == '-') {
            (*index)++;
            if (!parseT(input, index)) {
                return false;
            }
        }
        return true;
    }
    return false;
}

bool parseT(char input[], int* index) {
    if (parseF(input, index)) {
        while (input[*index] == '*' || input[*index] == '/') {
            (*index)++;
            if (!parseF(input, index)) {
                return false;
            }
        }
        return true;
    }
```

```
    return false;
}

bool parseF(char input[], int* index) {
    if (input[*index] == '(') {
        (*index)++;
        if (parseE(input, index) && input[*index] == ')') {
            (*index)++;
            return true;
        }
        return false;
    } else if (isdigit(input[*index])) {
        while (isdigit(input[*index])) {
            (*index)++;
        }
        return true;
    }
    return false;
}

int main() {
    char input[MAX_EXPRESSION_SIZE];
    int index = 0;

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

    if (parseE(input, &index) && input[index] == '\0') {
        printf("Valid expression\n");
    } else {
        printf("Invalid expression\n");
    }

    return 0;
}
```

Output:

```
PS D:\adit\Sem7\CD> cd "d:\adit\Sem7\CD\" ; if ($?) { g++ prac9.cpp -o prac9
Enter an arithmetic expression: 9*5
Valid expression
PS D:\adit\Sem7\CD> cd "d:\adit\Sem7\CD\" ; if ($?) { g++ prac9.cpp -o prac9
Enter an arithmetic expression: 9*(4+3)/(4-2)
Valid expression
PS D:\adit\Sem7\CD>
```

Practical 10

- **Write a C program for constructing of LL (1) parsing.**

Code:

Grammar:
E -> E + T | E - T | T
T -> T * F | T / F | F
F -> (E) | num

Code:

```
#include <stdio.h>
#include <stdbool.h>
#include <ctype.h>
#define MAX_EXPRESSION_SIZE 100
// Function prototypes
bool parseE(char input[], int* index);
bool parseT(char input[], int* index);
bool parseF(char input[], int* index);

bool match(char expected, char input[], int* index) {
    if (input[*index] == expected) {
        (*index)++;
        return true;
    }
    return false;
}

bool parseF(char input[], int* index) {
    if (input[*index] == '(') {
        (*index)++;
        if (parseE(input, index) && match(')', input, index)) {
            return true;
        }
        return false;
    } else if (isdigit(input[*index])) {
        (*index)++;
        return true;
    }
    return false;
}

bool parseT(char input[], int* index) {
    if (parseF(input, index)) {
        while (input[*index] == '*' || input[*index] == '/') {
```

```
            char op = input[*index];
            (*index)++;
            if (!parseF(input, index)) {
                return false;
            }
        }
        return true;
    }
    return false;
}

bool parseE(char input[], int* index) {
    if (parseT(input, index)) {
        while (input[*index] == '+' || input[*index] == '-') {
            char op = input[*index];
            (*index)++;
            if (!parseT(input, index)) {
                return false;
            }
        }
        return true;
    }
    return false;
}

int main() {
    char input[MAX_EXPRESSION_SIZE];
    int index = 0;
    printf("Enter an arithmetic expression: ");
    scanf("%s", input);
    if (parseE(input, &index) && input[index] == '\0') {
        printf("Valid expression\n");
    } else {
        printf("Invalid expression\n");
    }
    return 0;
}
```

Output:

