

ASSIGNMENT

Course Code	CSC301A
Course Name	Compilers
Programme	B.Tech
Department	Computer Science and Engineering
Faculty	Faculty of Engineering and Technology

Name of the Student	Shikhar Singh
Reg. No	17ETCS002168
Semester/Year	6 th Semester/ 2020
Course Leader/s	Ms. Suvidha

Declaration Sheet			
Student Name	Shikhar singh		
Reg. No	17ETCS002168		
Programme	B.tech	Semester/Year	6 th sem/ 2020
Course Code	CSC301A		
Course Title	Compilers		
Course Date		to	
Course Leader	Ms. Suvidha		
<p>Declaration</p> <p>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.</p>			
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	

Declaration Sheet	2
Contents	3
List of Tables	4
List of Figures	5
Question No. 1	6
1.1 Introduction:	6
1.2 Identification and grouping of tokens:	7
1.3 Implementation in Lex:	8
1.4 Design of context free Grammar:	9
1.5 Implementation in Yacc:	11
1.6 Results and Comments:	13
Bibliography	17
Appendix A	18
Appendix B	21
Appendix C	22

Table 1: table to list all tokens and lexemes.....	7
--	---

Figure 1: phases of a compiler	6
Figure 2: Lex implementation	8
Figure 3: context free grammar	9
Figure 4: context free grammar	10
Figure 5: context free grammar	10
Figure 6: yacc implementation – tokens, start and declaration block	11
Figure 7: yacc implementation – assignment and function call block	11
Figure 8: yacc implementation – array, function, functional args, statement, while and for	12
Figure 9: yacc implementation – if, else, struct, printf, scanf, case, break, switch, expression and expression list.	12
Figure 10: yacc implementation – basic C code to print the debug output and parser result.....	13
Figure 11: test files	13
Figure 12: Makefile commands.....	14
Figure 13: run command	14
Figure 14: output for test file 3.....	14
Figure 15: output for test4.c	15
Figure 16: output for test1.c file	15
Figure 17: test2.c output	16
Figure 18: test1.c source code.	18
Figure 19: test2.c source code (analogous of test1.c with errors.)	18
Figure 20: source code of test3.c	19
Figure 21: test code of test4.c.....	20
Figure 22: source code.	21
Figure 23: running the command.	21

Solution to Question No. 1:**1.1 Introduction:**

The world is driven on programming languages. All software and mathematical computation require a framework to take instructions and these instructions are written in some programming language. But a programming language is a human extension of computing. It is not easily understandable by a computer or any computing machine at all, hence, we use computer programs (software) known as **compilers**. Compiler is a computer program that transforms source code written in a programming language into another computer language, with the latter often having a binary form known as object code. The most common reason for converting a source code is to create an executable program. The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language.

This assignment focuses on building a subset of an actual compiler (a prototype), for which we will need the following:

The phases of compilation or the compiler cycle as shown in figure 1:

Compiler tools:

1. Flex – it is a tool used to generate scanners, programs which recognizes lexical patterns in the text. Flex reads the given input file or standard input in case if no input file names are given, for a description of scanner to generate.
2. Bison – it is a general purpose parser that converts an annotated context-free grammar into deterministic LR or general LR parser which has LR (1), LALR (1) parser tables.

