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;
            }
        }
```

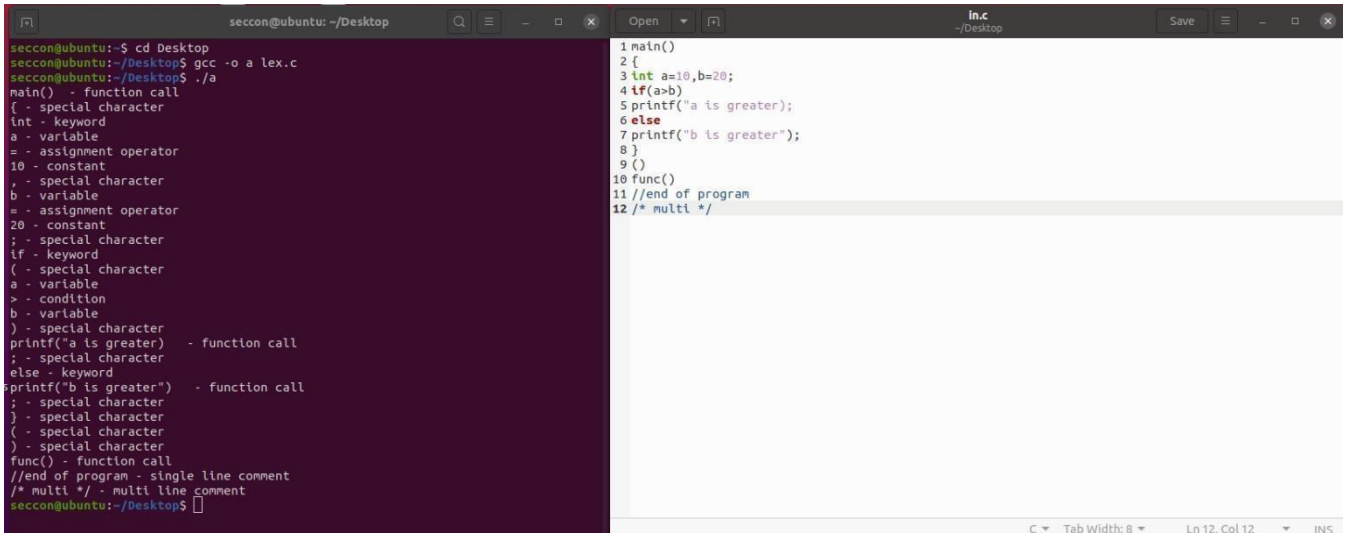
```
            for(int j = 1; j < condition; j++)
            {
                printf("%c", *(tempvar + j));
            }
            printf(" - variable\n");              char* tempvar1 =
strstr(tempvar, "<");              char* tempvar2 = strstr(tempvar,
">");             if(tempvar1
!= NULL) tempvar = 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



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 [-
+]?[09]+(\.[0-9]+)?   char_const \'.\' string_const
\".*\"  identifier [azA-Z_][a-zA-Z0-9_]* function
[a-zAZ_][a-zAZ0-9]*[(].*[)] 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:

