# Compiler Design

## Report Of Phase - 2

## Parser / Syntax Checker

**By : Shishir Gangwar (15CO147)**
**Saumyadip Mandal (15CO144)**

# Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

# Specification Of Yacc

Like lex, yacc has its own specification language. A yacc specification is structured along the same lines as a Lex specification.

```
%{
  /* C declarations and includes */
%}
  /* Yacc token and type declarations */
%%
  /* Yacc Specification
 in the form of grammar rules like this:
  */
  symbol    :       symbols tokens
                {  $$ = my_c_code($1); }
       ;
%%
  /* C language program (the rest) */
```

The Yacc Specification rules are the place where you "glue" the various tokens together that lex has conveniently provided to you.

Each grammar rule defines a symbol in terms of:

1) other symbols
2) tokens (or terminal symbols) which come from the lexer.

Each rule can have an associated action, which is executed after all the component symbols of the rule have been parsed. Actions are basically C-program statements surrounded by curly braces.

# Lexer

The lexer will provide us with the following tokens:

1) Each possible keyword (TITLE, MENU, etc) as a separate token
2) A LABEL token for representing labels
3) A EXEC token for executable commands
4) Icon file names will be represented as 3 separate tokens: '<' LABEL '>'

To aid in error-recovery, newline characters will be considered significant, and will be passed up as a separate token (unless they are preceded by a '\').

The key aspects of the lexer are:

Each keyword returns a separate, unique token

A LABEL is either an identifier (loosely speaking) or an arbitrary string in quotes, or any alternating sequence of these two things

1) An EXEC token is identified by the fact that:
2) it is not a keyword.
3) it appears after we have scanned a LABEL token on the same line.
4) This is achieved by setting the start-condition ACT when we scan the LABEL token.

We use lex's "first match" rule to ensure that keywords get priority over the corresponding LABEL and EXEC interpretations, and that the EXEC interpretation gets priority over the LABEL interpretation in the state ACT.

It is thus essential that, where a keyword may appear on a line, the length of the other rules (for LABEL or EXEC) be no longer than the keyword rule. Otherwise, lex's "longest match" rule would override the "first match" rule.

# Tokens

We have included a lot of return statements, but have not defined the tokens in the brackets. A token is simply a unique integer which yacc associates with an item that the lexer has found. Within yacc, all tokens must be declared in the "Yacc Declarations" section of our yacc specification, like this:

```
%token    TITLE
%token    MENU
```

# The yyerror() function

The yyerror() function is called when yacc encounters an invalid syntax. The yyerror() is passed a single string (char*) argument. Unfortunately, this string usually just says "parse error", so on its own, it's pretty useless. Error recovery is an important topic which we will cover in more detail later on. For now, we just want a basic yyerror() function like this:

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

# Implementations Details

First we retrieved all the tokens from the lexical analyser.The strings which matches the the grammar of lex are known as tokens.Also in the lexer part the comments part has been taken care of.The comment is ignored whenever it is encountered.All the data types , arrays , functions, if else conditions and loops which are there in our abstract are written in lex language.Operators are also taken care.Now in the parser file we have called the tokens which are returned from the lexical phase.All the operators are written in the priority wise so that the problem of precedence is taken care off. In the operator's %left indicate that the precedence is from left to right and %right indicate right to left.Also %left means left recursive and %right means right recursive.