```
PS D:\adit\Sem7\CD> cd "d:\adit\Sem7\CD\" ; if ($?) { g++ prac10.cpp -o prac10
Enter an arithmetic expression: 2*3/8
Valid expression
PS D:\adit\Sem7\CD> cd "d:\adit\Sem7\CD\" ; if ($?) { g++ prac10.cpp -o prac10
Enter an arithmetic expression: (4*3)+9/(2*5)
Valid expression
PS D:\adit\Sem7\CD>
```

Practical 11

- **Implement a C program to implement operator precedence parsing.**

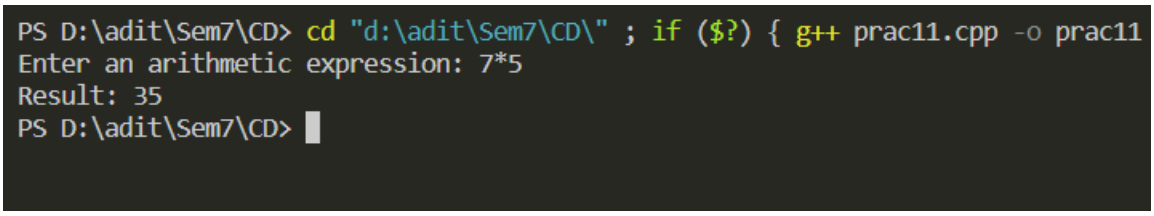
Code:

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

#define MAX_EXPRESSION_SIZE 100

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

// Function to perform binary operations
int applyOperator(int operand1, char operator, int operand2) {
    switch (operator) {
        case '+':
            return operand1 + operand2;
        case '-':
            return operand1 - operand2;
        case '*':
            return operand1 * operand2;
        case '/':
            if (operand2 != 0) {
                return operand1 / operand2;
            } else {
                printf("Error: Division by zero\n");
                exit(1);
            }
        default:
            printf("Error: Invalid operator\n");
            exit(1);
    }
```

```
}

// Operator precedence parsing function
int precedenceParse(char input[]) {
    int operandStack[MAX_EXPRESSION_SIZE];
    char operatorStack[MAX_EXPRESSION_SIZE];
    int operandTop = -1;
    int operatorTop = -1;
    int index = 0;

    while (input[index] != '\0') {
        if (isdigit(input[index])) {
            int operand = 0;
            while (isdigit(input[index])) {
                operand = operand * 10 + (input[index] - '0');
                index++;
            }
            operandStack[++operandTop] = operand;
        } else if (input[index] == '+' || input[index] == '-' || input[index] == '*' ||
input[index] == '/') {
            while (operatorTop >= 0 &&
getPrecedence(operatorStack[operatorTop]) >= getPrecedence(input[index]))
{
                int operand2 = operandStack[operandTop--];
                int operand1 = operandStack[operandTop--];
                char op = operatorStack[operatorTop--];
                int result = applyOperator(operand1, op, operand2);
                operandStack[++operandTop] = result;
            }
            operatorStack[++operatorTop] = input[index];
            index++;
        } else {
            printf("Error: Invalid character in expression\n");
            exit(1);
        }
    }

    while (operatorTop >= 0) {
        int operand2 = operandStack[operandTop--];
        int operand1 = operandStack[operandTop--];
        char op = operatorStack[operatorTop--];
        int result = applyOperator(operand1, op, operand2);
        operandStack[++operandTop] = result;
    }

    return operandStack[0];
```

```
}

int main() {
    char input[MAX_EXPRESSION_SIZE];

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

    int result = precedenceParse(input);
    printf("Result: %d\n", result);

    return 0;
}
```

Output:



Practical 12

- **Given a parsing table, Parse the given input using Shift Reduce Parser for any unambiguous grammar.**

Code:

Grammar:
1. S -> aSb
2. S -> ε

	‘+’	‘*’	‘(’	‘)’	‘id’	‘$’
0	S2		S4		S5	
1	S6	S7				
2	R2	S8		R2		R2
3	R4	R4		R4		R4
4	S6	S7				
5	R6	R6		R6		R6
6	S2		S4		S5	
7	R1	R1		R1		R1
8	R3	R3		R3		R3

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

#define MAX_STACK_SIZE 100
#define MAX_INPUT_SIZE 100

// Define the parsing table
int parsingTable[9][7] = {
    {2, -1, 4, -1, 5, -1, -1},
    {6, 7, -1, -1, -1, -1, -1},
    {-1, 8, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1},
    {6, 7, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1},
    {2, -1, 4, -1, 5, -1, -1},
```

```
    {-1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1}
};

// Define a stack for the parser
int stack[MAX_STACK_SIZE];
int top = -1;

