RAMAIAH UNIVERSITY OF APPLIED SCIENCES

ASSIGNMENT - 2

Course Code	19CSC305A
Course Name	Compilers
Programme	B. Tech
Department	CSE
Faculty	FET

Name of the Student	K Srikanth
Reg. No	17ETCS002124
Semester/Year	5th Semester/ 3rd Year
Course Leader/s	Mr. Hari Krishna S. M.

Declaration Sheet				
Student Name	K Srikanth			
Reg. No	17ETCS002124			
Programme	B. Tech		Semester/Year	5th / 3rd
Course Code	19CSC305A			
Course Title	Compilers			
Course Date	14/09/2020	to	16/02/2021	
Course Leader	Mr. Hari Krishna S. M.			

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

RAMAIAH UNIVERSITY OF APPLIED SCIENCES

Faculty of Engineering and Technology			
Ramaiah University of Applied Sciences			
Department	Computer Science and Engineering	Programme	B. Tech in Computer Science and Engineering
Semester/Batch	05th /2018		
Course Code	19CSC305A	Course Title	Compilers
Course Leader	Mr. Hari Krishna S. M. & Ms. Suvidha		

Assignment					
Register No.		17ETCS002124	**Name of the Student**	K Srikanth	
Sections		**Marking Scheme**	**Marks**		
			Max Marks	**First Examiner Marks**	**Moderator**
Part A 1					
	A 1.1	Identification and grouping of Tokens	05		
	A 1.2	Implementation in *Lex*	03		
	A 1.3	Design of Context Free Grammar	05		
	A 1.4	Implementation in *Yacc*	07		
	A 1.5	Results and Comments	05		
		Part-A 1 Max Marks	**25**		
		Total Assignment Marks	**25**		

Course Marks Tabulation				
Component- CET B Assignment	**First Examiner**	**Remarks**	**Second Examiner**	**Remarks**
A.1				
Marks (out of 25)				

Signature of First Examiner **Signature of Moderator**

RAMAIAH UNIVERSITY OF APPLIED SCIENCES

Please note:

1. Documental evidence for all the components/parts of the assessment such as the reports, photographs, laboratory exam / tool tests are required to be attached to the assignment report in a proper order.
2. The First Examiner is required to mark the comments in RED ink and the Second Examiner's comments should be in GREEN ink.
3. The marks for all the questions of the assignment have to be written only in the **Component – CET B: Assignment** table.
4. If the variation between the marks awarded by the first examiner and the second examiner lies within +/- 3 marks, then the marks allotted by the first examiner is considered to be final. If the variation is more than +/- 3 marks then both the examiners should resolve the issue in consultation with the Chairman BoE.

Assignment -2

Instructions to students:

1. The assignment consists of **1** questions: Part A – **1** Question.
2. Maximum marks is **25**.
3. The assignment has to be neatly word processed as per the prescribed format.
4. The maximum number of pages should be restricted to **15**.
5. The printed assignment must be submitted to the course leader.
6. **Submission Date: 16th Jan 2021**
7. **Submission after the due date is not permitted.**
8. **IMPORTANT**: It is essential that all the sources used in preparation of the assignment must be suitably referenced in the text.
9. Marks will be awarded only to the sections and subsections clearly indicated as per the problem statement/exercise/question

Preamble:

The aim of this course is to train the students in the design and implementation of compilers and various components of a compiler, including a scanner, parser, and code generator. The students are exposed to GNU compiler, construction tools and their application. Students are trained to design and implement a compiler for a simple language.

Part A

Q-A1.1)

Grouping of tokens was similar to what we did in our **previous assignment there were more keywords to add so that it can be recognize it as tokens.**

Lex Implementation

```
%{
    int yylex();
    void yyerror(char *s);
%}
alphabet [a-zA-Z] // substitute definitions
digit [0-9]
underscore \_
whitespace [ \t\r\f\v]+
%x PREPROCESSING
%x MULTILINECOMMENT
%x SINGLELINECOMMENT
%%
<<EOF>> {exit(0);}
^"#include" {BEGIN PREPROCESSING; printf("%10s PREPROCESSING\n",yytext); }
<PREPROCESSING>{whitespace} ;
<PREPROCESSING>"<"[^<>\n]*">" {BEGIN INITIAL;}
<PREPROCESSING>\"[^<>\n]*\" {BEGIN INITIAL;}
<PREPROCESSING>"\n" {yylineno++; BEGIN INITIAL;}
<PREPROCESSING>. {yyerror("Mistake in Header");}
"/*" {BEGIN MULTILINECOMMENT; printf("%10s MULTILINECOMMENT\n",yytext); }
<MULTILINECOMMENT>.|{whitespace} ;
<MULTILINECOMMENT>\n {yylineno++;}
<MULTILINECOMMENT>"*/" {BEGIN INITIAL;}
<MULTILINECOMMENT>"/*" {yyerror("Comment format invalid");}
"//" {BEGIN SINGLELINECOMMENT; printf("%10s SINGLELINECOMMENT\n",yytext); }
<SINGLELINECOMMENT>\n {yylineno++; BEGIN INITIAL;}
<SINGLELINECOMMENT>. ;
\+ {printf("%10s PLUS\n",yytext);}
\- {printf("%10s MINUS\n",yytext);}
\* {printf("%10s MULT\n",yytext);}
\/ {printf("%10s DIV\n",yytext);}
\^ {printf("%10s POW\n",yytext);}
"%" {printf("%10s MOD ARITHMETIC OPERATOR\n",yytext);}
"--" {printf("%10s DECREMENT ARITHMETIC OPERATOR\n",yytext);}
"++" {printf("%10s INCREMENT ARITHMETIC OPERATOR\n",yytext);}
">" {printf("%10s GT COMPARISION OPERATOR\n",yytext);}
"<" {printf("%10s LT COMPARISION OPERATOR\n",yytext);}
">=" {printf("%10s GT_EQ COMPARISION OPERATOR\n",yytext);}
"<=" {printf("%10s LT_EQ COMPARISION OPERATOR\n",yytext);}
"==" {printf("%10s EQUAL COMPARISION OPERATOR\n",yytext);}
"!=" {printf("%10s NOT_EQUAL COMPARISION OPERATOR\n",yytext);}
"||" {printf("%10s OR LOGICAL OPERATOR\n",yytext);}
"&&" {printf("%10s AND LOGICAL OPERATOR\n",yytext);}
"=" {printf("%10s EQUAL OPERATOR\n",yytext);}
"!" {printf("%10s NOT LOGICAL OPERATOR\n",yytext);}
main {printf("%10s MAIN\n",yytext);}
printf {printf("%10s PRINTF KEYWORD\n",yytext);}
scanf {printf("%10s SCANF KEYWORD\n",yytext);}
return {printf("%10s RETURN TYPE\n",yytext);}
if {printf("%10s IF CONTROL STATEMENT\n",yytext);}
else {printf("%10s ELSE CONTROL STATEMENT\n",yytext);}
while {printf("%10s WHILE LOOPING CONSTRUCT\n",yytext);}
for {printf("%10s FOR LOOPING CONSTRUCT\n",yytext);}
switch {printf("%10s SWITCH STATEMENT\n",yytext);}
case {printf("%10s CASE STATEMENT\n",yytext);}
default {printf("%10s DEFAULT STATEMENT\n",yytext);}
break {printf("%10s BREAK STATEMENT\n",yytext);}
int {printf("%10s INT DATATYPE\n",yytext);}
float {printf("%10s FLOAT DATATYPE\n",yytext);}
char {printf("%10s CHAR DATATYPE\n",yytext);}
";" {printf("%10s SEMICOLON\n",yytext);}
[\(\)\{\}\,\[\]] {printf("%10s SEPARATOR\n",yytext);}
\".*\" {printf("%s STRING LITERAL",yytext);}
([_a-zA-Z]+[_a-zA-Z0-9]*) {printf("%10s VARIABLE\n",yytext);}
{digit}+ { printf("%10s INT VALUE\n",yytext);}
{digit}+[\.]{digit}+ { printf("%10s FLOAT VALUE\n",yytext);}
\n ;
{whitespace} ;
. {printf("%10s CHAR_LITERAL\n",yytext);}
%%
int yywrap(){ return 1;}
void yyerror (char *s) {fprintf (stderr, "%s at line %d\n", s, yylineno);}
int main()
{
    yyin = fopen("InputFile.c", "r");
    if(yyin==NULL) printf("\nError\n");
    else{
    printf("\Started Tokenizing\n"); printf("17ETCS002124 K Srikanth\n");yylex();}
    fclose(yyin);
    return 0;
}
```

