Department of CSE
SSN College of Engineering

Vishakan Subramanian - 18 5001 196 - Semester VI

22 January 2021

UCS 1602 - Compiler Design

Exercise 1: Lexical Analyser Using C

Aim:

To write a program using C to perform the basic functionalities of a **Lexical Analyser**.

Code:

```c
/*  C Program that performs a basic lexical analysis of a given string  */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <fcntl.h>

int isOperator(char ch);
int isSeparator(char ch);
int isDelimiter(char ch);
int isValidIdentifier(char *str);
int isInteger(char *str);
int isKeyword(char *str);
int isPreprocessorDirective(char ch);
char *subString(char *str, int start, int end);
int printOperator(char ch1, char ch2);
int lexicalParse(char *str);

int main(void){
    int status = 0, len, fp;
    char text[10000], file[100];

    printf("\n\t\t\tLexical Analyser Using C\n");
    printf("\n\t\tEnter file name to parse: ");
    scanf("%[^\n]", file);

    fp = open(file, O_RDONLY);

    if(fp < 0){
        printf("\nError: File does not exist.\n");
        return 0;
    }

    len = read(fp, text, 10000);
    close(fp);

    printf("\nText to be parsed:\n\n%s\n", text);

    status = lexicalParse(text);

    if(status){
        printf("\n\n\t\tThe given expression is lexically valid.\n");
    }

    else{
```

```c
        printf("\n\n\t\tThe given expression is lexically invalid.\n");
    }

    return 0;
}

int isOperator(char ch){
    //Checks if the character is a valid operator

    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=' || ch == '%' || ch == '!' ){
            return 1;
        }

    return 0;
}

int isSeparator(char ch){
    //Checks if the character is a valid separator

    if (ch == ';'|| ch == '{' || ch == '}' || ch == ','){
            return 1;
        }

    return 0;
}

int isDelimiter(char ch){
    //Checks if the character is a valid delimiter

    if (ch == ' ' || ch == '(' || ch == ')'
        || isSeparator(ch) == 1 || isOperator(ch) == 1){
            return 1;
        }

    return 0;
}

int isValidIdentifier(char *str){
    //Checks if the character is a valid identifier

    if(isdigit(str[0]) > 0 || isDelimiter(str[0]) == 1){
        //First character shouldn't be a digit or a special character
        return 0;
    }

    return 1;
}

int isInteger(char *str){
```

```c
    //Checks if the string is a valid integer

    int i = 0, len = strlen(str);

    if(!len){
        return 0;
    }

    for(i = 0; i < len; i++){
        if(!isdigit(str[i])){
            return 0;
        }
    }

    return 1;
}

int isKeyword(char *str){
    //Checks if the string is a valid keyword

    if(!strcmp(str, "if") || !strcmp(str, "else") || !strcmp(str, "while") ||
        !strcmp(str, "for") || !strcmp(str, "do") || !strcmp(str, "break") ||
        !strcmp(str, "switch") || !strcmp(str, "continue") || !strcmp(str, "return") ||
        !strcmp(str, "case") || !strcmp(str, "default") || !strcmp(str, "void") ||
        !strcmp(str, "int") || !strcmp(str, "char") || !strcmp(str, "bool") ||
        !strcmp(str, "struct") || !strcmp(str, "goto") || !strcmp(str, "typedef") ||
        !strcmp(str, "unsigned") || !strcmp(str, "long") || !strcmp(str, "short") ||
        !strcmp(str, "float") || !strcmp(str, "double") || !strcmp(str, "sizeof")){
            return 1;
        }

    return 0;
}

int isPreprocessorDirective(char ch){
    //Checks if the string is a valid preprocessor directive

    if(ch == '#'){
        //Basic check, works for header files, macros and const declarations
        return 1;
    }
    return 0;
```

```
}

char *subString(char *str, int start, int end){
    //Get a substring from the given string
    int i = 0;
    char *sub = (char *)malloc(sizeof(char) * (end - start + 2));

    for(i = start; i <= end; i++){
        sub[i - start] = str[i];
    }

    sub[end - start + 1] = '\0';

    return sub;
}

int printOperator(char ch1, char ch2){
    //Print the details of the parsed operator

    switch(ch1){
        case '+':
            if(ch2 == '='){
                printf("ASSIGN ");
            }
            else if(ch2 == ' '){
                printf("ADD ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;


        case '-':
            if(ch2 == '='){
                printf("SUB-ASSIGN ");
            }
            else if(ch2 == ' '){
                printf("SUB ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;

        case '*':
            if(ch2 == '='){
                printf("PRODUCT-ASSIGN ");
            }
```

```c
            else if(ch2 == ' '){
                printf("PRODUCT ");
            }
            else{
                printf("INVALID-OP");
                return 0;
            }
            break;

        case '/':
            if(ch2 == '='){
                printf("DIVISION-ASSIGN ");
            }
            else if(ch2 == ' '){
                printf("DIVISION ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;

        case '%':
            if(ch2 == '='){
                printf("MODULO-ASSIGN ");
            }
            else if(ch2 == ' '){
                printf("MODULO ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;

        case '=':
            if(ch2 == '='){
                printf("EQUALITY ");
            }
            else if(ch2 == ' '){
                printf("ASSIGN ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;

        case '>':
            if(ch2 == '='){
                printf("GT-EQ ");
```

```c
            }
            else if(ch2 == ' '){
                printf("GT ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;

        case '<':
            if(ch2 == '='){
                printf("LT-EQ ");
            }
            else if(ch2 == ' '){
                printf("LT ");
            }
            else{
                printf("INVALID-OP ");
                return 0;
            }
            break;

        case '!':
            printf("NOT ");
            break;

        default:
            printf("INVALID-OP ");
            return 0;
    }

    return 1;
}

int lexicalParse(char *str){
    //Parse the given string to check for validity
    int left = 0, right = 0, len = strlen(str), status = 1, i;

    printf("\nLexical Analysis:\n\t");

    while(right <= len && left <= right){
        //While we are within the valid bounds of the string, check:

        while(isPreprocessorDirective(str[right]) == 1){
                //Check if string is preprocessor directive
                printf("PPDIR ");

                for(right; str[right] != '\n'; right++);
                right++;
                left = right;
```

