CS6612 – Compiler Lab

Ex no : 1 **Name : Sreedhar V**

Date : 03.02.2021 **Reg no: 185001161**

Specification

Develop a Lexical analyser to recognize the patterns namely, identifiers, constants, comments and operators using the following regular expressions.

Code

(predefined.h file)

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include<limits.h>
#include<stdbool.h>
#include<ctype.h>
const int T = 1;
const int F = 0;
// Creating a global array of keywords , operators, symbols

char keywords[][100]={"return","int","float","long","double","char","if
","else"};
char operators[][10]={"+","-
","*","/","^","%","<",">","!","?","==","<=",">=","||","&&"};
char symbol[]={'{','}',';',',','.',':',')','('};
int sym_size = 9;
int key_size = 10;
int op_size = 15;
int relop_size = 7;
int arthop_size = 6;
char arth_operators[][10]={"+","-","*","/","^","%",};
int logop_size = 4;
char log_operators[][10] = {"||","&&","&","|"};
char rel_operators[][10]={"<",">","!","?","==","<=",">="};
```

(Analyser.c file)

```
#include"predefined.h"


/*Code checks for the following patterns
Identifier
Constant
Comments
Operators
Keywords*/

bool issymbol(char ch[100])
{
    int i;
    for(i=0;i<sym_size;i++)
    {
        if(ch[0]==symbol[i])
            return true;
    }
    return false;
}
bool check_function(char str[100])
{
    int i=0;
    bool a,b;
    for(i=0;i<strlen(str);i++)
    {
        if(str[i]=='(')
            a=true;
        if(str[i]==')')
            b=true;
    }
    if(a && b)
        return true;
    return false;
}
bool check_operator(char str[100])
{
    int i;
    for(i=0;i<op_size;i++)
    {
        if(strcmp(str,operators[i])==0)
            return true;
    }
    return false;
```

```
}
bool check_RELOP(char *str)
{
    int i;
    for(i=0;i<relop_size;i++)
    {
        if(strcmp(str,rel_operators[i])==0)
            return true;
    }
    return false;

}
bool check_LOGOP(char *str)
{
    int i;
    for(i=0;i<logop_size;i++)
    {
        if(strcmp(str,log_operators[i])==0)
            return true;
    }
    return false;

}
bool check_ARTHOP(char *str)
{
    int i;
    for(i=0;i<arthop_size;i++)
    {
        if(strcmp(str,arth_operators[i])==0)
            return true;
    }
    return false;

}
bool check_comment(char *str)
{
    if(str[0]=='\\' || str[1]=='\\')
        return true;
    return false;
}
bool check_keyword(char str[100])
{
    int i;
    for(i=0;i<key_size;i++)
    {
        if(strcmp(str,keywords[i])==0)
```

```
            return true;

    }
    return false;
}
bool check_assign(char ch)
{
    if(ch=='=')
        return true;
    return false;
}
bool check_num(char str[100])
{
    int len = strlen(str);
    int i=0;
    while(i<len)
    {
        if(!isdigit(str[i]))
            return false;
        i++;
    }
    return true;
}
bool check_char(char str[100])
{
    if((str[0]=='\"' || str[0]=='\'' ) && (str[strlen(str)-
1]=='\"' || str[strlen(str)-1]=='\''))
        return true;
    return false;
}

void analyser(char input[100000])
{
    int i=0,temp=0;
    int len = strlen(input);
    char line[100][1000];
    int l=0;
    int flag=0;
    char *token = strtok(input,"\n");
    while(token!=NULL)
    {
        strcpy(line[l++],token);
        token = strtok(NULL,"\n");
    }
    int l1=0;
    while(l1<l)
```

```
{
    if((line[l1][0]=='/') && (line[l1][1]=='/'))
    {
            printf(" SL CMT\n");
            l1++;
            continue;
    }
    if((line[l1][0]=='/') && (line[l1][1]=='*') && (flag == 0))
    {
            printf(" ML CMT STARTS\n");
            l1++;
            flag=1;
            continue;
    }
    if((line[l1][0]=='*') && (line[l1][1]=='/') && (flag == 1))
    {
            printf(" ML CMT ENDS\n");
            l1++;
            flag=0;
            continue;
    }
    if(flag)
    {
        l1++;
        continue;
    }
    token = strtok(line[l1]," ");
    if(strlen(token)==1 && token[0]=='\n')
        continue;
    while(token!=NULL)
    {
        if(check_keyword(token))
            printf(" KW ");
        else if(check_function(token))
            printf(" FC");
        else if(check_assign(token[0]))
            printf(" ASSIGN");
        else if(check_operator(token))
        {
            if(check_RELOP(token))
                printf(" RELOP");
            else if(check_LOGOP(token))
                printf(" LOGDOP");
            else if(check_ARTHOP(token))
                printf(" ARTHOP");
            else
```

```
                    printf(" OP");
            }
            else if(issymbol(token))
                printf(" SP");
            else if(check_num(token))
                printf(" NUMCONST");
            else if(check_char(token))
                printf(" CHARCONST");
            else if(!isdigit(token[0]))
                printf(" ID");
            else
                printf(" INVALID CHA");
            token = strtok(NULL," ");
        }
        printf("\n");
        l1++;
    }
}
void main()
{
    char code[100000];
    FILE * file = fopen("input.txt", "r");
    char  c;
    int idx=0;
    while (fscanf(file , "%c" ,&c) == 1)
    {
        code[idx] = c;
        idx++;
    }
    code[idx] = '\0';
    printf("\n");
    analyser(code);
}
```

(Sample input file)

```
/*
Multi line comment..
hi
hi
hi
*/
main()
{
int a = 5 , b = 10 ;
if ( a > b )
printf(\"a_is_greater\");
else
printf(\"b_is_greater\");
sum = add(a,b) ;
// This is a comment......
double 7aid ;
false = u > "8" ? a && b : a || b ;
}
```

(Output)

```
[Running] cd "g:\Academics\SSN\6th Sem\Compiler Design\Ex1\" && gcc sim
ple_analyser.c -
o simple_analyser && "g:\Academics\SSN\6th Sem\Compiler Design\Ex1\"sim
ple_analyser

 ML CMT STARTS
 ML CMT ENDS
 FC
 SP
 KW  ID ASSIGN NUMCONST SP ID ASSIGN NUMCONST SP
 KW  SP ID RELOP ID SP
 FC
 KW
 FC
 ID ASSIGN FC SP
 SL CMT
 KW  INVALID CHA SP
 ID ASSIGN ID RELOP CHARCONST RELOP ID LOGDOP ID SP ID LOGDOP ID SP
 SP

[Done] exited with code=18 in 2.238 seconds
```

Learning Outcome:

- I've learnt how the lexical analyser works and its basic functionalities.
- I've learnt how to tokenize an entire C program.
- I've learnt how to identify and group the lexemes into specific categories.
- I've learnt how to recognize the pattern (regular expression) and separate them into tokens for a program.
- I've learnt the regular expression for identifiers, constants, comments and operators.

CS6612 – Compiler Lab

Ex no : 2 **Name : Sreedhar V**

Date : 03.02.2021 **Reg no: 185001161**

Specification

Develop a Lexical analyzer to recognize the patterns namely, identifiers, constants, comments and operators using the following regular expressions. Construct symbol table for the identifiers with the following information using LEX tool.

Code