Grouping of Tokens

Input C File

```c
/*
This is a Multiline Comment
*/
#include<stdio.h>
void main(){
int a[100][20];
float r[30];
int var=23,b=2124,c=0044;
c=2/5+2-(4^3+(24%3))+2*15/3+3;
char h='s';
float d;
d=23.01;
c++;
scanf("%d",&c);
scanf("%f",&b);
printf("Enter c");
printf("%d",c);
printf("%s",h);
printf("%f",d);
if(b<4 || b==4){
}
if(c==4 && b>2){
}
else{
}
while(var!=3){
}
for(var=0;var<=3;var++){
}
} // This is a Single Line Comment
```

Tokenizing Input File

```
Started Tokenizing
17ETCS002124 K Srikanth
       /* MULTILINECOMMENT
  #include PREPROCESSING
     void VARIABLE
     main MAIN
        ( SEPARATOR
        ) SEPARATOR
        { SEPARATOR
      int INT DATATYPE
        a VARIABLE
        [ SEPARATOR
      100 INT VALUE
        ] SEPARATOR
        [ SEPARATOR
       20 INT VALUE
        ] SEPARATOR
        ; SEMICOLON
    float FLOAT DATATYPE
        r VARIABLE
        [ SEPARATOR
       30 INT VALUE
        ] SEPARATOR
        ; SEMICOLON
      int INT DATATYPE
      var VARIABLE
        = EQUAL OPERATOR
```

Figure 1 Lex Program Generating Tokens

Figure 2 Lex Program Generating Tokens (Continued)

```
       23 INT VALUE
        , SEPARATOR
        b VARIABLE
        = EQUAL OPERATOR
     2124 INT VALUE
        , SEPARATOR
        c VARIABLE
        = EQUAL OPERATOR
     0044 INT VALUE
        ; SEMICOLON
        c VARIABLE
        = EQUAL OPERATOR
        2 INT VALUE
        / DIV
        5 INT VALUE
        + PLUS
        2 INT VALUE
        - MINUS
        ( SEPARATOR
        4 INT VALUE
        ^ POW
        3 INT VALUE
        + PLUS
        ( SEPARATOR
       24 INT VALUE
        % MOD ARITHMETIC OPERATOR
        3 INT VALUE
        ) SEPARATOR
        ) SEPARATOR
```

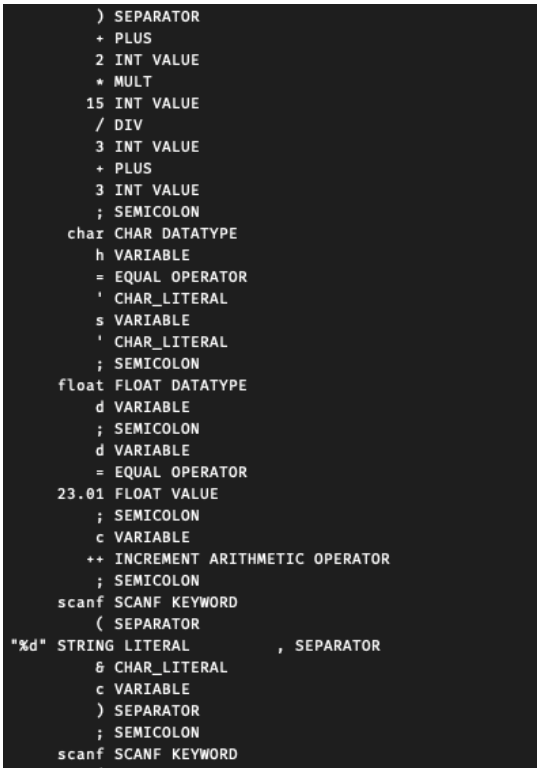

Figure 3 Lex Program Generating Tokens (Continued)

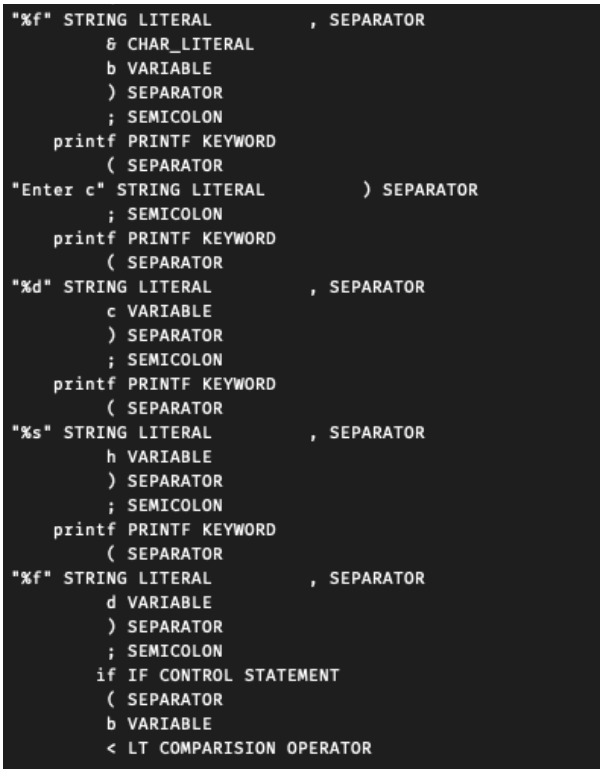

Figure 4 Lex Program Generating Tokens (Continued)

