CSE-306L Lab Copy

Perumalla Dharan
CSE_D
AP21110010201

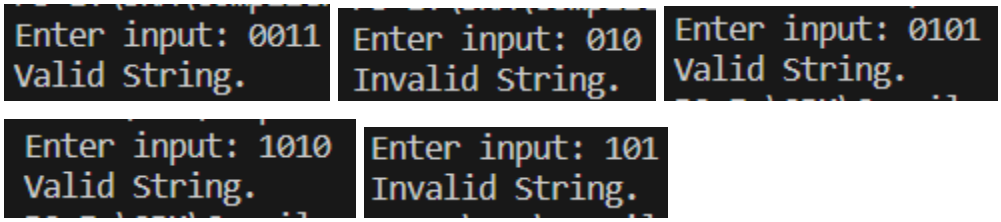
1. **Week 1 - Language recognizer**
   Write a program in C that recognizes the following languages.

a. **Set of all strings over binary alphabet containing even number of 0's and even number of 1's.**

```c
    // Week 1 Question 1
    // Set of all strings over binary alphabet containing even number of 0's and even number of 1's.

    #include <stdio.h>

    int isValidString(const char *str) {
        int count0 = 0, count1 = 0;
        while (*str != '\0') {
            if (*str == '0') {
                count0++;
            } else if (*str == '1') {
                count1++;
            } else {
                return 0;
            }
            str++;
        }
        return (count0 % 2 == 0) && (count1 % 2 == 0);
    }

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

```
        printf("Enter input: ");
        scanf("%s", input);
        if (isValidString(input)) {
            printf("Valid String.\n");
        } else {
            printf("Invalid String.\n");
        }
        return 0;
    }
```

Outputs-



b. **Lab Assignment: Set of all strings ending with two symbols of the same type.**

```
    // Week 1 Question 2
// Lab Assignment: Set of all strings ending with two symbols of
the same type.

#include <stdio.h>

int isValidString(const char *str)
{
    int length = 0;
    while (str[length] != '\0')
    {
        length++;
    }
    if (length >= 2 && (str[length - 1] == str[length - 2]))
```

```
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int main()
{
    char input[100];
    printf("Enter input: ");
    scanf("%s", input);
    if (isValidString(input))
    {
        printf("Valid String.\n");
    }
    else
    {
        printf("Invalid String.\n");
    }
    return 0;
}
```

Outputs-

```
Enter input: 101
Invalid String.
```

```
Enter input: 0011
Valid String.
```

```
Enter input: 1100
Valid String.
```

2. Week 2 - Implementation of Lexical analyzer using C

```
// Week 2 Question 1
// Lab Assignment: Lexical Analyzer using C
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_KEYWORDS 100
#define MAX_INDENT_LEN 100

char keywords[MAX_KEYWORDS][MAX_INDENT_LEN]={
    "int",
    "float",
    "char",
    "double",
    "if",
    "else",
    "for",
    "while",
    "do",
    "switch",
    "main()"
};

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

int isIdentifier(char *str){
    if(!((str[0]>='a' && str[0]<='z') || (str[0]>='A' &&
str[0]<='Z'))){
        return 0;
```

```
    }
    for(int i=1;i<strlen(str);i++){
        if(!((str[i]>='a' && str[i]<='z') || (str[i]>='A' &&
str[i]<='Z') || (str[i]>='0' && str[i]<='9'))){
            return 0;
        }
    }
    return 1;
}

int isInteger(char *str){
    for(int i=0;i<strlen(str);i++){
        if(!(str[i]>='0' && str[i]<='9')){
            return 0;
        }
    }
    return 1;
}

int isRelationalOperator(char *str){
    if(strcmp(str,"<")==0 || strcmp(str,">")==0 ||
strcmp(str,"<=")==0 || strcmp(str,">=")==0 ||
strcmp(str,"==")==0 || strcmp(str,"!=")==0){
        return 1;
    }
    return 0;
}

int isParenthesis(char *str){
    if(strcmp(str,"(")==0 || strcmp(str,")")==0 ||
strcmp(str,"{")==0 || strcmp(str,"}")==0){
        return 1;
    }
    return 0;
```

```
}

int main(){
    FILE *fp;
    fp=fopen("input.txt","r");
    if(fp==NULL){
        printf("Error opening file\n");
        exit(0);
    }
    char ch;
    char str[MAX_INDENT_LEN];
    int i=0;
    FILE *fp2=fopen("output.txt","w");
    while((ch=fgetc(fp))!=EOF){
        if(ch==' ' || ch=='\n' || ch=='\t'){
            if(i>0){
                str[i]='\0';
                if(isKeyword(str)){
                    // printf("%s is a keyword\n",str);
                    fprintf(fp2,"Keyword             :
%s\n",str);
                }
                else if(isIdentifier(str)){
                    // printf("%s is an identifier\n",str);
                    fprintf(fp2,"Identifier          :
%s\n",str);
                }
                else if(isInteger(str)){
                    // printf("%s is an integer\n",str);
                    fprintf(fp2,"Interger            :
%s\n",str);
                }
                else if(isRelationalOperator(str)){
```