```
%{
#include<stdio.h>
#include<string.h>
int i = 0;
int address=1000;
int size =0;
int flag =1;
char buffer[100];
struct table{
    char symbol[50];
    char type[50];
    int address;
    char value[100];
    int size;
}t[100];
void add_symbol(char a[]);
int lookup(char a[]);
void add_value(char val[],int s);
void display();
void update(char a[]);
%}
/* Rules Section*/
MCMT "/*"([^*]|\*+[^*/])*\*+"/"
ARTHOP [+|-|*|/|^|%]
FC [a-zA-Z]+[(].*[)]
ASSIGN ["="]
RELOP [<|>|!|?|==|<=|>=]
```

```
LOGOP [&&|"||"|"|"|<<|>>|~]
SYM ['{'|'}'|';'|'|'|'.'|':'|')'|'('|,]
INT [-]?[0-9]+
FLOAT [0-9]*"."[0-9]+
ID [a-zA-Z_][a-zA-Z0-9_]*
STR ["][a-zA-Z0-9]["]
SCMT [/][/].*
CHAR ['][a-zA-Z0-9]['] 
%%
return|int|float|long|double|char|if|else {printf("KW ");update(yytext)
;}
{MCMT} {printf("MULTI LINE CMT");}
{FC} {printf("FC ");}
{ASSIGN} {printf("ASSIGN ");flag=1;}
{STR} {printf("STRING ");}
{CHAR} {printf("CHAR ");add_value(yytext,1);address++;}
{SCMT} {printf("SINGLE LINE CMT");continue;}
{ARTHOP} {printf("ARTHOP ");}
{RELOP} {printf("RELOP ");}
{LOGOP} {printf("LOGOP ");}
{SYM} {printf("SYM ");flag=0;}
{FLOAT} {printf("FLOAT ");if(flag)add_value(yytext,4);address+=4;}
{INT} {printf("INT ");if(flag)add_value(yytext,2);address+=2;}
{ID} {printf("ID ");if(lookup(yytext))add_symbol(yytext);}
%%
int yywrap(void){}
int lookup(char a[])
{
    int i=0;
    for(int i=0;i<size;i++)
    {
        if(!strcmp(t[i].symbol,a))
            return 0;
    }
    return 1;
}
void add_value(char val[],int s)
{
    size--;
    strcpy(t[size].value,val);
    t[size].size = s;
    size++;
}
void add_symbol(char a[])
{
    strcpy(t[size].symbol,a);
```

```
    strcpy(t[size].value,"NULL");
    strcpy(t[size].type,buffer);
    t[size].address = address;
    size++;
}
void update(char a[])
{
    strcpy(buffer,a);
}
void display()
{
    int i=0,j;
    printf("\n Starting Address = 1000");
    printf("\n\n SYMBOL TABLE\n");
    printf("\nSYMBOL\tValue\tType  \tAddr\tSize\n");
    for(i=0;i<40;i++)printf("-");
    printf("\n");
    for(i=0;i<size;i++)
    {
        printf("%-6s\t%-5s\t%-6s\t%-
7d\t%d\n",t[i].symbol,t[i].value,t[i].type,t[i].address,t[i].size);
    }
    printf("\n\n");
}

int main()
{
    // The function that starts the analysis
    yyin = fopen("input.c","r");
    yylex();
    display();
    return 0;
}
```

(Sample input file)

```
/*
Multi line comment..
hi
hi
hi
*/
int main()
{
    int a=5;
```

```
    float b=10.13;
    float c;
    if(a>b)
        printf("a_is_greater");
    else
        printf("b is greater");
    add(a,b);
    // This is a comment......
    char out = 'd' > "8" ? a & b : a || b ;
    int var =0, v =9;
}
```

(Output)

```
OUTPUT   DEBUG CONSOLE   TERMINAL

PS G:\Academics\SSN\6th Sem\Compiler Design\Lex> ./a

MULTI LINE CMT
KW  FC
SYM
    KW  ID ASSIGN INT SYM
    KW  ID ASSIGN FLOAT SYM
    KW  ID SYM
    FC
        FC SYM
    KW
        FC SYM
    FC SYM
    SINGLE LINE CMT
    KW  ID  ASSIGN  CHAR  RELOP  STRING  RELOP  ID  LOGOP  ID  SYM  ID  ARTHOP ARTHOP  ID  SYM
    KW  ID  ASSIGN INT SYM  ID  ASSIGN INT SYM
SYM

 Starting Address = 1000

 SYMBOL TABLE

SYMBOL  Value   Type    Addr    Size
----------------------------------------
a       5       int     1000    2
b       10.13   float   1002    4
c       NULL    float   1006    0
out     'd'     char    1006    1
var     0       int     1007    2
v       9       int     1009    2
```

Learning Outcome:

- I’ve learnt how the lexical analyser works and its basic functionalities.
- I’ve learnt how to tokenize an entire C program using lex tool.
- I’ve learnt how to identify and group the lexemes into specific categories.
- I’ve learnt how to construct the symbol table for the identifiers.
- I’ve learnt how to recognize the pattern (regular expression) and separate them into tokens for a program.
- I’ve learnt the regular expression for identifiers, constants, comments and operators.

CS6612 – Compiler Lab

Ex no : 3 **Name : Sreedhar V**

Date : 20.02.2021 **Reg no: 185001161**

Specification

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

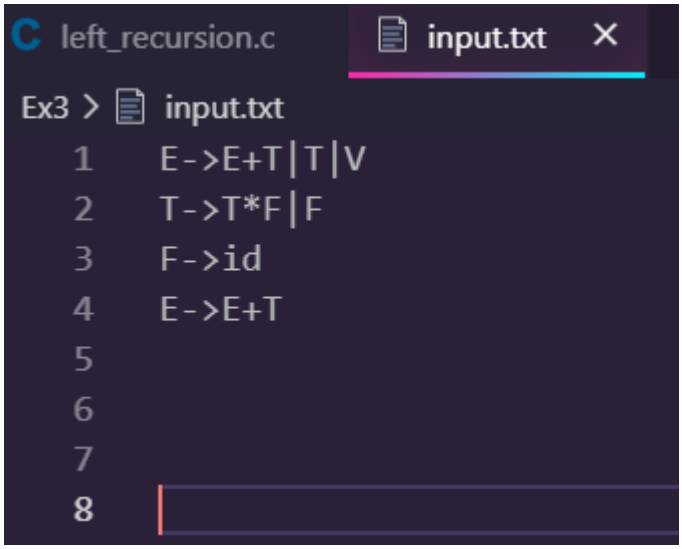
```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void detect(char prod[][30],int n)
{
    for(int i=0;i<n;++i)
   {
        char p = prod[i][0];
        if(p==prod[i][3])
        {
            char *token = strtok(prod[i],"|");
            char alpha[10];
            int j=0;
            for(int itr=4;itr<strlen(token);itr++)
                alpha[j++]=token[itr];
            char beta[10][20];
            j=0;
            char buffer[20];
            while(token!=NULL)
            {
                strcpy(buffer,token);
                token=strtok(NULL,"|");
                if(token!=NULL)
                    strcpy(beta[j++],token);
                else
                    strcpy(beta[j++],buffer);
            }
            j--;
            if(!j)
```

