

|  |  |
| --- | --- |
| **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 | | | | |
|  | | | |  | | | | |



|  |  |  |  |
| --- | --- | --- | --- |
| **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** | | | | | | | | | | | | | |



**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();}

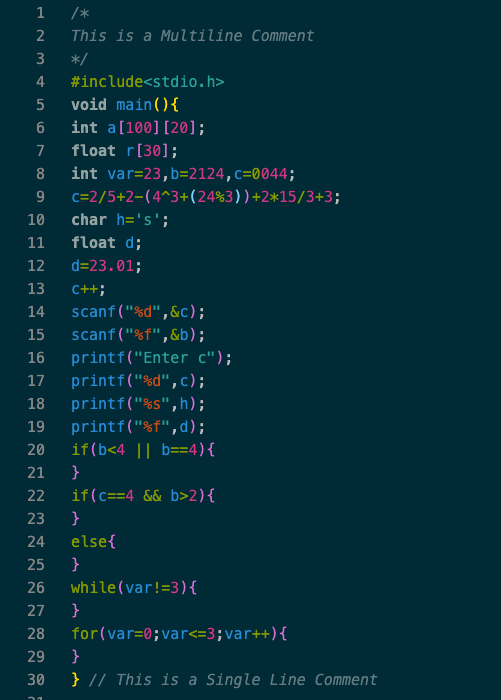
fclose(yyin);

return 0;

}

**Grouping of Tokens**

**Input C File**



**Tokenizing Input File**

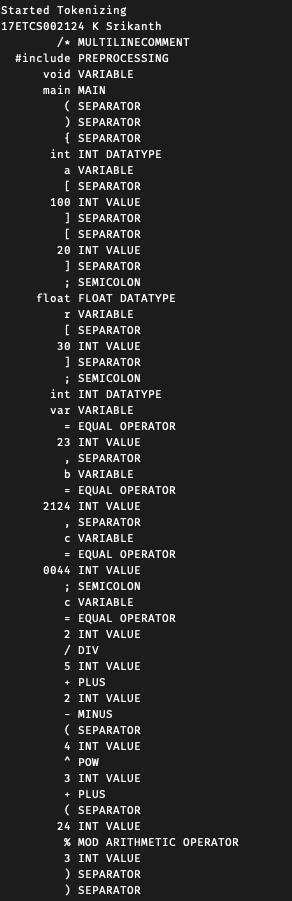
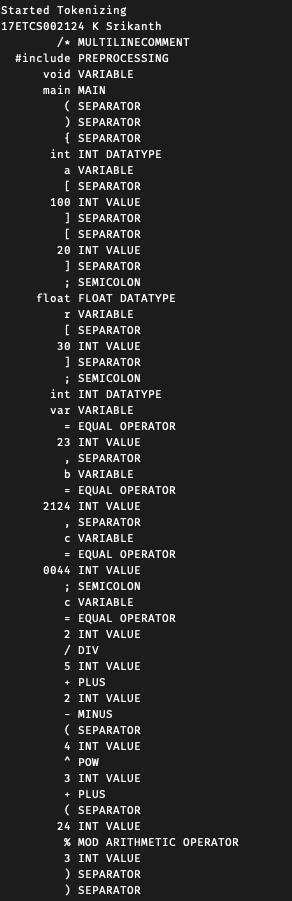


Figure 2 Lex Program Generating Tokens (Continued)