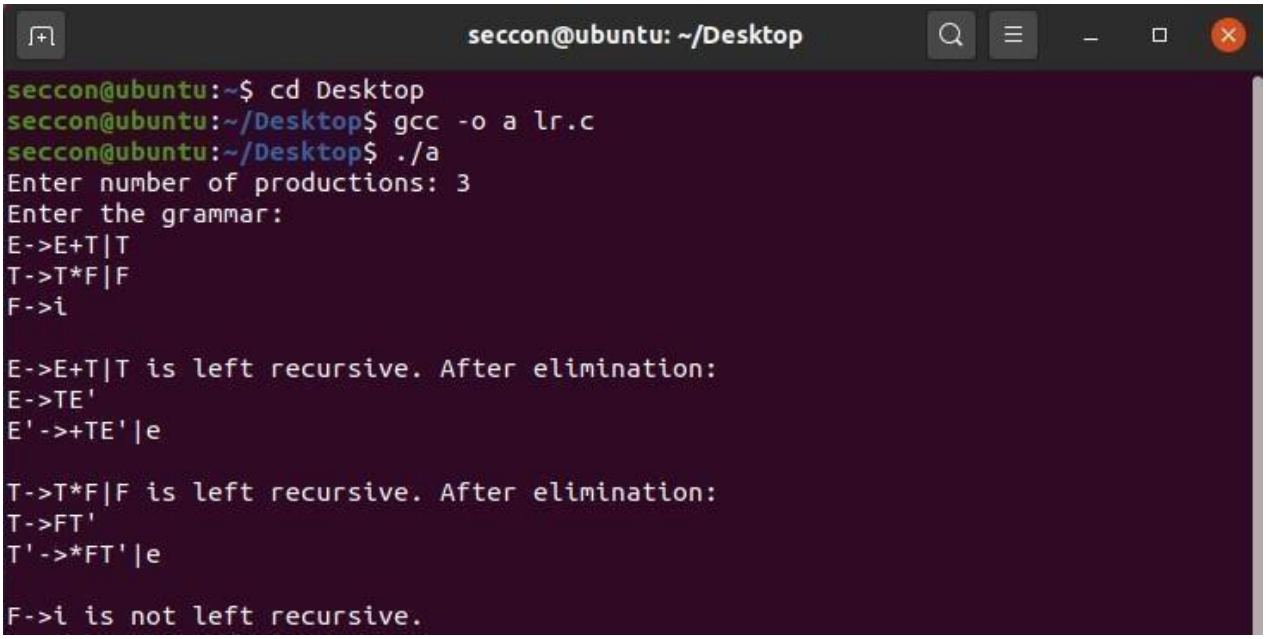
Write a program in C to find whether the given grammar is Left Recursive or not. If it is found to be left recursive, convert the grammar in such a way that the left recursion is removed.

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:



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.

SSN COLLEGE OF ENGINEERING

ASSIGNMENT 4: Recursive Descent Parser using C

-SRINITHYEE S K
185001166

Aim:

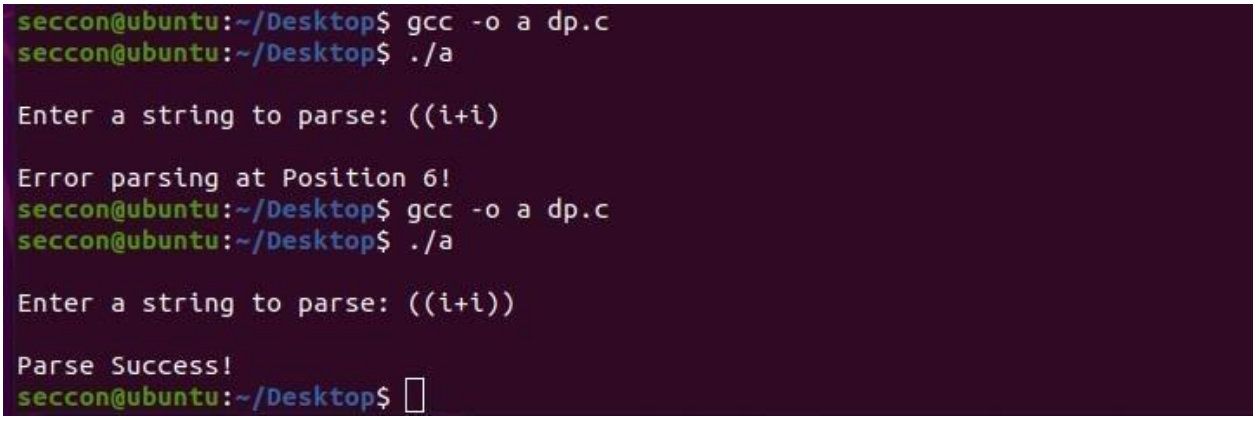
To implement a recursive descent parser using C

Code:

```
#include<stdio.h>
#include<stdlib.h
> void E(); void
Eprime(); void
T(); void
Tprime(); void F();
char s; int pos =
0;
void parse(char c)
{     if(s == c)
{
        s = getchar();
    }
else {
        printf("Error at position %d!\n", pos);
exit(0);
    }
}
void E()
{
    T();
    Eprime();
}
void Eprime()
{     if(s == '+') {
pos++;
parse('+');
T();
        Eprime();
    }
}
void T()
{
    F();
    Tprime();
```

```
}
void Tprime()
{     if(s == '*') {
pos++;
parse('*');
F();
        Tprime();
    }
}
void F()
{     if(s == '(') {
pos++;
parse('(');         E();
pos++;
parse(')');     }
else if(s == 'i') {
pos++;         parse('i');
parse('d');     }
else {
        printf("Error at position %d!\n", pos);
exit(0);
    }
}
int main()
{
    printf("Enter string to parse: ");
s = getchar();     E();
printf("Parse Success!\n");
    return 0;
}
```