```
            {
                printf("\n%c -> %c'",p,p);
                char alpha[10];
                int j=0;
                for(int itr=4;itr<strlen(prod[i]);itr++)
                    alpha[j++]=prod[i][itr];
                printf("\n%c'->%s %c' | (null)\n",p,alpha,p);
                continue;
            }
            printf("\n%c ->%s %c'",p,beta[0],p);
            for(int i=1;i<j;i++)
                printf("|%s %c'",beta[i],p);
            printf("\n%c'->%s %c' | (null)\n",p,alpha,p);
        }
        else
            printf("\n%s\n",prod[i]);
    }
}
int main()
{
    int n=0;
    char prod[20][30];

    printf("\n\tLeft Recursion_Elimation\n");
    int i=0;
    FILE *file = fopen("input.txt","r");
    char c;
    printf("\nGiven grammar\n");
    while(fscanf(file,"%c",&c)==1)
    {
        if(c=='\n')
        {
            prod[n][i]='\0';
            printf("\n%s\n",prod[n]);
            n++;
            i=0;
        }
        else
        {
            prod[n][i]=c;
            i++;
        }
    }
    printf("The set of productions in grammer after left recursion:\n")
;
    detect(prod,n);
```

```
    printf("\n");
    return 0;
}
```

(Sample input file)



(Output)

```
PS G:\Academics\SSN\6th Sem\Compiler Design\Ex3> gcc left_recursion.c -o a
PS G:\Academics\SSN\6th Sem\Compiler Design\Ex3> ./a

        Left Recursion_Elimation

Given grammar

E->E+T|T|V

T->T*F|F

F->id

E->E+T




The set of productions in grammer after left recursion:

E ->T E'|V E'
E'->+T E' | (null)

T ->F T'
T'->*F T' | (null)

F->id

E -> E'
E'->+T E' | (null)
```

Learning Outcome:

- I’ve learnt how to identify the left recursion in the production of the grammar and construct a new grammar with removing such productions.
- I’ve learnt how to implement the same using C code which identifies whether the grammar is left recursive or not and converts the grammar in a such a way that the left recursion is removed.
- I’ve learnt the Elimination of Immediate Left Recursion using the rule if the production is in the form A->Aα | β, then A-> βA’,A’->Ɛ | αA’.

CS6612 – Compiler Lab

Ex no : 4 **Name : Sreedhar V**

Date : 28.02.2021 **Reg no: 185001161**

Specification

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().

G: E→E+T|E-T|T

T→T*F | T/F|F

F→(E)|i

Code

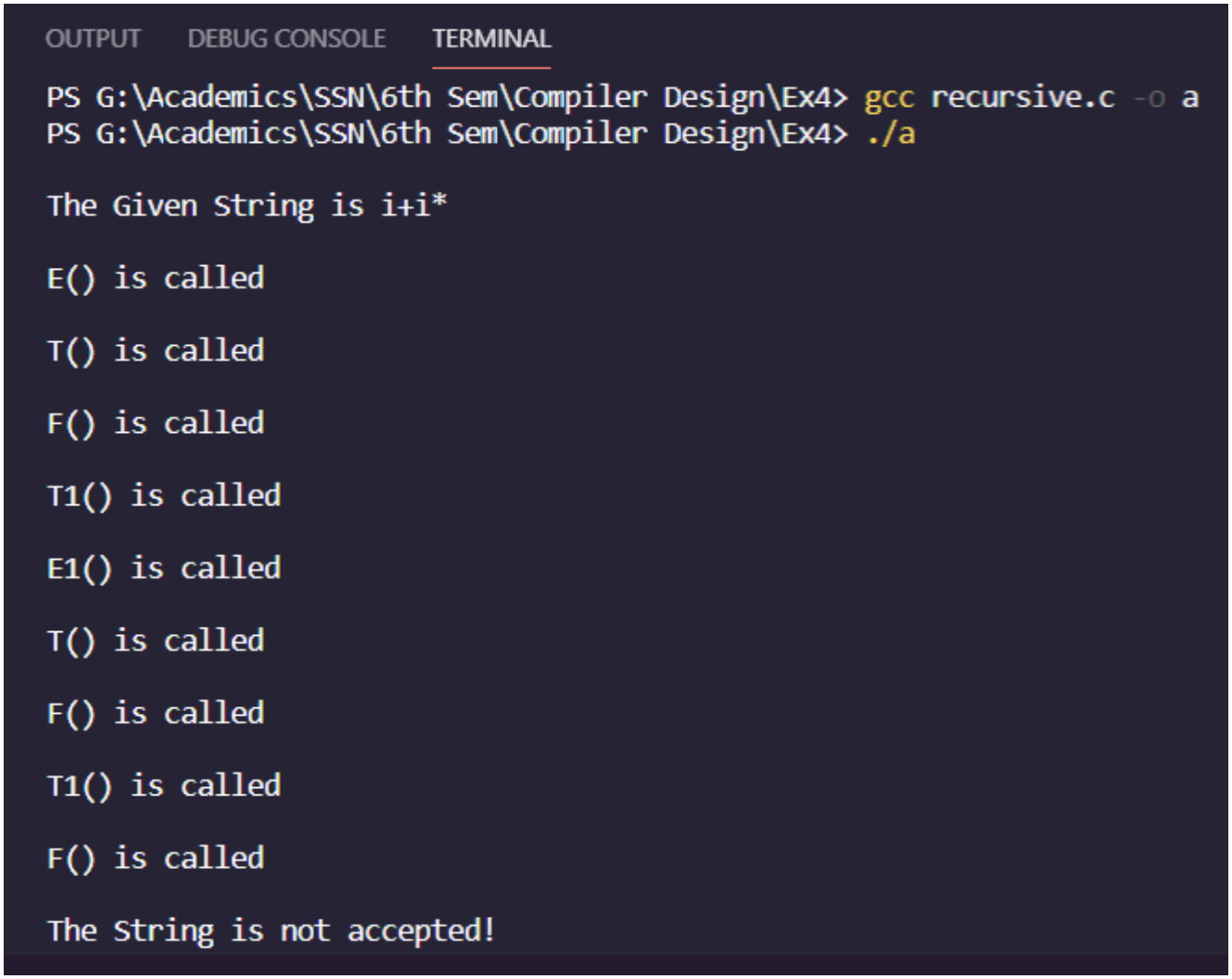
```
/*E->TE'
E'-> +TE' | -TE' | epsilon
T->FT'
T'->*FT'| /FT' | epsilon
F->(E) | id*/
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<string.h>
char str[100];
int i=0;
int E();
int E1();
int T();
int T1();
int F();
void print_tab(int depth);
void main()
{
    strcpy(str,"i+i");
    printf("\nThe Given String is %s\n",str);
    if(E(0))
```

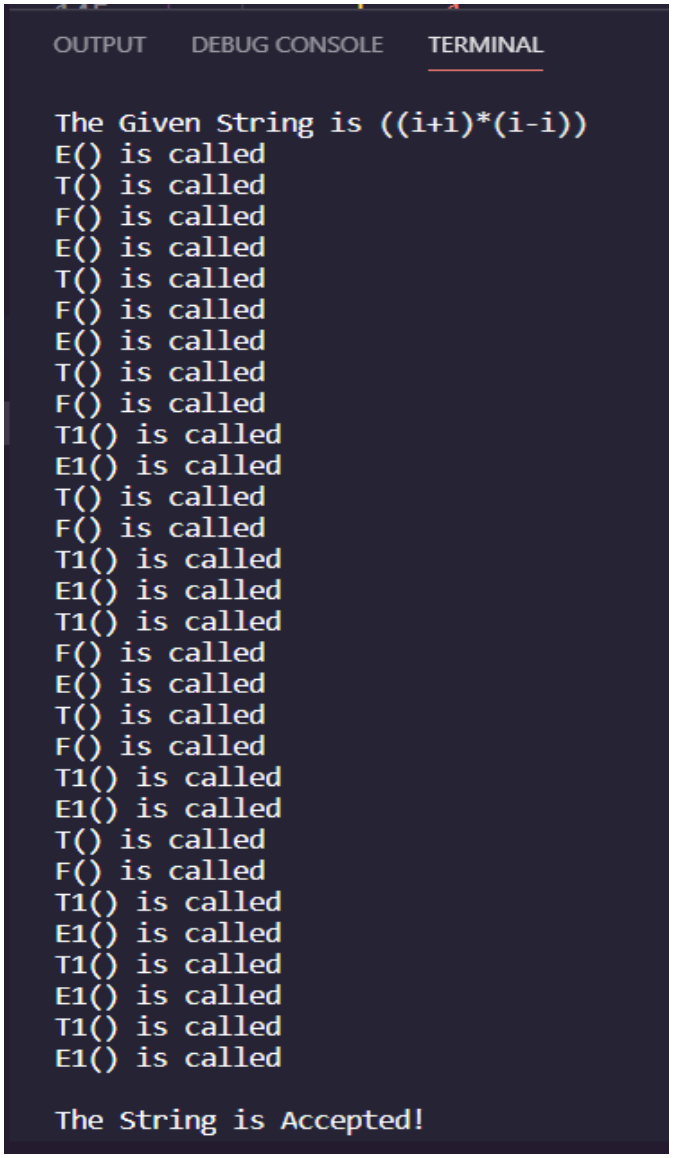
```
        printf("\nThe String is Accepted!\n");
    else
        printf("\nThe String is not accepted!\n");
}
int E()
{
    printf("\nE() is called\n");
    if(T())
    {
        if(E1())
            return 1;
        else
            return 0;
    }
    else
        return 0;
}
int E1()
{
    printf("\nE1() is called\n");
    if(str[i]=='+')
    {
        i++;
        if(T())
        {
            if(E1())
                return 1;
            else
                return 0;
        }
        else
            return 0;
    }
    else if(str[i]=='-')
    {
        i++;
        if(T())
        {
            if(E1())
                return 1;
            else
                return 0;
        }
        else
            return 0;
    }
```