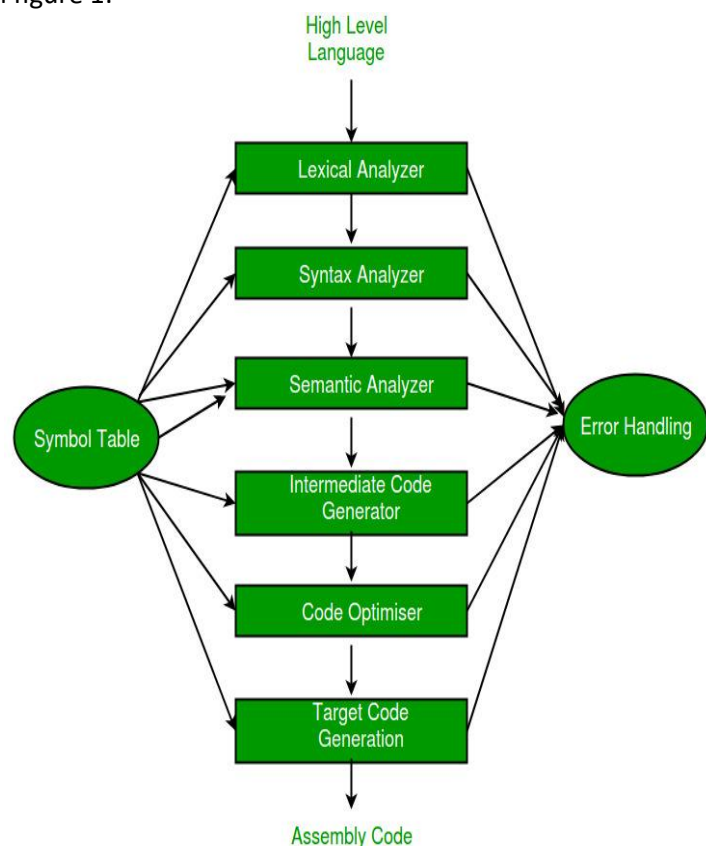


Figure 1: phases of a compiler

1.2 Identification and grouping of tokens:

Table 1: table to list all tokens and lexemes.

RE	Token
DATATYPES	
Int	INT
float	FLOAT
char	CHAR
double	DOUBLE
KEYWORDS	
Void	VOID
Main	MAIN
If	IF
Else	ELSE
switch	SWITCH
case	CASE
break	BREAK
While	WHILE
For	FOR
printf	PRINTF
scanf	SCANF
RELATIONAL OPERATORS	
>=	GE
==	EQ
<	LT
>	GT
!=	NE
<=	LE
ARITHMETIC OPERATORS	
%	PERCENT
&	AMPERSEND
++	INC
--	DEC

LOGICAL OPERATORS	
&&	AND
	OR
LITERALS	
[a-zA-Z]	Alpha
[0-9]	digit
{digit}+	NUM
{alpha}({alpha} {digit})*	ID (identifier)

Appendix C contains the tokens generated and enumerated in yytoken in a.tab.c file.

1.3 Implementation in Lex:

```

alpha [a-zA-Z]
digit [0-9]

%%
[ \t]      ;          /*Assigning tab space here*/
[ \n]      {yylineno = yylineno + 1;}
int return INT;
float return FLOAT;
char return CHAR;
double return DOUBLE;    /*grouping data types*/

for return FOR;
while return WHILE;      /*grouping looping statements*/

if return IF;
else return ELSE;
switch return SWITCH;
case return CASE;
break return BREAK;      /*grouping control statements */

printf return PRINTF;
scanf return SCANF;      /*grouping Input-output functions */

void return VOID;
^"#include ".+ ;          /*grouping Reserved keywords*/
{digit}+      { yylval=atoi(yytext); return NUM ; }    /*Defining NUM as a set of digit which already had been declared above*/
{alpha}({alpha}|{digit})* return ID; /*Defining ID as a set of digit and alpha which already had been declared above*/
"<=" return LE;
">=" return GE;
"==" return EQ;
"!=" return NE;
">" return GT;
"<" return LT;
"." return DOT;
"%" return PERCENT;
"&" return AMPERSEND;
"++" return INC;
"--" return DEC;        /*grouping all the operators*/
VV.* ;
V\*(.*\n)*.*\V ;      /* To eat up all the other spaces tabs and other escape sequence characters in the input clutter*/
.      return yytext[0]; /*Return yytext to 0 when the lex file execute without any error*/
%%

```

Figure 2: Lex implementation

1.4 Design of context free Grammar:

List of statements included in the compiler:

❖ data types:

- int
- float
- char
- double
- void

❖ control statements:

- if
- else
- switch-case

❖ looping statements:

- for
- while

❖ input-output statements:

- scanf
- printf

❖ compound statement and 1-D/2-D array:

- { block }
- array [index]
- array [index] [index]

```
start: Function
      | Declaration
      ;

/* Declaration block */
Declaration: Type Assignment ';'
            | Assignment ';'
            | FunctionCall ';'
            | ArrayUsage ';'
            | Type ArrayUsage ';'
            | error
            ;

/* Assignment block */
Assignment: ID '=' Assignment
           | ID '=' FunctionCall
           | ID '=' ArrayUsage
           | ArrayUsage '=' Assignment
           | ID ',' Assignment
           | NUM ',' Assignment
           | ID '+' Assignment
           | ID '-' Assignment
           | ID '*' Assignment
           | ID '/' Assignment
           | NUM '+' Assignment
           | NUM '-' Assignment
           | NUM '*' Assignment
           | NUM '/' Assignment
           | INC Assignment
           | Assignment INC
           | DEC Assignment
           | Assignment DEC
           | '\'' Assignment '\''
           | '(' Assignment ')'
           | '-' '(' Assignment ')'
           | '{' Assignment '}'
           | '-' NUM
           | '-' ID
           | NUM
           | ID
           ;

/* Function Call Block */
FunctionCall : ID '(' ')'
             | ID '(' Assignment ')'
             ;
```

Figure 3: context free grammar

Figure 3 shows the context free grammar for start, which recognizes a function which is how we write a statement block in C i.e. a driver function like main. And a declaration which consists of all possible chunks of code that lies outside of a function.

The declaration block consists of various inline statements that can be written in C language. These statements are easily understood by the naming given to them in the declaration block.

The assignment block contains all the final grammar rules which the parse tree can finally divulge into.

At last is the general function call block which governs the rules for a function call.

```

/* Array Usage */
ArrayUsage: ID '[' Assignment ']'
| ID '[' Assignment ']' '[' Assignment ']'
;

/* Function block */
Function: Type ID '(' ArgList ')' CompoundStmt
;
ArgList: ArgList ',' Arg
;
Arg: Type ID
;
CompoundStmt: '{' StmtList '}'
;
StmtList: StmtList Stmt
;
;
Stmt: WhileStmt
| Declaration
| ForStmt
| IfStmt
| PrintFunc
| ScanFunc
| SwitchFunc
| CaseStmt
| BreakFunc
| Assignment
| ';'
;

/* Type Identifier block */
Type: INT
| FLOAT
| CHAR
| DOUBLE
| VOID
;

/* While Blocks */
WhileStmt: WHILE '(' Expr ')' Stmt
| WHILE '(' Expr ')' CompoundStmt
;

```

Figure 4: context free grammar

Figure 5 have all the remaining statement blocks i.e., for loop, if statement, structure statement, print function, scan function, case statement, switch statement, break statement. This is lastly followed by an expression list and an expression block which further calls the assignment and array usage block whenever and wherever necessary.

Appendix C contains the grammar generated.

Figure 4 contains the grammar rules for array declaration (for both 1D and 2D array), below that is the function block which contains the rules for writing a simple C function.

This block is followed by an argument list block, which is further followed by an argument block.

a statement list block is an extension of compound statement which can generate multiple statements whenever and wherever necessary.

The statement block contains all the rules of possible statements that can be used in this subset of C compiler.

Below the statement block are the individual blocks for each kind of statement listed in the statement block.

```

/* For Block */
ForStmt: FOR '(' Type Expr ';' Expr ';' Expr ')' '{' StmtList '}'
| FOR '(' Expr ';' Expr ';' Expr ')' CompoundStmt
| FOR '(' Expr ')' Stmt
| FOR '(' Expr ')' CompoundStmt
;

/* IfStmt Block */
IfStmt: IF '(' Expr ')' Stmt ELSE Stmt
| IF '(' Expr ')' Stmt ELSE CompoundStmt
| IF '(' Expr ')' CompoundStmt
| IF '(' Expr ')' CompoundStmt ELSE CompoundStmt
;

/* Struct Statement */
StructStmt: STRUCT ID '{' Type Assignment '}'
;

/* Print Function */
PrintFunc: PRINTF '(' '\n' Expr '\n' ')' ';'
| PRINTF '(' '\n' ExprList '\n' ')' ';'
| PRINTF '(' '\n' Expr '\n' ',' Expr ')' ';'
;

/* Scan Function */
ScanFunc: SCANF '(' '\n' ExprList '\n' ',' Expr ')' ';'
;

/* Case Statement */
CaseStmt: CASE NUM ':' Stmt
;

/* Break Function */
BreakFunc: BREAK ';'
;

/* Switch Function */
SwitchFunc: SWITCH '(' Expr ')' '{' StmtList '}'
;

/* Expression Block */
ExprList: ExprList Expr
| Expr
;
Expr: Assignment
| ArrayUsage
;