OUTPUT:



SSN COLLEGE OF ENGINEERING

Learning Outcome:

- Learnt the working of Recursive Descent Parser
- Understood why it doesn’t support Left Recursive Grammars
- Successfully implemented a Recursive Descent Parser using c using return handling and recursion.

SSN COLLEGE OF ENGINEERING

ASSIGNMENT 5: Implementation of Desk Calculator using Yacc Tool

-SRINITHYEE S K
185001166

Aim:

To implement a Desk Calculator using Yacc Tool

Code:

Calculator.l

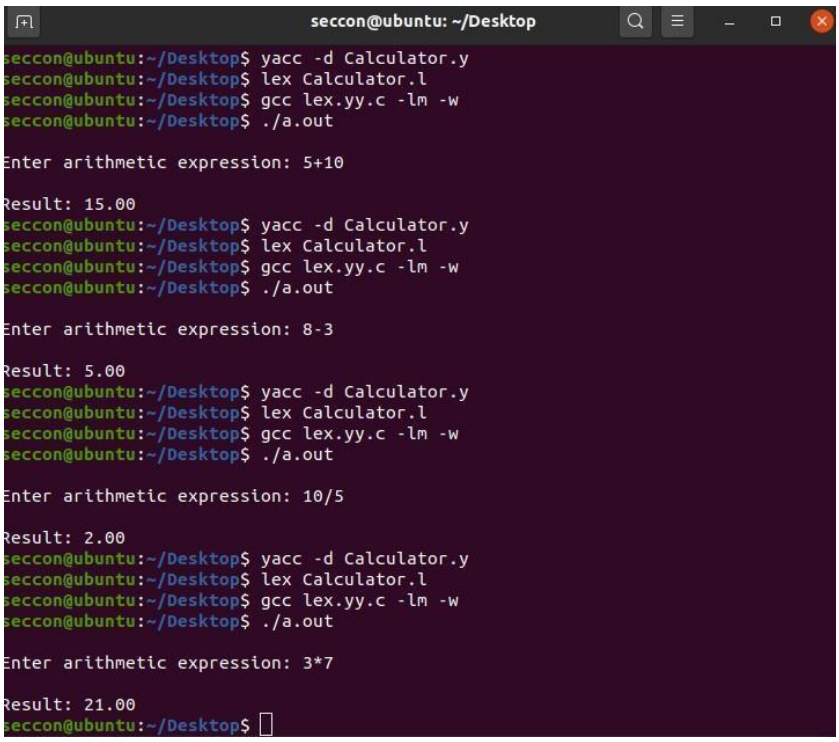
```
%{
#include<stdio.h>  #include<stdlib.h>
#include<string.h> #include
"y.tab.h" extern int
yylval;
%}
%% [0-9]+ {    yylval =
atoi(yytext);
return INTEGER;
}
(" "|"\t") { }
("+"|"-"|"*"|"/"|"^"|"("|")"|"\n") { return *yytext; }
. {   char err[25];
  sprintf(err, "Invalid character: %s\n", yytext);
yyerror(err);
}
```

Calculator.y

```
%{
#include<stdio.h>
#include<stdlib.h>
#include<math.h> int
yylex(void); #include
"y.tab.h"
%}
%token INTEGER
%% program: line
program
| line  line: expr '\n' {
printf("%d\n", $1); }
expr: expr '+' mulex { $$ = $1 + $3; }
```

```
| expr '-' mulex { $$ = $1 - $3; }
    | mulex { $$ = $1; }  mulex: mulex
'*' powex { $$ = $1 * $3; }       |
mulex '/' powex { $$ = $1 / $3; }
     | powex { $$ = $1; }  powex: powex '^'
term { $$ = pow($1, $3); }       | term {
$$ = $1; } term: '(' expr ')' { $$ = $2; }
    | INTEGER { $$ = $1; }
%%
int yyerror(char* s)
{    fprintf(stderr, "%s\n",
s);
return 0; } int yywrap() {   return
1; } int main() {   yyparse();
return 0; }
```

OUTPUT:



Learning Outcome:

- Learnt about Yacc Parser Generator and that it is LALR(1) parser.
- Learnt to visualize parser's working using scanner.
- Learnt to integrate Yacc and Lex in one file.
- Successfully implemented a basic calculator using Yacc tool, understanding it's syntax.