```
    else if(str[i]=='\0')
    {
        //i++;
        return 1;
    }
}
int T()
{
    printf("\nT() is called\n");
     if(F())
    {
        if(T1())
            return 1;
        else
            return 0;
    }
    else
        return 0;
}
int T1()
{
    printf("\nT1() is called\n");
    if(str[i]=='*')
    {
        i++;
        if(F())
        {
            if(T1())
                return 1;
            else
                return 0;
        }
        else
            return 0;
    }
    else if(str[i]=='/')
    {
        i++;
        if(F())
        {
            if(T1())
                return 1;
            else
                return 0;
        }
        else
```

```
            return 0;
    }
    else if(str[i]=='\0')
    {
        //i++;
        return 1;
    }
}
int F()
{
    printf("\nF() is called\n");
    if(str[i]=='(')
    {
        i++;
        if(E())
        {
            if(str[i]==')')
            {
                i++;
                return 1;
            }
            else
                return 0;
        }
        else
            return 0;
    }
    else if(str[i]=='i')
    {
        i++;
        return 1;
    }
}
void print_tabs(int depth)
{
    int i=0;
    for(i=0;i<depth;i++)
        printf("\t");
}
```

(Output)


OUTPUT
DEBUG CONSOLE
TERMINAL
PS G:\Academics\SSN\6th Sem\Compiler Design\Ex4> gcc recursive.c -o a
PS G:\Academics\SSN\6th Sem\Compiler Design\Ex4> ./a
The Given String is i+i*
E() is called
T() is called
F() is called
T1() is called
E1() is called
T() is called
F() is called
T1() is called
F() is called
The String is not accepted!



Learning Outcome:

- I've learnt how to construct the recursive decent parser for the given input grammar
- I've learnt the internal working of the recursive parser and able to trace the input string whether it's accepted or not.

CS6612 – Compiler Lab

Ex no : 5 **Name : Sreedhar V**

Date : 10.03.2021 **Reg no: 185001161**

- Implementation of Desk Calculator using Yacc Tool

Specification

Write 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, + , - ,* , / , ^ , (,).Precedence and associativity has to be preserved. Yacc is available as a command in linux. Thegrammar should have non terminals E, Op and a terminal id.

Code

(lex file)

```
%{
#include "calc.tab.h"
#include <stdio.h>
#include<string.h>
void yyerror(char *);
int yylval;
%}
num [0-9]+
tab "\t"|"\n"
GT ">="
LT "<="
LS "<<"
RS ">>"
LAND "&&"
LOR "||"
EQ "=="
NQ "!="
%%
{num} {yylval=atoi(yytext);return NUM;}
{tab} {return 0;}
. return yytext[0];
{GT} {return GT;}
{LT} {return LT;}
{LS} {return LS;}
{RS} {return RS;}
```