// Function to perform a shift operation
void shift(int state) {
    top++;
    stack[top] = state;
}

// Function to perform a reduce operation
void reduce(int production) {
    int reduceLength;
    switch (production) {
        case 1:
            reduceLength = 3;
            break;
        case 2:
            reduceLength = 1;
            break;
        case 3:
            reduceLength = 3;
            break;
        case 4:
            reduceLength = 1;
            break;
        case 5:
            reduceLength = 3;
            break;
        case 6:
            reduceLength = 1;
            break;
        default:
            printf("Error: Invalid production\n");
            return;
    }

    // Pop elements from the stack based on the reduce length
    for (int i = 0; i < reduceLength; i++) {
        top--;
    }
```

```
    // Determine the non-terminal to push onto the stack
    int nonTerminal;
    switch (production) {
        case 1:
            nonTerminal = 0; // E
            break;
        case 2:
            nonTerminal = 0; // E
            break;
        case 3:
            nonTerminal = 1; // T
            break;
        case 4:
            nonTerminal = 1; // T
            break;
        case 5:
            nonTerminal = 2; // F
            break;
        case 6:
            nonTerminal = 2; // F
            break;
        default:
            printf("Error: Invalid production\n");
            return;
    }

    // Push the non-terminal onto the stack based on the goto entry in the
parsing table
    int newState = parsingTable[stack[top]][nonTerminal];
    stack[++top] = newState;
}

// Function to perform the parsing
bool parse(char input[]) {
    int index = 0;
    stack[++top] = 0; // Push the initial state onto the stack
    int state, action;

    while (true) {
        state = stack[top];
        action = parsingTable[state][getInputIndex(input[index])];

        if (action == -1) {
            printf("Error: Invalid input\n");
            return false;
        } else if (action == 0) {
```

```
            printf("Error: Parsing completed with an error\n");
            return false;
        } else if (action < 0) { // Reduce
            reduce(-action);
        } else { // Shift
            shift(action);
            index++;
        }

        if (input[index] == '\0' && state == 1) {
            printf("Parsing successful\n");
            return true;
        }
    }
}

// Function to get the index for the parsing table based on the input
int getInputIndex(char input) {
    switch (input) {
        case '+':
            return 0;
        case '*':
            return 1;
        case '(':
            return 2;
        case ')':
            return 3;
        case 'id':
            return 4;
        case '$':
            return 5;
        default:
            printf("Error: Invalid input symbol\n");
            exit(1);
    }
}

int main() {
    char input[MAX_INPUT_SIZE];

    printf("Enter an input string (e.g., 'id + id $'): ");
    scanf("%s", input);

    // Append '$' to the end of the input string
    strcat(input, "$");
```

```
    if (parse(input)) {
        printf("Parsing succeeded!\n");
    } else {
        printf("Parsing failed.\n");
    }

    return 0;
}
```

	Action					Goto		
	+	*	(	)	id	E	T	F
0	s2		s4		s5	1	3	6
1		s7						
2	r2	r2		r2	r2			
3								
4	s2		s4		s5		8	6
5								
6								
7	s2		s4		s5			9
8	r4	r4		r4	r4			
9	r6	r6		r6	r6			

Stack	Input	Action
0	id+id*id$	Shift (State 5)
0id	+id*id$	Reduce using production 6 (F->id)
0F	+id*id$	Shift (State 2)
0F+	id*id$	Shift (State 4)
0F+id	*id$	Reduce using production 6 (F->id)
0F+F	*id$	Reduce using production 4 (T->F)
0F	*id$	Reduce using production 1 (E->T)
0T	*id$	Shift (State 8)
0T*	id$	Shift (State 4)
0T*id	$	Reduce using production 6 (F->id)
0T*F	$	Reduce using production 4 (T->F)
0T	$	Reduce using production 3 (T->T*F)
0T*	$	Reduce using production 2 (E->T)
0E	$	Accept (Parsing successful)

Practical 13

- **Introduction to YACC and generate calculator program.**

Code:

Yacc:

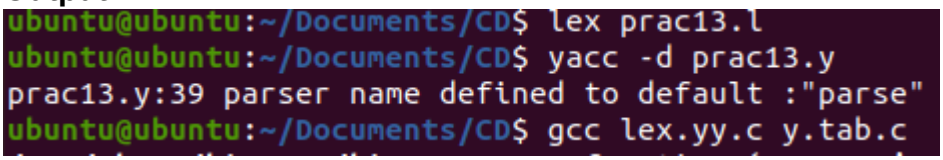
```
%{
/* Definition section */
#include<stdio.h>
int flag=0;
%}