Figure 1 Lex Program Generating Tokens

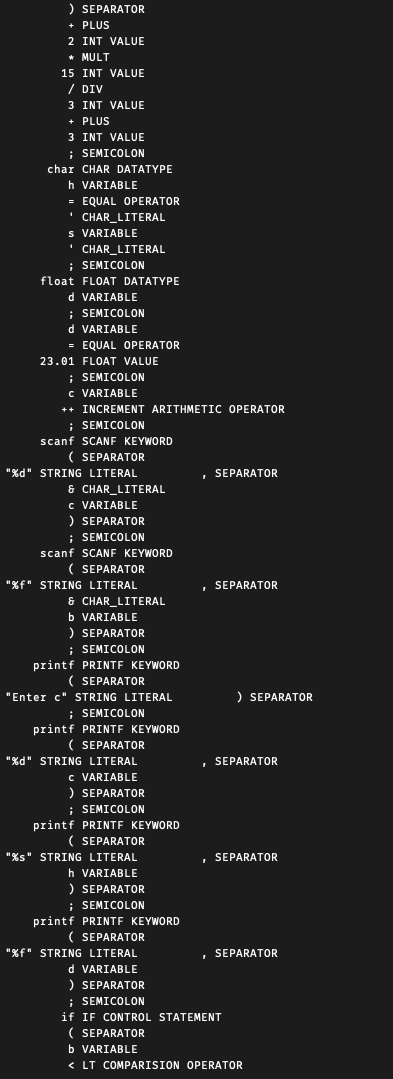
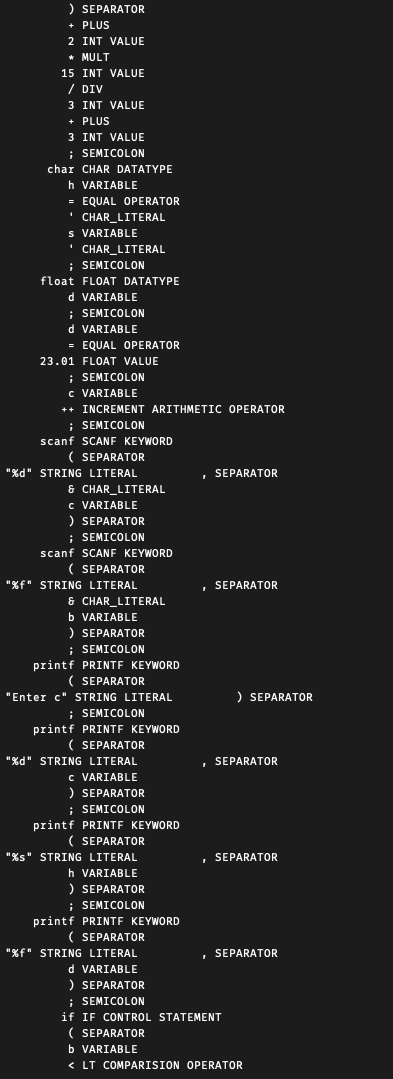
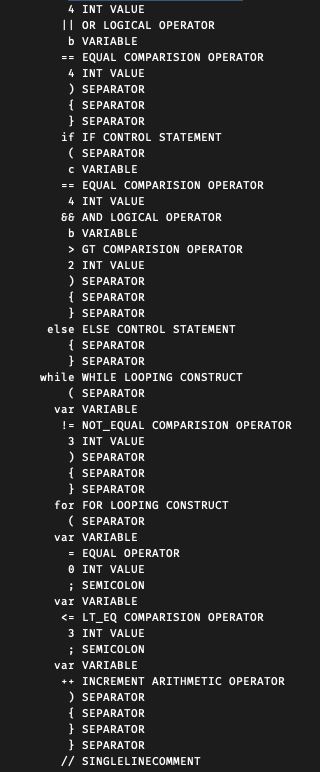


Figure 3 Lex Program Generating Tokens (Continued)

Figure 4 Lex Program Generating Tokens (Continued)

****

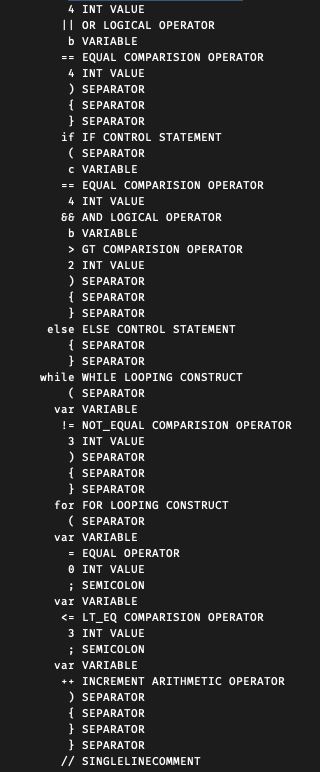


Figure 6 Lex Program Generating Tokens (Continued)

Figure 5 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);

