RAMAIAH UNIVERSIT
OF APPLIED SCIENCE

ASSIGNMENT

Course Code	CSC310A
Course Name	COMPILERS
Programme	B.TECH
Department	CSE
Faculty	FET

Name of the Student	AISHWARYA
Reg. No	17ETCS002015
Semester/Year	6TH SEM / 3RD YEAR
Course Leader/s	Mr. Hari Krishna S. M.

:

Declaration Sheet			
Student Name	AISHWARYA		
Reg. No	17ETCS002015		
Programme	B.TECH	Semester/Year	6TH SEM/ 3RD YEAR
Course Code	CSC310A		
Course Title	COMPILERS		
Course Date		to	
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	Aishwarya	Date	02/04/2020
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	06th /2017		
Course Code	CSC310A	Course Title	Compilers
Course Leader	Mr. Hari Krishna S. M. & Ms. Suvidha		

Assignment			
Register No.	**17ETCS002015**	**Name of the Student**	**AISHWARYA**

Sections		**Marking Scheme**	**Marks**		
			Max Marks	**First Examiner Marks**	**Moderator**
Part A 1					
	A 1.1	Identification and grouping of Tokens	04		
	A 1.2	Implementation in *Lex*	02		
	A 1.3	Design of Context Free Grammar	04		
	A 1.4	Implementation in *Yacc*	06		
	A 1.5	Results and Comments	04		
		Part-A 1 Max Marks	**20**		
		Total Assignment Marks	**20**		

Course Marks Tabulation

Component- CET B Assignment	**First Examiner**	**Remarks**	**Second Examiner**	**Remarks**
A.1				
Marks (out of 20)				

Signature of First Examiner **Signature of Moderator**

PART – A

A1.1 Identification and grouping of Tokens

Token is the smallest independent unit defined by a parser or the lexical analyser. It can contain a data, language keyword, identifier, datatype, etc of a language syntax. Tokens are defined by the regular expressions, which is understood by the lexical analyser that is lex, which reads in a stream of characters, identifies the lexeme in the stream, and categorizes them in token.

Tokens can be classified as follows:

- Keywords: they are pre-defined or reserved words in a programming language.
- Identifiers: terminology for naming of variables, functions and arrays.
- Constants: whose values cannot be modified after the program execution starts.
- Strings: they are sequence of characters ending with a null character.
- Special Symbols: [] () {} , ; * = #
- Operators: symbols that triggers an action when applied to C variables and other objects. +, - , *, /

The tokens used in the implementation of the lex file are: -

LEXEME	TOKENS
0-9(float valu	constant
0-9(integer)	constant
“A-Z”,a-z	string
int	datatype
float	datatype
;	Semi-colon(separator)
,	Comma(separator)
:	Colon(separator)
IF	keyword
else	keyword
elseif	keyword
while	keyword
do	keyword
switch	keyword
case	keyword
default	keyword
break	keyword
for	keyword

printf	keyword
scanf	keyword
+	arithmetic operator
-	arithmetic operator
$	arithmetic operator
&	Special Symbol
*	arithmetic operator
/	arithmetic operator
(	open parenthesis- Special symb
)	Close parenthesis- Special symb
{	Open braces- Special symbol
}	Closed braces- Special symbol
[	Open bracket- Special symbol
]	Close bracket- Special symbol
<	Less than relational operator
=	Equal-to relational operator
>	Greater-than relational operat
!	Exclamation mark

A1.2 Implementation in *Lex*

Lex is a data file format, which contains shared lexicons and store the language specified data. It is designed to generate scanners, known s tokenizers, used to recognize the lexical patterns in the text. It transforms an input stream into the sequence of tokens and reads the input stream and produces the source code through implementing the lexical analyser in C program.