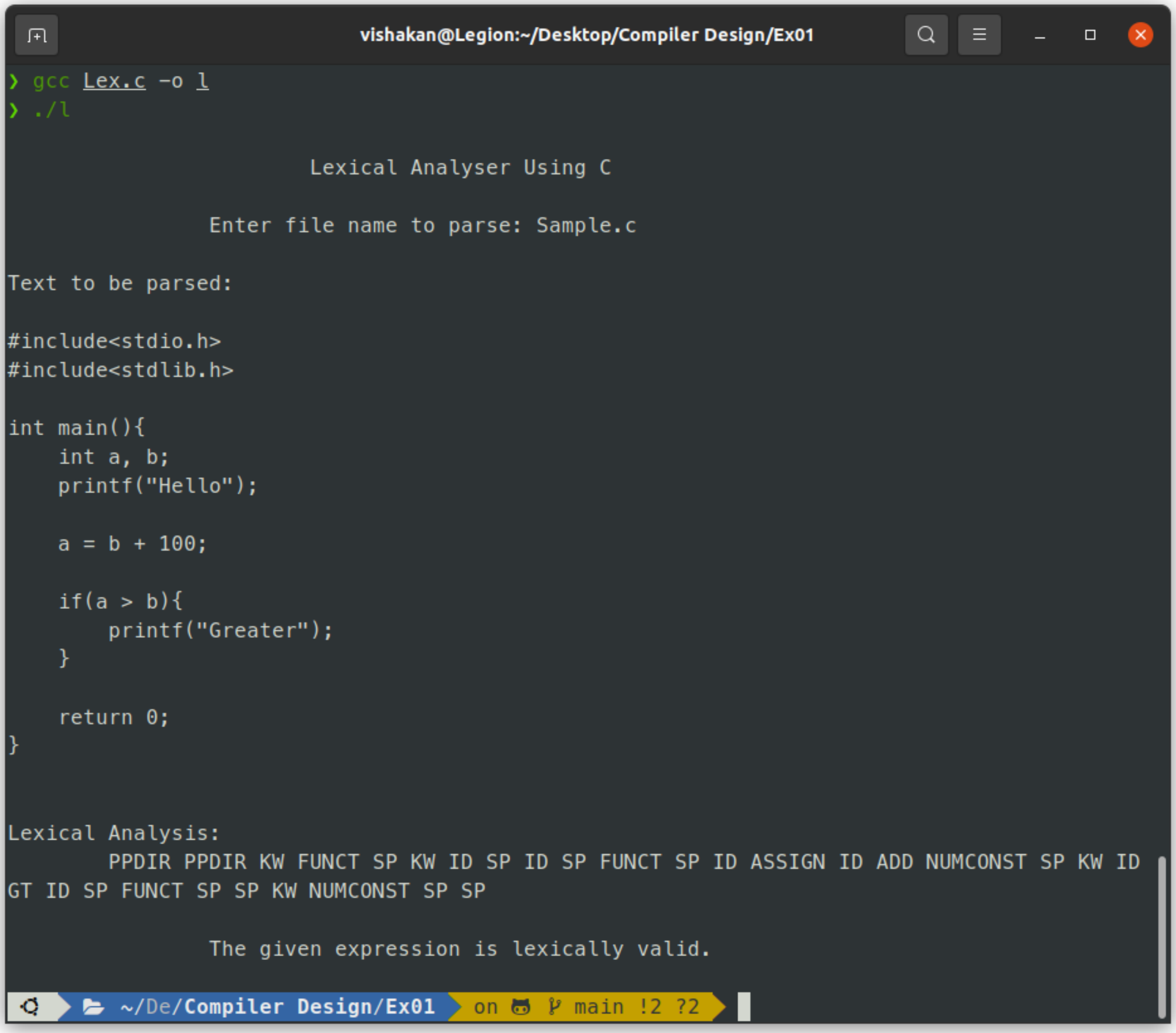
```
    }

    for(i = right; i < len; i++){
        //Clearing linebreaks & tabs to spaces for efficient processing
        if(str[i] == '\n' || str[i] == '\t'){
            str[i] = ' ';
        }
    }

    if(isDelimiter(str[right]) == 0){
        //If we do not encounter a delimiter, keep moving forward
        //"right" points to the next character
        right++;
    }

    else if(isDelimiter(str[right]) == 1 && left == right){
        //If it is a delimiter, and we haven't parsed it yet

        if(isSeparator(str[right]) == 1){
            //Check if the delimiter is a separator
            printf("SP ");
        }

        else if(isOperator(str[right]) == 1){
            //Check if the delimiter is an operator
            if((right + 1) <= len && isOperator(str[right + 1]) == 1){
                //Check if the next character is also an operator
                status = status & printOperator(str[right], str[right + 1]);
                right++;
            }

            else{
                //Next character is not an operator
                status = status & printOperator(str[right], ' ');
            }

            //printf("\n\t\t'%c' is an operator.", str[right]);
        }

        right++;
        left = right;
    }

    else if(str[right] == '(' && left != right || (right == len && left != right)){
        //Special case, to check for functions

        char *sub = subString(str, left, right - 1);

```

```
            if(isKeyword(sub) == 1){
                //Check if the function is a keyword based function, like "if" & "for"
                printf("KW ");
                left = right;
                continue;   //Go ahead with the next check
            }

            //Otherwise, its some other function, parse it.

            for(i = right + 1; i < len; i++){
                if(str[i] == ')'){
                    //Finish parsing till the end of the block and break
                    printf("FUNCT ");
                    right = i + 1;
                    left = right;
                    status = status & 1;
                    break;
                }
            }
        }

        else if(isDelimiter(str[right]) == 1 && left != right || (right == len && left != right)){
            //We encountered a delimiter in the "right" position, but left != right
            //thus a chunk of unparsed characters exist between left and right

            //Make a substring of the unparsed characters
            char *sub = subString(str, left, right - 1);

            if(isInteger(sub) == 1){
                //Check if substring is an integer
                printf("NUMCONST ");
            }
            else if(isKeyword(sub) == 1){
                //Check if substring is a keyword
                printf("KW ");
            }
            else if(isValidIdentifier(sub) == 1){
                //Check if substring is a valid identifier
                printf("ID ");
            }
            else if(isValidIdentifier(sub) == 0 && isDelimiter(str[right - 1]) == 0){
                //Otherwise, print that it is not a valid identifier
                status = status & 0;
                printf("INVALID-ID");
            }

```

```
            left = right;   //We have parsed the chunk, thus "left" = "right"
        }

    }

    return status;
}
```

Output - Valid Case:

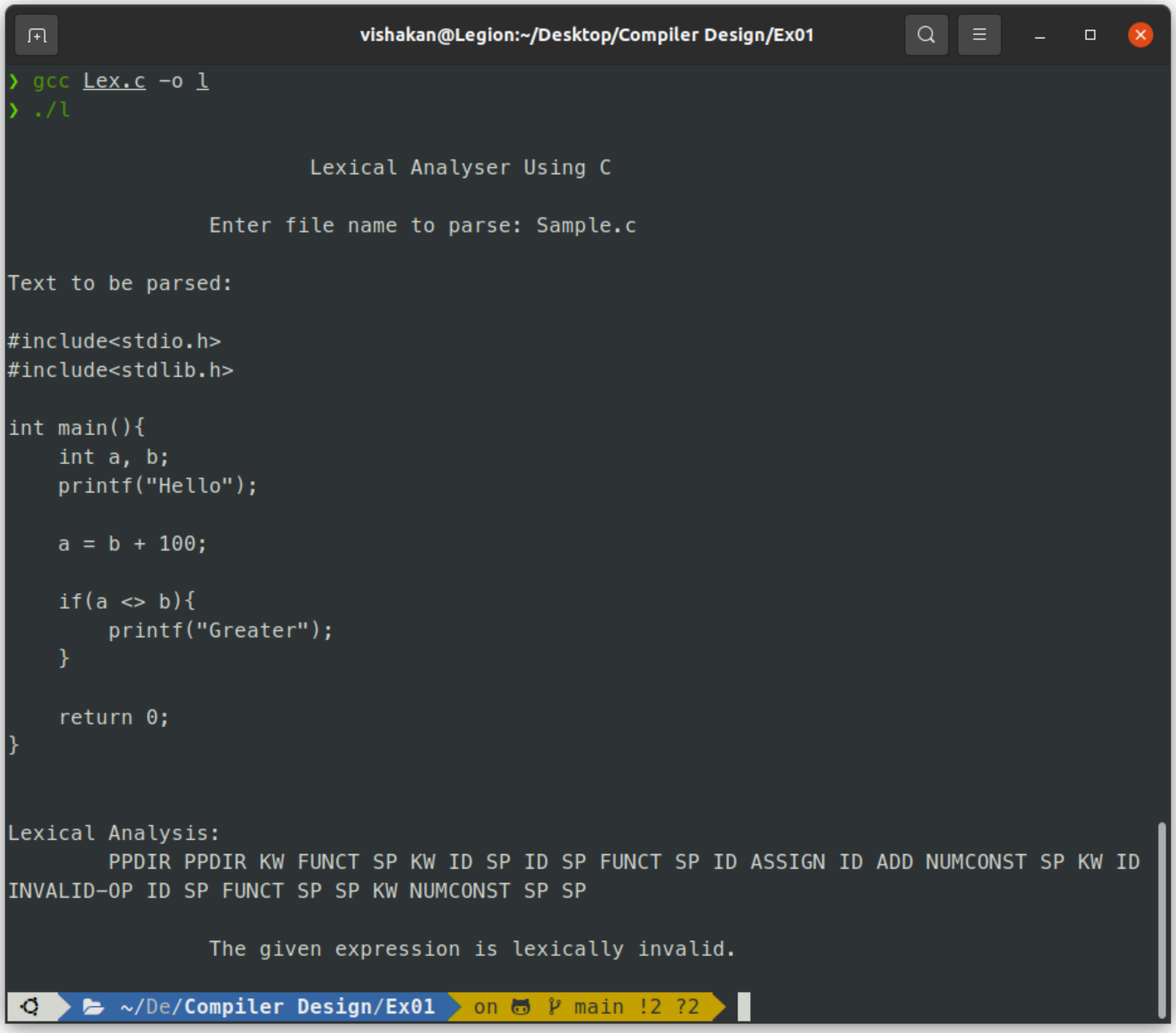
Figure 1: Console Output for a Valid Program.



```
vishakan@Legion:~/Desktop/Compiler Design/Ex01
❯ gcc Lex.c -o l
❯ ./l

                        Lexical Analyser Using C

                Enter file name to parse: Sample.c

Text to be parsed:

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

int main(){
    int a, b;
    printf("Hello");

    a = b + 100;

    if(a > b){
        printf("Greater");
    }

    return 0;
}


Lexical Analysis:
        PPDIR PPDIR KW FUNCT SP KW ID SP ID SP FUNCT SP ID ASSIGN ID ADD NUMCONST SP KW ID
GT ID SP FUNCT SP SP KW NUMCONST SP SP

                The given expression is lexically valid.

 ~/De/Compiler Design/Ex01  on  main !2 ?2
```