```
{LAND} {return LAND;}
{LOR} {return LOR;}
{NQ} {return NQ;}
{EQ} {return EQ;}
%%
int yywrap(void){return 1;}
```

(yacc file)

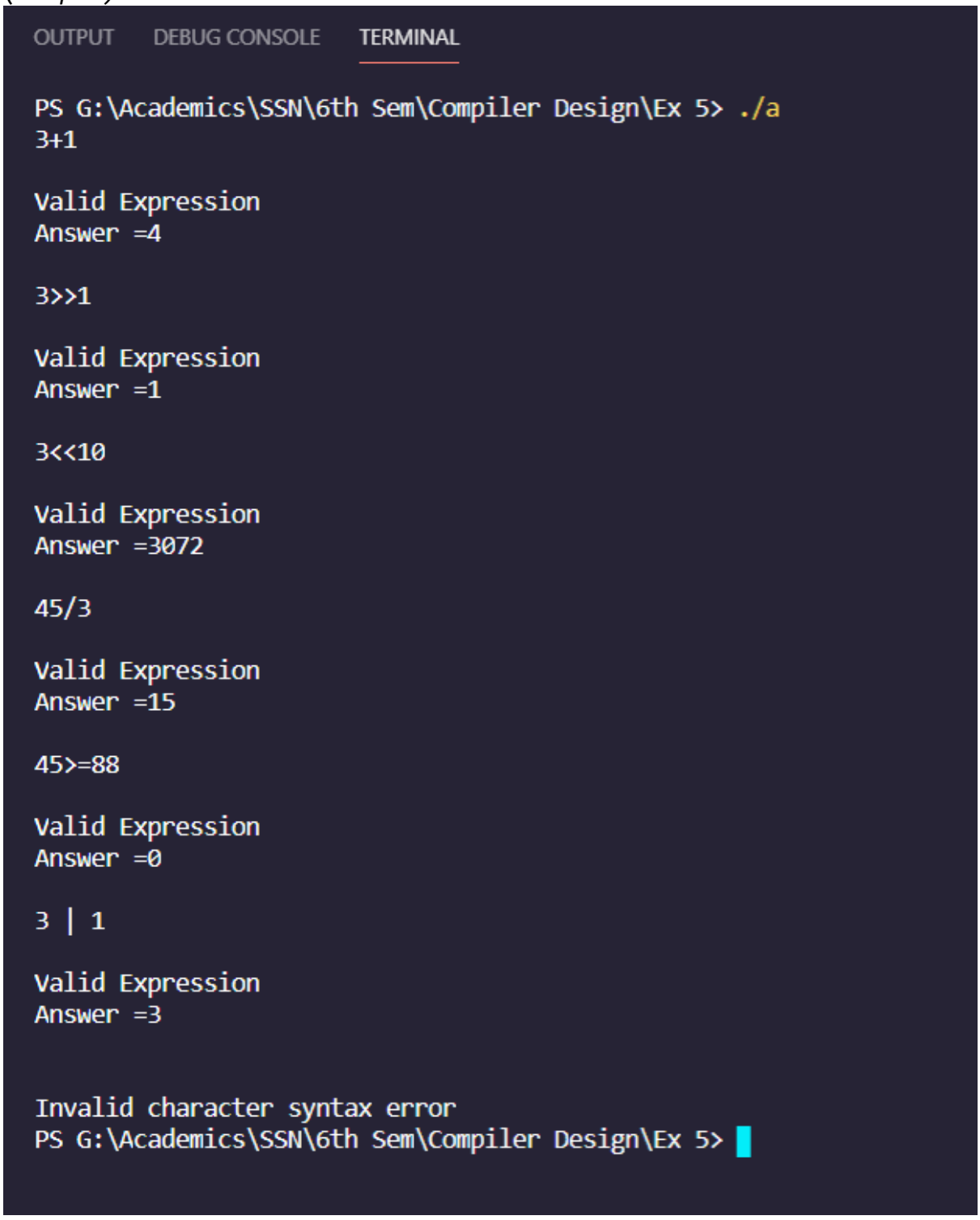
```
%{
#include <stdio.h>
int flag=0;
int yyerror(char *er);
int yylex(void);
#include <math.h>
#include<stdlib.h>
%}
%token NUM
%token LS
%token RS
%token GT
%token LT
%token LAND
%token EQ
%token NQ
%token LOR
%left LOR
%left LAND
%left '|'
%left '&'
%left EQ NQ
%left '>' GT
%left '<' LT
%left LS RS
%left '+' '-'
%left '*' '^' '/' '%'
%left '(' ')'

%%
P : E {if(!flag){printf("\nValid Expression\n");}printf("Answer =%d\n\n
",$$);return 0;};
E : NUM { $$ = $1;}
  | E '+' E {$$ = $1 + $3; }
  | E '-' E {$$ = $1 - $3; }
  | E '*' E {$$ = $1 * $3; }
  | E '/' E {$$ = $1 / $3; }
```

```
  | E '%' E {$$ = $1 % $3; }
  | E '^' E {$$ = pow($1,$3); }
  | '(' E ')' {$$ = $2 ; }
  | E GT E {$$ = $1>= $3; }
  | E '>' E {$$ = $1> $3; }
  | E '<' E {$$ = $1< $3; }
  | E LT E {$$ = $1<= $3; }
  | E RS E {$$ = $1>>$3; }
  | E LS E {$$ = $1<< $3; }
  | E '&' E {$$ = $1 & $3; }
  | E '|' E {$$ = $1 | $3; }
  | E LAND E {$$ = $1 && $3; }
  | E LOR E {$$ = $1 || $3; }
%%
int main()
{
    while(1)
    {
        yyparse();
    }


    return 0;
}
int yyerror(char *er)
{
    flag=1;
    printf("\nInvalid character %s\n",er);
    exit(0);
}
```

(Output)



Learning Outcome:

- I've learnt how to implement the calculator considering all its precedence and associativity while execution.

- I’ve learnt the basic syntax of the yacc program and how to implement the grammar rules in c code.
- I’ve learnt the associativity and precedence of the operators and how to evaluate whether the given expression is a valid one or not.
- I’ve leant how the lex program sends the token based on its syntax and yacc program evaluates the stream of tokens based on the given grammar rules and produces the result.

CS6612 – Compiler Lab

Ex no : 6 **Name : Sreedhar V**

Date : 02.03.2021 **Reg no: 185001161**

Programming Assignment-6 - Implementation of Syntax Checker using Yacc Tool

---

Develop a Syntax checker to recognize the tokens necessary for the following statements by writing suitable grammars

Assignment statement

Conditional statement

Looping statement

Code

```
(%{
#include<stdio.h>
#include<string.h>
#include"syn_ch.tab.h"
%}

id ([a-zA-Z_][a-zA-Z0-9_]*|[0-9]+)
rl ("<"|"<="|">"|">="|"=="|"!=")
op ("+"|"-"|"*"|"/"|"%")
un ("++"|"--")
nl "\n"
ts "\t"|" "

%%
"if" {return IF;}
"else" {return ELSE;}
"while" {return WHILE;}
"do" {return DO;}
"for" {return FOR;}
{id} {return ID;}
{rl} {return RL;}
{op} {return OP;}
{un} {return UN;}
{nl} {return NL;}
{ts} ;
. return yytext[0];
```