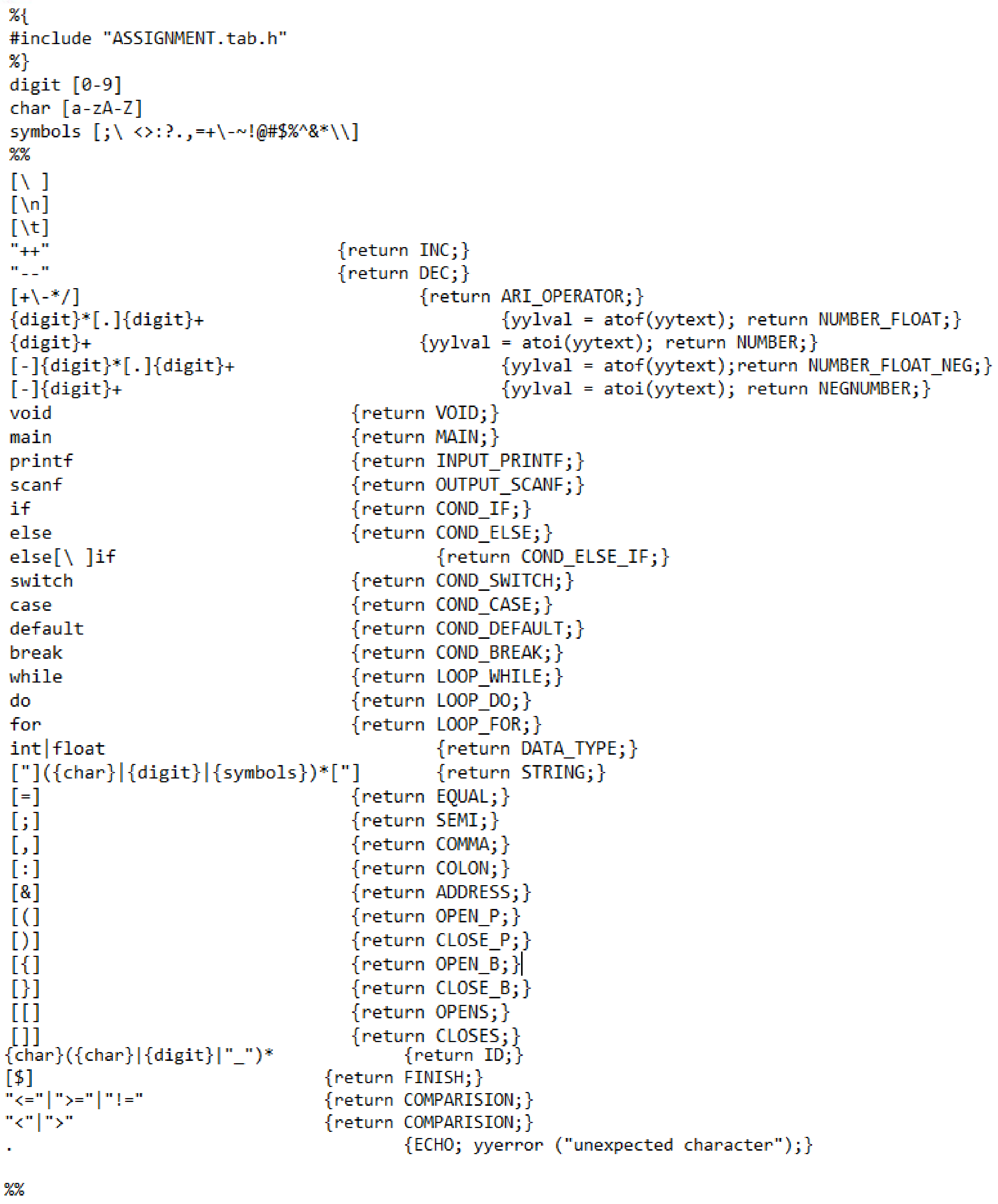
When a lexical analyser reads a source code, it scans the code letter by letter and when whitespace, operator symbol or special symbols, it decides a word is completed.

The input to lex file is divided into three sections, the sections are:

- **Definition section**
- **Rules section**
- **User subroutines**

The definition section includes the literal block, start conditions and translations. Lines starts with the whitespace are copied to the C file.

The rules section contains the pattern lines and C code, a line that starts with the sections above are separated through the lines containing **%%**.



```
%{
#include "ASSIGNMENT.tab.h"
%}
digit [0-9]
char [a-zA-Z]
symbols [;\ <>:?.,=+\-~!@#$%^&*\\]
%%
[\ ]
[\n]
[\t]
"++"                            {return INC;}
"--"                            {return DEC;}
[+\-*/]                                 {return ARI_OPERATOR;}
{digit}*[.]{digit}+                             {yylval = atof(yytext); return NUMBER_FLOAT;}
{digit}+                                {yylval = atoi(yytext); return NUMBER;}
[-]{digit}*[.]{digit}+                          {yylval = atof(yytext);return NUMBER_FLOAT_NEG;}
[-]{digit}+                                     {yylval = atoi(yytext); return NEGNUMBER;}
void                             {return VOID;}
main                             {return MAIN;}
printf                           {return INPUT_PRINTF;}
scanf                            {return OUTPUT_SCANF;}
if                               {return COND_IF;}
else                             {return COND_ELSE;}
else[\ ]if                               {return COND_ELSE_IF;}
switch                           {return COND_SWITCH;}
case                             {return COND_CASE;}
default                          {return COND_DEFAULT;}
break                            {return COND_BREAK;}
while                            {return LOOP_WHILE;}
do                               {return LOOP_DO;}
for                              {return LOOP_FOR;}
int|float                                {return DATA_TYPE;}
["]({char}|{digit}|{symbols})*["]        {return STRING;}
[=]                              {return EQUAL;}
[;]                              {return SEMI;}
[,]                              {return COMMA;}
[:]                              {return COLON;}
[&]                              {return ADDRESS;}
[(]                              {return OPEN_P;}
[)]                              {return CLOSE_P;}
[{]                              {return OPEN_B;}
[}]                              {return CLOSE_B;}
[[]                              {return OPENS;}
[]]                              {return CLOSES;}
{char}({char}|{digit}|"_")*          {return ID;}
[$]                             {return FINISH;}
"<="|">="|"!="                  {return COMPARISION;}
"<"|">"                         {return COMPARISION;}
.                                       {ECHO; yyerror ("unexpected character");}

%%
```

Fig. lex file (ASSIGNMENT.L)

A1.3 Design of Context Free Grammar for each function

Here, for the solution CFG used for each function, along with CFG regular grammar is used since, regular language is a subset of context free grammar.

A context-free grammar (CFG) is a set of recursive rewriting rules (or *productions*) used to generate patterns of strings. To generate a string of terminal symbols from a CFG:

- Begin with a string consisting of the start symbol;
- Apply one of the productions with the start symbol on the left-hand size, replacing the start symbol with the right-hand side of the production;
- Repeat the process of selecting nonterminal symbols in the string, and replacing them with the right-hand side of some corresponding production, until all nonterminal have been replaced by terminal symbols.

CFG has a set of four components: non-terminals, terminal symbols, productions and the start symbol. The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

The CFG used in the functions above are: -

The rules corresponding to the terminals but still n capital format are tokens such as ID, NUMBER etc. in the RHS of the production rules which will be taken from the lex file.

1. **Data Types**

For this, chosen two data-types namely, integer, float.

NON_DECLARE → DATA_TYPE ID;

NON_INITILISE → DATA_TYPE ID = T

| ID = T

2. **Control Statements**

For this, chosen two control statements, if-else(along with elseif()) and switch-case control statement.