Output - Invalid Case:

Figure 2: Console Output for an Invalid Program.



```
vishakan@Legion:~/Desktop/Compiler Design/Ex01
❯ gcc Lex.c -o l
❯ ./l

                        Lexical Analyser Using C

                Enter file name to parse: Sample.c

Text to be parsed:

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

int main(){
    int a, b;
    printf("Hello");

    a = b + 100;

    if(a <> b){
        printf("Greater");
    }

    return 0;
}


Lexical Analysis:
        PPDIR PPDIR KW FUNCT SP KW ID SP ID SP FUNCT SP ID ASSIGN ID ADD NUMCONST SP KW ID
INVALID-OP ID SP FUNCT SP SP KW NUMCONST SP SP

                The given expression is lexically invalid.

 ~/De/Compiler Design/Ex01  on  main !2 ?2
```

Learning Outcome:

- From the experiment, I understood how a basic **Lexical Analyser** works.
- I was able to formulate ideas on how to implement recognition of specific tokens in programs for identification by the Lexical Analyser.
- I was able to implement simple regular expressions in C.
- I learnt how to parse a program for lexical validity, utilising the concept of **lexemes**.
- I was able to visualize the complexity that goes behind the compilation process and the significance of a Lexical Analyser phase in the compilation flow.

Department of CSE
SSN College of Engineering

Vishakan Subramanian - 18 5001 196 - Semester VI

12 February 2021

UCS 1602 - Compiler Design

Exercise 2: Lexical Analyser Using Lex Tool

Aim:

To write a program using Lex to perform the basic functionalities of a **Lexical Analyser**, and to form a symbol table on the parsed program.

Code:

```
/* Lexical Analyser Using Lex Tool */

/*Definitions*/

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

struct symbol{
    char type[10];
    char name[20];
    char value[100];
};  //For Symbol Table

typedef struct symbol sym;

sym sym_table[1000];
int cur_size = -1;
char current_type[10];
%}

number_const [-+]?[0-9]+(\.[0-9]+)?
char_const \'.\'
string_const \".*\"
identifier [a-zA-Z_][a-zA-Z0-9_]*
function [a-zA-Z_][a-zA-Z0-9]*[(].*[)]
keyword (int|float|char|unsigned|typedef|struct|return|continue|break|if|else|for|while|do|extern|auto|case|switch|enum|goto|long|double|sizeof|void|default|register)
pp_dir ^[#].*[>]$
rel_ops ("<"|">"|"<="|">="|"=="|"!=")
assign_ops ("="|"+="|"-="|"%="|"/="|"*=")
arith_ops ("+"|"-"|"%"|"/"|"*")
single_cmt [/][/].*
multi_cmt ([/][/].*)|([/][*](.|[\n\r])*[*][/])
spl_chars [{}(),;\[\]]

/*Rules*/

%%

{pp_dir} {
    printf("PPDIR ");
    strcpy(current_type, "INVALID");
}

```

```
{keyword} {
    printf("KW ");

    if(strcmp(yytext, "int") == 0){
        strcpy(current_type, "int");
    }
    else if(strcmp(yytext, "float") == 0){
        strcpy(current_type, "float");
    }
    else if(strcmp(yytext, "double") == 0){
        strcpy(current_type, "double");
    }
    else if(strcmp(yytext, "char") == 0){
        strcpy(current_type, "char");
    }
    else{
        strcpy(current_type, "INVALID");
    }
}

{function} {
    printf("FUNCT ");
}

{identifier} {
    printf("ID ");

    if(strcmp(current_type, "INVALID") != 0){
        cur_size++;
        strcpy(sym_table[cur_size].name, yytext);
        strcpy(sym_table[cur_size].type, current_type);

        if(strcmp(current_type, "char") == 0){
            strcpy(sym_table[cur_size].value, "NULL");
        }
        else if(strcmp(current_type, "int") == 0){
            strcpy(sym_table[cur_size].value, "0");
        }
        else{
            strcpy(sym_table[cur_size].value, "0.0");
        }
    }
}

{single_cmt} {
    printf("SCMT ");
}

{multi_cmt} {
    printf("MCMT ");
}
```

```

{number_const} {
    printf("NUM_CONST ");

    if(strcmp(current_type, "INVALID") != 0){
        strcpy(sym_table[cur_size].value, yytext);
    }
}

{char_const} {
    printf("CHAR_CONST ");

    if(strcmp(current_type, "char") == 0){
        strcpy(sym_table[cur_size].value, yytext);
    }
}

{string_const} {
    printf("STR_CONST ");
}

{rel_ops} {
    printf("REL_OP ");
}

{arith_ops} {
    printf("ARITH_OP ");
}

{assign_ops} {
    printf("ASSIGN_OP ");
}

{spl_chars} {
    if(strcmp(yytext, ";") == 0){
        strcpy(current_type, "INVALID");
    }

 }

\n {
    printf("\n");
}

[ \t] { }


%%

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

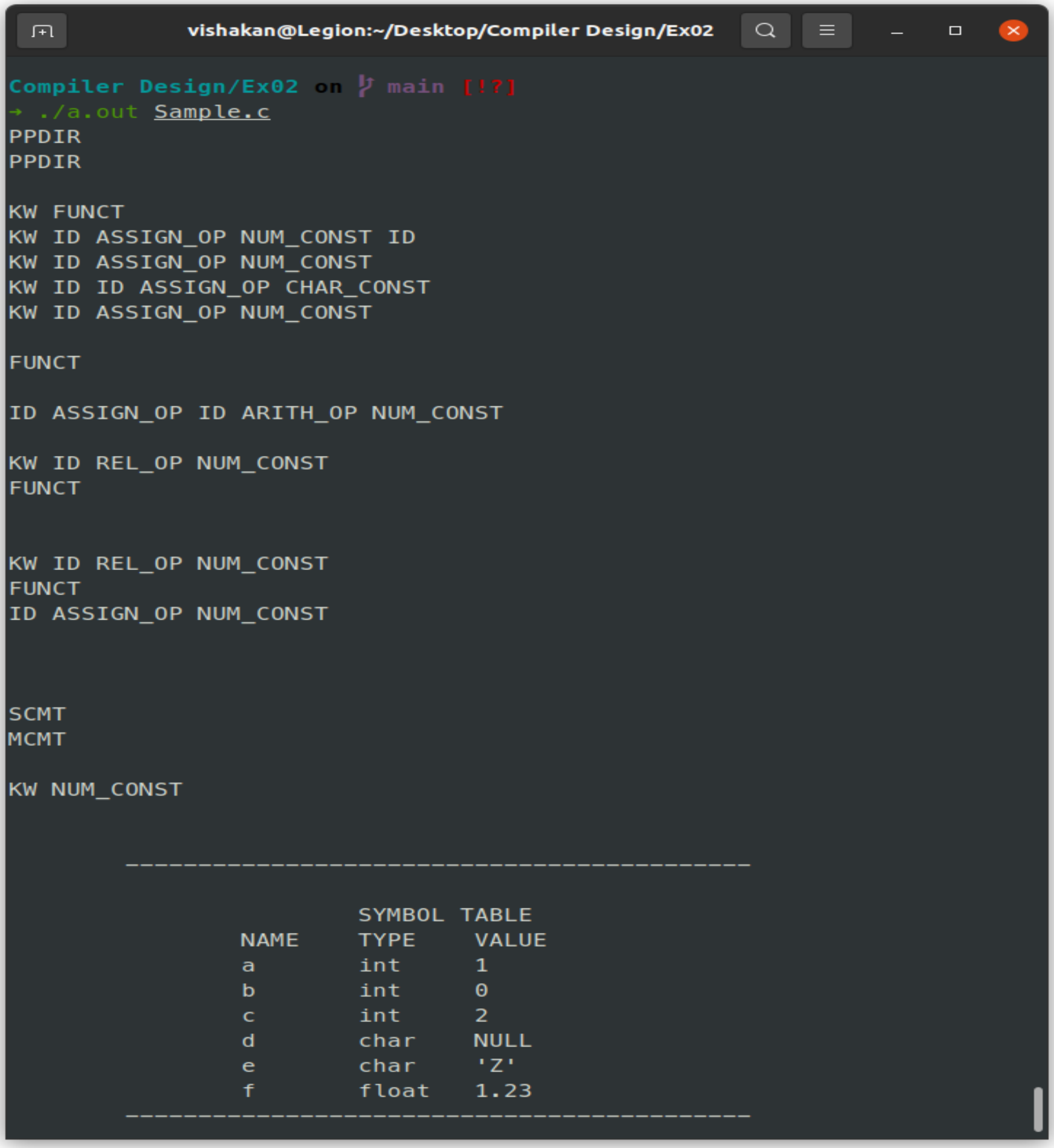
```
}