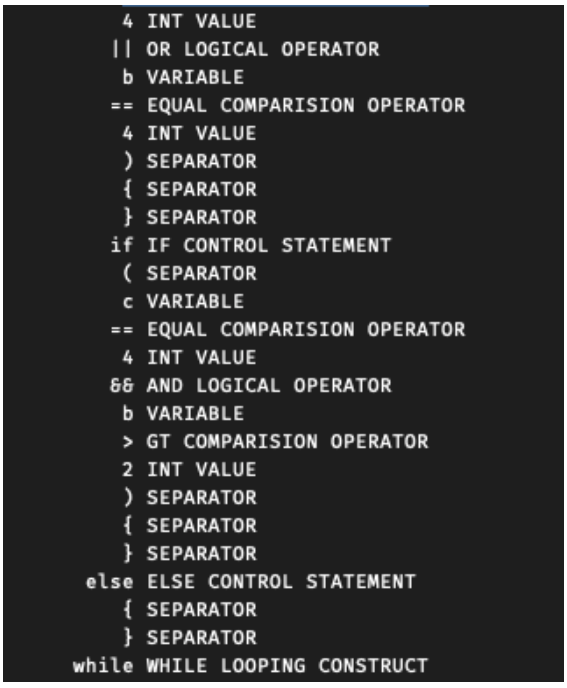

Figure 5 Lex Program Generating Tokens (Continued)

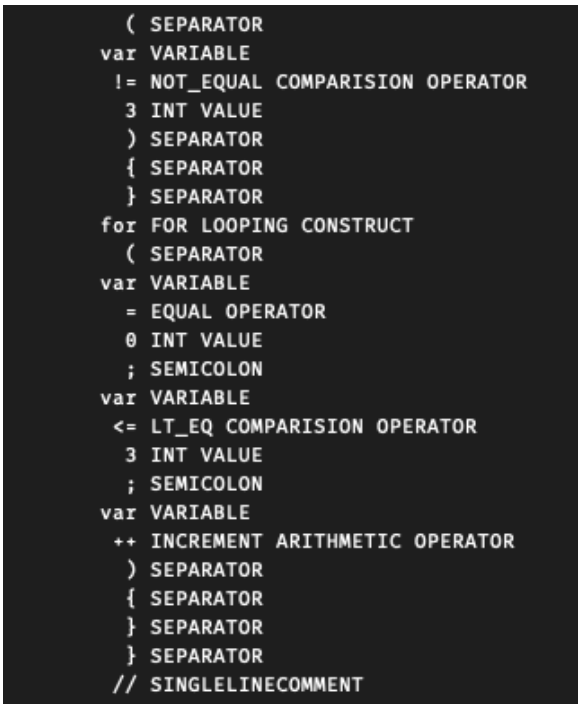

Figure 6 Lex Program Generating Tokens (Continued)

Q-A1.2)

Implementation of Lex Program for Yacc Parser

Start of declaration.

```
%{
#include "Yacc_File.tab.h"
int yylineno=1;
extern void yyerror (char *s);
%}
// K Srikanth 17ETCS002124
alphabet [a-zA-Z]
digit [0-9]
underscore _
whitespace [ \t\r\f\v] +
%x PREPROCESSING
%x MULTILINECOMMENT
%x SINGLELINECOMMENT
%%
```

yytext gives the matched input stream, this can be passed to semantic values of the tokens of any union type declared in YACC program, with the help of yylval shared between lexer and parser.

Regular expression part one: for end of file, preprocessing section, multiline comment and single line comment.

```
^"#include"                                  {BEGIN PREPROCESSING;}
<PREPROCESSING>{whitespace}                   ;
<PREPROCESSING>"<"[^<>\n]*">"                {BEGIN INITIAL;}
<PREPROCESSING>\"[^<>\n]*\"                  {BEGIN INITIAL;}
<PREPROCESSING>"\n"                          {yylineno++; BEGIN INITIAL;}
<PREPROCESSING>.                             {yyerror("Mistake in Header");}

"/*"                                         {BEGIN MULTILINECOMMENT;}
<MULTILINECOMMENT>.|{whitespace}             ;
<MULTILINECOMMENT>\n                         {yylineno++;}
<MULTILINECOMMENT>"*/"                       {BEGIN INITIAL;}
<MULTILINECOMMENT>"/*"                       {yyerror("Comment format invalid");}
"//"                                         {BEGIN SINGLELINECOMMENT;}
<SINGLELINECOMMENT>\n                       {yylineno++; BEGIN INITIAL;}
<SINGLELINECOMMENT>.                        ;
```

Regular expressions Type formats for input and output statements

```
\"%d\"          {return INT_FORMAT;}
\"%f\"          {return FLOAT_FORMAT;}
\"%s\"          {return STRING_FORMAT;}
\".*\"          {yylval.string=strdup(yytext);return STRING_LITERAL;}
[ \t]+              ;
```

Regular expressions for arithmetic operators

```
\+                  {return PLUS;}
\-                  {return MINUS;}
\*                  {return MULT;}
\/                  {return DIV;}
\^                  {return POW;}
\%                  {return MOD;}
"--"                {return DECREMENT;}
"++"                {return INCREMENT;}
```

Regular expressions for comparison operators

```
">"                                 {return GT;}
"<"                                 {return LT;}
">="                                {return GT_EQ;}
"<="                                {return LT_EQ;}
"=="                                {return EQUAL;}
"!="                                {return NOT_EQUAL;}
```

Regular expressions for Logical operators

```
"||"                                {return OR;}
"&&"                                {return AND;}
"!"                                 {return NOT;}
```

Regular expressions for Keywords

```
return                { return RETURN;}
break                 { return BREAK;}
main                  { return MAIN;}
if                        { yylval.string=strdup(yytext); return IF;}
else                    { yylval.string=strdup(yytext); return ELSE;}
while                 { yylval.string=strdup(yytext); return WHILE;}
for                     { yylval.string=strdup(yytext); return FOR;}
int                 { yylval.string=strdup(yytext); return INT;}
float                   { yylval.string=strdup(yytext); return FLOAT;}
char                   { yylval.string=strdup(yytext); return CHAR;}
void                    { yylval.string=strdup(yytext); return VOID;}
printf                { return PRINTF;}
scanf                 { return SCANF;}
```

Regular expressions for identifiers, numbers and characters

```
({underscore}|{alphabet})({underscore}|{alphabet}|{digit})*  {
  yylval.string=strdup(yytext);
  return VAR;
  }

{digit}+                    {
  yylval.int_val=atoi(yytext);
  return INT_VALUE;
  }

{digit}+[\.]{digit}+            {
  yylval.float_val=atof(yytext);
  return FLOAT_VALUE;
 }

(\"{alphabet}\"|'{alphabet}')     {
  yylval.string=strdup(yytext);
  return CHAR_LITERAL;
  }

. { return yytext[0];}

%%
```

This Lex program does not have a main () or call to yylex () as it will be called from yyparse function. Before we start programming lets define a simple symbol table to record the name, type and value of the identifiers or variables declared. The symbol table can be built as an array of structures. We will also define some utility functions to modify data in symbol table.

Q-A1.3)

Let's define grammar solving our objectives one by one.