- **if-else**

NON_IF → if (NON_CONDITION) { NON_STATEMENTS } ;

NON_ELSE → else if (NON_CONDITION) { NON_STATEMENTS } NON_ELSE

| else { NON_STATEMENTS }

| ϵ

- **switch-case**

NON_SWITCH → switch (ID) { NON_CASES }

NON_CASES → CASE | CASE DEFAULT

CASE → case NUMBER : NON_STATEMENTS break ; CASE | ϵ ;

DEFAULT → default : NON_STATEMENTS break ;

3. <u>Looping statements</u>

For this, we have chosen two looping statements while and do-while

- **while loop**

NON_WHILE → while (NON_CONDITION) { NON_STATEMENTS }

- **do-while loop**

NON_DOWHILE → do { NON_STATEMENTS }while (NON_CONDITION) ;

- **for loop**

NON_FOR → for (NON_INITILISE ; NON_CONDITION ; NON_INCREMENT) {NON_STATEMENTS }

```
NON_INCREMENT → ID ++
              | ++ ID
              | ID --
              |- - ID
```

4. <u>For input and output functions</u>

For output function, we have taken 'printf()'. The following is the grammar of the function printf().

NON_PRINTF → printf (STRING VARIABLES);

```
VARIABLES → ,  ID VARIABLES
          | ϵ
```

For the input function, we have taken 'scanf()'. The following is the grammar of the function scanf().

NON_SCANF → scanf (STRING , &ID) ;

5. <u>For the 2D ARRAY</u>

```
NON_TWO_ARRAY→ DATA_TYPE ID [ C ] [ C ] ;
          | DATA_TYPE ID [ C ] [ C ] = { NON_INPUT } ;
          |NON_TWODASSIGN
```

NON_TWODASSIGN → ID [C] [C] = T ;

NON_INPUT→ T , NON_INPUT | T | ϵ

6. **Compound statements**

For this part, we have created two production rules that are solely recursive production rules.

E→ S $

S → void main ( ) NON_STATEMENTS }

NON_STATEMENTS → NON_STATEMENTS STMT | ϵ ;

STMT → NON_INITILISE ;

|NON_DECLARE ;

|NON_INCREMENT

|NON_TWO_ARRAY

|NON_SWITCH | NON_IF

|NON_FOR | NON_WHILE | NON_DOWHILE

|NON_PRINTF | NON_SCANF

| ϵ

7. **Other Needed Grammar**

T→ T ARI_OPERATOR T | (T)
| NUMBER_FLOAT | NUMBER
| NUMBER_FLOAT_NEG
| NEGNUMBER | ID

NON_CONDITION → C COMPARISION C | C == C

C → ID | NUMBER

The grammar starts with the start symbol E and moves to the non-terminal statements for having multiple statements. Here, S is used to select the statement to be executed such as the loops, conditional statements, input output functions. In these statements, the non-terminal statements are called to having multiple statement such as loop. The program will stop with ($) last symbol of the starting symbol production rule, until and unless not in the string or statement. The program will stop when there is syntax error in the program. No restrictions upon the use of space such as tab-space, new-line or some extra space. In the lex file [\], [\n], [\t] is not returning anything hence, while entering the input in case few blank lines or tab is left then this won't give any error.

A1.4 Implementation in YACC

Yacc requires token names to be declared as such using the keyword `%token`.
Declaration of the start symbol using the keyword `%start`
C declarations: included files, global variables, types.

C code between `% {`and `%}`.

A rule has the form:

```
nonterminal: sentential form
             | sentential form
             .................
             | sentential form
             ;
```

ALGORITHM:

Step1: A Yacc source program has three parts as follows:

Declarations %% translation rules %% supporting C routines

Step2: Declarations Section: This section contains entries that:

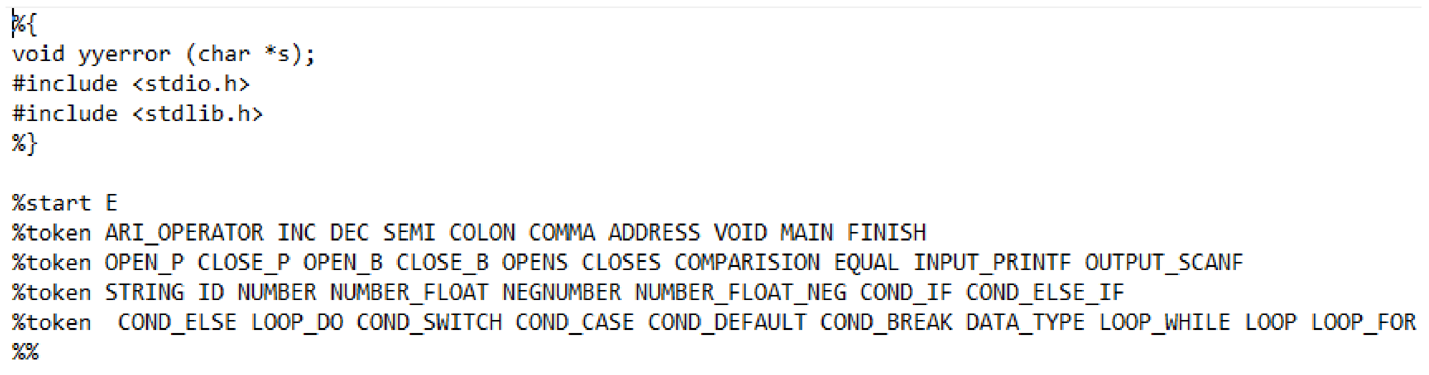
- Include standard I/O header file.
- Define global variables.
- Define the list rule as the place to start processing.
- Define the tokens used by the parser. v. Define the operators and their precedence.

