SSN COLLEGE OF ENGINEERING

ASSIGNMENT 1 : LEXICAL ANALYSER USING C

-SRINITHYEE S K

185001166

Aim:

To write a program in C that simulates a Lexical Analyser.

Code:

```
#include<stdio.h>
#include<string.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdlib.h>
#include<ctype.h> int
main()
{
   FILE* fp;     int count
= 0;     char* line =
NULL;     size_t len = 0;
ssize_t linelen;     char
store1[10][100];     char
store2[10][100];     fp =
fopen("./in.c", "r");     int
dtype[10], cnt = 0;
   while((linelen = getline(&line, &len, fp)) != -1)
   {
      if(line[0] == '#')
      {
         for(int i = 0; i < strlen(line); i++)
         {
            if(line[i] != '\n') printf("%c", line[i]);
         }
         printf(" - preprocessor directive\n");
      }
      char* int1 = strstr(line,"int ");
      char* float1 = strstr(line,"float
      "); char* for1 = strstr(line,"for(");
      char* if1 = strstr(line,"if("); char*
```

```
        else1 = strstr(line,"else"); int
        declare = 0; int conditional = 0;
        if(int1 != NULL)
        {
           declare = 1;
        printf("int - keyword\n");
           char* p = int1;
char str[10];            int slen = 0;
char* t = p;             int jumplen =
strlen("int ");
           t = t + 4;
while(*t != '\0')            {
char c = *t;
str[slen++] = c;
t = t + 1;               if(*t
== '=')
            {                 dtype[cnt++]
= 0;                   t = t + 1;
str[slen] = '\0';
strcpy(store1[count], str);
slen = 0;                  str[0] = '\0';
while(isdigit(*t) || *t == '.')
               {
char c = *t;
str[slen++] = c;
t = t + 1;                  }
str[slen] = '\0';
slen = 0;
               strcpy(store2[count], str);
            }
            if(*t ==',' | *t == ';')
            {
               count = count + 1;
t = t + 1;
            }
         }
      }
        if(float1 != NULL)
        {
           declare = 1;
        printf("float - keyword\n");
           char* p =
           float1; char
           str[10]; int slen
           = 0; char* t = p;
           int jumplen = strlen("float ");
           t = t + 6;
           while(*t !=
```