- **Declare and assign values to variables of three data-types, INT, FLOAT and CHAR**

  To declare a variable of any one of the types above, the data-type should precede the variable list.

  DECL: TYPE VARLIST

Variable list can be a single variable or a variable followed by variable list separated by comma and NONE represents nothing.

VARLIST: VAR |VAR , VARLIST

Type can of three types as discussed.

TYPE: INT | FLOAT | CHAR

A variable during declaration can be initialized with the same TYPE values, VARLIST could be written as

VARLIST: VAR EQUAL_PART|VAR EQUAL_PART , VARLIST

Variable can be assigned to int, float, char values, another variable, int expression, float expression or

be not assigned to anything, so EQUAL_PART can become

EQUAL_PART: = VAR

| = INT_VALUE | = FLOAT_VALUE| = CHAR_LITERAL

|A_INTEXP | A_FLOATEXP| NONE

- **Declare 1D and 2D arrays of the three data-types.**

To declare a 1 or 2 dimensional array, we can just add it in the VARLIST, 1d array will have size in INT_VALUE, and a 2d array will have order of rows × columns, rows and columns are INT_VALUEs as well.

VARLIST: VAR EQUAL_PART | VAR EQUAL_PART , VARLIST

|VAR [INT_VALUE] | VAR [INT_VALUE] [INT_VALUE]

Type for this array would be taken from the type of the VARLIST, which will make the declaration complete.

DECL: TYPE VARLIST

- **Evaluate Arithmetic expressions containing integer and floating variables or values, and store them in a variable.**

Arithmetic expressions can be of two types, integer and floating operations

Let them be **A_INTEXP and A_FLOATEXP**

Let's fix the associativity and precedence before solving the expressions,

+, -,*, / and % are left associative while, ^ (POWER) is right associative. Writing in the ascending order of operator precedence,

+ - * / % ^,

+ having the least precedence and ^ having the most.

As we can process Ambiguous grammar with precedence rules and associativity in YACC, let's define ambiguous grammar for our expression evaluation. The end terminals would be either variable or integer.

A_INTEXP: A_INTEXP + A_INTEXP | A_INTEXP - A_INTEXP | A_INTEXP * A_INTEXP | A_INTEXP / A_INTEXP

| A_INTEXP % A_INTEXP

| A_INTEXP ^ A_INTEXP | VAR | INT_VALUE

Similarly for A_FLOATEXP, following same precedence and associativity for grammar

A_FLOATEXP: A_FLOATEXP + A_FLOATEXP | A_FLOATEXP - A_FLOATEXP | A_FLOATEXP * A_FLOATEXP | A_FLOATEXP / A_FLOATEXP

| A_FLOATEXP % A_FLOATEXP

| A_FLOATEXP ^ A_FLOATEXP | VAR

| INT_VALUE | FLOAT_VALUE

Storing these expressions inside a Variable for later use, that makes an assignment action

ASSIGN_EXP: VAR = A_INTEXP | VAR = A_FLOATEXP

Unary operators also make an arithmetic expression, thus:

UNARY_EXP: VAR ++ | VAR - - | ++VAR | --VAR

- **Evaluate Logical expressions containing AND, OR, NOT**

A logical expressions can operate on truth values, the truth values in the program can be obtained from CONDTIONs, and the results of logical expressions are another truth values and hence logical expressions involving AND, OR, NOT are also CONDITIONs, thus we can write.

CONDITION: CONDITION AND CONDITION

|CONDITION OR CONDITION

|NOT CONDITION

- **Evaluate comparisons like >=, <=, >,< ,== ,!=**

The comparisons can happen between integers, variables and float values and the results are truth values, so they can be evaluated as CONDITION

CONDITION: TERM != TERM

|TERM == TERM

|TERM > TERM

|TERM < TERM

|TERM >=TERM

|TERM <=TERM

A term can be a integer float or a variable,

TERM: INT_VALUE | FLOAT_VALUE | VAR

All the comparison and logical operators are left associative, operator precedence as follows

OR AND == !=>=<=><

With < the highest and OR with the least precedence.

- **Check the syntax of two control statements, IF and IF_ELSE and evaluate conditions given.**

Control statements shift the control depending on the CONDITION it evaluates.

Now we consider two of them, IF and IF_ELSE

IF checks the condition, if the condition is true it executes the statement or compound statement following it. Otherwise passes the control to the first statement after the IF block.

COMPOUND_OR_STATEMENT: STATEMENT | COMPOUND_STATEMENT

A statement is any valid expression ending with semicolon or a construct and also compound statement is the list of statements within two flower braces.

COMPOUND_STATEMENT: { STATEMENTS_LIST }

A list of statements could be a single statement or a statement followed by statements list.

STATEMENTS_LIST: STATEMENT | STATEMENT STATEMENTS_LIST If statement can be defined as,

IF_STMT: IF (CONDITION) COMPOUND_OR_STMT

IF_ELSE checks the condition, if the condition is true it executes the statement or compound statement following it. Otherwise it executes the statement or compound statement after ELSE token.

IF_STMT: IF (CONDITION) COMPOUND_OR_STMT

ELSE COMPOUND_OR_STMT

- **Check the syntax two Looping constructs, WHILE and FOR loops evaluate conditions given.**

WHILE statement conditionally repeats a statement or a compound statement following it. While checks whether or not the condition in the parenthesis is true, if true it executes the statement or compound statement below it, then comes back to check the condition, this repeats until the given condition become false.

WHILE_STMT: WHILE (CONDITION) COMPOUND_OR_STMT

FOR statement does the same, conditionally repeats a statement or a compound statement following it. Unlike WHILE it has three expressions separated by semicolons within the open and closed parenthesis after FOR token. Each one has a specific type,

Any of those sections can be left blank. In case all the sections are empty , the loop is run infinitely.

First one can either be declaration with initialization section or an assignment expression. To initialize the iterating value for one time.

DECL_OR_ASSIGN: DECL | ASSIGN_EXP

Second one is a condition, which when fails FOR loop terminates CONDITION_OR_BLANK: CONDITION

Third one is unary operation or an assignment expression, runs after execution of every loop.

UNARY_OR_ASSIGN: DECL | ASSIGN_EXP

If all the expressions are left blank then it is an infinite loop

FOR_STMT:

FOR (DECL_OR_ASSIGN ; CONDITION_OR_NONE ; UNARY_OR_ASSIGN)

COMPOUND_OR_STMT | FOR (; ;)

COMPOUND_OR_STMT

- **Offer Print function to print () three data-types variables (i.e. INT, FLOAT and CHAR) and strings.**

Although printf can do a wide variety of printing operations, our printf is limited at operation, can process only one format at a time. (i.e “%d”, “%f”, “%s” or string literal within a pair of double Quotes) Printf function checks the first token after open parenthesis, if that is a string literal, it directly prints it. If it finds any format (“%d”, “%f”, “%s”), it goes past the comma after that and print the value found. This value can be a variable, integer or a float expressions.