Step3: Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.

Step4: Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.

Step5: Main- The required main program that calls the yyparse subroutine to start the program.

Step6: yyerror(s) -This error-handling subroutine only prints a syntax error message.



```
%{
void yyerror (char *s);
#include <stdio.h>
#include <stdlib.h>
%}

%start E
%token ARI_OPERATOR INC DEC SEMI COLON COMMA ADDRESS VOID MAIN FINISH
%token OPEN_P CLOSE_P OPEN_B CLOSE_B OPENS CLOSES COMPARISION EQUAL INPUT_PRINTF OUTPUT_SCANF
%token STRING ID NUMBER NUMBER_FLOAT NEGNUMBER NUMBER_FLOAT_NEG COND_IF COND_ELSE_IF
%token  COND_ELSE LOOP_DO COND_SWITCH COND_CASE COND_DEFAULT COND_BREAK DATA_TYPE LOOP_WHILE LOOP LOOP_FOR
%%
```

Fig1: yacc file(ASSIGNMENT.Y)

Fig 1 shows the headers for the yacc program along with the start symbol E and the tokens considered. The yyerror() function is called by YACC if it finds an error. The function yywrap() can be used to continue reading from another file.

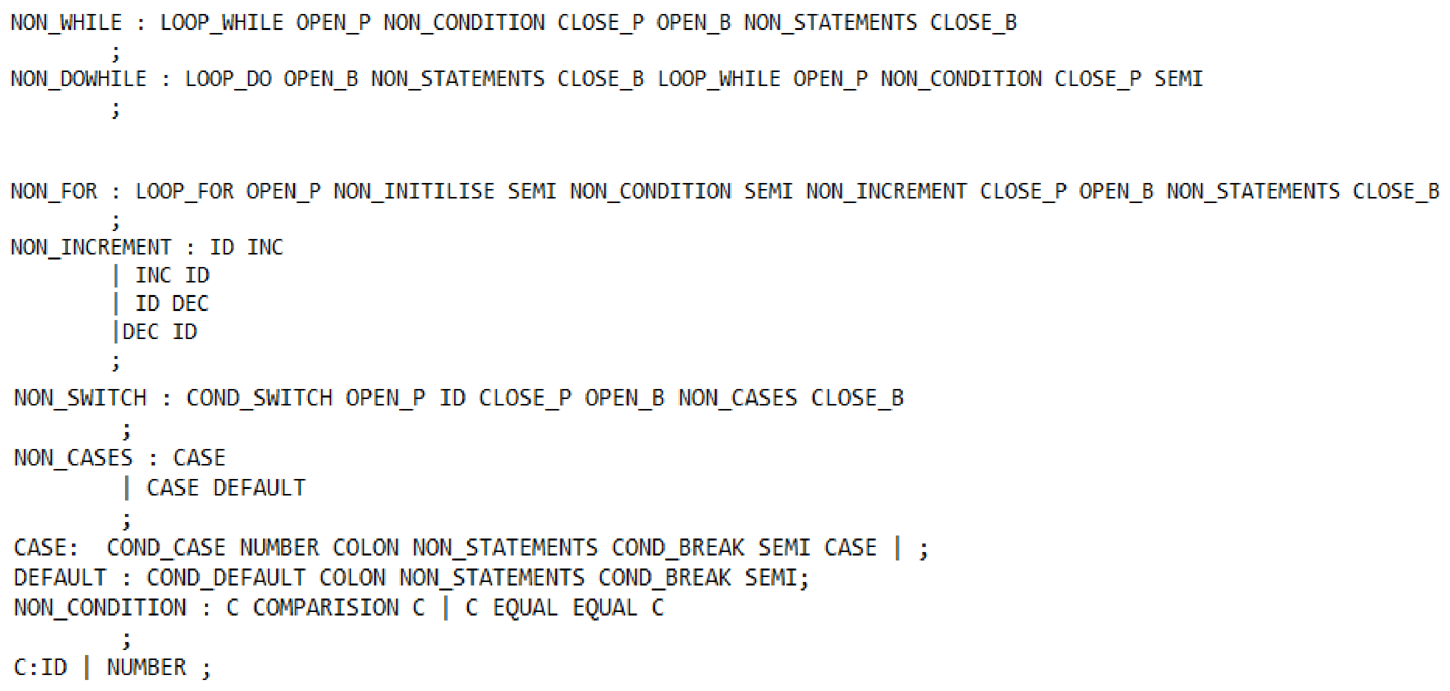
```
E: S FINISH
        ;
S : VOID MAIN OPEN_P CLOSE_P OPEN_B NON_STATEMENTS CLOSE_B FINISH        {printf("Code Accepted\n");exit(0);}
        ;
NON_STATEMENTS : NON_STATEMENTS STMT | ;

STMT :NON_INITILISE SEMI
        |NON_DECLARE SEMI
        |NON_INCREMENT
        |NON_TWO_ARRAY
        |NON_SWITCH | NON_IF
        |NON_FOR | NON_WHILE | NON_DOWHILE
        |NON_PRINTF | NON_SCANF
        |    ;

NON_DECLARE : DATA_TYPE ID;
NON_INITILISE : DATA_TYPE ID EQUAL T
        | ID EQUAL T
        ;

NON_PRINTF : INPUT_PRINTF OPEN_P STRING VARIABLES CLOSE_P SEMI
        ;
VARIABLES : COMMA ID VARIABLES
        |
        ;
NON_SCANF : OUTPUT_SCANF OPEN_P STRING COMMA ADDRESS ID CLOSE_P SEMI
        ;
T: T ARI_OPERATOR T
        |OPEN_P T CLOSE_P
        |NUMBER_FLOAT
        |NUMBER
        |NUMBER_FLOAT_NEG
        |NEGNUMBER
        |ID
        ;
NON_TWO_ARRAY: DATA_TYPE ID OPENS C CLOSES OPENS C CLOSES SEMI
        |DATA_TYPE ID OPENS C CLOSES OPENS C CLOSES EQUAL OPEN_B NON_INPUT CLOSE_B SEMI
        |NON_TWODASSIGN
        ;
NON_INPUT: T COMMA NON_INPUT | T | ;
NON_TWODASSIGN : ID OPENS C CLOSES OPENS C CLOSES EQUAL T SEMI
        ;


NON_IF : COND_IF OPEN_P NON_CONDITION CLOSE_P OPEN_B NON_STATEMENTS CLOSE_B  NON_ELSE
        ;
NON_ELSE : COND_ELSE_IF OPEN_P NON_CONDITION CLOSE_P OPEN_B NON_STATEMENTS CLOSE_B  NON_ELSE
        | COND_ELSE OPEN_B NON_STATEMENTS CLOSE_B
        |
        ;
```