SSN COLLEGE OF ENGINEERING

ASSIGNMENT 6: IMPLEMENTATION OF SYNTAX CHECKER USING YACC TOOL

Aim:

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

Assignment statement

Conditional statement

Looping statement

Code:

SyntaxCheck.y

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

%token  NUM ASSIGN ID
%token  RELOP LOGIC ARITH INCDEC
%token  IF ELIF ELSE
%token  FOR WHILE

%%
Lines   :   Block Lines
        |   Block
        ;

Block   :   Loop '{' Block
        |   ConStmt '{' Block
        |   Expr ';'
        |   '}'
        ;

Loop    :   FOR '(' Expr ';' Condns ';' Expr ')'
        |   FOR '(' ';' Condns ';' ')'
        |   WHILE '(' Condns ')'
        ;
```

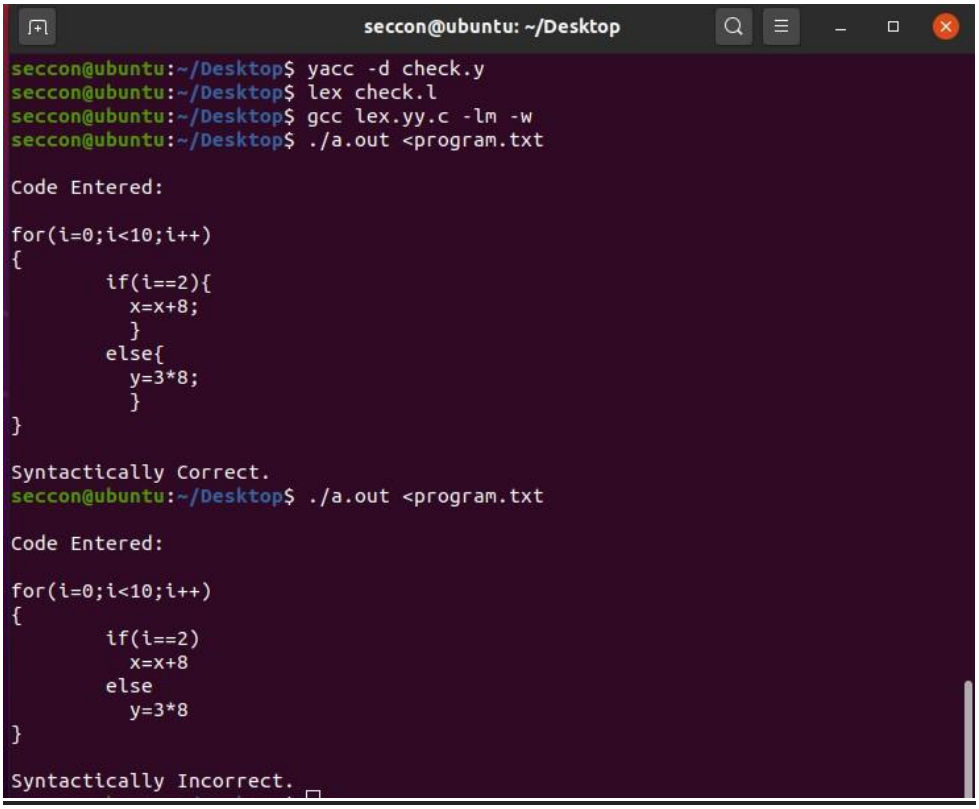
```
ConStmt :   IF '(' Condns ')'
        |   ELIF '(' Condns ')'
        |   ELSE
        ;

Condns  :   Condn LOGIC Condns
        |   Condn
        ;

Condn   :   ID RELOP ID
        |   ID RELOP NUM
        |   ID
        ;