/*User Subroutines*/

int main(int argc, char *argv[]){
    int i = 0;

    yyin = fopen(argv[1], "r");
    yylex();

    printf("\n\t------------------------------------------\n");

    printf("\n\t\t\tSYMBOL TABLE");
    printf("\n\t\tNAME\tTYPE\tVALUE\n");
    for(i = 0; i <= cur_size; i++){
        printf("\t\t%s\t%s\t%s\n", sym_table[i].name, sym_table[i].type, sym_table[i].value);
    }

    printf("\t-------------------------------------------\n");

    return 0;
}
```

Parsed C Code:

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

int main(){
    int a = 1, b;
    int c = 2;
    char d, e = 'Z';
    float f = 1.23;

    printf("Hello to %d", c);

    a = b + 100;

    if (c > 100){
        printf("Greater");
    }

    while (c > 0) {
        printf("Hello to Lex!");
        c -= 1;
    }


    //a is GREATER than b!
    /* Multi-line
    comment */

    return 0;
}
```

Output:

Figure 1: Console Output



Learning Outcome:

- From the experiment, I understood the basics of Lex tool.
- I was able to implement recognition for regular expressions using Lex terminology.
- I understood the working of a Lex program.
- I learnt about the three sections of a Lex program, namely, definitions, rules and user subroutines.
- I learnt to implement a basic symbol table using Lex on the parsed C program.
- I understood that Lex tool is more powerful and easy-to-use for Lexical Analysis task compared to conventional C programming.

Department of CSE
SSN College of Engineering

Vishakan Subramanian - 18 5001 196 - Semester VI

20 February 2021

UCS 1602 - Compiler Design

Exercise 3: Elimination of Left Recursion Using C

Aim:

Write a program in C to find whether the given grammar is **Left Recursive** or not. If it is found to be left recursive, convert the grammar in such a way that the left recursion is removed.

Code:

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

int main(){
    /*
        Sample Input Format:    E->E+T|T
                                T->T*F|F
                                F->i
    */

    char productions[100][100], sub_prods[100][100];
    char non_terminal;
    int num_prods, i, j, k, flag = 0;

    printf("\n\t\tElimination of Left Recursion\n");
    printf("\nEnter the number of Productions: ");
    scanf("%d", &num_prods);

    printf("\nEnter the Grammar:\n");

    for(i = 0; i < num_prods; i++){
        //Getting Input
        scanf("%s", productions[i]);
    }

    printf("\nGiven Grammar:\n");

    for(i = 0; i < num_prods; i++){
        //Printing the Grammar, and checking for left recursions
        printf("%s\n", productions[i]);

        if(productions[i][0] == productions[i][3]){
            flag = 1;
        }
    }

    if(flag == 0){
        //If Grammar is not left recursive, exit
        printf("\nGrammar is not Left Recursive.");
        return 0;
    }

    //Otherwise, Grammar is left recursive, parse and remove it
    printf("\nGrammar is Left Recursive.");
    printf("\n\nGrammar after removal of Left Recursion:");

```

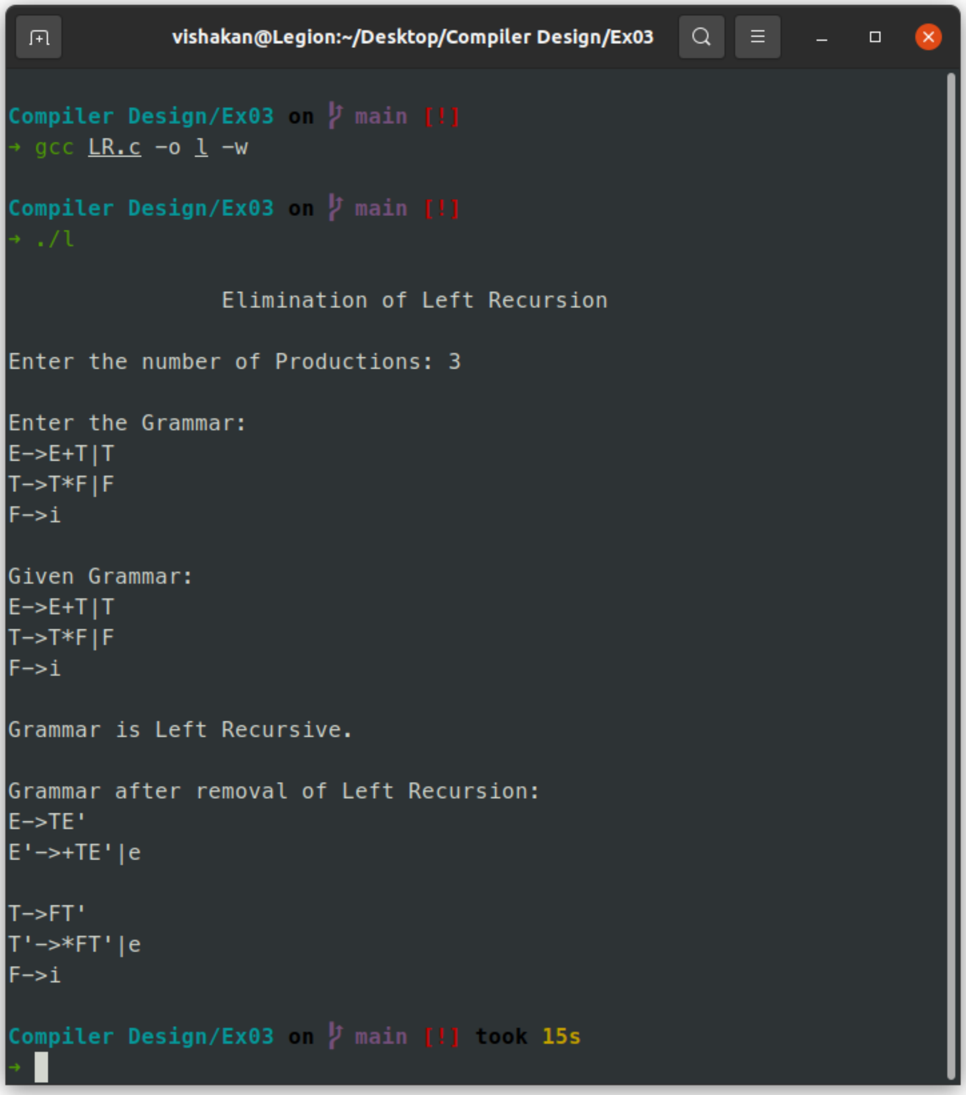
```
    for(i = 0; i < num_prods; i++){
        //Parse each production one by one
        non_terminal = productions[i][0];

        char *split, production[100];
        flag = 0;