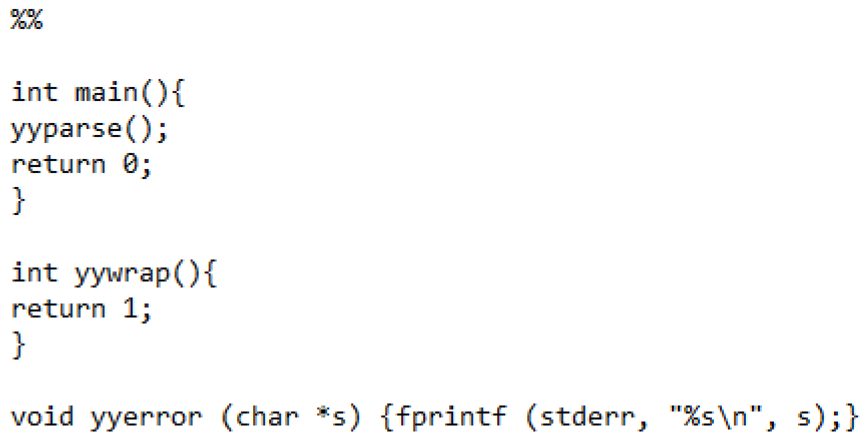


```
NON_WHILE : LOOP_WHILE OPEN_P NON_CONDITION CLOSE_P OPEN_B NON_STATEMENTS CLOSE_B
        ;
NON_DOWHILE : LOOP_DO OPEN_B NON_STATEMENTS CLOSE_B LOOP_WHILE OPEN_P NON_CONDITION CLOSE_P SEMI
        ;


NON_FOR : LOOP_FOR OPEN_P NON_INITILISE SEMI NON_CONDITION SEMI NON_INCREMENT CLOSE_P OPEN_B NON_STATEMENTS CLOSE_B
        ;
NON_INCREMENT : ID INC
        | INC ID
        | ID DEC
        |DEC ID
        ;
NON_SWITCH : COND_SWITCH OPEN_P ID CLOSE_P OPEN_B NON_CASES CLOSE_B
        ;
NON_CASES : CASE
        | CASE DEFAULT
        ;
CASE:  COND_CASE NUMBER COLON NON_STATEMENTS COND_BREAK SEMI CASE | ;
DEFAULT : COND_DEFAULT COLON NON_STATEMENTS COND_BREAK SEMI;
NON_CONDITION : C COMPARISION C | C EQUAL EQUAL C
        ;
C:ID | NUMBER ;
```

Fig2: yacc file (ASSIGNMENT.Y)

The above screenshots contain the grammar from the yacc file, where the rules are specified for the control statements, loops, input output functions the two-dimensional array. Each rule consists of a single name on the left-hand side of the ":" operator, a list of symbols and action code on the right-hand side. A semicolon indicates the end of the rule. Print Function is Printf followed by a semicolon or Printf. Scan Function is Scanf followed by a semicolon. The tokens are declared. IF, Else, Printf, Scanf, Float, Void are the tokens returned by the lex.



```
%%

int main(){
yyparse();
return 0;
}

int yywrap(){
return 1;
}

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

Fig3: yacc file (ASSIGNMENT.Y)

Main function of the yacc program. The programs section contains the following subroutines. Because these subroutines are included in this file, you do not need to use the yacc library when processing this file.

The file contains the following sections:

Declarations section. This section contains entries that:

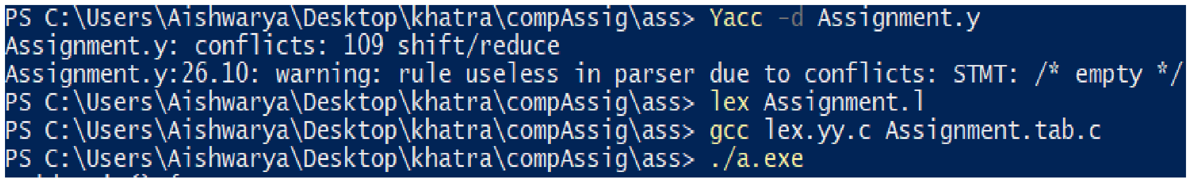
- Include standard I/O header file
- Define global variables
- Define the list rule as the place to start processing
- Define the tokens used by the parser
- Define the operators and their precedence

Rules section: - The rules section defines the rules that parse the input stream.

- *%start* - Specifies that the whole input should match stat.
- *%union* - By default, the values returned by actions and the lexical analyzer are integers. yacc can also support values of other types, including structures. In addition, yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The yacc value stack is declared to be a union of the various types of values desired. The user declares the union, and associate's union member names to each token and nonterminal symbol having a value.

Programs section: - The programs section contains the following subroutines. Because these subroutines are included in this file, you do not need to use the yacc library when processing this file

A1.5 Results and comments