```

Figure 5: context free grammar

1.5 Implementation in Yacc:

```
%{
#include <stdio.h>
#include <stdlib.h>
extern FILE *fp;
%}
%token INT FLOAT CHAR DOUBLE VOID
%token FOR WHILE
%token SWITCH CASE BREAK
%token IF ELSE PRINTF SCANF
%token STRUCT
%token NUM ID
%token INCLUDE
%token DOT
%right '='
%left AND OR
%left '<' '>' LE GE EQ NE LT GT INC DEC
%left PERCENT AMPERSEND
%%
/* start: Function
| Declaration
;
*/
/* Declaration block */
/* Declaration: Type Assignment ';'
| Assignment ';' { printf("\nExpression accepted and evaluated");printf("\nResult=%d\n", $$); }
| FunctionCall ';'
| ArrayUsage ';'
| Type ArrayUsage ';'
| StructStmt ';'
| error
;
;
```

Figure 6: yacc implementation – tokens, start and declaration block

```
/* Assignment block */
/* Assignment: ID '=' Assignment
| ID '=' FunctionCall
| ID '=' ArrayUsage
| ArrayUsage '=' Assignment
| ID ',' Assignment
| NUM ',' Assignment
| ID '+' Assignment
| ID '-' Assignment
| ID '*' Assignment
| ID '/' Assignment
| NUM '+' Assignment { $$ = $1 + $3; }
| NUM '-' Assignment { $$ = $1 - $3; }
| NUM '*' Assignment { $$ = $1 * $3; }
| NUM '/' Assignment { $$ = $1 / $3; }
| NUM OR Assignment { $$ = $1 || $3; }
| NUM AND Assignment { $$ = $1 && $3; }
| NUM LE Assignment { $$ = $1 <= $3; }
| NUM GE Assignment { $$ = $1 >= $3; }
| NUM NE Assignment { $$ = $1 != $3; }
| NUM EQ Assignment { $$ = $1 == $3; }
| NUM GT Assignment { $$ = $1 > $3; }
| NUM LT Assignment { $$ = $1 < $3; }
| ID OR Assignment
| ID AND Assignment
| ID LE Assignment
| ID GE Assignment
| ID NE Assignment
| ID EQ Assignment
| ID GT Assignment
| ID LT Assignment
| INC Assignment
| Assignment INC
| DEC Assignment
| Assignment DEC
| PERCENT Assignment
| AMPERSEND Assignment
| '{' Assignment '}'
| NUM
| ID
;
*/
/* Function Call Block */
/* FunctionCall: ID '(' ')'
| ID '(' Assignment ')'
;
;
```

Figure 7: yacc implementation – assignment and function call block

```

/* Array Usage */
ArrayUsage: ID '[' Assignment ']'
| ID '[' Assignment ']' '[' Assignment ']'
;

/* Function block */
Function: Type ID '(' AngList ')' CompoundStmt
;
AngList: AngList ',' Ang
| Arg
;
Ang: Type ID
;
CompoundStmt: '{' StmtList '}'
;
StmtList: StmtList Stmt
|
;
Stmt: WhileStmt
| Declaration
| ForStmt
| IfStmt
| PrintFunc
| ScanFunc
| SwitchFunc
| CaseStmt
| BreakFunc
| Assignment
| ';'
;

/* Type Identifier block */
Type: INT
| FLOAT
| CHAR
| DOUBLE
| VOID
;

/* While Blocks */
WhileStmt: WHILE '(' Expr ')' Stmt
| WHILE '(' Expr ')' CompoundStmt
;

/* For Block */
ForStmt: FOR '(' Type Expr ';' Expr ';' Expr ')' '{' StmtList '}'
| FOR '(' Expr ';' Expr ';' Expr ')' CompoundStmt
| FOR '(' Expr ')' Stmt
| FOR '(' Expr ')' CompoundStmt
;