        //Store the RHS of the production alone
        for(j = 0; productions[i][j + 3] != '\0'; j++){
            production[j] = productions[i][j + 3];
        }

        production[j] = '\0';
        j = 0;

        //Split at the sub-expression level when there is an OR operator
        split = strtok(production, "|");

        while(split != NULL){
            //Store the subexpression in a new productions array
            strcpy(sub_prods[j], split);

            if(split[0] == non_terminal && flag == 0){
                //Seeing an immediate left recursion, with no other
 productions
                //for the same non-terminal
                //This type of Left Recursion cannot be removed
                flag = 1;
            }
            else if(split[0] != non_terminal && flag == 1){
                //Already seen a left recursion, but now we have seen
                //another production with some terminal symbol
                //for the same non-terminal
                flag = 2;
            }

            j++;
            split = strtok(NULL, "|");
            //split and loop till all productions are parsed
        }

        if(flag != 2){
            //flag == 0 => no LR
            //flag == 1 => LR of the form A->Ab which cannot be removed
            printf("%s\n", productions[i]);
        }

        if(flag == 2){
            //Remove the left recursion if there's another production with
  terminal symbol
            printf("\n");
```

```
        flag = 0;

        for(k = 0; k < j; k++){
            if(sub_prods[k][0] != non_terminal){
                //Loop until the non-terminal causing the LR is not found, for 1st production rule
                if(flag != 0){
                    //Removed the LR by starting with the other non-terminal/ID,
                    //thus add the remaining sub-productions
                    printf("|%s%c\'", sub_prods[k], non_terminal);
                }
                else{
                    //No left recursion with that particular sub-production
                    //thus make it as a new production with a new non-terminal
                    flag = 1;
                    printf("%c->%s%c\'", non_terminal, sub_prods[k], non_terminal);
                }
            }
        }
        printf("\n");
        flag = 0;

        for(k = 0; k < j; k++){
            if(sub_prods[k][0] == non_terminal){
                //Loop until the non-terminal causing the LR is found, for 2nd production rule
                if(flag != 0){
                    //Add the remaining sub-productions, since the LR has been removed
                    printf("|%s%c\'", sub_prods[k] + 1, non_terminal);
                }
                else{
                    //k sub-production contains the LR causing term, thus first print the
                    //next sub-production followed by a new non-terminal as a new production
                    //2D Array Manipulation, sub_prods[k] + 1 essentially prints
                    //the string sub_prods[k][1] till sub_prods[k][n]
                    flag = 1;
                    printf("%c\'->%s%c\'", non_terminal, sub_prods[k] + 1, non_terminal);
                }
            }
        }
        printf("|e\n");
    }
```

```
    }


    return 0;
}

/*
OUTPUT:

gcc LR.c -o l -w
./l

        Elimination of Left Recursion

Enter the number of Productions: 3

Enter the Grammar:
E->E+T|T
T->T*F|F
F->i

Given Grammar:
E->E+T|T
T->T*F|F
F->i

Grammar is Left Recursive.

Grammar after removal of Left Recursion:
E->TE'
E'->+TE'|e

T->FT'
T'->*FT'|e
F->i

*/
```

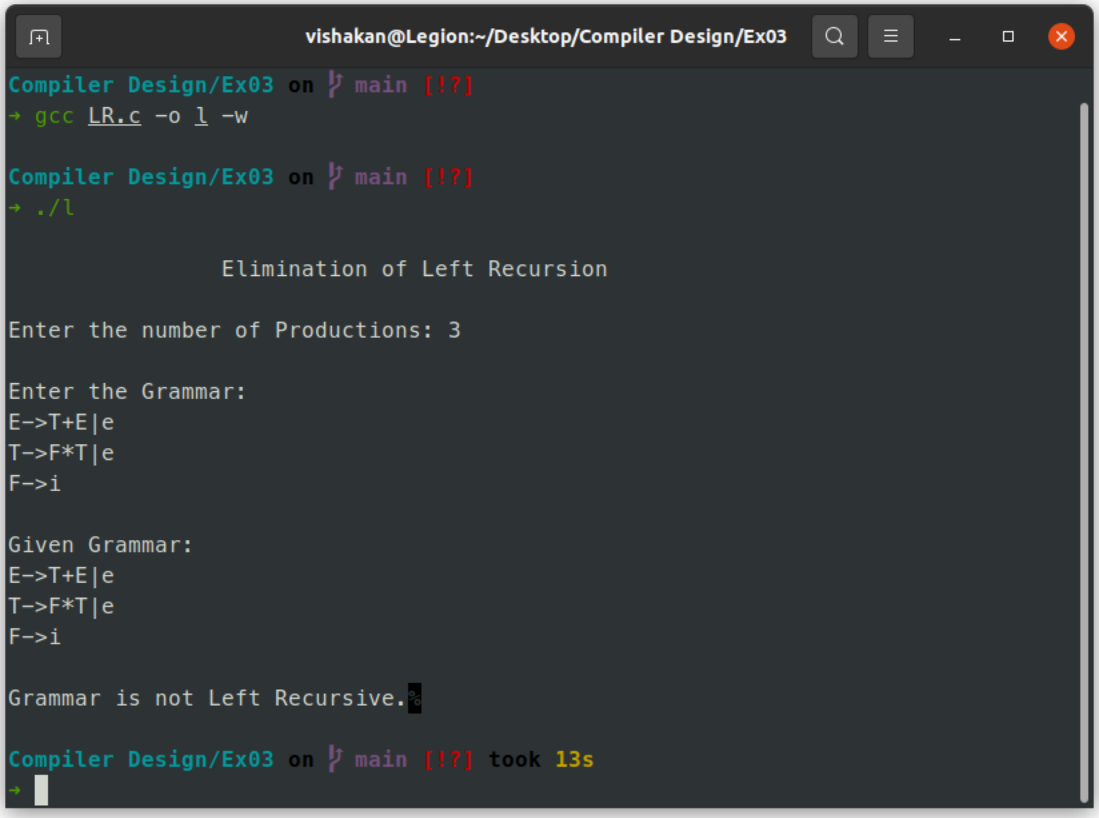
Output - Left Recursive Grammar:

Figure 1: Console Output for a Left Recursive Grammar.



Output - Non Left Recursive Grammar:

Figure 2: Console Output for a Non Left Recursive Grammar.



Learning Outcome:

- I understood about left recursive grammars.
- I understood the need for this type of conversion, as top-down parsers cannot handle left recursive grammars.
- I was able to perform a check of whether or not a grammar is left recursive using C.
- I implemented a conversion in C which converts left recursive grammar to non left recursive grammar.
- I refreshed my 2D-char array manipulation concepts in C.

Department of CSE
SSN College of Engineering

Vishakan Subramanian - 18 5001 196 - Semester VI

26 February 2021

UCS 1602 - Compiler Design

Exercise 4: Recursive Descent Parser Using C

Aim:

Write a program in C to construct **Recursive Descent Parser** for the following grammar which is for arithmetic expression involving + and *. Check the Grammar for left recursion and convert into suitable for this parser. Write recursive functions for every non-terminal. Call the function for start symbol of the Grammar in main().

G1:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow i$

Extend this parser to include division, subtraction and parenthesis operators.

G2:

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid i$

Code - Grammar 1:

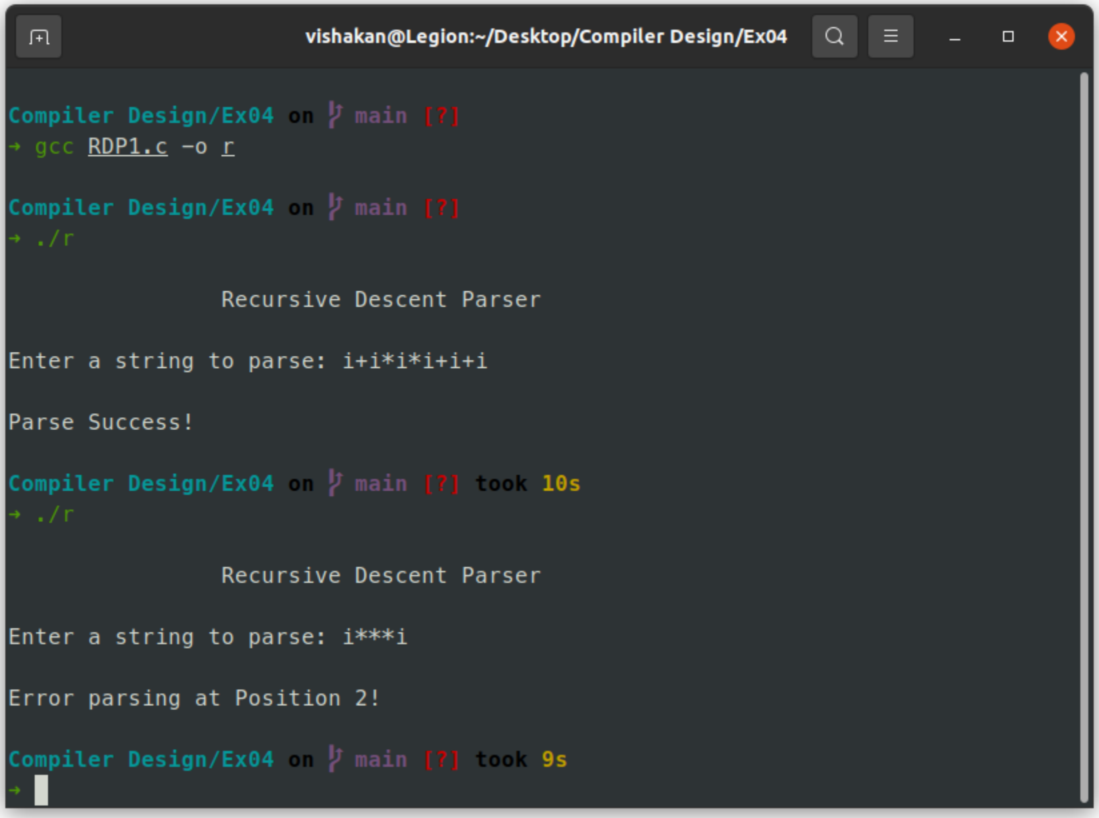
```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