```
                    // printf("%s is a relational
operator\n",str);
                    fprintf(fp2,"Relational Operator :
%s\n",str);
                }
                else if(isParenthesis(str)){
                    // printf("%s is a parenthesis\n",str);
                    fprintf(fp2,"Parenthesis         :
%s\n",str);
                }
                else{
                    // printf("%s is an invalid token\n",str);
                    fprintf(fp2,"Invalid Token       :
%s\n",str);
                }
                i=0;
            }
        }
        else{
            str[i++]=ch;
        }
    }
    fclose(fp);
    fclose(fp2);
    return 0;
}
```

Outputs-
Input file

```
int main ( ) {
    printf ( " Hello World " ) ;
    return 0 ;
}
```

Output file

```
Keyword              : int
Identifier           : main
Parenthesis          : (
Parenthesis          : )
Parenthesis          : {
Identifier           : printf
Parenthesis          : (
Invalid Token        : "
Identifier           : Hello
Identifier           : World
Invalid Token        : "
Parenthesis          : )
Invalid Token        : ;
Identifier           : return
Interger             : 0
Invalid Token        : ;
```

3. Week 3 - Introduction to LEX tool

Implement the following programs using Lex tool

a. Identification of Vowels and Consonants

b. count number of vowels and consonants

c. Count the number of Lines in given input

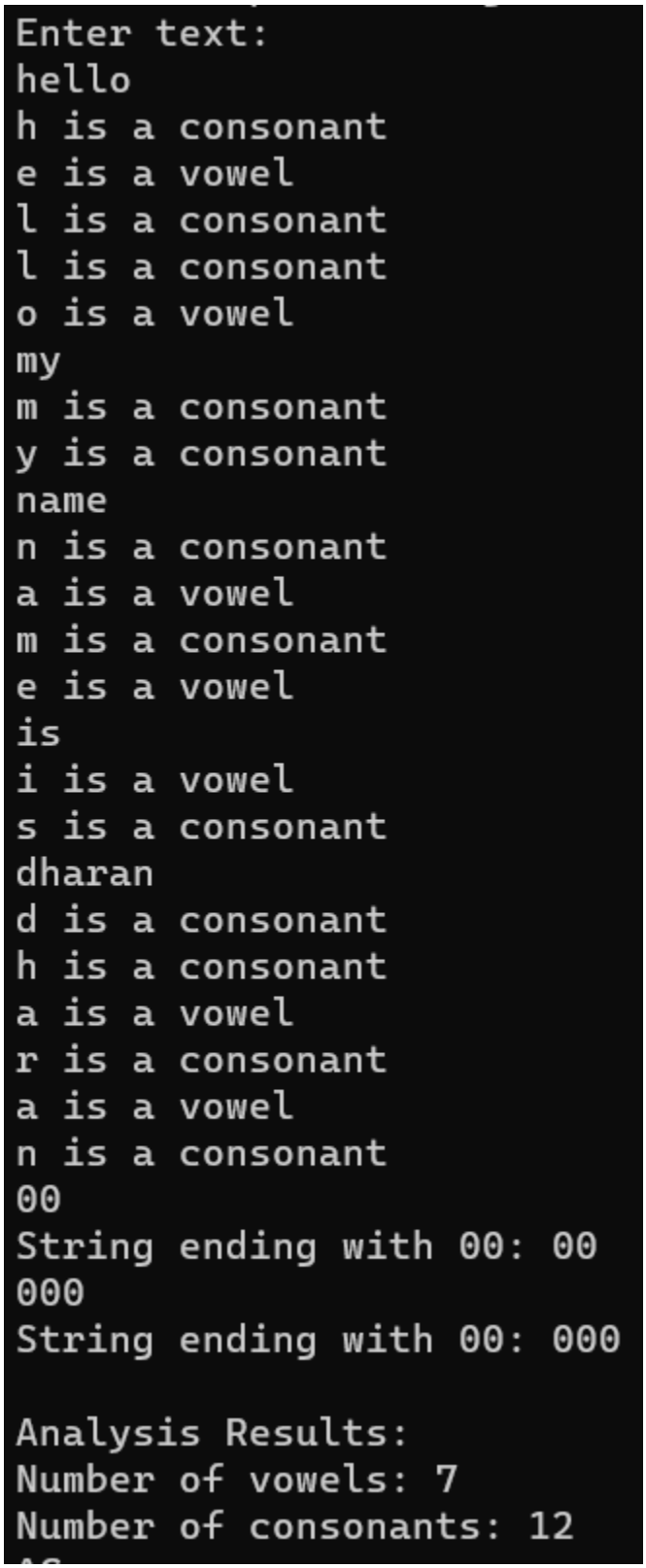
d. Recognize strings ending with 00

e. Recognize a string with three consecutive 0's