Expr    :   Init
        |   ID ASSIGN ID ARITH ID
        |   ID ASSIGN ID ARITH NUM
        |   ID ASSIGN NUM ARITH NUM
        |   ID INCDEC
        |   INCDEC ID
        ;

Init    :   ID ASSIGN Init
        |   ID ASSIGN ID
        |   ID ASSIGN NUM
        ;
%%

int yyerror(char *s){
flag = 1;
   //fprintf(stderr, "%s\n", s);
return 1;
}

int main(void){

   printf("\nCode Entered:\n\n");
system("cat program.txt");     yyparse();

if(flag){
        printf("\nSyntactically Incorrect.\n");
   }

   else{
        printf("\nSyntactically Correct.\n");
   }
```

```
    return 0;
}
```

SyntaxCheck.l

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

assign                    ("=")     relop
("=="|"!="|">="|"<="|"<"|">")     arithop
("+"|"-"|"/"|"%"|"*") incdec      ("++"|"--
") logical     ("||"|"&&")
identifier  [a-zA-Z_][a-zA-Z0-9_]*


%%

[0-9]+          {return NUM;}
{assign}        {return ASSIGN;}
{relop}         {return RELOP;}
{logical}       {return LOGIC;}
{arithop}       {return ARITH;}
{incdec}        {return INCDEC;}
"if"            {return IF;}
"else if"       {return ELIF;}
"else"          {return ELSE;}
"for"           {return FOR;}
"while"         {return WHILE;}
{identifier}    {return ID;}


[ \t]           {;}
[\n]            {;}
.               {return *yytext;}

%%

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

Output:



Learning Outcome:

- Understood how to construct grammar for a program syntax checker.
- Realised that Yacc is LALR(1) parser
- Successfully implemented a syntax checker using Yacc parser

SSN COLLEGE OF ENGINEERING

ASSIGNMENT 7: INTERMEDIATE CODE GENERATION USING LEX AND YACC

Aim:

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 program written using declaration, conditional and assignment statements in new language **Pascal-2021**

Code:

Code.txt

```
i: integer=1;
a: real=4.2;
b: char='c';
c: integer=63;
d: real=24.88;
x: integer;