PRINTF_STMT:

PRINTF (“%d” , VAR)

|PRINTF (“%f” , VAR)

|PRINTF (“%s” , VAR)

|PRINTF (STRING_LITERAL)

|PRINTF (“%d” , A_INTEXP)

|PRINTF (“%f” , A_FLOATEXP)

|PRINTF (“%d” , CONDITION)

- **Check the syntax of the scan statement and scan the corresponding input.**

Our scanf works just like the printf explained above, except it pushes the value to the variable located at address obtained by ampersand operator.

SCANF_STMT:

SCANF (“%d”, & VAR)

| SCANF (“%f”, & VAR)

| SCANF (“%s”, & VAR)

Now let’s define the program order, starting with statement

STATEMENT:

DECL ';'

|ASSIGN_EXP ';'

|PRINTF_STMT ';'

|SCANF_STMT ';'

|UNARY_EXP ';'

|IF_STMT ';'

|IF_ELSE_STMT ';'

|WHILE_STMT ';'

|FOR_STMT

|RETURN VAR_OR_VALUE ';' ';'

|BREAK ';' |

Return statement can return any type,

VAR_OR_VALUE: VAR | INT_VALUE | FLOAT_VALUE | CHAR_LITERAL | STRING_LITERAL

Program can contain global declaration, preprocessors or main()

PROGRAM:

RETURN_TYPE MAIN ()

|DECL ; PROGRAM

|PREPROCESSING PROGRAM

As seen above a function can have return statements, return different data-types

RETURN_TYPE: INT | FLOAT | CHAR | VOID

Q-A1.4)

Implementation of Yacc Program for Yacc Parser

Start of declaration

```
%{
void yyerror (char *s);
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define max_lexemes 100

extern FILE *yyin;
extern int yyparse();
extern int yylex();
extern int yylineno;
int count=0;

struct symbol{
    char type[10];
    char value[30];
    char name[10];
}symbol_table[max_lexemes];
```

Let's continue to define some utility functions,

```
int lookup(char *name) {

int i,flag=1;
for(i=0;i<=count;i++)
{
if(strcmp(symbol_table[i].name,name)==0) {
  flag=0;
break;
} }
if(flag==1)
{
       printf("Undeclared variable %s in line %d\n",name,yylineno);
exit(0); }
return i;
```

Definition of function lookup to return the index of the variable in symbol table with the help of variable name.

```
void addvar(char* type, char* name){
++count;
strcpy(symbol_table[count].type,type);
strcpy(symbol_table[count].name,name);
}
```

Definition of function addVar to add the new variable with given name and type associated with it.

```
char* getValue(char* name){
       return symbol_table[lookup(name)].value;
}
```

Definition of function getValue to return the value stored in that variable using the variable name.

```
char* getTypeString(char* name){
      return symbol_table[lookup(name)].type;
}
```

Definition of function getTypeString to return the data-type of the variable using variable name.

```
int getTypeInt(char* name){
      if(strcmp(symbol_table[lookup(name)].type,"int")==0)
            return 1;
      else if(strcmp(symbol_table[lookup(name)].type,"float")==0)
            return 2;
      else if(strcmp(symbol_table[lookup(name)].type,"char")==0)
            return 3;
      return 0;
}
```

Definition of function setValue to store the value of type specified in integer type (i.e., 1 2 or 3) the variable located at index in the symbol table.

```
void setValue(char* value,int type,int index){
       char* expected_type=symbol_table[index].type;
       //variable type from symbol table
       if(type==1){
              if(strcmp(expected_type,"int")==0
                  || strcmp(expected_type,"float")==0){  // floating values can
have integer values also
                     strcpy(symbol_table[index].value,value);
                     return ;
              }
              else{
                     printf("Type mismatch at %d",yylineno);
                     exit(0);
              }
       }
       if(type==2){
              if(strcmp(expected_type,"float")==0){
                     strcpy(symbol_table[index].value,value);
                     return;
              }
              else{
                     printf("type mismatch at %d",yylineno);
                     exit(0);
              }
       }
       if(type==3){
              if(strcmp(expected_type,"char")==0){
                     strcpy(symbol_table[index].value,value);
                     return;
              }
              else{
                     printf("type mismatch at %d",yylineno);
                     exit(0);
              }
       }
              printf("Undefined type encountered at line %d\n",yylineno);
              exit(0);
}
```

Definition of function to display the data symbol table.

```
void display()
{
       int i;
       printf("SYMBOL TABLE\n");
       printf("%20s\t%20s\t%20s\n","Variable Name","Type","value");
       for(i=0;i<=count;i++)
       printf("\n%20s\t%20s\t%20s\n",symbol_table[i].name,symbol_table[i
].type,symbol_table[i].value);
}
```

Check if the given word is not a reserved keyword, returns 1 on success.

```
int notReservedKeyword(char* word){
if(strcmp("int",word)==0 || strcmp("float",word)==0 || strcmp("char",word)==0 ||
 strcmp("while",word)==0 || strcmp("do",word)==0 || strcmp("for",word)==0||
    strcmp("if",word)==0 || strcmp("else",word)==0){
    printf("Reserved keyword %s used as identifier at line %d \n",word,yylineno);
    exit(0);

}
```

Declaring start token and define operator precedence and associativity.

```
%}
%start START
%left ','
%right '='
%left OR
%left AND
%left EQUAL NOT_EQUAL
%left GT LT LT_EQ GT_EQ
%left PLUS MINUS
%left MULT DIV MOD
%right POW
```

Defining three data-types to be extracted from lex program

```
%union {
    int int_val;
    char *string;
    float float_val;
}
```

Declaring all the typeless tokens obtained from lex program.

```
%token SCANF PRINTF STRING_FORMAT FLOAT_FORMAT INT_FORMAT CHAR_FORMAT
%token UNDERSCORE NOT IF FOR ELSE INCREMENT DECREMENT WHILE
%token MAIN PREPROCESSING RETURN BREAK VOID
```

Declaring all the type specific tokens obtained from lex program.

```
%token <string> VAR FLOAT INT CHAR CHAR_LITERAL STRING_LITERAL
%token <int_val> INT_VALUE
%token <float_val> FLOAT_VALUE
```

Declaring type for some non- terminals

```
%type <string> TYPE UNARY_EXP
%type <int_val> A_INTEXP CONDITION TERM
%type <float_val> A_FLOATEXP
```

When a program successfully reaches START, means program ran successfully. And so, the action is print statement

```
%%
START: PROGRAM  {printf("Program with no syntax error! \npass!\n");display();}
```

Program can contain preprocessor directives or global declarations before the main() begins. Main() is associated with a return type explaining the type of the data that function returns.

```
PROGRAM: RETURN_TYPE MAIN '(' ')' COMPOUND_STATEMENT {printf("Function
completed\n");} |DECL ';' PROGRAM
       {printf("Global declaration section\n");}
|PREPROCESSING PROGRAM
{printf("PREPROCESSING started\n");}
;
```

Return type can be int or float or char or void data-type.

```
RETURN_TYPE:
INT
|FLOAT
|CHAR
|VOID ;
```

Compound statement is the statements list enclosed within a pair of flower braces. statements_list can be a list of statements a single statement.

```
COMPOUND_STATEMENT: '{' STATEMENTS_LIST '}'
;
STATEMENTS_LIST: STATEMENT STATEMENTS_LIST
       |STATEMENT
;
```