```
%{
```

```
#include <stdio.h>

int vowelCount = 0;
int consonantCount = 0;
int lineCount = 0;

%}

%%
[aeiouAEIOU]    { printf("%s is a vowel\n", yytext);
vowelCount++; }
[a-zA-Z]        { printf("%s is a consonant\n", yytext);
consonantCount++; }
\n              { lineCount++; }
.*00$           { printf("String ending with 00: %s\n",
yytext); }
.*000.*         { printf("String with three consecutive 0's:
%s\n", yytext); }
.               { /* Ignore other characters */ }

%%

int yywrap() {
    return 1; // Signal that there are no more files to
process
}

int main() {
    printf("Enter text:\n");
    yylex();

    printf("\nAnalysis Results:\n");
    printf("Number of vowels: %d\n", vowelCount);
    printf("Number of consonants: %d\n", consonantCount);
    printf("Number of lines: %d\n", lineCount);

    return 0;
}
```

Outputs-



```
Enter text:
hello
h is a consonant
e is a vowel
l is a consonant
l is a consonant
o is a vowel
my
m is a consonant
y is a consonant
name
n is a consonant
a is a vowel
m is a consonant
e is a vowel
is
i is a vowel
s is a consonant
dharan
d is a consonant
h is a consonant
a is a vowel
r is a consonant
a is a vowel
n is a consonant
00
String ending with 00: 00
000
String ending with 00: 000

Analysis Results:
Number of vowels: 7
Number of consonants: 12
```

4. Week 4 - Implementation of lexical analyzer using LEX

Implement lexical analyzer using LEX for recognizing the following tokens:

- **A minimum of 10 keywords of your choice**
- **Identifiers with the regular expression : letter(letter | digit)***
- **Integers with the regular expression: digit+**
- **Relational operators: <, >, <=, >=, ==, !=**
- **Ignores everything between multi line comments (/* …. */)**
- **Storing identifiers in symbol table**
- **Using files for input and output.**

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

#define MAX_IDENT_LEN 50
#define MAX_KEYWORDS 10

struct SymbolTable {
    char name[MAX_IDENT_LEN];
    int tokenType;
};

struct SymbolTable symbolTable[MAX_KEYWORDS];
int symbolTableSize = 0;

%}

%option noyywrap

letter      [a-zA-Z]
digit       [0-9]
id          {letter}({letter}|{digit})*
integer     {digit}+
```

```
rel_op      "<"|">"|"<="|">="|"=="|"!="

%%

"/*"[\s\S]*?"*/" ;

{rel_op}     { printf("Relational Operator: %s\n", yytext); }

{integer}    { printf("Integer: %s\n", yytext); }

{id} {
    int isKeyword = 0;
    char keywords[10][10] = {"if", "else", "while", "for",
"int", "float", "return", "break", "continue", "switch"};

    for (int i = 0; i < 10; i++) {
        if (strcmp(yytext, keywords[i]) == 0) {
            printf("Keyword: %s\n", yytext);
            isKeyword = 1;
            break;
        }
    }

    if (!isKeyword) {
        strcpy(symbolTable[symbolTableSize].name, yytext);
        symbolTable[symbolTableSize].tokenType = 1;
        symbolTableSize++;
        printf("Identifier: %s\n", yytext);
    }
}

.   { }

%%

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s input_file\n", argv[0]);
```

```
        return 1;
    }

    FILE *inputFile = fopen(argv[1], "r");
    if (!inputFile) {
        perror("Error opening input file");
        return 1;
    }

    yyin = inputFile;
    yylex();

    printf("\nSymbol Table:\n");
    for (int i = 0; i < symbolTableSize; i++) {
        printf("Name: %s, Token Type: %d\n",
symbolTable[i].name, symbolTable[i].tokenType);
    }

    fclose(inputFile);

    return 0;
}
```

Outputs-

Input file

```
int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Output in cmd