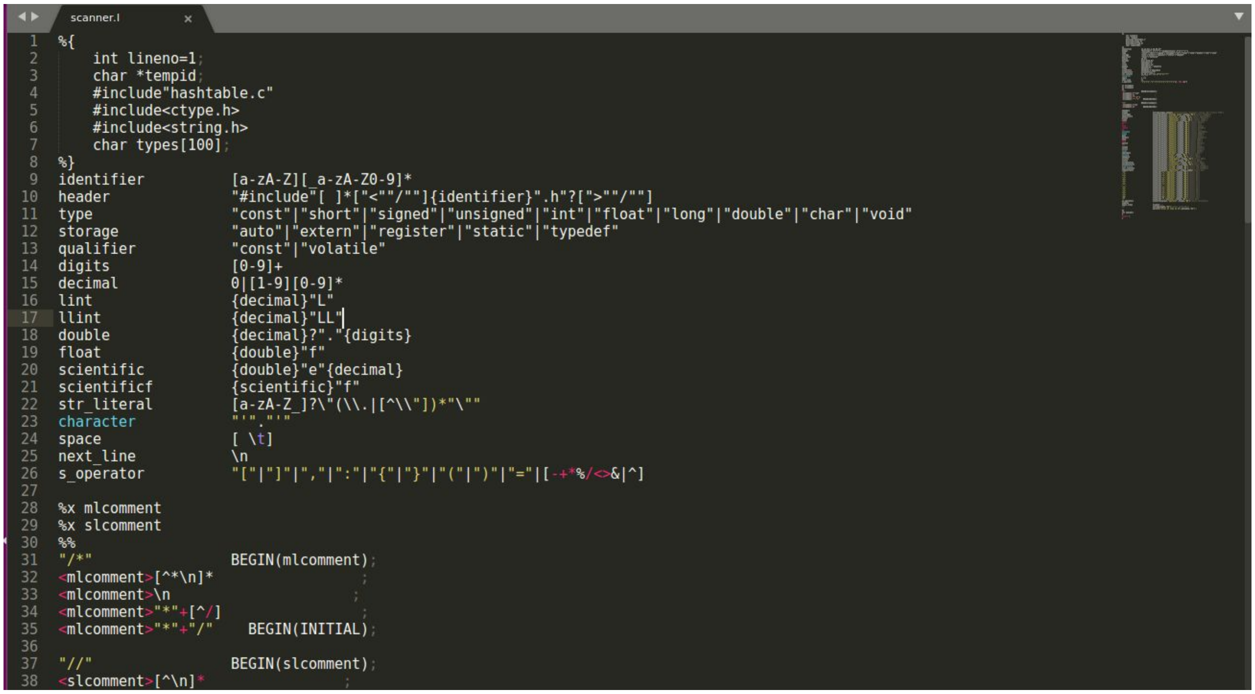
Now we have started our grammar by the start keyword,then followed by again start and function or declaration or start or it can can be empty also.The empty part is taken care by including the keyword empty which means there is nothing which matches the grammar.For structure the STRUCT keyword is defined followed by ‘ { ‘ start ‘ } ‘ ,which means inside the structure we can define the start state. The function declaration takes type , ID then argument declaration in the parameter part then followed by the compound statements.statement can contain statement again or for statement or while statement or if statement or return statement in it followed by semicolon.Then we have defined the function by defining the grammar as first will be return type then variable name which is ID here the parenthesis then followed by argument declaration then statements.Then we have declaration statements which can be expression , structure or function call.We have also used error keyword which is for handling multi line errors.

Then we have defined the statement , it can be any compound statement or while loop statement, for loop statement, declaration, print, scan statement.Compound statement is a statement list enclose in curly braces.Declaration statement is defined by ID followed by assign which means assigning a value to the variable.