```
%%

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

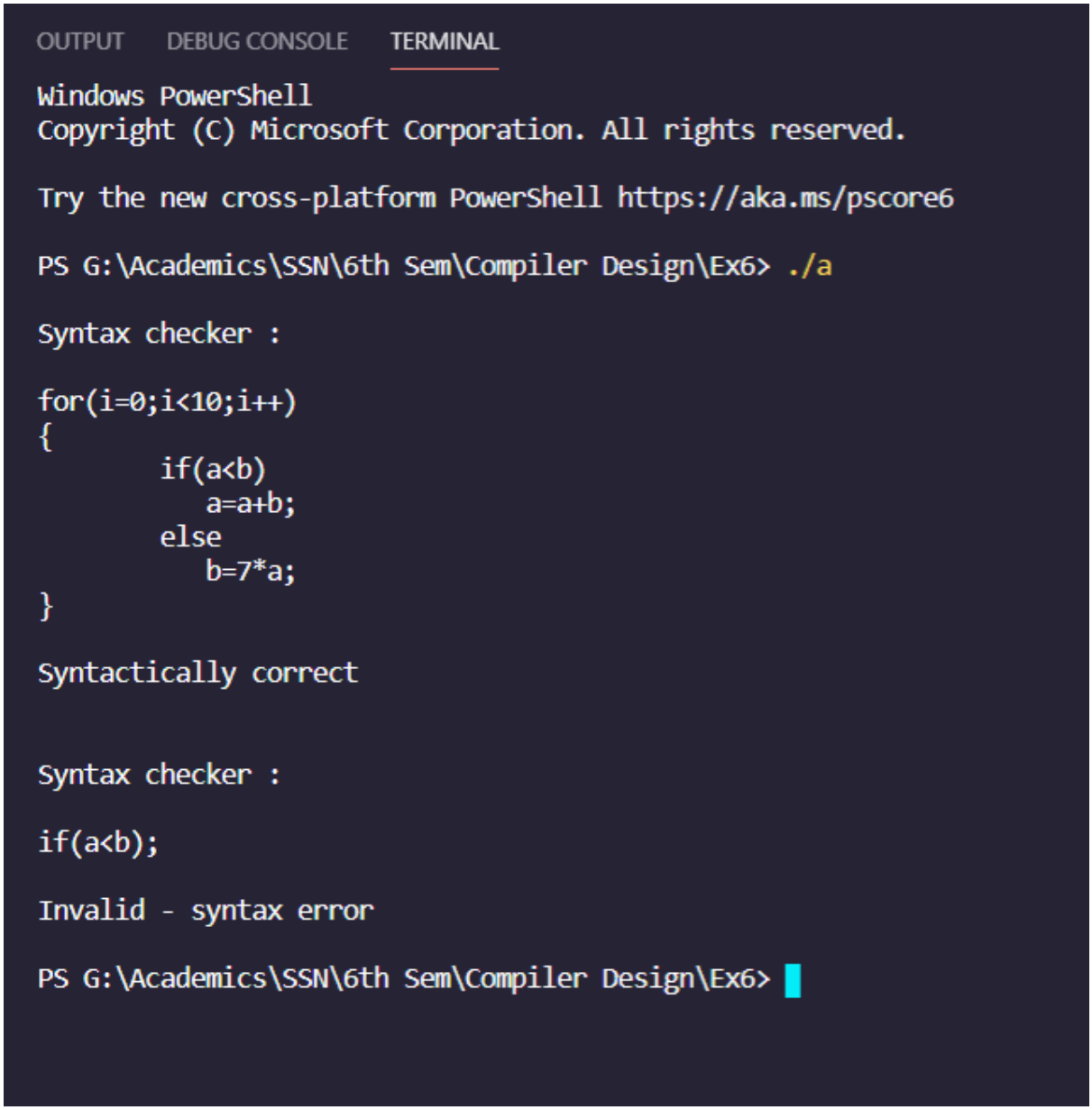
file)

(yacc file)

```
%{
#include <stdio.h>
int yyerror(char *er);
int yylex(void);
#include <math.h>
#include<stdlib.h>
%}
%token INT STR ID RELOP ARITHOP UNOP DATATYPE IF ELSE

%%
S   : DATATYPE VAR';'|VAR';'|CONDT
VAR : VAR ',' INIT | INIT
INIT : ID | EXPR
EXPR : ID'='ST | ID UNOP | ID'='ID ARITHOP ST | ID'='ID | ID ARITHOP '=
' INT
ST : INT|STR|ID
CONDT :IF '(' CONDT_EXP ')' | ELSE
CONDT_EXP : ID RELOP CONDT_EXP | INT RELOP CONDT_EXP | ID | INT
%%
int main()
{
    while(1)
    {
        yyparse();
    }
    return 0;
}
```

(Output)



Learning Outcome:

- I've learnt how to implement the syntax checker considering all its grammar rules and syntax for C language while execution.

- I’ve learnt the basic syntax of the yacc program and how to implement the grammar rules in c code.
- I’ve leant how the lex program sends the token based on its syntax and yacc program evaluates the stream of tokens based on the given grammar rules and produces the result.

CS6612 – Compiler Lab

Ex no : 7 **Name : Sreedhar V**

Date : 23.04.2021 **Reg no: 185001161**

Programming Assignment-7 - Generation of Intermediate Code using Lex and Yacc

---

The new Language **Pascal-2021** is introduced with the following programming constructs

Data types

integer

real

char

Operators

+, -, * and /

Precedence → * and / have lesser priority than + and –

Associativity → * and / → right , + and - → left

Declaration statement

var: type;

var: type=constant;

Example

a: integer;

b: integer = 5;

Generate Intermediate code (TAC sequences) for the code involving conditional and assignment statements.

Conditional Statement

if condition then

---

else

---

end if

Generate Intermediate code in the form of Three Address Code sequence for the sample input program written using declaration, conditional and assignment statements in new language **Pascal-2021**, Following is the sample input