```

Figure 8: yacc implementation – array, function, functional args, statement, while and for

```

/* IfStmt Block */
IfStmt: IF '(' Expr ')' Stmt ELSE Stmt
| IF '(' Expr ')' Stmt
| IF '(' Expr ')' Stmt ELSE CompoundStmt
| IF '(' Expr ')' CompoundStmt
| IF '(' Expr ')' CompoundStmt ELSE CompoundStmt
;

/* Struct Statement */
StructStmt: STRUCT ID '{' Type Assignment '}'
;

/* Print Function */
PrintFunc: PRINTF '(' '\n' Expr '\n' ')' ';'
| PRINTF '(' '\n' ExprList '\n' ')' ';'
| PRINTF '(' '\n' Expr '\n' ',' Expr ')' ';'
;

/* Scan Function */
ScanFunc: SCANF '(' '\n' ExprList '\n' ',' Expr ')' ';'
;

/* Case Statement */
CaseStmt: CASE NUM ':' Stmt
;

/* Break Function */
BreakFunc: BREAK ';'
;

/* Switch Function */
SwitchFunc: SWITCH '(' Expr ')' '{' StmtList '}'
;

/* Expression Block */
ExprList: ExprList Expr
| Expr
;
Expr: Assignment
| ArrayUsage
;

```

Figure 9: yacc implementation – if, else, struct, printf, scanf, case, break, switch, expression and expression list.

```

#include"lex.yy.c"
#include<ctype.h>

int flag, flag2=1;
int main(int argc, char *argv[])
{
    yyin = fopen(argv[1], "r");
    printf("\n----OUTPUT----\n");
    flag = yyparse();
    if(!flag)
    {
        if(flag2)
        {
            printf("\nParsing completed successfully without any errors.\n");
        }
        else
            printf("\nParsing unsuccessful\n");
    }

    else
        printf("\nParsing failed\n");

    fclose(yyin);
    return 0;
}

yyerror(char const *s) {
    flag2=0;
    printf("=> line %d: %s %s\n", yylineno, s, yytext );
}

```

Figure 10: yacc implementation – basic C code to print the debug output and parser result.

1.6 Results and Comments:

NOTE: The source code of all the files can be seen in the APPENDIX A at the end of this report.

Different test cases are created to analyze and validate the compiler as seen in the figure 11:



Figure 11: test files

Here, **test3.c** contains all the required test cases in the problem statement with syntactically correct C code, whereas **test4.c** contains the same code with syntax errors present in them.

Furthermore, **test1.c** contains a regular expression that can be evaluated with syntactically correct code.

Whereas, **test2.c** contains the same expression with syntax errors.

RUNNING THE CODE:

In order to make the program execution process time efficient, a **Makefile** is created to automate the task of writing a series of repetitive steps.

The source code of the make file is as follows:

```

all:
    lex program/a.l
    bison program/a.y -vgk
    gcc a.tab.c -ll -ly -w
    ./a.out test/${test}

```

Figure 12: Makefile commands

NOTE: The detailed explanation of Makefile is in Appendix B.

We can simply pass the file name to **make** command to run the program in the following manner:

“make test=[filename] -e”

The final piece of code for the same can be seen in figure 13.

```

> /mnt/c/Users/Shikhar/Desktop/compiler_assignment/compiler > master !7 ?1
make test=test4.c -e

```

Figure 13: run command

RESULT AND ANALYSIS:

Passing **test3** to the compiler yields the following output:

```

> make test=test3.c -e
lex program/a.l
bison program/a.y -vgk
gcc a.tab.c -ll -ly -w
./a.out test/test3.c

---OUTPUT---

Parsing completed successfully without any errors.
> /mnt/c/Users/S/De/compiler_assignment/compiler > master !7 ?1

```

Figure 14: output for test file 3

test3.c contains test cases for all the features as required in the problem statement. And as it can be seen from figure 13, the code gets parsed successfully without any error.