```
Keyword: int
Identifier: main

Identifier: printf
Identifier: Hello
Identifier: world
Identifier: n

Keyword: return
Integer: 0



Symbol Table:
Name: main, Token Type: 1
Name: printf, Token Type: 1
Name: Hello, Token Type: 1
Name: world, Token Type: 1
Name: n, Token Type: 1
```

5. Week 5 - Lexical Analyzer

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

#define MAX_KEYWORDS 100
#define MAX_INDENT_LEN 100

char keywords[MAX_KEYWORDS][MAX_INDENT_LEN]={
    "int",
    "if",
    "else",
    "for",
```

```
    "while",
    "then",
    "endif",
    "print",
};

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

int isDigit(char *str)
{
    for (int i = 0; i < strlen(str); i++)
    {
        if (!(str[i] >= '0' && str[i] <= '9'))
        {
            return 0;
        }
    }
    return 1;
}

int isLetter(char *str)
{
    if (!((str[0] >= 'a' && str[0] <= 'z') || (str[0] >= 'A' &&
str[0] <= 'Z')))
```

```
    {
        return 0;
    }
    for (int i = 1; i < strlen(str); i++)
    {
        if (!((str[i] >= 'a' && str[i] <= 'z') || (str[i] >= 'A'
&& str[i] <= 'Z')))
        {
            return 0;
        }
    }
    return 1;
}

int isMulop(char *str){
    if(strcmp(str,"*")==0 || strcmp(str,"/")==0){
        return 1;
    }
    return 0;
}

int isAddop(char *str){
    if(strcmp(str,"+")==0 || strcmp(str,"-")==0){
        return 1;
    }
    return 0;
}

int isRelop(char *str){
    if(strcmp(str,"<")==0 || strcmp(str,">")==0 ||
strcmp(str,"<=")==0 || strcmp(str,">=")==0 ||
strcmp(str,"==")==0 || strcmp(str,"!=")==0){
        return 1;
    }
```

```
    return 0;
}

int isAssignmnetop(char *str){
    if(strcmp(str,"=")==0){
        return 1;
    }
    return 0;
}

int isParenthesis(char *str){
    if(strcmp(str,"(")==0 || strcmp(str,")")==0 ||
strcmp(str,"{")==0 || strcmp(str,"}")==0){
        return 1;
    }
    return 0;
}

int isIdentifier(char *str){
    if(!((str[0]>='a' && str[0]<='z') || (str[0]>='A' &&
str[0]<='Z'))){
        return 0;
    }
    for(int i=1;i<strlen(str);i++){
        if(!((str[i]>='a' && str[i]<='z') || (str[i]>='A' &&
str[i]<='Z') || (str[i]>='0' && str[i]<='9'))){
            return 0;
        }
    }
    return 1;
}

int isComment(char *str) {
    if (str[0] == '/' && str[1] == '/') {
```

```
        return 1;
    } else if (str[0] == '/' && str[1] == '*') {
        int i = 2;
        while (str[i] != '\0') {
            if (str[i] == '*' && str[i + 1] == '/') {
                return 1;
            }
            i++;
        }
    }
    return 0;
}

int main()
{
    FILE *fp;
    fp = fopen("input.txt", "r");
    if (fp == NULL)
    {
        printf("Error opening file\n");
        exit(0);
    }
    char ch;
    char str[MAX_INDENT_LEN];
    int i = 0;
    FILE *fp2 = fopen("output.txt", "w");
    while ((ch = fgetc(fp)) != EOF)
    {
        if (ch == ' ' || ch == '\n' || ch == '\t')
        {
            if (i > 0)
            {
                str[i] = '\0';
                if (isKeyword(str))
```

```
                {
                    fprintf(fp2, "Keyword : %s\n", str);
                }
                else if(isDigit(str))
                {
                    fprintf(fp2, "Integer : %s\n", str);
                }
                else if(isLetter(str))
                {
                    fprintf(fp2, "Letter : %s\n", str);
                }
                else if(isMulop(str))
                {
                    fprintf(fp2, "Multiplication Operator :
%s\n", str);
                }
                else if(isAddop(str))
                {
                    fprintf(fp2, "Addition Operator : %s\n",
str);
                }
                else if(isRelop(str))
                {
                    fprintf(fp2, "Relational Operator : %s\n",
str);
                }
                else if(isAssignmnetop(str))
                {
                    fprintf(fp2, "Assignment Operator : %s\n",
str);
                }
                else if(isParenthesis(str))
                {
                    fprintf(fp2, "Parenthesis : %s\n", str);
```

```
                }
                else if(isIdentifier(str))
                {
                    fprintf(fp2, "Identifier : %s\n", str);
                }
                else if(isComment(str))
                {
                    fprintf(fp2, "Comment : %s\n", str);
                }
                else
                {
                    fprintf(fp2, "Invalid : %s\n", str);
                }
                i=0;
            }
        }
        else
        {
            str[i++] = ch;
        }
    }
    fclose(fp);
    fclose(fp2);
    return 0;
}
```