Code

Lex file(tac.l)

```
%{
struct info{
        char *var;
        char *code;
        int val;
    };
#include <stdio.h>
#include<string.h>
#include "tac.tab.h"
void yyerror(char *);
extern YYSTYPE yylval;
%}
id ([a-zA-Z_][a-zA-Z0-9_]*|[0-9]+)
num [0-9]+
GT ">="
LT "<="
LS "<<"
RS ">>"
LAND "&&"
LOR "||"
EQ "=="
NQ "!="
CHAR ['][a-zA-Z0-9]['] 
%%
begin {return BEG;}
end {return END;}
if {return IF;}
then {return THEN;}
else {return ELSE;}
end_if {return ENDIF;}
integer {return INT;}
real {return REAL;}
char {return CHAR;}
var {return VAR;}
{num} {yylval.temp.val=atoi(yytext);return NUM;}
{id} {yylval.temp.var=(char*)malloc(10);strcpy(yylval.temp.var,yytext);
return ID;}
{GT} {return GT;}
{LT} {return LT;}
{LS} {return LS;}
```

```
{RS} {return RS;}
{LAND} {return LAND;}
{LOR} {return LOR;}
{NQ} {return NQ;}
{EQ} {return EQ;}
[{};()]  {return *yytext;}
[-+*/^()=&|%:;] {return *yytext;}
{CHAR} {yylval.temp.var=(char*)malloc(10);strcpy(yylval.temp.var,yytext
);return CH;}
[\t] ;
[\n] ;
[ ] ;

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

yacc file(tac.y)

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

#include <math.h>
#include<stdlib.h>
#include<string.h>
struct table{
    char var[20];
    int val;
    char type[20];
}symbol[10];
int l=0,t=1,n=0;
int yyerror(char *er);
int yylex(void);
void display_table()
{
    int j=0;
    printf("\tSYMBOL TABLE\n");
    printf("Name      Type      Value\n");
    for(j=0;j<n;j++)
    {
        printf("%-10s %-10s %-
10d\n",symbol[j].var,symbol[j].type,symbol[j].val);
```

```
    }
}
struct info{
        char *var;
        char *code;
        int val;
};

%}
%token NUM LS RS GT LT LAND EQ NQ LOR ID IF THEN BEG END ELSE INT CHAR
REAL CH ENDIF VAR

%union{

    struct info temp;
    int val;
    char *code;
}

%type<code> S BLOCK ASSIGNMENT CONDITION
%type<temp> E C ID
%type<val> NUM

%right '='
%left '!'
%left LOR
%left LAND
%left '|'
%left '&'
%left EQ NQ
%left '>' GT
%left '<' LT
%left LS RS
%right '*' '^' '/' '%'
%left '+' '-'
%left '(' ')'

%%
S : DEC BEG BLOCK END {printf("\nBEGIN %s\n END\n Syntactically Correct
\n",&$3);display_table();return 0;}
DEC   :DEC DEC
      | VAR ID ':' INT '=' NUM ';' {strcpy(symbol[n].var,$2.var);strcpy
(symbol[n].type,"INT");symbol[n++].val=$6;}
      | VAR ID ':' REAL '=' NUM ';' {strcpy(symbol[n].var,$2.var);strcp
y(symbol[n].type,"REAL");symbol[n++].val=$6;}
```

```
      | VAR ID ':' REAL ';'         {strcpy(symbol[n].var,$2.var);strcp
y(symbol[n].type,"REAL");symbol[n++].val=0;}
      | VAR ID ':' INT ';'          {strcpy(symbol[n].var,$2.var);strcp
y(symbol[n].type,"INT");symbol[n++].val=0;}
      | VAR ID ':' CHAR ';'         {strcpy(symbol[n].var,$2.var);strcp
y(symbol[n].type,"CHAR");symbol[n++].val=0;}

BLOCK : CONDITION {$$=(char*)malloc(2000);sprintf($$,"%s\n",$1);}
      | ASSIGNMENT';' {$$=(char*)malloc(2000);sprintf($$,"%s\n",$1);}
      | BLOCK BLOCK {$$=(char*)malloc(2000);sprintf($$,"%s%s\n",$1,$2);
}
      | {$$=(char*)malloc(2000);sprintf($$,"");}

ASSIGNMENT : ID '=' E {$$=(char*)malloc(2000);sprintf($$,"%s %s=%s\n",$
3.code,$1.var,$3.var);}
           | ID '+''+' {$$=(char*)malloc(2000);sprintf($$,"%s++\n",$1.v
ar);}
           | ID '-''-' {$$=(char*)malloc(2000);sprintf($$,"%s--
\n",$1.var);}


CONDITION : IF '(' C ')' THEN BLOCK ELSE BLOCK ENDIF {$$=(char*)malloc(
2000);sprintf($$,"   if %s goto L%d\n   goto L%d\nL%d:\n%s   goto L%d\n
L%d:\n%sL%d:\n",$3.code,l,l+1,l,$6,l+2,l+1,$8,l+2);l+=3;}
          | IF '(' C ')' THEN BLOCK ENDIF {$$=(char*)malloc(2000);sprin
tf($$,"   if %s goto L%d\n   goto L%d\nL%d:\n%sL%d:\n",$3.code,l,l+1,l,
$6,l+1);l+=2;}

E : NUM {$$.var=(char*)malloc(3);sprintf($$.var,"%d",$1);$$.code=(char*
)malloc(1);strcpy($$.code,"");}
  | E '+' E {$$.var=(char*)malloc(3);sprintf($$.var,"t%d",t);t+=1;$$.co
de=(char*)malloc(300);sprintf($$.code,"%s%s   %s = %s + %s\n",$1.code,$
3.code,$$.var,$1.var,$3.var);}
  | E '-
' E {$$.var=(char*)malloc(3);sprintf($$.var,"t%d",t);t+=1;$$.code=(char
*)malloc(300);$$.code=(char*)malloc(300);sprintf($$.code,"%s%s   %s = %
s - %s\n",$1.code,$3.code,$$.var,$1.var,$3.var);}
  | E '*' E {$$.var=(char*)malloc(3);sprintf($$.var,"t%d",t);t+=1;$$.co
de=(char*)malloc(300);sprintf($$.code,"%s%s   %s = %s * %s\n",$1.code,$
3.code,$$.var,$1.var,$3.var);}
  | E '/' E {$$.var=(char*)malloc(3);sprintf($$.var,"t%d",t);t+=1;$$.co
de=(char*)malloc(300);sprintf($$.code,"%s%s   %s = %s / %s\n",$1.code,$
3.code,$$.var,$1.var,$3.var);}
  | E '%' E {$$.var=(char*)malloc(3);sprintf($$.var,"t%d",t);t+=1;$$.co
de=(char*)malloc(300);sprintf($$.code,"%s%s   %s = %s % %s\n",$1.code,$
3.code,$$.var,$1.var,$3.var);}
```

```
  | E '^' E {$$.var=(char*)malloc(3);sprintf($$.var,"t%d",t);t+=1;$$.co
de=(char*)malloc(300);sprintf($$.code,"%s%s   %s = %s ^ %s\n",$1.code,$
3.code,$$.var,$1.var,$3.var);}
  | E RS E {$$.var=(char*)malloc(3);sprintf($$.var,"t%d",t);t+=1;$$.cod
e=(char*)malloc(300);sprintf($$.code,"%s%s   %s = %s >> %s\n",$1.code,$
3.code,$$.var,$1.var,$3.var);}
  | E LS E {$$.var=(char*)malloc(3);sprintf($$.var,"t%d",t);t+=1;$$.cod
e=(char*)malloc(300);sprintf($$.code,"%s%s   %s = %s << %s\n",$1.code,$
3.code,$$.var,$1.var,$3.var);}
  | '(' E ')' {$$.var=(char*)malloc(3);sprintf($$.var,"%s",$2.var);$$.c
ode=(char*)malloc(300);sprintf($$.code,"%s\n",$2.code);}
  | ID  {$$.var=(char*)malloc(3);sprintf($$.var,"%s",$1.var);$$.code=(c
har*)malloc(300);strcpy($$.code,"");}

C : E GT E {$$.code=(char*)malloc(300);sprintf($$.code,"%s >= %s",$1.va
r,$3.var);}
    | E '>' E {$$.code=(char*)malloc(300);sprintf($$.code,"%s > %s",$1.
var,$3.var);}
    | E '<' E {$$.code=(char*)malloc(300);sprintf($$.code,"%s < %s",$1.
var,$3.var);}
    | E LT E {$$.code=(char*)malloc(300);sprintf($$.code,"%s <= %s",$1.
var,$3.var);}
    | E '&' E {$$.code=(char*)malloc(300);sprintf($$.code,"%s & %s",$1.
var,$3.var);}
    | E '|' E {$$.code=(char*)malloc(300);sprintf($$.code,"%s | %s",$1.
var,$3.var);}
    | E LAND E {$$.code=(char*)malloc(300);sprintf($$.code,"%s && %s",$
1.var,$3.var);}
    | E LOR E {$$.code=(char*)malloc(300);sprintf($$.code,"%s || %s",$1
.var,$3.var);}
    | '!' E {$$.code=(char*)malloc(300);sprintf($$.code,"! %s",$2.var);
}
    | E EQ E {$$.code=(char*)malloc(300);sprintf($$.code,"%s == %s",$1.
var,$3.var);}
    | E NQ E {$$.code=(char*)malloc(300);sprintf($$.code,"%s != %s",$1.
var,$3.var);}


%%
int main()
{
    printf("\n\n\nIntermediate code generation PASCAL-
2021 language\n");
    yyparse();
}
int yyerror(char *er)
```

```
{
    printf("\nInvalid character %s\n",er);
    exit(0);
}
```

Learning Outcome:

- I’ve learnt how to implement the syntax checker considering all its grammar rules , operator precedence and syntax for a custom language while execution.
- I’ve learnt the basic syntax of the yacc program and how to implement the grammar rules in c code.
- I’ve learnt how to give use different datatypes for the top of the stack using union.
- I’ve leant how the lex program sends the token based on its syntax and yacc program evaluates the stream of tokens based on the given grammar rules and produces the result.