Different statement formats.

```
STATEMENT:  DECL ';'
             |ASSIGN_EXP ';'
;
|PRINTF_STMT ';' |SCANF_STMT ';' |UNARY_EXP ';'
|IF_STMT
|IF_ELSE_STMT |WHILE_STMT
|FOR_STMT
|RETURN VAR_OR_VALUE ';' |BREAK ';'
|
```

Different types that can be returned.

```
VAR_OR_VALUE: VAR|INT_VALUE|FLOAT_VALUE|CHAR_LITERAL|STRING_LITERAL;
```

When an IF statement is encountered, it can be followed by a compound statement or a single statement, the semantic action displays the line number, truth value of condition and next statement to where the control to be shifted. If or out

```
IF_STMT:
IF '(' CONDITION ')' COMPOUND_OR_STMT {
if($3)
printf("line %d: IF control encountered, condition evaluated to
true\n",yylineno); else
       printf("line %d: IF control encountered, condition evaluated to false"
                     " shifting control to next statement\n",yylineno);
} ;
```

A non-terminal to represent compound statement or statement

```
COMPOUND_OR_STMT: COMPOUND_STATEMENT   |   STATEMENT
;
```

When an IF statement is encountered, it can be followed by a compound statement or a single statement, the semantic action displays the line number, truth value of condition and next statement to where the control to be shifted. If or else

```
IF '(' CONDITION ')' COMPOUND_OR_STMT ELSE COMPOUND_OR_STMT {
if($3)
       printf("line %d: IF ELSE control encountered,
                     condition evaluated to true\n",yylineno);
else
       printf("line %d: IF ELSE control encountered, condition evaluated to
false" ", running else part\n",yylineno);
} ;
```

When a WHILE statement is encountered, it can be followed by a compound statement or a single statement, the semantic action displays the line number, truth value of condition and next statement to where the control to be shifted. Inside while or out

```
WHILE_STMT:
WHILE '(' CONDITION ')' COMPOUND_OR_STMT {
if($3)

printf("line %d: WHILE loop encountered, condition evaluated to
true\n",yylineno); else

       printf("line %d: WHILE loop encountered, condition evaluated to false"
                     ", shifting to next statement after WHILE
loop\n",yylineno);

} ;
```

When a FOR statement is encountered, it can be followed by a compound statement or a single statement, the semantic action displays the line number, truth value of condition and next statement to where the control to be shifted. Inside for or out. For also executes declaration or assignment or unary operation. FOR with no expressions in the parenthesis runs infinitely.

```
FOR_STMT:
FOR '(' DECL_OR_ASSIGN ';' CONDITION ';' UNARY_OR_ASSIGN ')' COMPOUND_OR_STMT {
if($5)
       printf("line %d: FOR loop encountered, condition evaluated to
true\n",yylineno);
else
       printf("line %d: FOR loop encountered, condition evaluated to false"
                     ", shifting to next statement after FOR loop\n",yylineno);
       }
| FOR '(' ';' ';' ')'{printf("line %d: INFINITE FOR loop
encountered\n",yylineno);}
;
```

Non terminals to represent declaration or assignment expression and unary expression or assignment expression.

```
DECL_OR_ASSIGN: DECL    |  ASSIGN_EXP ;
UNARY_OR_ASSIGN: UNARY_EXP  |  ASSIGN_EXP;
```

Assignment expressions stores the value of Arithmetic expressions on the right of '=' to the variable in the left of it. Semantic actions convert's the integer or floating value from arithmetic expression into string only type allowed in our symbol table and then making function call to setValue function.

```
ASSIGN_EXP:
VAR '=' A_INTEXP  {
                }
|VAR '=' A_FLOATEXP {
;
printf("line %d:Expression evaluated to %d and"
       " assigned to variable %s\n",yylineno,$3,$1);
if(getTypeInt($1)){
       printf("%s = %d\n",$1,$3);
       char intstring[10];
       sprintf(intstring,"%10d",$3);
       setValue(intstring,getTypeInt($1),lookup($1));}
printf("line %d:Expression evaluated to %f and"
        "assigned to variable %s\n",yylineno,$3,$1);
if(getTypeInt($1)){
       printf("%s = %f\n",$1,$3);
}
char floatstring[10];
sprintf(floatstring,"%7.3f",$3);
setValue(floatstring,getTypeInt($1),lookup($1));}
```

Declaration statements are type followed by variable list. The type could be INT or FLOAT or CHAR, semantic actions are to forward this information to higher non terminals. This could be asserted to varlist.

```
DECL: TYPE VARLIST ;

TYPE:
INT {strcpy($$,$1);} |FLOAT {strcpy($$,$1);} |CHAR {strcpy($$,$1);}
;
```

Varlist can contain var or varlist followed by var, semantic actions add the variables to the symbol table. Otherwise say declaration happened.

```
VARLIST:
VAR  {
|VARLIST ',' VAR {
printf("line %d: variable %s declared\n",yylineno,$1);
                            if(notReservedKeyword($1))
                                      addvar($0,$1);}
                                           EQUAL_PART
printf("line %d: variable %s declared\n",yylineno,$3);
                            if(notReservedKeyword($3))
                                      addvar($0,$3);}
                                           EQUAL_PART
|VAR '[' INT_VALUE ']' {
                 printf("line %d: %s type 1D array named %s declared with size %d
\n ",

yylineno,$0,$1,$3);}
|VAR '[' INT_VALUE ']' '[' INT_VALUE ']' {
          printf("line %d: %s type 2D array named %s declared with order %d X %d
\n ",

;

yylineno,$0,$1,$3,$6);}
```

Arithmetic expression evaluation containing integers and variables, semantic actions define the evaluation and pass the value to non-terminals.

```
A_INTEXP:
'(' A_INTEXP ')'
|A_INTEXP PLUS A_INTEXP {$$=$2;}

|A_INTEXP MINUS A_INTEXP {$$=$1+$3;}

|A_INTEXP MULT A_INTEXP {$$=$1-$3;}

|A_INTEXP MOD A_INTEXP {$$=$1*$3;}

|A_INTEXP DIV A_INTEXP {$$=$1%$3;}

|A_INTEXP POW A_INTEXP {if($3==0)yyerror("Division by zero");
            else $$=$1/$3;}
{$$=pow($1,$3);}

|INT_VALUE. {$$=$1;}

|VAR  {if(getTypeInt($1)==1) $$=atoi(getValue($1));
            else yyerror("Type mismatch");}a

;
```