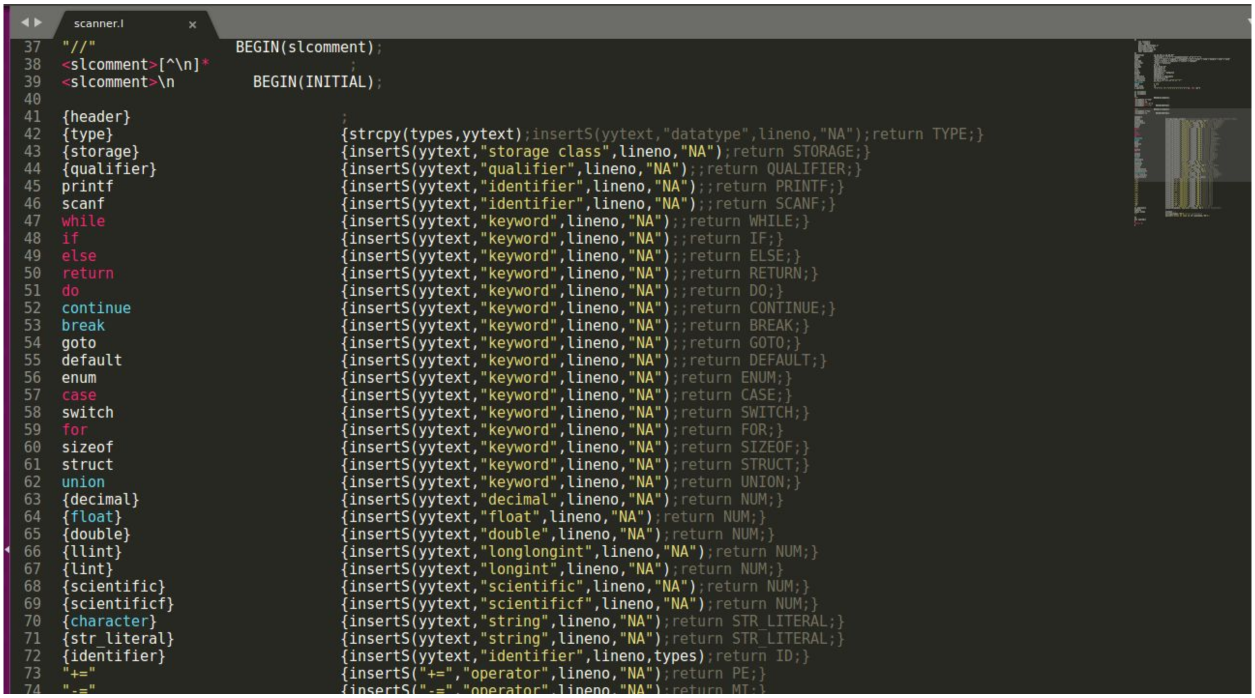
Then we have declared rules for while loop, for loop,do while cases separately in the grammar.Also we have declared rules for if and else if and else.Grammar for if statement is if statement declared then brackets then condition.Also dangling if-else part is taken care off in the yacc part.
We have also written grammar for array declaration by giving ID first followed by ‘ [ ‘ then expression then ‘ ] ’.The grammar for printf and scanf is also given in the code of the grammar.Also expr and rexpr are also defined so that it handle the cases with operators part.Also less than equal to , greater than equal to , not equal to cases are handled.Then there are operators such as ‘+’ , ‘-’ , ‘*’ , ‘/’ , etc are handled.

At the end if parsing is completed then we have shown that Parsing is complete.If parsing is not complete than we have shown that parsing error is there.Also we have declared the error line where the error will occur.We have also displayed the symbol table in which we have shown that symbol name , type , attribute and line number.

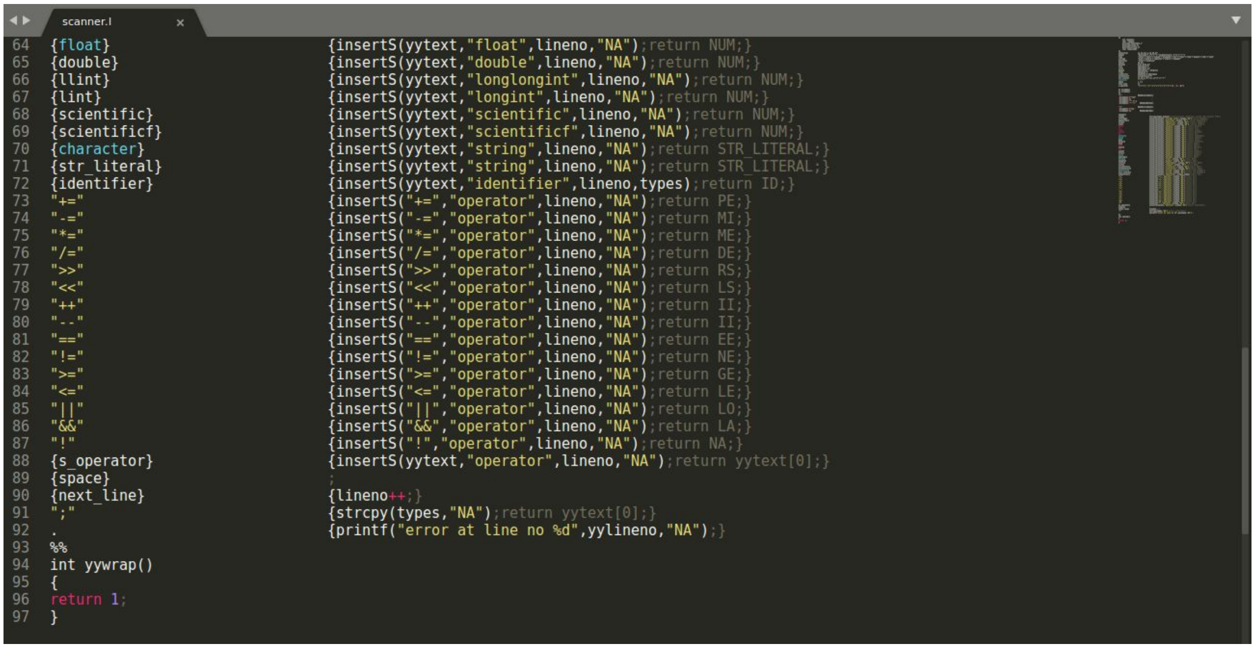
# Code for Lexical Phase



```
%{
    int lineno=1;