begin
   if (i>0) then
      x=a+b*c/d;
   else
      x=a*b*c-d;
   end if
end
```

Tac.y

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

   int yylex(void);
   int yyerror(char *);
   int yywrap();

   int vars = 0, labels = 0;

   struct info{
      char *var;
      char *code;
      int intval;
      float floatval;
      char charval;
   };

   typedef struct info node;

   node *makeNode(){
      //creating a new node to store intermediate code

      node *n = (node *)malloc(sizeof(node));
      n->intval = 0;
      n->floatval = 0;
      n->charval = 0;
      n->var = (char *)malloc(50 * sizeof(char));
      n->code = (char *)malloc(5000 * sizeof(char));
```

```
        return n;
    }
%}

/*Declaration of tokens and precedence*/
%token BGN END IF THEN ELSE INT CHAR
%token REAL CHCONST VAR NUM RELOP ADDOP MULOP

/*Increasing precedence*/
%right MULOP
%left ADDOP

/*Declaration of the types that YYSTYPE can take with the union*/
%union{
    int intval;
    float floatval;
    char ch;
    char *str;
    struct info *Node;
}

/*Declaring types for the tokens*/
%type<str>      VAR RELOP ADDOP MULOP
%type<intval>   NUM
%type<floatval> REAL
%type<ch>       CHCONST
%type<Node>     Program Structure Declarations Statements
%type<Node>     Declaration Type Value Statement
%type<Node>     Assignment Conditional Condition Expr
%type<Node>     E T F

%%

Program         :   Structure{
                    printf("\nL%-5d - |\n%s", 0, $$->code);
                }
                ;

Structure       :   Declarations BGN Statements END{
                    sprintf($$->code, "%s%10s\n%s", $1->code, "|", $3->code);
                }
                ;

Declarations    :   Declaration Declarations{
                    $$ = makeNode();
                    sprintf($$->code, "%s%s", $1->code, $2->code);
                }

                |   Declaration{
                    $$ = $1;
                }
                ;

Declaration     :   VAR ':' Type ';' {
                    $$ = makeNode();
```

```
            sprintf($$->code, "%10s %-5s := %s\n", "|", $1, $3->var);
          }

        | VAR ':' Type '=' Value ';'{
            $$ = makeNode();
            sprintf($$->code, "%10s %-5s := %s\n", "|", $1, $5->var);
          }
        ;

Type    : INT{
            $$ = makeNode();
            $$->intval = 0;
            sprintf($$->var, "%d", 0);
            sprintf($$->code, "");
          }

        | REAL{
            $$ = makeNode();
            $$->floatval = 0.0;
            sprintf($$->var, "%.2f", 0.0);
            sprintf($$->code, "");
          }

        | CHAR{
            $$ = makeNode();
            $$->charval = 0;
            sprintf($$->var, "%s", "NULL");
            sprintf($$->code, "");
          }
        ;

Value   : NUM{
            $$ = makeNode();
            $$->intval = $1;
            sprintf($$->var, "%d", $1);
            sprintf($$->code, "");
          }

        | REAL{
            $$ = makeNode();
            $$->floatval = $1;
            sprintf($$->var, "%.2f", $1);
            sprintf($$->code, "");
          }

        | CHCONST{
            $$ = makeNode();
            $$->intval = $1;
            sprintf($$->var, "%c", $1);
            sprintf($$->code, "");
          }
        ;

Statements  : Statement Statements{
            $$ = makeNode();
```

```
            sprintf($$->code, "%s%s", $1->code, $2->code);
          }

        | Statement{
            $$ = $1;
          }
        ;

Statement    : Assignment {
            $$ = $1;
          }

        | Conditional{
            $$ = $1;
          }
        ;

Assignment    : VAR '=' Expr ';'{
            $$ = makeNode();
            char tac[100];
            sprintf($$->var, "%s", $1);
            sprintf(tac, "%10s %-5s := %s\n", "|", $$->var, $3->var);
            sprintf($$->code, "%s%s", $3->code, tac);
          }
        ;

Expr        : E{
            $$ = $1;
          }
        ;

E           : T MULOP E{
            $$ = makeNode();
            char tac[100];
            sprintf($$->var, "x%d", ++vars);
            sprintf(tac, "%10s %-5s := %s %s %s\n", "|", $$->var, $1->var, $2, $3->var);
            sprintf($$->code, "%s%s%s", $1->code, $3->code, tac);
          }

        | T{
            $$ = $1;
          }

        | F{
            $$ = $1;
          }
        ;

T           : T ADDOP F{
            $$ = makeNode();
            char tac[100];
            sprintf($$->var, "x%d", ++vars);
            sprintf(tac, "%10s %-5s := %s %s %s\n", "|", $$->var, $1->var, $2, $3->var);
            sprintf($$->code, "%s%s%s", $1->code, $3->code, tac);
          }
```

```
            |  F{
                  $$ = $1;
               }
            ;

F            :  VAR{
                  $$ = makeNode();
                  sprintf($$->var, "%s", $1);
                  sprintf($$->code, "");
               }

            |  NUM{
                  $$ = makeNode();
                  $$->intval = $1;
                  sprintf($$->var, "%d", $1);
                  sprintf($$->code, "");
               }

            |  REAL{
                  $$ = makeNode();
                  $$->floatval = $1;
                  sprintf($$->var, "%.2f", $1);
                  sprintf($$->code, "");
               }

            |  CHCONST{
                  $$ = makeNode();
                  $$->charval = $1;
                  sprintf($$->var, "'%c'", $1);
                  sprintf($$->code, "");
               }
            ;

Conditional    :  IF '(' Condition ')' THEN Statements ELSE Statements END IF{
                  $$ = makeNode();
                  int condnBlock = ++labels;
                  int endBlock = ++labels;
                  sprintf($$->code, "%s%10s if %s then goto L%d\n%s%10s goto L%d\n%10s\nL%-5d -
|\n%s%10s\nL%-5d - |\n", $3->code, "|", $3->var, condnBlock, $8->code, "|", endBlock, "|", condnBlock, $6->code,
"|", endBlock);
                                                }
            ;

Condition      :  Expr RELOP Expr{
                  $$ = makeNode();
                  char tac[100];
                  sprintf($$->var, "%s%s%s", $1->var, $2, $3->var);
                                          sprintf($$->code, "%s%s", $1->code, $3->code);
               }
            ;
%%

int yyerror(char* str){
   printf("\n%s", str);
```