```
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass> Yacc -d Assignment.y
Assignment.y: conflicts: 109 shift/reduce
Assignment.y:26.10: warning: rule useless in parser due to conflicts: STMT: /* empty */
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass> lex Assignment.l
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass> gcc lex.yy.c Assignment.tab.c
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass> ./a.exe
```

Command for the execution of yaccfile

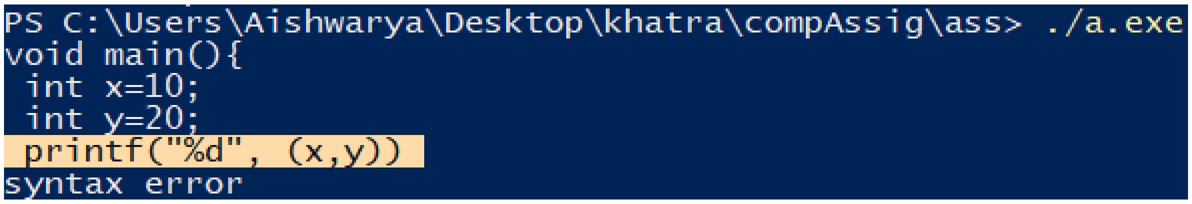
In the above screenshot, command to compile the yacc file.

Yacc -d assignment.y where the assingment.y is a yacc file as .y extension specifies yacc file. Here, -d produces file y.tab.h. this contains #define statements that associate the yacc-assigned token codes without token names. This allows source files other than y.tab.c to access the token codes by including header file. It draws the yacc rules from the assignment.y file, and places the output in the y.tab.c. the name of the tokens are returned.

y.tab.c contains an output file.

./a.exe is used to obtain the output, if there is error in the lex or the yacc file it will show error.

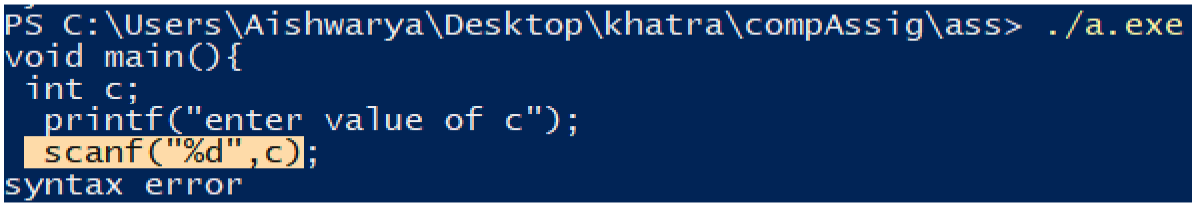
Shift/reduce conflict occurs when an ambiguous grammar is used for expressions that doesnot specify the associativities and the precedance leve if its operators. When enough terms have been read and a grammar rule can be recognized according to the terms obtained. In this situation, the parser makes a reduction. If, however, there is also another grammar rule which calls for more terms to be accumulated and the look ahead token is just what the second grammar rule expected. In this situation, the parser may make a shift operation.



```
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass> ./a.exe
void main(){
 int x=10;
 int y=20;
 printf("%d", (x,y))
syntax error
```

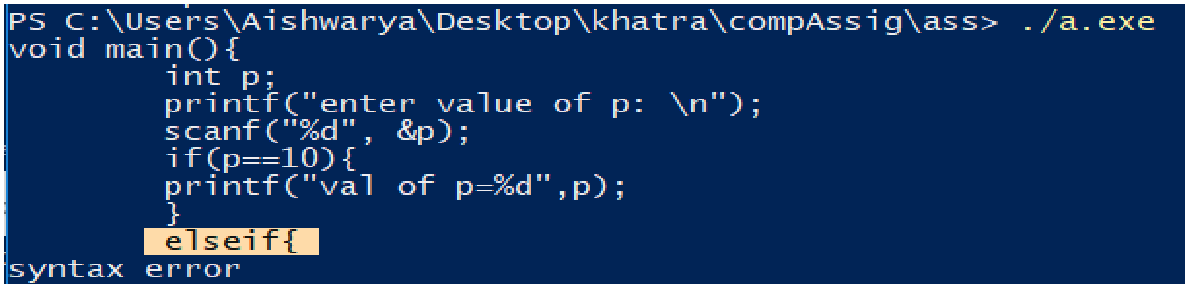
Output 1: semicolon is missing in the output statement

The above screenshot error is been shown, where the semi-colon is missing at the end of the output statement i.e., printf statement, resulting in the parser to fail and display the syntax error.