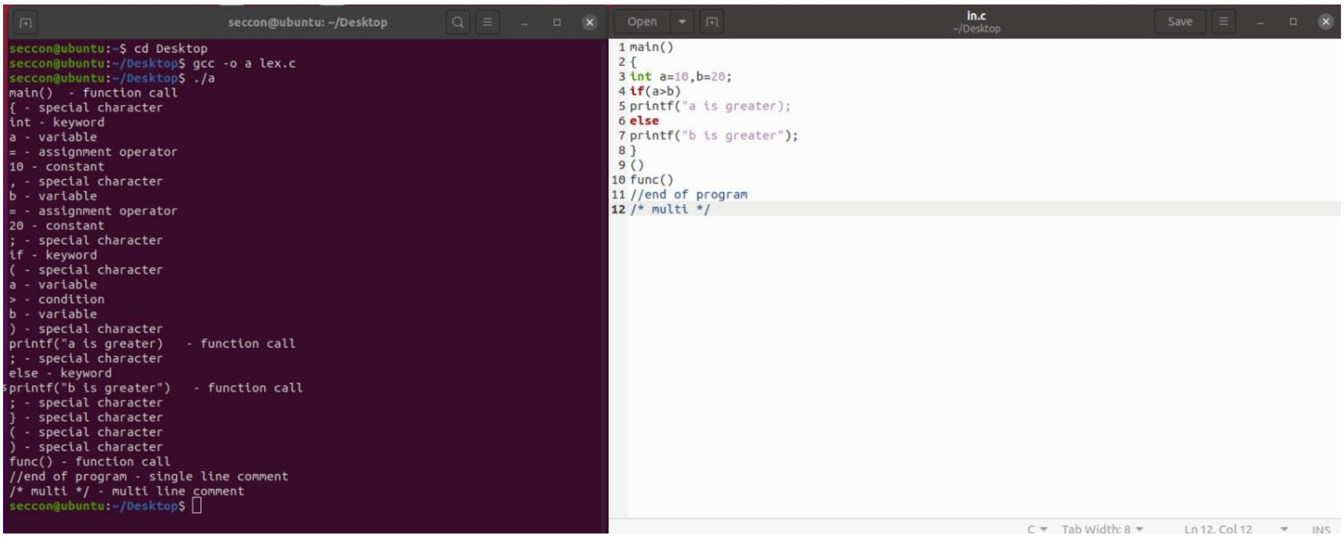
```
            '\0') {     char
            c = *t;
                str[slen++]=c;
t = t + 1;              if(*t
== '=')
                {
dtype[cnt++] = 1;                   t = t +
1;                  str[slen] = '\0';
strcpy(store1[count], str);
slen = 0;                  str[0] = '\0';
while(isdigit(*t) || *t == '.')
                    {                       char c =
*t;                         str[slen++] = c;
t = t + 1;                      }
str[slen] = '\0';                   slen = 0;
strcpy(store2[count], str);
                }
                if(*t == ',' | *t == ';')
                {
                    count = count + 1;
t = t + 1;
                }
            }
        }
        if(for1 != NULL)
printf("for - keyword\n");
        if(if1 != NULL)
        {           printf("if -
keyword\n");
            conditional = 1;
        }
        if(else1 != NULL)
printf("else - keyword\n"); char*
templine; templine = line; int first
= 1; if(declare == 1)
        {
            while(templine != NULL)
            { if(first == 1)
                {     templine = strstr(templine,"
             ");        first = 0;
                }
            else
                {                   printf(", - special
character\n");
                }
int equindex;
                for(int z = 0; z < strlen(templine); z++)
```

```
                {
                    if(*(templine + z) == '=')
                    {
equindex = z;
break;
                    }
                }
                for(int j = 1; j < equindex; j++)
                {
                    printf("%c", *(templine + j));
                }
                printf(" - variable\n");
printf("= - assignment operator\n");
templine = strstr(templine, "=");                   int
commaindex;
                for(int z = 0; z < strlen(templine); z++)
                {
                    if(*(templine + z) == ',')
                    {
                        commaindex = z;
break;
                    }
                }
                for(int j = 1; j < commaindex; j++)
                {
                    printf("%c", *(templine + j));
                }
                printf(" - constant\n");
                templine = strstr(templine, ",");
            }
        }
        char* main1 = strstr(line, "main("); char*
        printf1 = strstr(line, "printf(");
        if(main1 != NULL || printf1 != NULL)
        {     for(int i = 0; i < strlen(line);
        i++)
            {     if(line[i]=='\t' || line[i]==';' || line[i] ==
            '\n')
                {     printf("
                ");
            }
                else
                {         printf("%c",
            line[i]);
                }
            }             printf(" -
function call\n");
```

```
        }
        char* popen = strstr(line, "{");
        if(popen != NULL) printf("{ - special character\n");
char* semicolon = strstr(line, ";");
        if(semicolon != NULL) printf("; - special character\n");
char* pclose = strstr(line, "}");
        if(pclose != NULL) printf("} - special character\n");
char* bracket_open = strstr(line, "(");
        if(bracket_open != NULL && main1 == NULL && printf1 == NULL) printf("( -
special character\n");          char* tempvar;
        if(conditional == 1)
        {
            tempvar = strstr(line, "(");
int i;              int condition;
            for(int z = 0; z < strlen(tempvar); z++)
            {
                if(*(tempvar + z) == '<' || *(tempvar + z) == '>')
                {
                    condition = z;
break;
                }
            }
            for(int j = 1; j < condition; j++)
            {
                printf("%c", *(tempvar + j));
            }
            printf(" - variable\n");
            char* tempvar1 = strstr(tempvar, "<");
char* tempvar2 = strstr(tempvar, ">");
if(tempvar1 != NULL) tempvar = tempvar1;
if(tempvar2 != NULL) tempvar = tempvar2;
printf("%c - condition\n", *(tempvar));
            for(int z = 1; z < strlen(tempvar); z++)
            {
                if(*(tempvar + z) == ')')
                {
        condition = z;
                    break;
                }
                else
                {
                    printf("%c", *(tempvar + z));
                }
            }
            printf(" - variable\n");
        }         char* bracket_close =
strstr(line, ")");
```

```
        if(bracket_close != NULL && main1 == NULL && printf1 == NULL) printf(") -
special character\n");
    }
fclose(fp);
    return 0;
}
```

Output:



Learning Outcome:

- The role and operation of Lexical Analyser was understood.
- Implementation of Regular Expression has been learnt.
- Learnt to parse the program and token identification.
- Understood the role of a Lexical Analyser in compilation.
- Understood the significance of keywords and general structure of a C program.