Similarly, test4.c is the analogous of test3.c but contains syntax errors in a few statements. The output for the same can be seen in the figure 14 as shown below:

```
> make test=test4.c -e
lex program/a.l
bison program/a.y -vgk
gcc a.tab.c -ll -ly -w
./a.out test/test4.c

---OUTPUT---
=> line 6: syntax error ;
=> line 12: syntax error a
=> line 13: syntax error )
=> line 43: syntax error printf
=> line 47: syntax error for

Parsing unsuccessful

[mnt/c/Users/Shikhar/Desktop/compiler_assignment/compiler] master !7 ?1
```

Figure 15: output for test4.c

Test1.c contains a regular expression to be evaluated. Passing the equation in the statement block yields the result by evaluating the expression and printing the result on the screen.

```
> make test=test1.c -e
lex program/a.l
bison program/a.y -vgk
gcc a.tab.c -ll -ly -w
./a.out test/test1.c

---OUTPUT---
Expression accepted ==> Result = 17
Expression accepted ==> Result = 1
Expression accepted ==> Result = 0
Expression accepted ==> Result = 5
Parsing completed successfully without any errors.

[mnt/c/Users/Shikhar/Desktop/compiler_assignment/compiler] master !7 ?1
```

Figure 16: output for test1.c file

The first equation passed is “2+3*5”, as it can be seen in **APPENDIX A**. And it can be seen that result achieved is 17. The scanner accepts the equation by reading the .c file and passes it to the parser. When the parser encounters it, it passes it on to the grammar for validation which when gets matched, For e.g.: - when 3*5 is found, NUM*NUM gets matched and the equation is processed, and result is evaluated.

The following derivation shows the operator precedence and associativity that is maintained as it is written in the grammar file.

For 3*5:

```
Compound_Statement -> Statement_List
Compound_Statement -> Statement_List Statement
Compound_Statement -> Statement Statement
Compound_Statement -> Assignment ;
Compound_Statement -> NUM*NUM ;
Compound_Statement -> 3*5 ;
```

Similarly, test2.c is an analogous of test1.c with some syntax errors. The output can be seen here,

```
> make test=test2.c -e
lex program/a.l
bison program/a.y -vgk
gcc a.tab.c -ll -ly -w
./a.out test/test2.c

----OUTPUT----
=> line 6: syntax error int
=> line 7: syntax error ;

Parsing unsuccessful

[mnt/c/Users/Shikhar/Desktop/compiler_assignment/compile
```

Figure 17: test2.c output

Now the errors were in the lines 6 and line 7 as it can be seen in the source code in **APPENDIX A**. Hence, the output can be verified as correct.

LIMITATION:

This compiler is very limited and can only detect syntax errors of subset of statements of the original compiler.

It misses out on a ton of useful features like pointers, recursion and their implementation.

IMPROVEMENT:

This compiler can be further developed to implement all the statements and process them to give a real output rather than just telling errors. Also, it misses out on programming paradigms. Several programming features like OOPs, functional programming etc. aren't present which makes a language much more powerful and give it its identity.

All these things can be further added to the compiler and make it much better.

1. <https://www.geeksforgeeks.org/introduction-of-compiler-design/?ref=lbp>
2. <https://llvm.org/docs/tutorial/OCamlLangImpl2.html>
3. <http://www.aosabook.org/en/llvm.html>
4. <https://www.gnu.org/software/bison/>

```
compiler > test > C test1.c
1  #include <stdio.h>
2  void main()
3  {
4
5      2+3*5;    //equation 1
6      0 || 1;   //equation 2
7      1 && 0;   //equation 3
8      3+10/10-5; //equation 4
9
10 }
```

Figure 18: test1.c source code.

```
compiler > test > C test2.c
1  #include <stdio.h>
2  void main()
3  {
4      2+3*5;    //equation 1
5      0 || 1;   //equation 2
6      int 1 && 0; //equation 3
7      3+10/10-5 //equation 4
8
9  }
```