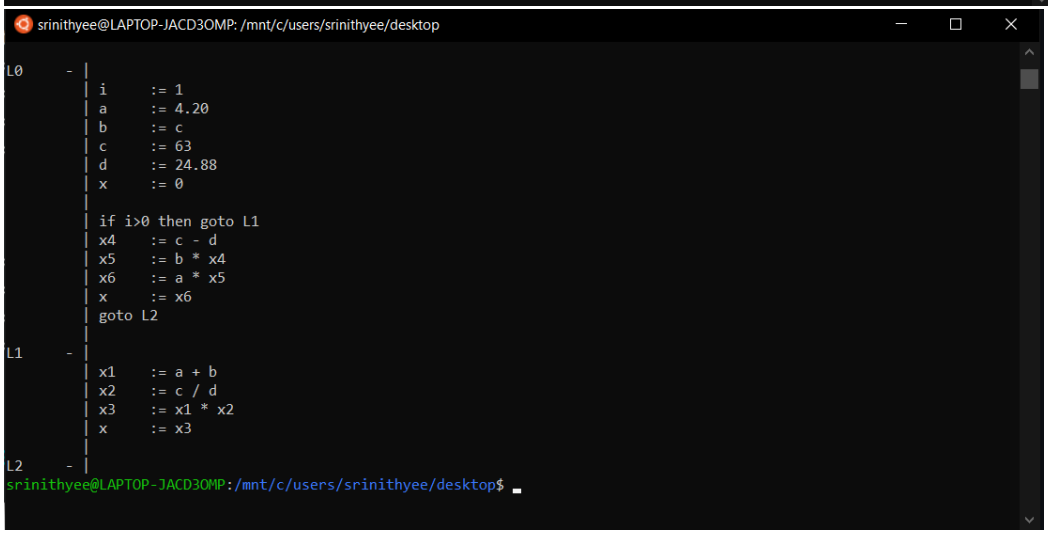
```
    return 0;
}

int yywrap(){
    return 1;
}

int main(){
    printf("\n\t\tIntermediate Code Generation\n");
    printf("\nYour Code:\n\n");
    system("cat Code.txt");
    printf("\n\nThree Address Code:\n");

    yyparse();
    return 0;
}
```

Tac.l

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include "y.tab.h"
%}

term    ([a-zA-Z\_][a-zA-Z\_0-9]*)
num     ([0-9]+)
real    {num}\.{num}
relop   ("<"|"<="|">"|">="|"=="|"!=")
addop   ("+"|"-")
mulop   ("*"|"/"|"%")
spl     (";"|","|"{"|"}"|"("|")"|"="|"&"|"|"|"!"|":")

%%
"begin"     {return BGN;}
"end"       {return END;}
"if"        {return IF;}
"then"      {return THEN;}
"else"      {return ELSE;}
"integer"   {return INT;}
"char"      {return CHAR;}
"real"      {return REAL;}
['].[']     {yylval.ch = yytext[1]; return CHCONST;}
{term}      {yylval.str = strdup(yytext); return VAR;}
{real}      {yylval.floatval = atof(yytext); return REAL;}
{num}       {yylval.intval = atoi(yytext); return NUM;}
{relop}     {yylval.str = strdup(yytext); return RELOP;}
{mulop}     {yylval.str = strdup(yytext); return MULOP;}
{addop}     {yylval.str = strdup(yytext); return ADDOP;}
{spl}       {return *yytext;}
[ \t\n]+    {;}

.           {char errmsg[100];
             strcpy(errmsg, "Invalid Character: ");
```

```
        strcat(errmsg, yytext);
        strcat(errmsg, "\n");
        yyerror(errmsg);}

%%
```

Output Snapshots:



Learning Outcome:

- Understood the working of a Yacc Parser Generator
- The purpose of %union was understood.
- The fact that precedence can be given only to tokens in Yacc has been understood.
- I was successfully able to implement Intermediate Code Generator using Yacc and Lex.

SSN COLLEGE OF ENGINEERING