%token NUMBER

%left '+' '-'

%left '*' '/' '%'

%left '(' ')'

/* Rule Section */
%%

ArithmeticExpression: E{

                        printf("\nResult=%d\n", $$);
                        return 0;
                        };
E:E'+'E {$$=$1+$3;}

|E'-'E {$$=$1-$3;}

|E'*'E {$$=$1*$3;}

|E'/'E {$$=$1/$3;}

|E'%'E {$$=$1%$3;}

|'('E')' {$$=$2;}

| NUMBER {$$=$1;}

;

%%
```

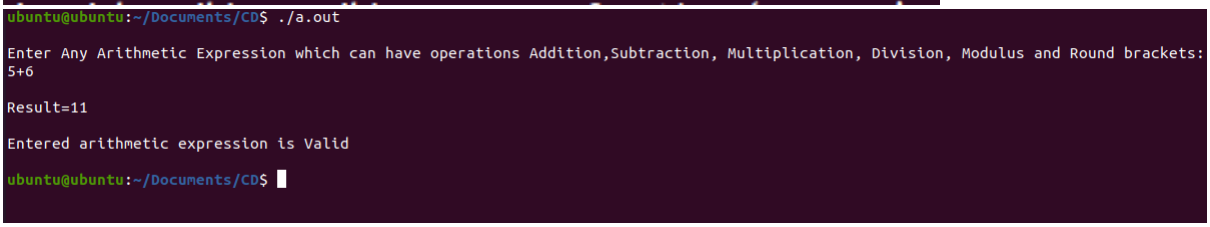
```
//driver code
void main()
{
printf("\nEnter Any Arithmetic Expression which can have operations Addition,
Subtraction, Multiplication, Division, Modulus and Round brackets:\n");

yyparse();
if(flag==0)
printf("\nEntered arithmetic expression is Valid\n\n");
}

int yyerror()
{
printf("\nEntered arithmetic expression is Invalid\n\n");
flag=1;
return 0;
}
```

Lex:

```
%{
/* Definition section */
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}

/* Rule Section */
%%
[0-9]+ {
                yylval=atoi(yytext);
                return NUMBER;

        }
[\t] ;

[\n] return 0;

. return yytext[0];

%%

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

Output:


ubuntu@ubuntu:~/Documents/CD$ lex prac13.l
ubuntu@ubuntu:~/Documents/CD$ yacc -d prac13.y
prac13.y:39 parser name defined to default :"parse"
ubuntu@ubuntu:~/Documents/CD$ gcc lex.yy.c y.tab.c


ubuntu@ubuntu:~/Documents/CD$ ./a.out
Enter Any Arithmetic Expression which can have operations Addition,Subtraction, Multiplication, Division, Modulus and Round brackets:
5+6
Result=11
Entered arithmetic expression is Valid
ubuntu@ubuntu:~/Documents/CD$

Practical 14

- **Generate 3-tuple intermediate code for given infix expression.**

Code:

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

// Structure to represent an intermediate code tuple
struct Tuple {
    char opcode;        // Operator (+, -, *, /, etc.)
    char operand1[5];   // Operand 1
    char operand2[5];   // Operand 2
    char result[5];     // Result
};