Figure 19: test2.c source code (analogous of test1.c with errors.)

```

#include <stdio.h>
void main()
{
    //data type
    int a,b;
    float f;
    double d;
    char c;
    c=a;

    // I/O statements
    scanf("%d",&a);
    printf("%d",a);

    //assignment operations
    c=a+b;

    // conditional statement
    if(a>b)
    {
        printf("%d",a);
        printf("Meow");
    }
    else
    {
        printf("Meow2");
        printf("Meow3");
    }

    switch(number)
    {
        case 1:printf("one");
        break;
    }

    //array declaration and definition
    int arr[a][b]= {1,2,3,4,5,6,7,8};
    float arr[a][b];

    //Loop statements
    while(1)
    {
        printf("Infinite");
    }
    printf("assignemt kjdsfbadsjkb asgf");

    for(int i=0; i<10; i--)
    {
        a=i;
        b=a+i;
    }
}

```

Figure 20: source code of test3.c

All the required statements from the problem in the question are listed here and grouped and labeled with comments.

```

compiler > test > C test4.c
1  #include <stdio.h>
2  void main()
3  {
4      //data type
5      int a,b;
6      float ; //syntax error
7      double d; // syntax error
8      char c;
9      //c=a;
10
11     // I/O statements
12     scanf("%d",a); //syntax error
13     printf("d,"); //syntax error
14
15     //assignment operations
16     c=a+b;
17
18     // conditional statement
19     if(a>b)
20     {
21         printf("%d",a);
22         printf("Meow");
23     }
24     else
25     {
26         printf("Meow2");
27         printf("Meow3");
28     }
29
30     switch(a) //syntax error
31     {
32         case 1:printf("one");
33         break;
34     }
35
36     //array declaration and definition
37     int arr[a][b]={1,2,3,4,5,6}; //syntax error
38     float arr[a][b]; //syntax error
39
40     //Loop statements
41     while( //syntax error
42     {
43         printf("Infinite");
44     }
45
46
47     for(int i= i<10; i--) //syntax error
48     {
49         a=i;
50         b=a+i;
51     }
52 }

```

Figure 21: test code of test4.c

This file contains the same code as in test1.c but with errors present in them. All the errors have been labeled using comments.



```
compiler > Makefile
1  all:
2      lex program/a.l
3      bison program/a.y -vgk
4      gcc a.tab.c -ll -ly -w
5      ./a.out test/${test}
6
```

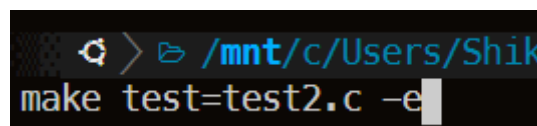
Figure 22: source code.

Makefile is a type oriented file used to automate tasks developed by GNU. It is marked by labels with a set of instructions in them. All the steps are listed in the order they are bound to be executed.

The test file name is a variable which should get the arguments from the terminal. Hence a variable test was made.

\$(test) gives the value stored in the variable.

When we run the command we give the filename to test variable and pass it with additional argument “-e”. This is used to override the environment variables so that makefile can receive the value of the variable as passed from the terminal.



```
/mnt/c/Users/Shik
make test=test2.c -e
```

Figure 23: running the command.

```

enum yytokentype
{
    INT = 258,
    FLOAT = 259,
    CHAR = 260,
    DOUBLE = 261,
    VOID = 262,
    FOR = 263,
    WHILE = 264,
    SWITCH = 265,
    CASE = 266,
    BREAK = 267,
    IF = 268,
    ELSE = 269,
    PRINTF = 270,
    SCANF = 271,
    STRUCT = 272,
    NUM = 273,
    ID = 274,
    INCLUDE = 275,
    DOT = 276,
    AND = 277,
    OR = 278,
    LE = 279,
    GE = 280,
    EQ = 281,
    NE = 282,
    LT = 283,
    GT = 284,
    INC = 285,
    DEC = 286,
    PERCENT = 287,
    AMPERSEND = 288
};
#endif

```

Above is the list of tokens generated in a.tab.c file.