    char *tempid;
    #include"hashtable.c"
    #include<ctype.h>
    #include<string.h>
    char types[100];
%}
identifier          [a-zA-Z][_a-zA-Z0-9]*
header              "#include"[ ]*["<""/""]{identifier}".h"?[">""/""]
type                "const"|"short"|"signed"|"unsigned"|"int"|"float"|"long"|"double"|"char"|"void"
storage             "auto"|"extern"|"register"|"static"|"typedef"
qualifier           "const"|"volatile"
digits              [0-9]+
decimal             0|[1-9][0-9]*
lint                {decimal}"L"
llint               {decimal}"LL"
double              {decimal}?"."{digits}
float               {double}"f"
scientific          {double}"e"{decimal}
scientificf         {scientific}"f"
str_literal         [a-zA-Z_]?\"(\\.|[^\\"])*"\""
character           "'"."'"
space               [ \t]
next_line           \n
s_operator          "["|"]"|","|":"|"{"|"}"|"("|")"|"="|[-+*%/<>&|^]

%x mlcomment
%x slcomment
%%
"/*"                BEGIN(mlcomment);
<mlcomment>[^*\n]*                ;
<mlcomment>\n                    ;
<mlcomment>"*"+[^/]               ;
<mlcomment>"*"+"/"    BEGIN(INITIAL);

"//"                BEGIN(slcomment);
<slcomment>[^\n]*                ;
```