Outputs-
Input file

```
{
    int a[3] , t1 , t2 ;
    t1 = 2 ; a[0] = 1 ; a[1] = 2 ; a[t1] = 3 ;

    t2 = - ( a[2] + t1 * 6 ) / a[2] -t1 ) ;
    if t2 > 5 then
        print ( t2 ) ;
    else {
        int t3 ;
        t3 = 99 ;
        t2 = -25 ;
        print ( - t1 + t2 * t3 ) ; /* this is a comment on 2 lines */
    } endif
}
```

Output file

```
Parenthesis : {
Keyword : int
Invalid : a[3]
Invalid : ,
Identifier : t1
Invalid : ,
Identifier : t2
Invalid : ;
Identifier : t1
Assignment Operator : =
Integer : 2
Invalid : ;
Invalid : a[0]
Assignment Operator : =
Integer : 1
Invalid : ;
Invalid : a[1]
Assignment Operator : =
Integer : 2
Invalid : ;
Invalid : a[t1]
Assignment Operator : =
Integer : 3
Invalid : ;
Identifier : t2
Assignment Operator : =
Addition Operator : -
Parenthesis : (
Invalid : a[2]
Addition Operator : +
Identifier : t1
Multiplication Operator : *
```

```
Integer : 6
Parenthesis : )
Multiplication Operator : /
Invalid : a[2]
Invalid : -t1
Parenthesis : )
Invalid : ;
Keyword : if
Identifier : t2
Relational Operator : >
Integer : 5
Keyword : then
Keyword : print
Parenthesis : (
Identifier : t2
Parenthesis : )
Invalid : ;
Keyword : else
Parenthesis : {
Keyword : int
Identifier : t3
Invalid : ;
Identifier : t3
Assignment Operator : =
Integer : 99
Invalid : ;
Identifier : t2
Assignment Operator : =
Invalid : -25
Invalid : ;
Keyword : print
Parenthesis : (
```

```
Addition Operator : -
Identifier : t1
Addition Operator : +
Identifier : t2
Multiplication Operator : *
Identifier : t3
Parenthesis : )
Invalid : ;
Invalid : /*
Letter : this
Letter : is
Letter : a
Letter : comment
Letter : on
Integer : 2
Letter : lines
Invalid : */
Parenthesis : }
Keyword : endif
```

6. Week 6 - Recursive Descent Parser

Implement the Recursive Descent Parser for the Expression Grammar given below.

- **E -> TE'**
- **E' -> +TE' | ϵ**
- **T -> FT'**
- **T' -> *FT' | ϵ**
- **F -> (E) | i**

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

```c
struct Parser {
    char* expression;
    int index;
};

void initParser(struct Parser* parser, char* input) {
    parser->expression = input;
    parser->index = 0;
}

int parseE(struct Parser* parser);
int parseE_prime(struct Parser* parser);
int parseT(struct Parser* parser);
int parseT_prime(struct Parser* parser);
int parseF(struct Parser* parser);

int match(struct Parser* parser, char token) {
    if (parser->expression[parser->index] == token) {
        parser->index++;
        return 1;
    }
    return 0;
}

int parseE(struct Parser* parser) {
    if (parseT(parser) && parseE_prime(parser)) {
        return 1;
    }
    return 0;
}

int parseE_prime(struct Parser* parser) {
    if (parser->expression[parser->index] == '+') {
```

```
        parser->index++;
        if (parseT(parser) && parseE_prime(parser)) {
            return 1;
        }
        return 0;
    }
    return 1;  // E' can be epsilon
}

int parseT(struct Parser* parser) {
    if (parseF(parser) && parseT_prime(parser)) {
        return 1;
    }
    return 0;
}

int parseT_prime(struct Parser* parser) {
    if (parser->expression[parser->index] == '*') {
        parser->index++;
        if (parseF(parser) && parseT_prime(parser)) {
            return 1;
        }
        return 0;
    }
    return 1;  // T' can be epsilon
}

int parseF(struct Parser* parser) {
    if (parser->expression[parser->index] == '(') {
        parser->index++;
        if (parseE(parser) && parser->expression[parser->index]
== ')') {
            parser->index++;
            return 1;
```

```
        }
    } else if (isalpha(parser->expression[parser->index])) {
        parser->index++;
        return 1;
    }
    return 0;
}

int parse(struct Parser* parser) {
    return parseE(parser) && parser->expression[parser->index]
== '\0';
}

int main() {
    char input[100];
    struct Parser parser;

    while (1) {
        printf("Enter an expression:\n");
        if (fgets(input, sizeof(input), stdin) == NULL) {
            break;
        }

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

        if (strlen(input) == 0) {
            break;  // Exit when an empty line is entered
        }

        initParser(&parser, input);
```

```
        if (parse(&parser)) {
            printf("Parsing successful\n");
        } else {
            printf("Parsing failed\n");
        }
    }

    return 0;
}
```