}

return 1; }

**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**

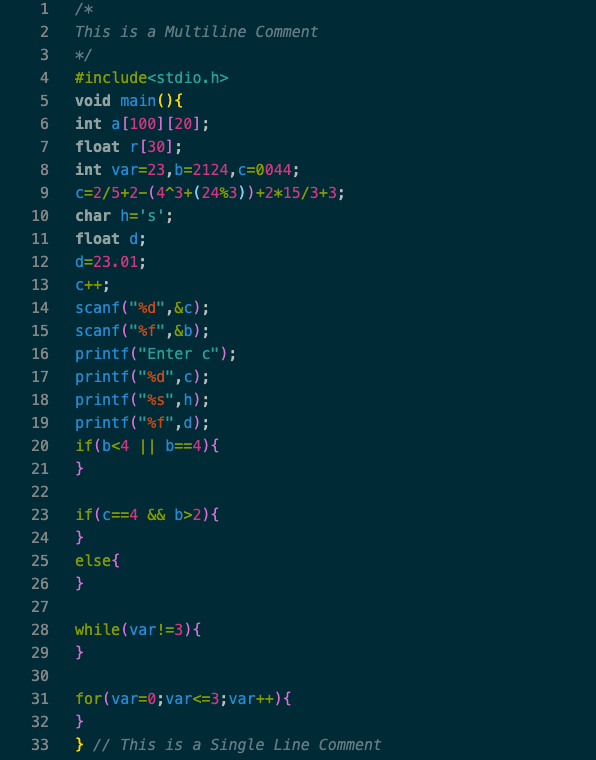
****

Figure 1 C Program for Yacc Parser

**To Run Lex and Yacc File**

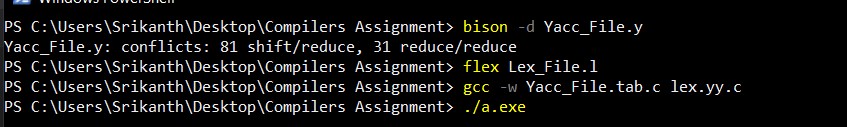
****

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**

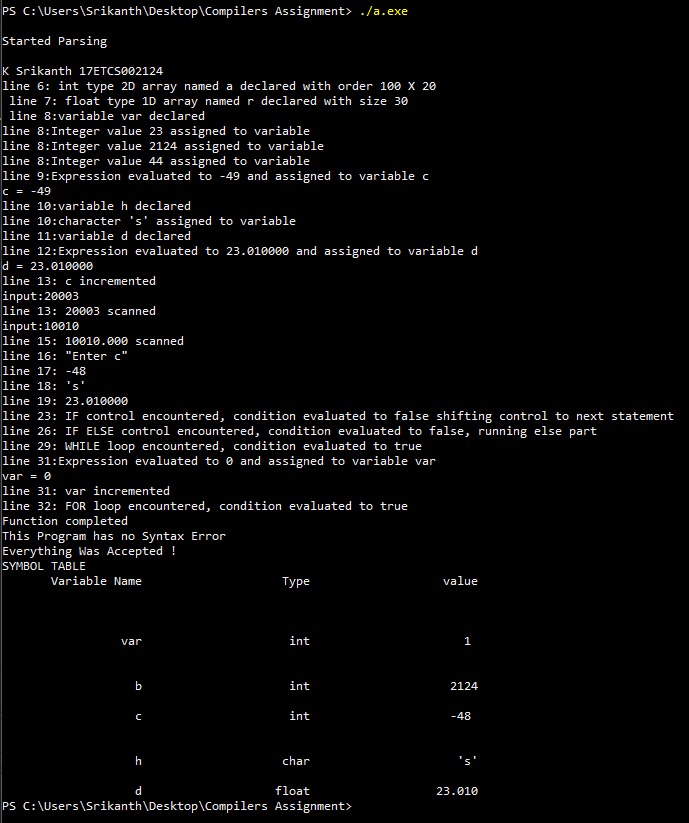
****

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**

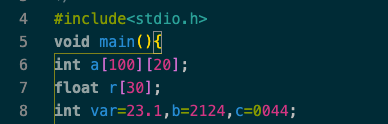
****

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**

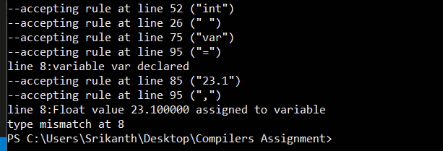
****

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**

****

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**

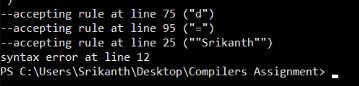
****

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.