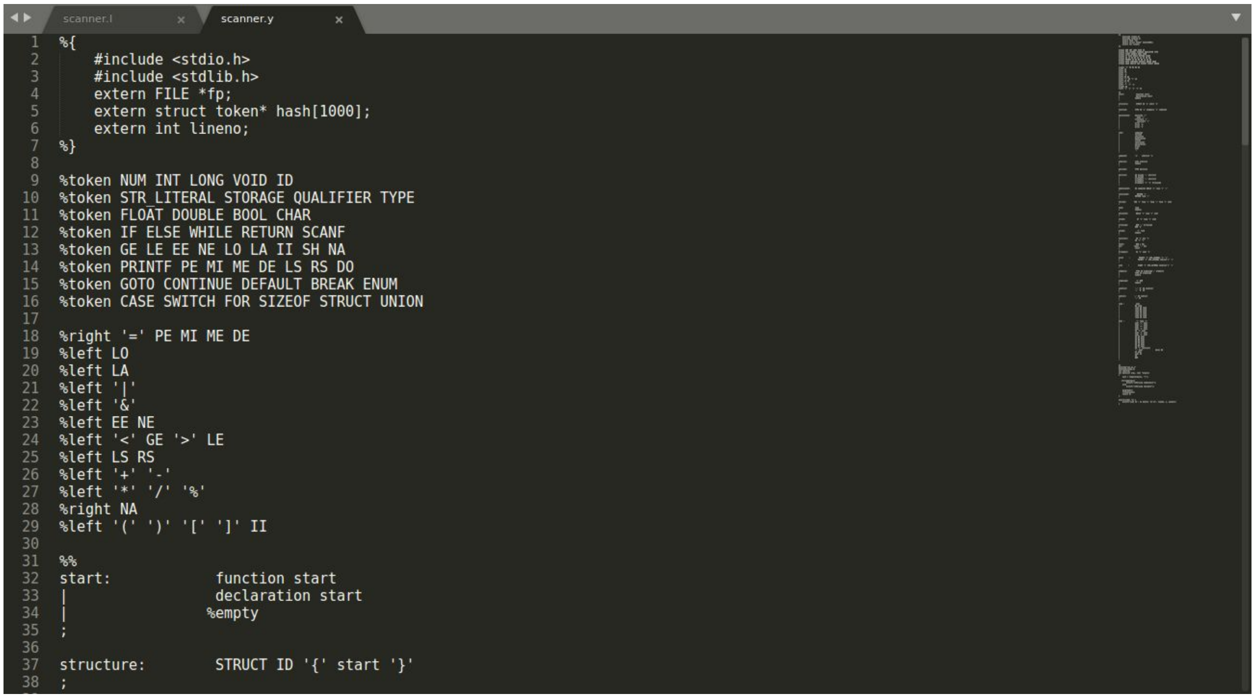


```
"//"                BEGIN(slcomment);
<slcomment>[^\n]*                ;
<slcomment>\n         BEGIN(INITIAL);

{header}                        ;
{type}                          {strcpy(types,yytext);insertS(yytext,"datatype",lineno,"NA");return TYPE;}
{storage}                       {insertS(yytext,"storage class",lineno,"NA");return STORAGE;}
{qualifier}                     {insertS(yytext,"qualifier",lineno,"NA");;return QUALIFIER;}
printf                          {insertS(yytext,"identifier",lineno,"NA");;return PRINTF;}
scanf                           {insertS(yytext,"identifier",lineno,"NA");;return SCANF;}
while                           {insertS(yytext,"keyword",lineno,"NA");;return WHILE;}
if                              {insertS(yytext,"keyword",lineno,"NA");;return IF;}
else                            {insertS(yytext,"keyword",lineno,"NA");;return ELSE;}
return                          {insertS(yytext,"keyword",lineno,"NA");;return RETURN;}
do                              {insertS(yytext,"keyword",lineno,"NA");;return DO;}
continue                        {insertS(yytext,"keyword",lineno,"NA");;return CONTINUE;}
break                           {insertS(yytext,"keyword",lineno,"NA");;return BREAK;}
goto                            {insertS(yytext,"keyword",lineno,"NA");;return GOTO;}
default                         {insertS(yytext,"keyword",lineno,"NA");;return DEFAULT;}
enum                            {insertS(yytext,"keyword",lineno,"NA");return ENUM;}
case                            {insertS(yytext,"keyword",lineno,"NA");return CASE;}
switch                          {insertS(yytext,"keyword",lineno,"NA");return SWITCH;}
for                             {insertS(yytext,"keyword",lineno,"NA");return FOR;}
sizeof                          {insertS(yytext,"keyword",lineno,"NA");return SIZEOF;}
struct                          {insertS(yytext,"keyword",lineno,"NA");return STRUCT;}
union                           {insertS(yytext,"keyword",lineno,"NA");return UNION;}
{decimal}                       {insertS(yytext,"decimal",lineno,"NA");return NUM;}
{float}                         {insertS(yytext,"float",lineno,"NA");return NUM;}
{double}                        {insertS(yytext,"double",lineno,"NA");return NUM;}
{llint}                         {insertS(yytext,"longlongint",lineno,"NA");return NUM;}
{lint}                          {insertS(yytext,"longint",lineno,"NA");return NUM;}
{scientific}                    {insertS(yytext,"scientific",lineno,"NA");return NUM;}
{scientificf}                   {insertS(yytext,"scientificf",lineno,"NA");return NUM;}
{character}                     {insertS(yytext,"string",lineno,"NA");return STR_LITERAL;}
{str_literal}                   {insertS(yytext,"string",lineno,"NA");return STR_LITERAL;}
{identifier}                    {insertS(yytext,"identifier",lineno,types);return ID;}
"+="                            {insertS("+=","operator",lineno,"NA");return PE;}
"-="                            {insertS("-=","operator",lineno,"NA");return MI;}
```