/*Recursive Descent Parser*/

/*
Grammar: G: E->E+T|T
            T->T*F|F
            F->i
*/

/*
Removed Left Recursion
Grammar G': E->TE'
            E'->+TE'|e
            T->FT'
            T'->*FT'|e
            F->i
*/

struct parse_struct{
    char str[100];
    int pos;
    int len;
};

typedef struct parse_struct parser;

parser E(parser p);
parser T(parser p);
parser EPrime(parser p);
parser F(parser p);
parser TPrime(parser p);
parser parse(parser p, char s);

int main(void){
    parser p;

    printf("\n\t\tRecursive Descent Parser\n");
    printf("\nEnter a string to parse: ");
    scanf("%s", p.str);

    p.len = strlen(p.str);
    p.pos = 0;

    p = E(p);
```

```
    if(p.pos == p.len){
        //All characters have been parsed
        printf("\nParse Success!\n");
    }

    else{
        //Some characters haven't been parsed, but returned to main
        printf("\nError parsing at Position %d!\n", p.pos);
    }


    return 0;
}

parser E(parser p){
    //printf("\nAt E");
    p = T(p);
    p = EPrime(p);

    return p;
}

parser T(parser p){
    //printf("\nAt T");
    p = F(p);
    p = TPrime(p);

    return p;
}

parser EPrime(parser p){
    //printf("\nAt EPrime");
    if(p.str[p.pos] == '+'){
        p = parse(p, '+');
        p = T(p);
        p = EPrime(p);
    }

    return p;
}

parser TPrime(parser p){
    //printf("\nAt TPrime");
    if(p.str[p.pos] == '*'){
        p = parse(p, '*');
        p = F(p);
        p = TPrime(p);
    }

    return p;
```

```
}

parser F(parser p){
    //printf("\nAt F");
    if(p.str[p.pos] == 'i'){
        p = parse(p, 'i');
    }
    else{
        printf("\nError parsing at Position %d!\n", p.pos);
        exit(0);
    }

    return p;
}

parser parse(parser p, char s){
    if(p.str[p.pos] != s){
        printf("\nError parsing at Position %d!\n", p.pos);
        exit(0);
    }
    else{
        p.pos++;
    }

    return p;
}
```

Output - Grammar 1:

Figure 1: Console Output for parsed strings of Grammar 1.



Code - Grammar 2:

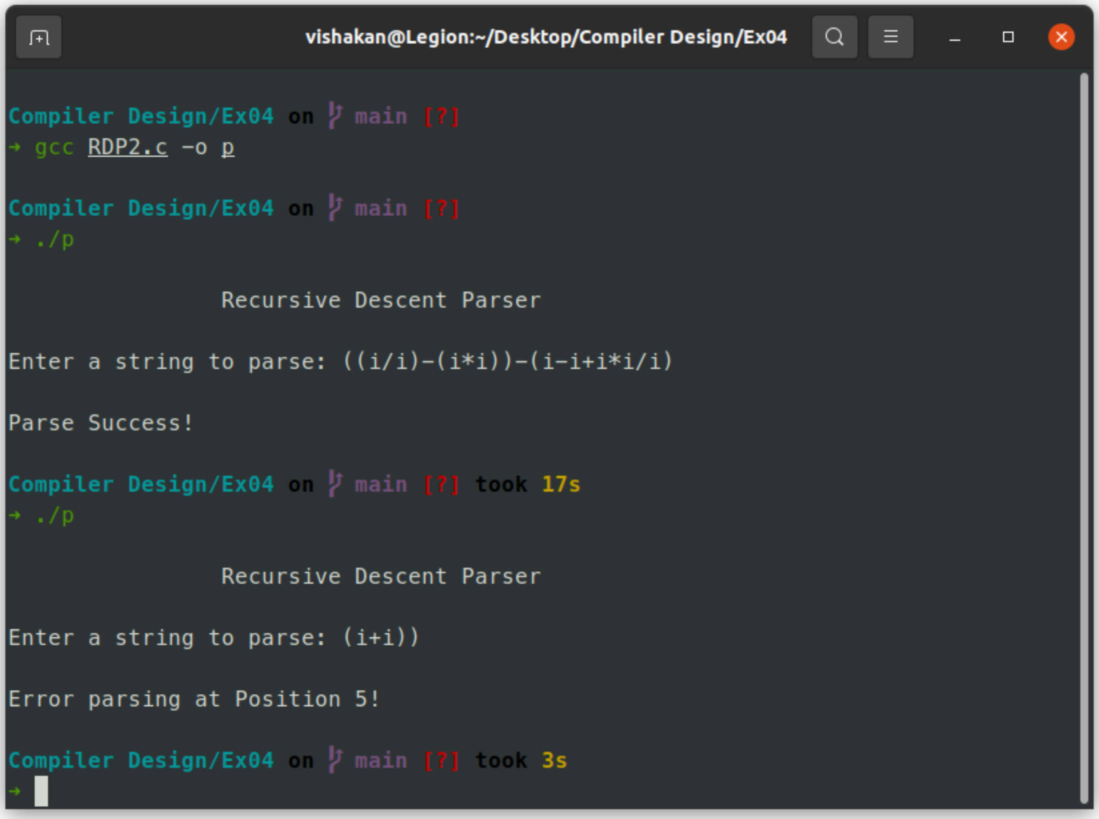
```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