SSN COLLEGE OF ENGINEERING

ASSIGNMENT 2: LEXICAL ANALYSER USING LEX TOOL

-SRINITHYEE S K
185001166

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:

```
 %{
#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|e
xtern|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;
}


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;
}
```

OUTPUT:

```
KW FUNCT
KW ID ASSIGN_OP NUM_CONST ID
KW ID ASSIGN_OP NUM_CONST
KW ID ID ASSIGN_OP CHAR_CONST
KW ID ASSIGN_OP NUM_CONST

FUNCT

ID ASSIGN_OP ID ARITH_OP NUM_CONST

KW ID REL_OP NUM_CONST
FUNCT


KW ID REL_OP NUM_CONST
FUNCT
ID ASSIGN_OP NUM_CONST



SCMT
MCMT

KW NUM_CONST


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

                        SYMBOL TABLE
                NAME    TYPE     VALUE
                a       int      1
                b       int      0
                c       int      2
                d       char     NULL
                e       char     'Z'
                f       float    1.23
        -------------------------------------------
```

Learning Outcome:

- Learnt the basics of Lex tool.
- Implement recognition for regular expressions using Lex terminology.
- Learnt to implement a basic symbol table using Lex on the parsed C program.
- Realized that Lex tool is more powerful and easy-to-use for Lexical Analysis.

SSN COLLEGE OF ENGINEERING

ASSIGNMENT 3: ELIMINATION OF LEFT RECURSION USING C

-SRINITHYEE S K
185001166

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:

```
#include<stdio.h>
#include<string.h>
int main()
{
    char non_terminal, productions[10][100], splits[10][10];
    int num;
    printf("Enter number of productions: ");
    scanf("%d", &num);
    printf("Enter the grammar:\n");
    for(int i = 0; i < num; i++)
    {
        scanf("%s", productions[i]);
    }
    for(int i = 0; i < num; i++)
    {
        printf("\n%s", productions[i]);
        non_terminal = productions[i][0];
        char production[100], *token;
        int j, flag = 0;
        for(j = 0; productions[i][j + 3] != '\0'; j++)
            production[j] = productions[i][j + 3];
        production[j] = '\0';
        j = 0;
        token = strtok(production, "|");
        while(token != NULL)
        {
            strcpy(splits[j], token);
            if(token[0] == non_terminal && flag == 0) flag = 1;
            else if(token[0] != non_terminal && flag == 1) flag = 2;
            j++;
            token = strtok(NULL, "|");
        }
        if(flag == 0) printf(" is not left recursive.\n");
        else if(flag == 1) printf(" is left recursive, cannot reduce.\n");
        else
        {
```

```
            printf(" is left recursive. After elimination:\n");
            flag = 0;
            for(int k = 0; k < j; k++)
            {
                if(splits[k][0] != non_terminal) {
                    if(flag != 0)
                    {
                        printf("|%s%c\'", splits[k], non_terminal);
                    }
                    else
                    {
                        flag = 1;
                        printf("%c->%s%c\'", non_terminal, splits[k], non_terminal);
                    }
                }
            }
            printf("\n");
            flag = 0;
            for(int k = 0; k < j; k++)
            {
                if(splits[k][0] == non_terminal) {
                    if(flag != 0)
                    {
                        printf("|%s%c\'", splits[k] + 1, non_terminal);
                    }
                    else
                    {
                        flag = 1;
                        printf("%c\'->%s%c\'", non_terminal, splits[k] + 1, non_terminal);
                    }
                }
            }
            printf("|e\n");
        }
    }
}
```

OUTPUT:

```
seccon@ubuntu: ~/Desktop
seccon@ubuntu:~$ cd Desktop
seccon@ubuntu:~/Desktop$ gcc -o a lr.c
seccon@ubuntu:~/Desktop$ ./a
Enter number of productions: 3
Enter the grammar:
E->E+T|T
T->T*F|F
F->i

E->E+T|T is left recursive. After elimination:
E->TE'
E'->+TE'|e

T->T*F|F is left recursive. After elimination:
T->FT'
T'->*FT'|e

F->i is not left recursive.
```

Learning Outcome:

- Learnt about left recursive grammars.
- Learnt to check if a grammar is left recursive using C.
- Successfully implemented a conversion in C which converts left recursive grammar to non left recursive grammar.