Outputs -

```
Enter an expression:
(i+i)*(i*i)
Parsing successful
```

```
Enter an expression:
(i*i)*i
Parsing successful
```

```
Enter an expression:
i+i
Parsing successful
```

Lab Assignment: Construct Recursive Descent Parser for the grammar G = ({S, L}, {(,), a, ,}, {S → (L) | a ; L → L, S | S}, S) and verify the acceptability of the following strings:
(a,(a,a))
(a,((a,a),(a,a)))
You can manually eliminate Left Recursion if any in the grammar.

```
// Construct Recursive Descent Parser for the grammar
// G = ({S, L}, {(, ), a, ,}, {S  (L) | a ; L L, S | S}, S) and
verify the acceptability of the
// following strings:
// i. (a,(a,a))
// ii. (a,((a,a),(a,a)))

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

```
int i = 0, error = 0;
char input[100];

void S();
void L();

void S()
{
    if (input[i] == '(')
    {
        i++;
        L();
        if (input[i] == ')')
        {
            i++;
        }
        else
        {
            error = 1;
        }
    }
    else if (input[i] == 'a')
    {
        i++;
    }
    else
    {
        error = 1;
    }
}

void L()
{
    S();
```

```
    if (input[i] == ',')
    {
        i++;
        S();
    }
    else
    {
        error = 1;
    }
}

int main()
{
    FILE *fp1, *fp2;
    fp1 = fopen("input.txt", "r");
    fp2 = fopen("output.txt", "w");
    // printf("Enter the string: ");
    // scanf("%s", input);
    fscanf(fp1, "%s", input);

    while (input[i] != '\0')
    {
        S();
        if (strlen(input) == i && error == 0)
        {
            // printf("String is accepted\n");
            fprintf(fp2, "String is accepted\n");
        }
        else
        {
            // printf("String is not accepted\n");
            fprintf(fp2, "String is not accepted\n");
        }
    }
```

```
    return 0;
}
```

Outputs -

Input file

```
(a,((a,a),(a,a)))
```

Output file

```
String is accepted
```

7. Week 7 - Predictive parser

Write a C program for the computation of FIRST and FOLLOW for a given CFG.

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

void followfirst(char, int, int);
void follow(char c);
void findfirst(char, int, int);

int count, n = 0;
char calc_first[10][100];
char calc_follow[10][100];
int m = 0;
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;
```

```
int main(int argc, char **argv)
{
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
    count = 8;

    int nn;
    scanf("%d", &nn);
    char cc[100];
    for (int ii = 0; i < nn; i++)
    {
        scanf("%s", cc);
        strcpy(production[ii], cc);
        printf("%s\n", production[ii]);
    }

    int kay;
    char done[count];
    int ptr = -1;

    for (k = 0; k < count; k++)
    {
        for (kay = 0; kay < 100; kay++)
        {
            calc_first[k][kay] = '!';
        }
    }
    int point1 = 0, point2, temp;

    for (k = 0; k < count; k++)
    {
        c = production[k][0];
```

```
        point2 = 0;
        temp = 0;

        for (kay = 0; kay <= ptr; kay++)
            if (c == done[kay])
                temp = 1;

        if (temp == 1)
            continue;

        findfirst(c, 0, 0);
        ptr += 1;

        done[ptr] = c;
        printf("\n First(%c) = { ", c);
        calc_first[point1][point2++] = c;

        for (i = 0 + jm; i < n; i++)
        {
            int lark = 0, chk = 0;

            for (lark = 0; lark < point2; lark++)
            {

                if (first[i] == calc_first[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0)
            {
                printf("%c, ", first[i]);
                calc_first[point1][point2++] = first[i];
```

```
            }
        }
        printf("}\n");
        jm = n;
        point1++;
    }
    printf("\n");
    printf("------------------------------------------------"
           "\n\n");
    char donee[count];
    ptr = -1;

    for (k = 0; k < count; k++)
    {
        for (kay = 0; kay < 100; kay++)
        {
            calc_follow[k][kay] = '!';
        }
    }
    point1 = 0;
    int land = 0;
    for (e = 0; e < count; e++)
    {
        ck = production[e][0];
        point2 = 0;
        temp = 0;

        for (kay = 0; kay <= ptr; kay++)
            if (ck == donee[kay])
                temp = 1;

        if (temp == 1)
            continue;
        land += 1;
```

```
        follow(ck);
        ptr += 1;

        donee[ptr] = ck;
        printf(" Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;

        for (i = 0 + km; i < m; i++)
        {
            int lark = 0, chk = 0;
            for (lark = 0; lark < point2; lark++)
            {
                if (f[i] == calc_follow[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0)
            {
                printf("%c, ", f[i]);
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }
}

void follow(char c)
{
    int i, j;
```

```
    if (production[0][0] == c)
    {
        f[m++] = '$';
    }
    for (i = 0; i < 10; i++)
    {
        for (j = 2; j < 10; j++)
        {
            if (production[i][j] == c)
            {
                if (production[i][j + 1] != '\0')
                {
                    followfirst(production[i][j + 1], i,
                                (j + 2));
                }

                if (production[i][j + 1] == '\0' && c !=
production[i][0])
                {
                    follow(production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c, int q1, int q2)
{
    int j;

    if (!(isupper(c)))
    {
        first[n++] = c;
```

```
    }
    for (j = 0; j < count; j++)
    {
        if (production[j][0] == c)
        {
            if (production[j][2] == '@')
            {
                if (production[q1][q2] == '\0')
                    first[n++] = '@';
                else if (production[q1][q2] != '\0' && (q1 != 0
|| q2 != 0))
                {
                    findfirst(production[q1][q2], q1,
                              (q2 + 1));
                }
                else
                    first[n++] = '@';
            }
            else if (!isupper(production[j][2]))
            {
                first[n++] = production[j][2];
            }
            else
            {
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

void followfirst(char c, int c1, int c2)
{
    int k;
```

```
    if (!(isupper(c)))
        f[m++] = c;
    else
    {
        int i = 0, j = 1;
        for (i = 0; i < count; i++)
        {
            if (calc_first[i][0] == c)
                break;
        }

        while (calc_first[i][j] != '!')
        {
            if (calc_first[i][j] != '@')
            {
                f[m++] = calc_first[i][j];
            }
            else
            {
                if (production[c1][c2] == '\0')
                {
                    follow(production[c1][0]);
                }
                else
                {
                    followfirst(production[c1][c2], c1,
                                c2 + 1);
                }
            }
            j++;
        }
    }
}
```