/*Recursive Descent Parser*/

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

/*
Removed Left Recursion
Grammar G': E->TE'
            E'->+TE'|-TE'|e
            T->FT'
            T'->*FT'|/FT'|e
            F->(E)|i
*/

struct parse_struct{
    char str[100];
    int pos;
    int len;
};

typedef struct parse_struct parser;

parser E(parser p);
parser T(parser p);
parser EPrime(parser p);
parser F(parser p);
parser TPrime(parser p);
parser parse(parser p, char s);

int main(void){
    parser p;

    printf("\n\t\tRecursive Descent Parser\n");
    printf("\nEnter a string to parse: ");
    scanf("%s", p.str);

    p.len = strlen(p.str);
    p.pos = 0;

    p = E(p);
```

```
    if(p.pos == p.len){
        //All characters have been parsed
        printf("\nParse Success!\n");
    }

    else{
        //Some characters haven't been parsed, but returned to main
        printf("\nError parsing at Position %d!\n", p.pos);
    }

    return 0;
}

parser E(parser p){
    //printf("\nAt E");
    p = T(p);
    p = EPrime(p);

    return p;
}

parser T(parser p){
    //printf("\nAt T");
    p = F(p);
    p = TPrime(p);

    return p;
}

parser EPrime(parser p){
    //printf("\nAt EPrime");
    if(p.str[p.pos] == '+'){
        p = parse(p, '+');
        p = T(p);
        p = EPrime(p);
    }
    else if(p.str[p.pos] == '-'){
        p = parse(p, '-');
        p = T(p);
        p = EPrime(p);
    }

    return p;
}

parser TPrime(parser p){
    //printf("\nAt TPrime");
    if(p.str[p.pos] == '*'){
        p = parse(p, '*');
        p = F(p);
```

```
        p = TPrime(p);
    }
    else if(p.str[p.pos] == '/'){
        p = parse(p, '/');
        p = F(p);
        p = TPrime(p);
    }

    return p;
}

parser F(parser p){
    //printf("\nAt F");
    if(p.str[p.pos] == '('){
        p = parse(p, '(');
        p = E(p);
        p = parse(p, ')');
    }
    else if(p.str[p.pos] == 'i'){
        p = parse(p, 'i');
    }
    else{
        printf("\nError parsing at Position %d!\n", p.pos);
        exit(0);
    }

    return p;
}

parser parse(parser p, char s){
    if(p.str[p.pos] != s){
        printf("\nError parsing at Position %d!\n", p.pos);
        exit(0);
    }
    else{
        p.pos++;
    }

    return p;
}
```

Output - Grammar 2:

Figure 2: Console Output for parsed strings of Grammar 2.



Learning Outcome:

- I understood about the working of a **Recursive Descent Parser**.
- I understood that Recursive Descent Parser, being a Top-Down Parser, does not work with Left-Recursive Grammars.
- I was able to implement a working Recursive Descent Parser for a simple grammar.
- I was able to extend the concept to implement a Recursive Descent Parser for a complicated grammar with more productions.
- I refreshed my concepts with recursion & return handling in functions with C.
- I understood how to manually perform the Recursive Descent Parsing Process.

Department of CSE
SSN College of Engineering

Vishakan Subramanian - 18 5001 196 - Semester VI

7 March 2021

UCS 1602 - Compiler Design

Exercise 5: Implementation of Desk Calculator Using Yacc Tool

Aim:

Write a Lex program to recognize relevant tokens required for the **Yacc** parser to implement desk calculator. Write the Grammar for the expression involving the operators namely, $+, -, *, /,$ ˆ , $($, $)$. Precedence and associativity has to be preserved. Yacc is available as a command in Linux. The grammar should have non terminals E, OP and a terminal id.
Verify your calculator with the following inputs:

1. $3 + 9$
2. $3 + 9 * 6$
3. $(3 + 4) * 7$
4. $(3 - 4) + (7 * 6)$
5. $5/7 + 2$
6. $(4^2)^1$
7. $(2^3)^2$

Code - Yacc Parser File:

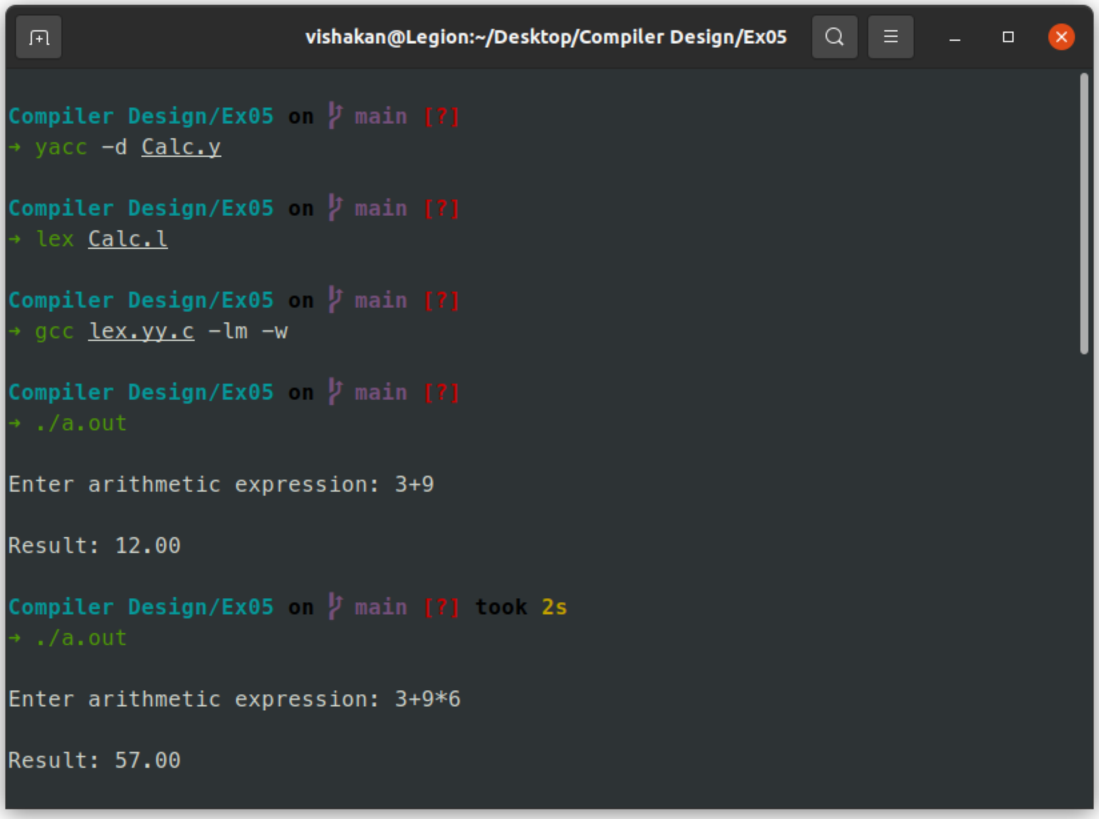
```
%{
   #include <stdio.h>
   #include <math.h>
   #define YYSTYPE double
   int flag = 0;
%}

%token NUM
/*Defining the precedence*/
%left '+' '-'
%left '/' '*'
%right '^'
%left '(' ')'

%%
Line   :  Expr             {printf("\nResult: %.2f\n", $$);}
Expr   :  Expr '+' Expr   {$$ = $1 + $3;}
       |  Expr '-' Expr   {$$ = $1 - $3;}
       |  Expr '*' Expr   {$$ = $1 * $3;}
       |  Expr '/' Expr   {$$ = $1 / $3;}
       |  Expr '^' Expr   {$$ = pow($1, $3);}
       |  '('Expr')'      {$$ = $2;}
       |  NUM             {$$ = $1;}
%%

int yyerror(){
   flag = 1;
   return 1;
}

int main(void){
   printf("\nEnter arithmetic expression: ");
   yyparse();

   if(flag){
      printf("\nEntered Unexpected Tokens.\n");
   }

   return 0;
}

/* Usage:
         Run yacc -d Calc.y
         Run lex Calc.l
         Run gcc lex.yy.c -lm -w
         Run ./a.out
*/
```

Code - Lex Grammar File:

```
%{
   #include <stdio.h>
   #include "y.tab.c"
   extern YYSTYPE yylval;
%}

%%

[0-9]+ {yylval = atoi(yytext); return NUM;}
[\t]   ;
[\n]   return 0;
.      return yytext[0];

%%

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

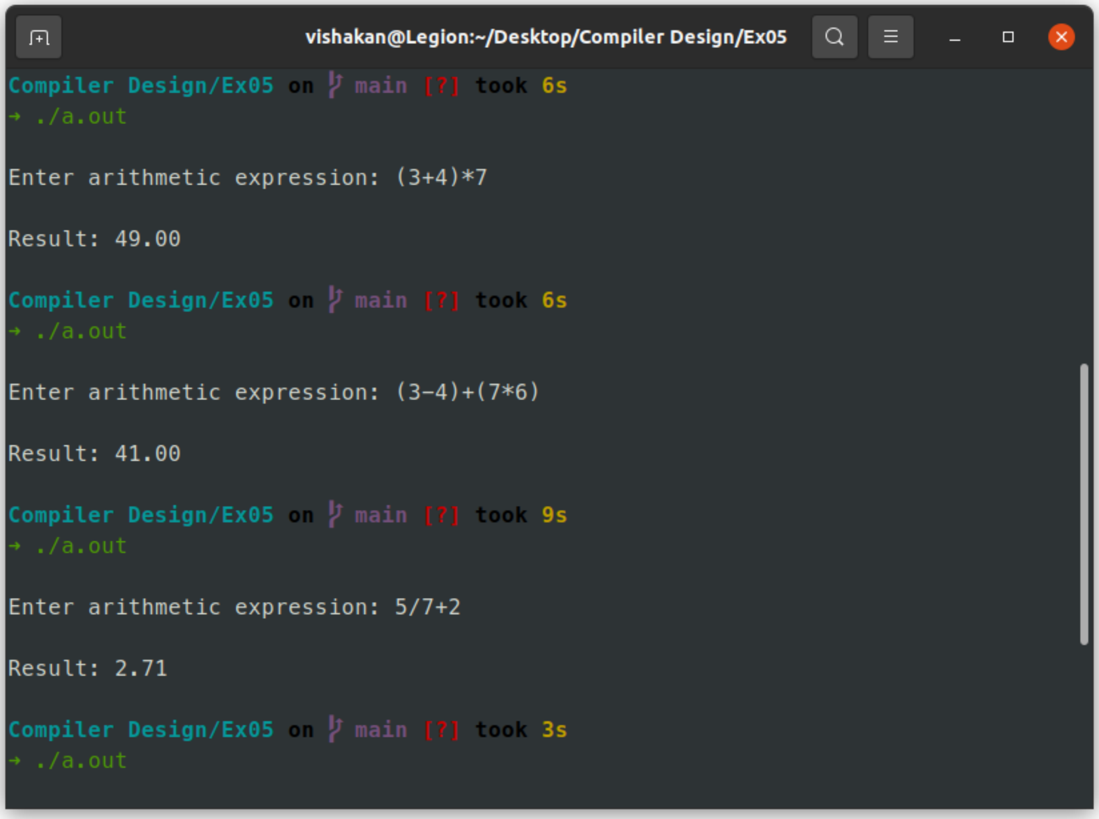
Output 1:

Figure 1: Console Output - 1.



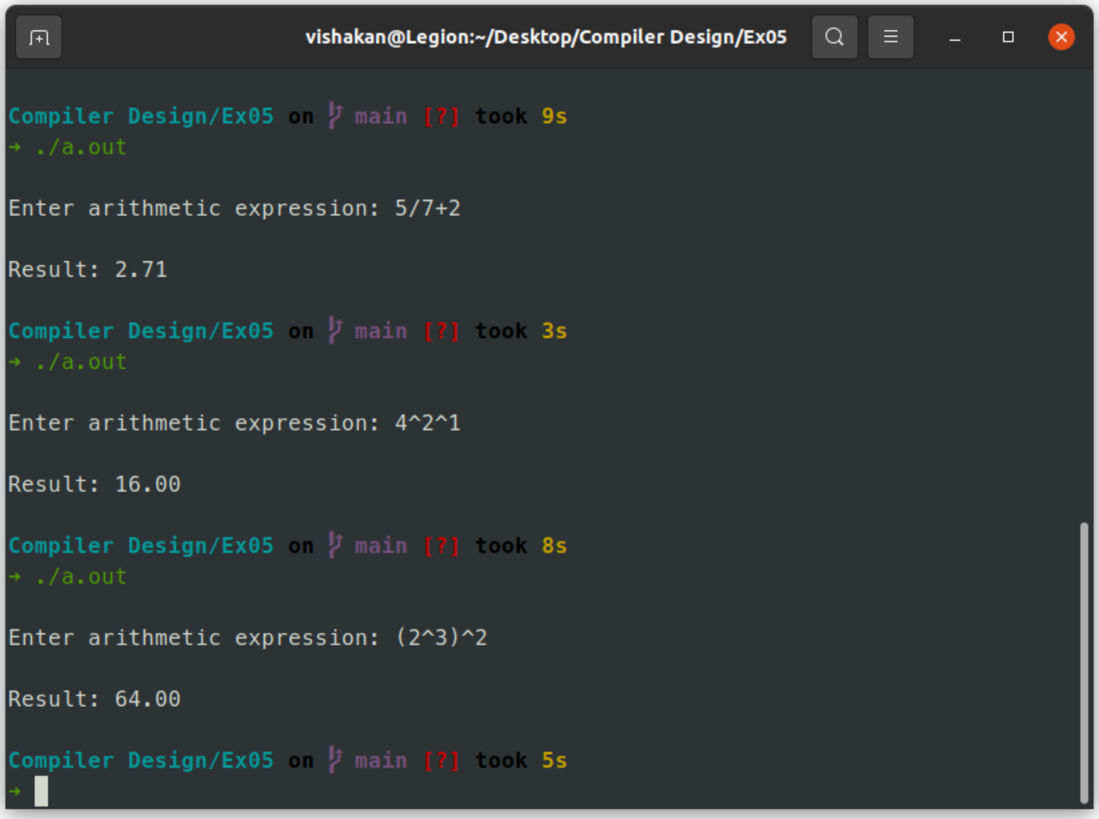
Output 2:

Figure 2: Console Output - 2.



Output 3:

Figure 3: Console Output - 3.



```
vishakan@Legion:~/Desktop/Compiler Design/Ex05

Compiler Design/Ex05 on main [?] took 9s
→ ./a.out

Enter arithmetic expression: 5/7+2

Result: 2.71

Compiler Design/Ex05 on main [?] took 3s
→ ./a.out

Enter arithmetic expression: 4^2^1

Result: 16.00

Compiler Design/Ex05 on main [?] took 8s
→ ./a.out

Enter arithmetic expression: (2^3)^2

Result: 64.00

Compiler Design/Ex05 on main [?] took 5s
→
```

Learning Outcome:

- I learnt the basic theory behind **Yacc Parser Generator**.
- I learnt that Yacc stands for Yet Another Compiler-Compiler.
- I understood that Yacc is a LALR(1) parser.
- I understood Yacc's basic syntax and programming logic.
- I learnt that Yacc needs a Lex file along with it to work as intended, to detect and give the tokens to the Yacc parser.
- I was able to visualize how the parser works with the scanner.
- I learnt how to define a simple grammar in Yacc's syntax.
- I was able to implement a parser with Yacc to mimic the features of a desk calculator with precedence logic.
- I understood how to compile and run the Yacc and Lex file together.