Arithmetic expression evaluation containing floating values and variables, semantic actions define the evaluation and pass the value to non-terminals.

```
A_FLOATEXP:
'(' A_FLOATEXP ')'
|A_FLOATEXP PLUS A_FLOATEXP
|A_FLOATEXP MINUS A_FLOATEXP
|A_FLOATEXP MULT A_FLOATEXP
|A_FLOATEXP DIV A_FLOATEXP
|A_FLOATEXP POW A_FLOATEXP
|FLOAT_VALUE
|INT_VALUE
|VAR {
       {$$=$2;}
       {$$=$1+$3;}
       {$$=$1-$3;}
       {$$=$1*$3;}
       {
if($3==0.0)yyerror("Division by zero");
else $$=$1/$3;}
       {$$=pow($1,$3);}
       {$$=$1;}
       {$$=(float)$1;}

;

if(getTypeInt($1)==1 || getTypeInt($1)==2)$$=atof(getValue($1));
else yyerror("Type mismatch");}
```

Logical and comparison expression evaluations containing integer value or variable, truth values to be passed to higher non terminals.

```
CONDITION:
FLOAT_VALUE NOT_EQUAL FLOAT_VALUE   {$$=$1 != $3;}
|FLOAT_VALUE  EQUAL FLOAT_VALUE     {$$=$1 == $3;}
|FLOAT_VALUE  GT FLOAT_VALUE        {$$=$1 > $3;}
|FLOAT_VALUE  GT_EQ FLOAT_VALUE     {$$=$1 >= $3;}
|FLOAT_VALUE  LT FLOAT_VALUE        {$$=$1 < $3;}
|FLOAT_VALUE  LT_EQ FLOAT_VALUE     {$$=$1 <= $3;}
;
```

Comparison expression evaluations containing floating value or variable, truth values to be passed to higher non terminals.

```
CONDITION:
FLOAT_VALUE NOT_EQUAL FLOAT_VALUE   {$$=$1 != $3;}
|FLOAT_VALUE  EQUAL FLOAT_VALUE     {$$=$1 == $3;}
|FLOAT_VALUE  GT FLOAT_VALUE        {$$=$1 > $3;}
|FLOAT_VALUE  GT_EQ FLOAT_VALUE     {$$=$1 >= $3;}
|FLOAT_VALUE  LT FLOAT_VALUE        {$$=$1 < $3;}
|FLOAT_VALUE  LT_EQ FLOAT_VALUE     {$$=$1 <= $3;}
;
```

Equal part can be a value or nothing. Semantic actions say value assigned and update the symbol table with new value.

```
EQUAL_PART:
'=' INT_VALUE {
       printf("line %d:Integer value %d assigned to variable\n",yylineno,$2);
       char value[15];
       sprintf(value,"%d",$2);
       setValue(value,1,count);}
|'=' FLOAT_VALUE {
       printf("line %d:Float value %f assigned to variable\n",yylineno,$2);
       char value[15];
       sprintf(value,"%f",$2);
       setValue(value,2,count);}
|'=' CHAR_LITERAL {
       printf("line %d:character %s assigned to variable\n",yylineno,$2);
       setValue($2,3,count);}
|'=' VAR {
       printf("line %d:variable %s assigned to variable\n",yylineno,$2);
       setValue(getValue($2),getTypeInt($2),count);}
|'=' A_INTEXP {
       printf("line %d:Expression evaluated to %d and assigned to
variable\n",yylineno,$2);
       char intstring[10];
       sprintf(intstring,"%10d",$2);
       setValue(intstring, 1 ,count);}
|'=' A_FLOATEXP {
       printf("line %d:Expression evaluated to %f and assigned to
variable\n",yylineno,$2);
       char floatstring[10];
       sprintf(floatstring,"%7.3f",$2);
       setValue(floatstring, 2 ,count);}
|
;
```

Printing statements print variety of values and expressions, but only one format at a time. Semantic actions print the values.

```
PRINTF_STMT:
      PRINTF '(' INT_FORMAT ',' VAR ')'              { printf("line %d:
%d\n",yylineno,atoi(getValue($5)));}
      |PRINTF '(' FLOAT_FORMAT ',' VAR ')'     { printf("line %d:
%f\n",yylineno,atof(getValue($5)));}
      |PRINTF '(' STRING_FORMAT ',' VAR ')'    { printf("line %d:
%s\n",yylineno,getValue($5));}
      |PRINTF '(' STRING_LITERAL ')'                 { printf("line %d: %s
\n",yylineno ,$3);}
      |PRINTF '(' INT_FORMAT ',' A_INTEXP ')' {printf("line %d:
%d\n",yylineno,$5);}
      |PRINTF '(' FLOAT_FORMAT ',' A_FLOATEXP ')' {printf("line %d:
%f\n",yylineno,$5);}
      |PRINTF '(' INT_FORMAT ',' CONDITION ')' {printf("line %d:
%d\n",yylineno,$5);}
;
```

Unary operators' evaluation, semantics print update process and update new value to symbol table

```
UNARY_EXP:
      VAR INCREMENT {char string_num[10];

      sprintf(string_num,"%d\n",atoi(getValue($1))+1);
                              setValue(string_num,1,lookup($1));
                              printf("line %d: %s
incremented\n",yylineno,$1);
                              }
      |VAR DECREMENT {char string_num[10];
                              sprintf(string_num,"%d\n",atoi(getValue($1))-
1);
                              setValue(string_num,1,lookup($1));
                              printf("line %d: %s
decremented\n",yylineno,$1);
                              }
      |INCREMENT VAR {char string_num[10];

      sprintf(string_num,"%d\n",atoi(getValue($2))+1);
                              setValue(string_num,1,lookup($2));
                              printf("line %d: %s
incremented\n",yylineno,$2);
                              }
      |DECREMENT VAR {char string_num[10];
                              sprintf(string_num,"%d\n",atoi(getValue($2))-
1);
                              setValue(string_num,1,lookup($2));
                              printf("line %d: %s
decremented\n",yylineno,$2);
                              }
;
```

Scanf checks the syntax of SCANF statement, accepts statement and continue the execution of the program

```
SCANF_STMT:
       SCANF '(' INT_FORMAT ',' '&' VAR ')' {  int num;

       printf("input:");

       scanf("%d",&num);

       printf("line %d: %d scanned\n",yylineno-1,num);
                                                                              }
       |SCANF '(' FLOAT_FORMAT ',' '&' VAR ')' {      float num;

       printf("input:");

       scanf("%f",&num);

       printf("line %d: %7.3f scanned\n",yylineno,num);
                                                                              }
       |SCANF '(' STRING_FORMAT ',' '&' VAR ')' {
        char character;

       printf("input:");

       scanf("%s",&character);

       printf("line %d: %s scanned\n",yylineno,&character);
                                                                              }
```

Main Function

```
int main(){
       //#ifdef YYDEBUG
       //yydebug = 1;
       //#endif
       yyin = fopen("InputFile.c", "r");
       if(yyin==NULL )
              printf("\nError\n");
       else{
              printf("\nStarted Parsing\n");
              printf("\nK Srikanth 17ETCS002124\n");
                     yyparse();
       }
       fclose(yyin);

    return 0;
}

int yywrap(){
    return 1;
}

void yyerror (char *s) {fprintf (stderr, "%s at line %d\n", s, yylineno);}
```

Q-A1.5)