Outputs -

```
S=AaAb
S=BbBa
A=@
B=@

 First(B) = { @, }

 First() = { }

-----------------------------------------------

 Follow(B) = { $, a,  }

 Follow() = { a, $,  }
```

8. Week 8 - Implement non-recursive Predictive Parser for the grammar
S -> aBa
B -> bB | ε

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

char input[100];
int pointer = 0;
int error = 0;

void S();
void B();
```

```
void match(char c) {
    if (input[pointer] == c) {
        pointer++;
    } else {
        error = 1;
    }
}

void S() {
    if (input[pointer] == 'a') {
        printf("S -> a");
        match('a');
        B();
        printf("a\n");
        if (input[pointer] == 'a') {
            match('a');
        } else {
            error = 1;
            printf("Error: Expected 'a' after 'B'\n");
        }
    } else {
        error = 1;
        printf("Error: Expected 'a' at the beginning\n");
    }
}

void B() {
    if (input[pointer] == 'b') {
        printf("B -> b");
        match('b');
        B();
        printf("b\n");
    } else {
        printf("B -> e\n");
```

```
    }
}

int main() {
    char grammar[100];
    char line[100];

    printf("Enter the grammar rules line by line. Type 'break'
on a new line to finish grammar input:\n");
    grammar[0] = '\0';

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

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

        strcat(grammar, line);
    }

    printf("Grammar input completed. Enter a string: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = 0;

    pointer = 0;
    error = 0;

    if (strcmp(grammar, "S -> aBa\nB -> bB | e\n") != 0) {
        printf("Grammar is not as expected.\n");
        return 1;
    }

    printf("Parsing steps:\n");
```

```
    S();

    if (error == 0 && input[pointer] == '\0') {
        printf("Parsing successful\n");
    } else {
        printf("Parsing error\n");
    }

    return 0;
}
```

Outputs -

```
Enter the grammar rules line by line. Type 'break' on a new line to finish grammar input:
S -> aBa
B -> bB | ε
break
Grammar input completed. Enter a string: aba
Parsing steps:
S -> aB -> bB -> e
b
a
Parsing successful

--------------------------------
Process exited after 101.9 seconds with return value 0
Press any key to continue . . .
```

Lab Assignment: Implement Predictive Parser using C for the Expression Grammar

E → TE'

E'→ +TE' | ε

T → FT'

T'→ *FT' | ε

F → (E) | d