Below is the grammar listed in the a.output file as generated through bison.

Grammar

0 \$accept: start \$end

1 start: Function

2 | Declaration

3 Declaration: Type Assignment ';'

4 | Assignment ';'

5 | FunctionCall ';'

6 | ArrayUsage ';'

7 | Type ArrayUsage ';'

8 | StructStmt ';'

9 | error

10 Assignment: ID '=' Assignment

11 | ID '=' FunctionCall

12 | ID '=' ArrayUsage

13 | ArrayUsage '=' Assignment

```

14      | ID ';' Assignment
15      | NUM ';' Assignment
16      | ID '+' Assignment
17      | ID '-' Assignment
18      | ID '*' Assignment
19      | ID '/' Assignment
20      | NUM '+' Assignment
21      | NUM '-' Assignment
22      | NUM '*' Assignment
23      | NUM '/' Assignment
24      | NUM OR Assignment
25      | NUM AND Assignment
26      | NUM LE Assignment
27      | NUM GE Assignment
28      | NUM NE Assignment
29      | NUM EQ Assignment
30      | NUM GT Assignment
31      | NUM LT Assignment
32      | ID OR Assignment
33      | ID AND Assignment
34      | ID LE Assignment
35      | ID GE Assignment
36      | ID NE Assignment
37      | ID EQ Assignment
38      | ID GT Assignment
39      | ID LT Assignment
40      | INC Assignment
41      | Assignment INC
42      | DEC Assignment
43      | Assignment DEC
44      | PERCENT Assignment
45      | AMPERSEND Assignment
46      | '{' Assignment '}'
47      | NUM
48      | ID

49 FunctionCall: ID '(' ')'
50      | ID '(' Assignment ')'

51 ArrayUsage: ID '[' Assignment ']'
52      | ID '[' Assignment ']' '[' Assignment ']'

53 Function: Type ID '(' ArgListOpt ')' CompoundStmt

54 ArgListOpt: ArgList
55      | %empty

56 ArgList: ArgList ',' Arg
57      | Arg

```

```

58 Arg: Type ID

59 CompoundStmt: '{' StmtList '}'

60 StmtList: StmtList Stmt
61         | %empty

62 Stmt: WhileStmt
63     | Declaration
64     | ForStmt
65     | IfStmt
66     | PrintFunc
67     | ScanFunc
68     | SwitchFunc
69     | CaseStmt
70     | BreakFunc
71     | Assignment
72     | ';'

73 Type: INT
74     | FLOAT
75     | CHAR
76     | DOUBLE
77     | VOID

78 WhileStmt: WHILE '(' Expr ')' Stmt
79         | WHILE '(' Expr ')' CompoundStmt

80 ForStmt: FOR '(' Type Expr ';' Expr ';' Expr ')' '{' StmtList '}'
81     | FOR '(' Expr ';' Expr ';' Expr ')' CompoundStmt
82     | FOR '(' Expr ')' Stmt
83     | FOR '(' Expr ')' CompoundStmt

84 IfStmt: IF '(' Expr ')' Stmt ELSE Stmt
85     | IF '(' Expr ')' Stmt
86     | IF '(' Expr ')' Stmt ELSE CompoundStmt
87     | IF '(' Expr ')' CompoundStmt
88     | IF '(' Expr ')' CompoundStmt ELSE CompoundStmt

89 StructStmt: STRUCT ID '{' Type Assignment '}'

90 PrintFunc: PRINTF '(' "" Expr "" ')' ';'
91     | PRINTF '(' "" ExprList "" ')' ';'
92     | PRINTF '(' "" Expr "" ',' Expr ')' ';'

93 ScanFunc: SCANF '(' "" ExprList "" ',' Expr ')' ';'

94 CaseStmt: CASE NUM ':' Stmt

95 BreakFunc: BREAK ';'

```


96 SwitchFunc: SWITCH '(' Expr ')' '{' StmtList '}'

97 ExprList: ExprList Expr

98 | Expr

99 Expr: Assignment

100 | ArrayUsage