C File for Yacc Parser

```c
/*
This is a Multiline Comment
*/
#include<stdio.h>
void main(){
int a[100][20];
float r[30];
int var=23,b=2124,c=0044;
c=2/5+2-(4^3+(24%3))+2*15/3+3;
char h='s';
float d;
d=23.01;
c++;
scanf("%d",&c);
scanf("%f",&b);
printf("Enter c");
printf("%d",c);
printf("%s",h);
printf("%f",d);
if(b<4 || b==4){
}

if(c==4 && b>2){
}
else{
}

while(var!=3){
}

for(var=0;var<=3;var++){
}
} // This is a Single Line Comment
```

Figure 1 C Program for Yacc Parser

To Run Lex and Yacc File

```
PS C:\Users\Srikanth\Desktop\Compilers Assignment> bison -d Yacc_File.y
Yacc_File.y: conflicts: 81 shift/reduce, 31 reduce/reduce
PS C:\Users\Srikanth\Desktop\Compilers Assignment> flex Lex_File.l
PS C:\Users\Srikanth\Desktop\Compilers Assignment> gcc -w Yacc_File.tab.c lex.yy.c
PS C:\Users\Srikanth\Desktop\Compilers Assignment> ./a.exe
```

Figure 2 Commands to Run Lex and Yacc Files

There are 4 scenarios that we will be seeing how would the compiler be working,

Scenario 1

In this scenario let's assume that our C program doesn't have any syntax errors and compiled successfully with respect to code given in **Image 1**

Running the following commands in **Image 2**

```
PS C:\Users\Srikanth\Desktop\Compilers Assignment> ./a.exe

Started Parsing

K Srikanth 17ETCS002124
line 6: int type 2D array named a declared with order 100 X 20
 line 7: float type 1D array named r declared with size 30
 line 8:variable var declared
line 8:Integer value 23 assigned to variable
line 8:Integer value 2124 assigned to variable
line 8:Integer value 44 assigned to variable
line 9:Expression evaluated to -49 and assigned to variable c
c = -49
line 10:variable h declared
line 10:character 's' assigned to variable
line 11:variable d declared
line 12:Expression evaluated to 23.010000 and assigned to variable d
d = 23.010000
line 13: c incremented
input:20003
line 13: 20003 scanned
input:10010
line 15: 10010.000 scanned
line 16: "Enter c"
line 17: -48
line 18: 's'
line 19: 23.010000
line 23: IF control encountered, condition evaluated to false shifting control to next statement
line 26: IF ELSE control encountered, condition evaluated to false, running else part
line 29: WHILE loop encountered, condition evaluated to true
line 31:Expression evaluated to 0 and assigned to variable var
var = 0
line 31: var incremented
line 32: FOR loop encountered, condition evaluated to true
Function completed
This Program has no Syntax Error
Everything Was Accepted !
```

Figure 3 Program has no Syntax Errors and Accepted !

As we can see the output starts from line 6, this is because as the lines from **1 to 3 area a multiline comment and followed by header** so as we wrote the regular expression in our lex file so it was not considered.

Now let's look at what all was accepted by our compiler.

- **Line 6:** As from the code we did declare a 2-Dimensional Array with size 100 and 20 and the type is Integer
- **Line 7:** As from the code we did declare a 1-Dimensional Array with size 30 and the type is float
- **Line 8:** As from the code you can see a variable was declared named **"var".** and on the same line we have assigned the values where **var = 23 and b = 2124 and c = 0044** all of type Integer.

- **Line 9:** As from the code we can see at line 9 there is an athematic expression and it is evaluated to -49 and assigned to variable c
- **Line 10:** As from the code you can see a variable was declared named **"h".** The type was **char** and was **initialized to "s"**
- **Line 11:** As from the code you can see a variable was declared named "d" and the type was float.
- **Line 12:** As from the code you can see the variable, we declared at **line 11 is now initialized to 23.010000.**
- **Line 13:** As from the code you can see the **variable "C" got incremented.**
- **Line 13 - 15:** As from the code you can see the we are **scanning with integer and float data types**
- **Line 16 - 19:** As from the code you can see the we are printing with **integer and float and String data types**
- **Line 20 – 21:** As from the code you can see we now have an **if control statement** with logical expressions and it is **evaluated to false**
- **Line 23 – 26:** As from the code you can see we now have an **if and else control statement** with logical expressions and it is **evaluated to false**
- **Line 28 – 29:** As from the code you can see we now have an **While lopping statement** with Condition **where (var!=3)** and it is **evaluated to false**
- **Line 31 – 32:** As from the code you can see we now have a **For lopping statement** with Condition where (var<=3) and **var is initialized to "0" and incremented and finally** it is **evaluated to true**
- **Line 33:** As from the code you can see we are end of the function.

Everything was Accepted without any Syntax error. Now we look at another scenario

Scenario 2

C Program

```
4   #include<stdio.h>
5   void main(){
6   int a[100][20];
7   float r[30];
8   int var=23.1,b=2124,c=0044;
```

Figure 4 C Program for Yacc Parser

In this scenario let's assume that our C program does have a syntax error and wasn't compiled successfully with respect to code given in **Image 4** looking at line 8 **we can see that var is initialized as float value.**

Output

```
--accepting rule at line 52 ("int")
--accepting rule at line 26 (" ")
--accepting rule at line 75 ("var")
--accepting rule at line 95 ("=")
line 8:variable var declared
--accepting rule at line 85 ("23.1")
--accepting rule at line 95 (",")
line 8:Float value 23.100000 assigned to variable
type mismatch at 8
PS C:\Users\Srikanth\Desktop\Compilers Assignment>
```

Figure 5 Program has a Mismatch Error at Line 8

Until line 7 it was accepted and line 8 we have a mismatch where it was supposed to be a integer value instead of float

Scenario 3

C Program

```
11  float d;
12  d="Srikanth";    a value of type "char *" cannot be assigned to an entity of type "float"
```

Figure 6 C Program for Yacc Parser

In this scenario let's assume that our C program does have a syntax error and wasn't compiled successfully with respect to code given in **Image 6** looking at line 12 **we can see that d is initialized as char array.**

Output

```
--accepting rule at line 75 ("d")
--accepting rule at line 95 ("=")
--accepting rule at line 25 (""Srikanth"")
syntax error at line 12
PS C:\Users\Srikanth\Desktop\Compilers Assignment>
```

Figure 7 Program has a Syntax Error at Line 12

Until line 11 it was accepted and line 12 we have a Syntax error where it was supposed to be a **Float value** instead of **char array.**

Conclusion

The regular expressions were framed to recognize lexemes from the input stream. The Lex program after compilation by FLEX generates a Scanner which reads input stream and generate tokens. The First Lex program was created to identify tokens in the program in .c file and simply print them. The second lex program was built to identify the tokens and pass them to parser. The parser is created by compiling YACC file containing Context free grammar through Bison. This parser parses the token stream obtained from Scanner and constructs a syntax tree. The syntax tree must run till start expression (i.e top of the tree should be the start) specified. If that fails, it calls yyerror function.
The context free grammar rules for parsing the given input program, was written in YACC program. The input was taken from InputFile.c. Output was given on command line interface. Program was run upon different files to check the correctness of our grammar. Results were satisfactory and observed drawbacks are listed below.