// Function to generate and print intermediate code
void generateIntermediateCode(char *expression) {
    struct Tuple code[100];
    int codeIndex = 0;
    char stack[100][5];
    int stackIndex = 0;
    int tempCount = 0;

    for (int i = 0; i < strlen(expression); i++) {
        char token = expression[i];

        if (token >= 'a' && token <= 'z') {
            // Operand
            sprintf(stack[stackIndex], "t%d", tempCount);
            tempCount++;
            stackIndex++;
        } else if (token == '+' || token == '-' || token == '*' || token == '/') {
            // Operator
            strcpy(code[codeIndex].operand2, stack[--stackIndex]);
            strcpy(code[codeIndex].operand1, stack[--stackIndex]);
            code[codeIndex].opcode = token;
            sprintf(code[codeIndex].result, "t%d", tempCount);
            strcpy(stack[stackIndex], code[codeIndex].result);
            stackIndex++;
            codeIndex++;
            tempCount++;
        }
    }
```

```
    // Print intermediate code
    printf("Intermediate Code:\n");
    for (int i = 0; i < codeIndex; i++) {
        printf("(%c, %s, %s, %s)\n", code[i].opcode, code[i].operand1,
code[i].operand2, code[i].result);
    }
}

int main() {
    char infixExpression[100];
    printf("Enter infix expression: ");
    scanf("%s", infixExpression);
    generateIntermediateCode(infixExpression);
    return 0;
}
```

Output:

```
PS D:\adit\Sem7\CD> cd "d:\adit\Sem7\CD\" ; if ($?) { g++ prac14.cpp -o prac14 } ; if ($?) { .\prac14 }
Enter infix expression: A+B*C
Intermediate Code:
(+, , N, t0)
(*, I, t0, t1)
```

Practical 15

- **Extract predecessor and successor from given control flow graph.**

Code:

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

#define MAX_BLOCKS 100

// Define a structure to represent a basic block
typedef struct {
    int label;  // Block label or identifier
    int successors[MAX_BLOCKS];
    int predecessors[MAX_BLOCKS];
    int numSuccessors;
    int numPredecessors;
} BasicBlock;

// Function to add a successor to a block
void addSuccessor(BasicBlock *block, int successor) {
    block->successors[block->numSuccessors++] = successor;
}

// Function to add a predecessor to a block
void addPredecessor(BasicBlock *block, int predecessor) {
    block->predecessors[block->numPredecessors++] = predecessor;
}

int main() {
    BasicBlock blocks[MAX_BLOCKS];
    int numBlocks = 0;

    // Initialize the CFG - you would populate this based on your actual CFG
structure
    numBlocks = 4;

    // For each block, set its label and successors (modify as needed)
    blocks[0].label = 0;
    addSuccessor(&blocks[0], 1);
    addSuccessor(&blocks[0], 2);

    blocks[1].label = 1;
    addSuccessor(&blocks[1], 2);
```

```
    blocks[2].label = 2;
    addSuccessor(&blocks[2], 3);

    blocks[3].label = 3;

    // Calculate predecessors for each block
    for (int i = 0; i < numBlocks; i++) {
        for (int j = 0; j < blocks[i].numSuccessors; j++) {
            int successor = blocks[i].successors[j];
            addPredecessor(&blocks[successor], i);
        }
    }

    // Print predecessors and successors
    for (int i = 0; i < numBlocks; i++) {
        printf("Block %d - Predecessors: ", blocks[i].label);
        for (int j = 0; j < blocks[i].numPredecessors; j++) {
            printf("%d ", blocks[i].predecessors[j]);
        }
        printf("- Successors: ");
        for (int j = 0; j < blocks[i].numSuccessors; j++) {
            printf("%d ", blocks[i].successors[j]);
        }
        printf("\n");
    }

    return 0;
}
```

Output:

```
PS D:\adit\Sem7\CD> cd "d:\adit\Sem7\CD\" ; if ($?) { g++ prac15.cpp -o prac15
Block 0 - Predecessors: - Successors: 1 2
Block 1 - Predecessors: 0 - Successors: 2
Block 2 - Predecessors: 0 1 - Successors: 3
Block 3 - Predecessors: 2 - Successors:
PS D:\adit\Sem7\CD>
```