```
// Week 8
// Implement Predictive Parser using C for the Expression
Grammar
```

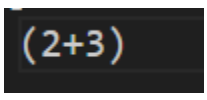
```c
// E  -> TE'
// E' -> +TE' | ε
// T  -> FT'
// T' -> *FT' | ε
// F  -> (E) | d

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

char input[100];
int i = 0;

bool E();
bool E_();
bool T();
bool T_();
bool F();

bool E()
{
    if (T())
    {
        if (E_())
        {
            return true;
        }
    }
    return false;
}

bool E_()
{
```

```
    if (input[i] == '+')
    {
        i++;
        if (T())
        {
            if (E_())
            {
                return true;
            }
        }
    }
    else
    {
        return true;
    }
    return false;
}

bool T()
{
    if (F())
    {
        if (T_())
        {
            return true;
        }
    }
    return false;
}

bool T_()
{
    if (input[i] == '*')
    {
```

```
        i++;
        if (F())
        {
            if (T_())
            {
                return true;
            }
        }
    }
    else
    {
        return true;
    }
    return false;
}

bool F()
{
    if (input[i] == '(')
    {
        i++;
        if (E())
        {
            if (input[i] == ')')
            {
                i++;
                return true;
            }
        }
    }
    else if (isdigit(input[i]))
    {
        i++;
        return true;
```

```
    }
    return false;
}

int main()
{
    FILE *fp1,*fp2;
    fp1 = fopen("input.txt","r");
    fp2 = fopen("output.txt","w");
    fscanf(fp1, "%s", input);
    if (E())
    {
        if (input[i] == '\0')
        {
            fprintf(fp2,"String is accepted\n");
        }
        else
        {
            fprintf(fp2,"String is not accepted\n");
        }
    }
    else
    {
        fprintf(fp2,"String is not accepted\n");
    }
    return 0;
}
```

Outputs -
Input file



Output file

```
String is accepted
```

9. Week 9 - Shift Reduce Parser

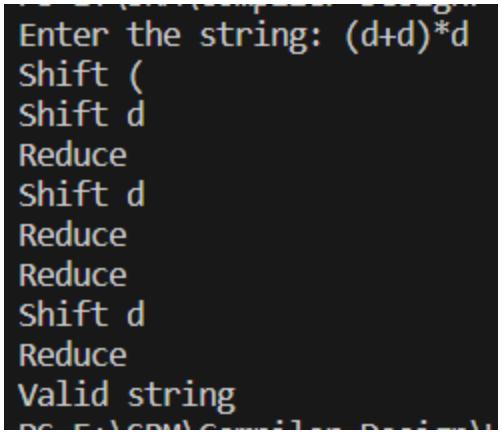
Implementation of Shift Reduce parser using C for the following grammar and illustrate the parser's actions for a valid and an invalid string.

E→E+E
E→E*E
E→(E)
E→d

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

void shift(char *input, int *ptr) {
    printf("Shift %c\n", input[*ptr]);
    (*ptr)++;
}

void reduce() {
    printf("Reduce\n");
}

void parse(char *input) {
    int ptr = 0;

    shift(input, &ptr);

    while (input[ptr] != '\0') {
        if (input[ptr] == 'd') {
            shift(input, &ptr);
```

```
        } else if (input[ptr] == '+' || input[ptr] == '*') {
            reduce();
            ptr++;
        } else if (input[ptr] == '(') {
            shift(input, &ptr);
        } else if (input[ptr] == ')') {
            reduce();
            ptr++;
        } else {
            printf("Invalid string\n");
            exit(0);
        }
    }

    reduce();

    printf("Valid string\n");
}

int main() {
    char str[100];
    printf("Enter the string: ");
    scanf("%s", str);

    parse(str);

    return 0;
}
```

Outputs -



Lab Assignment: Implementation of Shift Reduce parser using C for the following grammar and illustrate the parser's actions for a valid and an invalid string.
S –> 0S0 | 1S1 | 2

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

bool match(char expected, char input);
bool S(char input[], int *index);

int main() {
    char input[50];
    printf("Enter the input string: ");
    scanf("%s", input);

    int index = 0;
    if (S(input, &index) && input[index] == '\0') {
        printf("Accepted\n");
    } else {
        printf("Not Accepted\n");
    }
```

```
    return 0;
}

bool match(char expected, char input) {
    return expected == input;
}

bool S(char input[], int *index) {
    if (input[*index] == '0') {
        (*index)++;
        if (S(input, index) && input[*index] == '0') {
            (*index)++;
            return true;
        }
    } else if (input[*index] == '1') {
        (*index)++;
        if (S(input, index) && input[*index] == '1') {
            (*index)++;
            return true;
        }
    } else if (input[*index] == '2') {
        (*index)++;
        return true;
    }

    return false;
}
```

Outputs -