```
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass> ./a.exe
void main(){
 int c;
  printf("enter value of c");
  scanf("%d",c);
syntax error
```

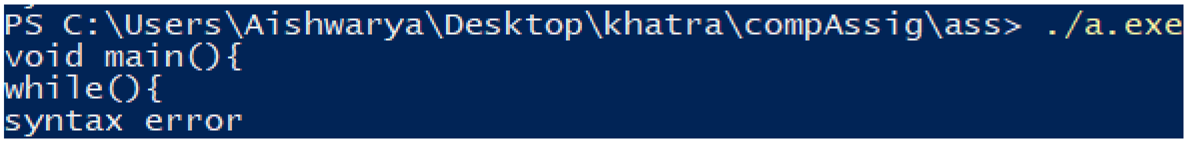
Output 2 : error as (&) operator is missing



```
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass> ./a.exe
void main(){
        int p;
        printf("enter value of p: \n");
        scanf("%d", &p);
        if(p==10){
        printf("val of p=%d",p);
        }
       elseif{
syntax error
```

Output 3: error as no space between else if

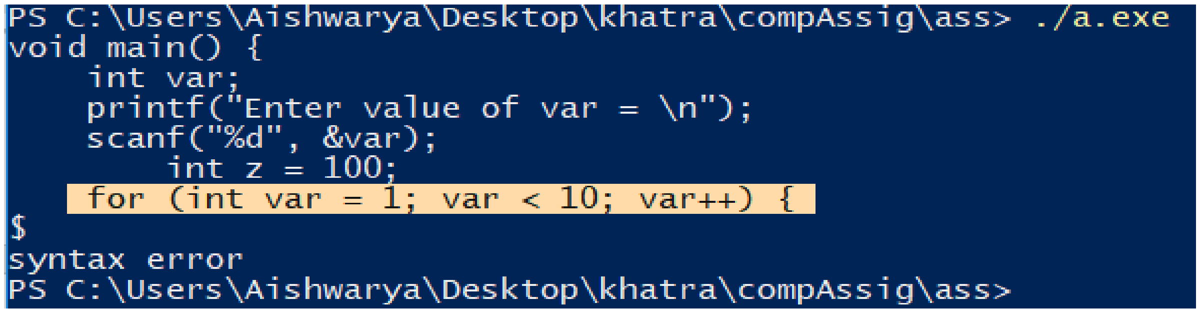
The above screenshot shows the syntax error as the statement, there is no space between the else and if which is not accepted by the grammar used. In C language (else if) is accepted.



```
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass> ./a.exe
void main(){
while(){
syntax error
```

Output4: error as no condition for while loop

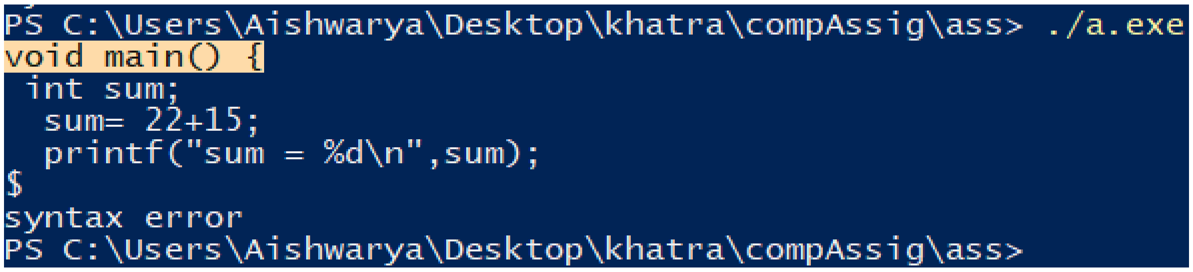
Syntax error is detected as for the while loop, condition is not specified. Hence parsing is failed and the syntax error is displayed.



```
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass> ./a.exe
void main() {
    int var;
    printf("Enter value of var = \n");
    scanf("%d", &var);
        int z = 100;
    for (int var = 1; var < 10; var++) {
$
syntax error
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass>
```

Output 5: error as no condition for the for loop

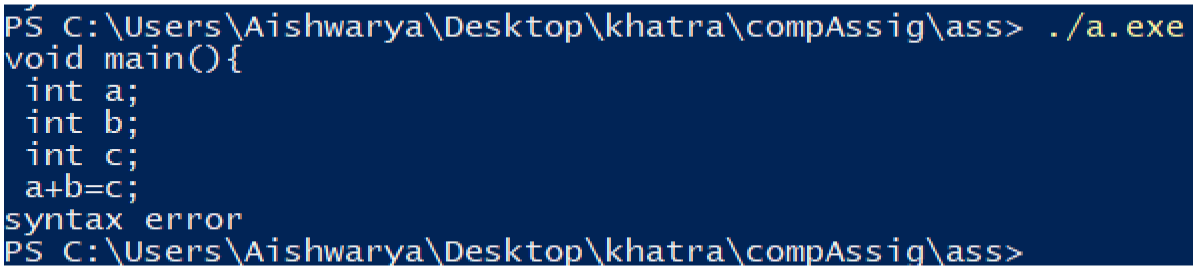
Here, error has occurred as for the loop the condition statement is not provided and according to the grammar the condition statement is required with the loop. Hence, parsing is failed and error is displayed.