CS6612 – Compiler Lab

Ex no : 8 **Name : Sreedhar V**

Date : 23.04.2021 **Reg no: 185001161**

Programming Assignment-8 - Code Optimization

Code:

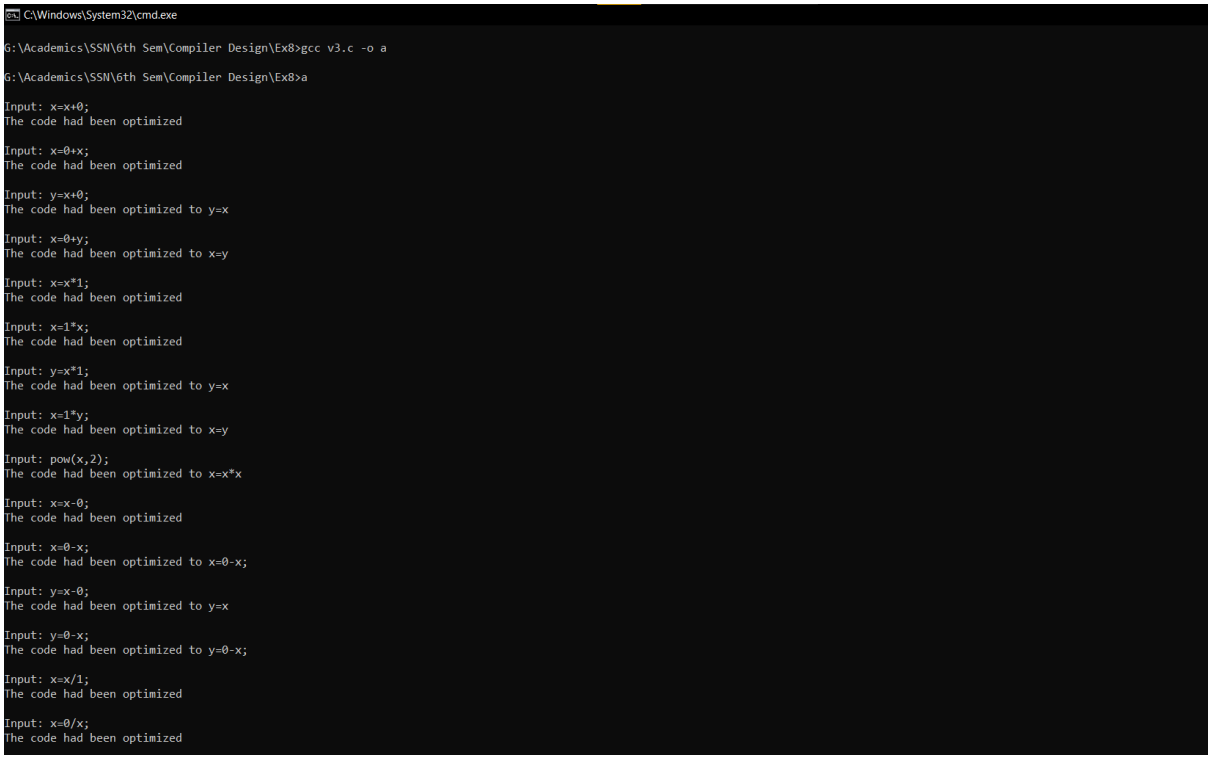
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    FILE *fin = fopen("input.txt", "r");
    char line[100];
    while (fgets(line, 100, fin))
    {
        printf("\nInput: %s", line);
        if (line[3] == '+' && (line[2] == '0' || line[4] == '0'))
        {
            if (line[4] == '0')
            {
                if (line[0] != line[2])
                    printf("The code had been optimized to %c=%c\n", li
ne[0], line[2]);
                else
                    printf("The code had been optimized\n");
            }
            else if (line[2] == '0')
            {
                if (line[0] != line[4])
                    printf("The code had been optimized to %c=%c\n", li
ne[0], line[4]);
                else
                    printf("The code had been optimized\n");
            }
            else
            {
                 printf("The code had been optimized\n");
            }
        }
        else if (line[3] == '*' && (line[2] == '1' || line[4] == '1'))
        {
```

```
            if (line[4] == '1')
            {
                if (line[0] != line[2])
                    printf("The code had been optimized to %c=%c\n", li
ne[0], line[2]);
                else
                    printf("The code had been optimized\n");
            }
            else if (line[2] == '1')
            {
                if (line[0] != line[4])
                    printf("The code had been optimized to %c=%c\n", li
ne[0], line[4]);
                else
                    printf("The code had been optimized\n");
            }
            else
            {
                 printf("The code had been optimized\n");
            }
        }
        else if (line[3] == '-' && (line[2] == '0' || line[4] == '0'))
        {
            if (line[4] == '0')
            {
                if (line[0] != line[2])
                    printf("The code had been optimized to %c=%c\n", li
ne[0], line[2]);
                else
                    printf("The code had been optimized\n");
            }
            else if (line[2] == '0')
            {
                printf("The code had been optimized to %s", line);
            }
            else
            {
                 printf("The code had been optimized\n");
            }
        }
        else if (line[3] == '/' && (line[2] == '0' || line[4] == '1'))
        {
            if (line[4] == '1')
            {
                if (line[0] != line[2])
```

```
                    printf("The code had been optimized to %c=%c\n", li
ne[0], line[2]);
                else
                    printf("The code had been optimized\n");
            }
            else if (line[4] == '1')
            {
                printf("The code had been optimized to %c=0\n", line[0]
);
            }
            else
                    printf("The code had been optimized\n");
        }
        else if (line[0] == 'p' && line[1] == 'o' && line[2] == 'w')
        {
            if (line[6] == '2')
            {
                printf("The code had been optimized to %c=%c*%c\n", lin
e[4], line[4], line[4]);
            }
            else
            {
                 printf("The code had been optimized\n");
            }
        }
        else
        {
                printf("The code had been optimized\n");
        }
    }
    fclose(fin);
    return 0;
}
```

Output:



Learning Outcome:

- I’ve learnt how to implement the code optimizer and analyse the code written in C and tried to optimize the code and memory by removing the useless arithmetic operations.
- I’ve learnt how to remove the unwanted assignment statement which affects the time and space complexity of the program.
- I’ve learnt to convert a C code which has redundant and unoptimized statements into an optimized program.