```
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass> ./a.exe
void main() {
 int sum;
  sum= 22+15;
  printf("sum = %d\n",sum);
$
syntax error
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass>
```

Output 6: error as no closing braces

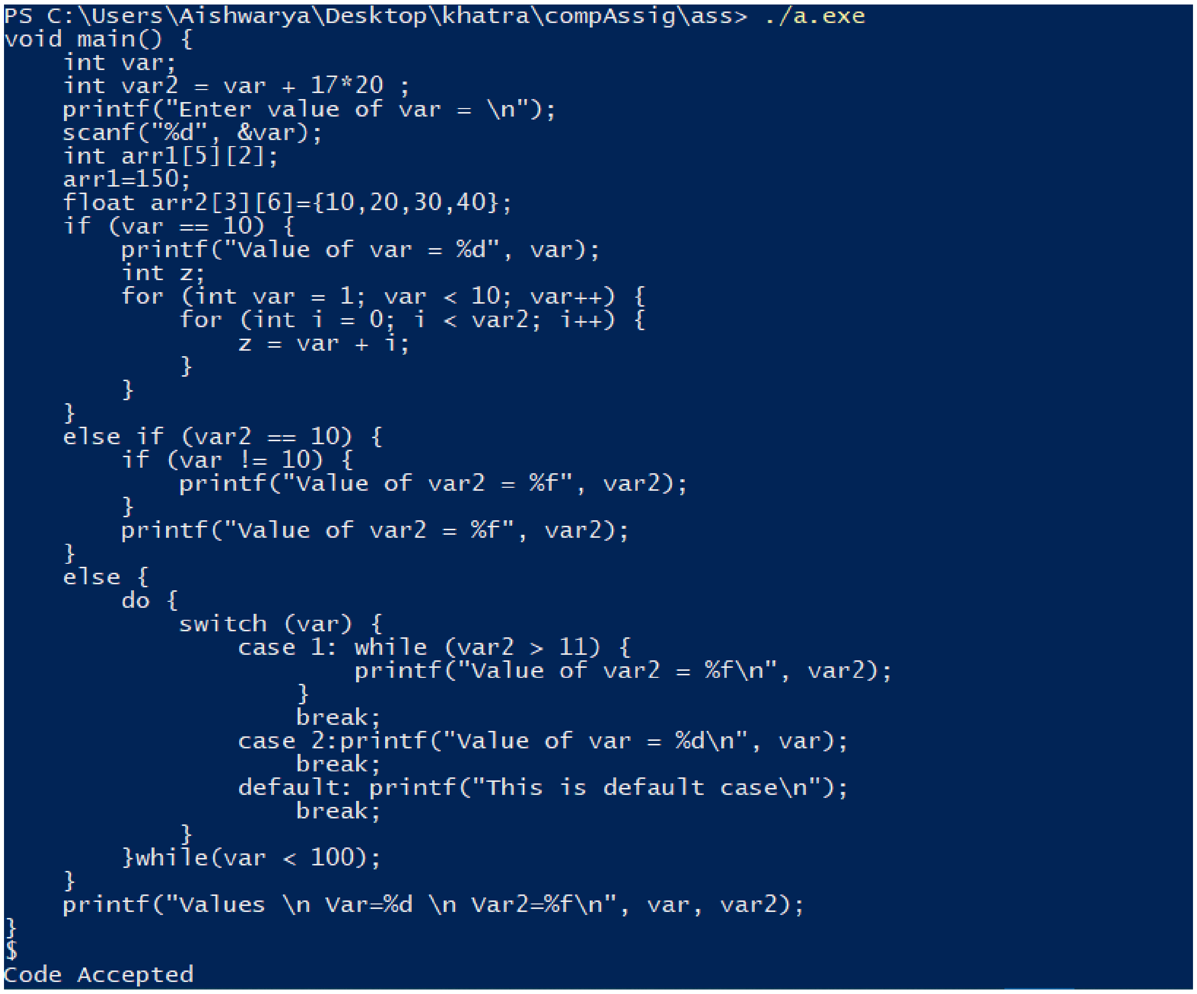
The above screenshot error has been encountered as there is no closing braces for the function started, resulting in the parser to fail and display of syntax error.



```
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass> ./a.exe
void main(){
 int a;
 int b;
 int c;
 a+b=c;
syntax error
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass>
```

Output 7: semantic error

This error occurs when the statement written in the program are not meaningful to the compiler.



```
PS C:\Users\Aishwarya\Desktop\khatra\compAssig\ass> ./a.exe
void main() {
    int var;
    int var2 = var + 17*20 ;
    printf("Enter value of var = \n");
    scanf("%d", &var);
    int arr1[5][2];
    arr1=150;
    float arr2[3][6]={10,20,30,40};
    if (var == 10) {
        printf("Value of var = %d", var);
        int z;
        for (int var = 1; var < 10; var++) {
            for (int i = 0; i < var2; i++) {
                z = var + i;
            }
        }
    }
    else if (var2 == 10) {
        if (var != 10) {
            printf("Value of var2 = %f", var2);
        }
        printf("Value of var2 = %f", var2);
    }
    else {
        do {
            switch (var) {
                case 1: while (var2 > 11) {
                        printf("Value of var2 = %f\n", var2);
                    }
                    break;
                case 2:printf("Value of var = %d\n", var);
                    break;
                default: printf("This is default case\n");
                    break;
            }
        }while(var < 100);
    }
    printf("Values \n Var=%d \n Var2=%f\n", var, var2);
}
$
Code Accepted
```

Output8: with no error

The screenshot above shows the code written in C programming language accepted by compiler developed, according to the problem where a compiler is to be developed for a subset of C language with the features such as minimum two datatype, control statement, looping statements, the input/output functions, compound statements and 2D array.

CONCLUSION:

A compiler is developed for the subset of C programming language for the features such as: -

Minimum two data types, minimum two control statements, minimum two looping statements, Input-output functions and Compound statements and two-dimensional Array. The compiler developed is able to handle nested loops, array(2D), input output functions as well as compound statements for C programming language. Here, first the resulting tokens are identified from the lex file and grouped. The lex program is translated to a program which reads an input stream, copying it to an output stream and dividing the input into strings which match the given expressions. As each string is recognized, the program fragment corresponding to that string is executed.

As the program developed is the subset of the C-compiler, hence all the functionalities of the C compiler are not considered. The functionalities lacking can be included such as including the header file in the input, the datatypes considered are float and int, the datatypes such as string, double is not included, the ternary operator is not included and arithmetic operator such as pow is not included which can be included with in further developments in the compiler. Hence, including these the flexibility of the program will increase for large programs which include multiple functions.