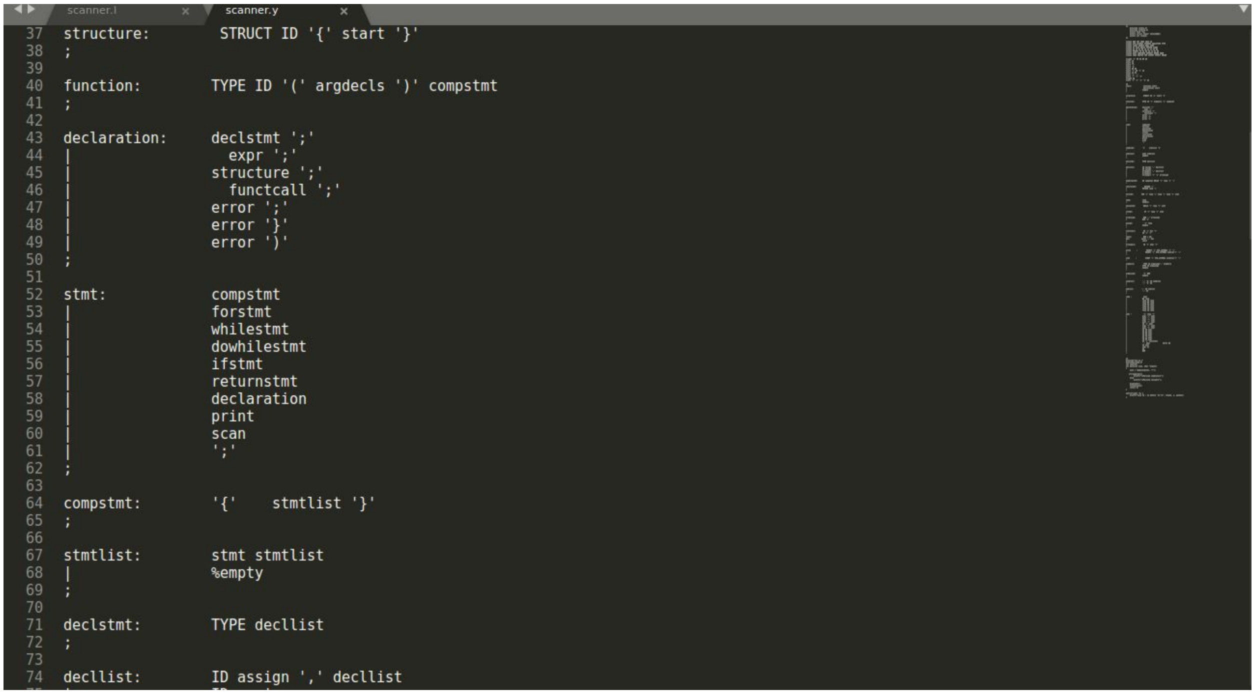


```
{float}                         {insertS(yytext,"float",lineno,"NA");return NUM;}
{double}                        {insertS(yytext,"double",lineno,"NA");return NUM;}
{llint}                         {insertS(yytext,"longlongint",lineno,"NA");return NUM;}
{lint}                          {insertS(yytext,"longint",lineno,"NA");return NUM;}
{scientific}                    {insertS(yytext,"scientific",lineno,"NA");return NUM;}
{scientificf}                   {insertS(yytext,"scientificf",lineno,"NA");return NUM;}
{character}                     {insertS(yytext,"string",lineno,"NA");return STR_LITERAL;}
{str_literal}                   {insertS(yytext,"string",lineno,"NA");return STR_LITERAL;}
{identifier}                    {insertS(yytext,"identifier",lineno,types);return ID;}
"+="                            {insertS("+=","operator",lineno,"NA");return PE;}
"-="                            {insertS("-=","operator",lineno,"NA");return MI;}
"*="                            {insertS("*=","operator",lineno,"NA");return ME;}
"/="                            {insertS("/=","operator",lineno,"NA");return DE;}
">>"                            {insertS(">>","operator",lineno,"NA");return RS;}
"<<"                            {insertS("<<","operator",lineno,"NA");return LS;}
"++"                            {insertS("++","operator",lineno,"NA");return II;}
"--"                            {insertS("--","operator",lineno,"NA");return II;}
"=="                            {insertS("==","operator",lineno,"NA");return EE;}
"!="                            {insertS("!=","operator",lineno,"NA");return NE;}
">="                            {insertS(">=","operator",lineno,"NA");return GE;}
"<="                            {insertS("<=","operator",lineno,"NA");return LE;}
"||"                            {insertS("||","operator",lineno,"NA");return LO;}
"&&"                            {insertS("&&","operator",lineno,"NA");return LA;}
"!"                             {insertS("!","operator",lineno,"NA");return NA;}
{s_operator}                    {insertS(yytext,"operator",lineno,"NA");return yytext[0];}
{space}                         ;
{next_line}                     {lineno++;}
";"                             {strcpy(types,"NA");return yytext[0];}
.                               {printf("error at line no %d",yylineno,"NA");}
%%
int yywrap()
{
return 1;
}
```

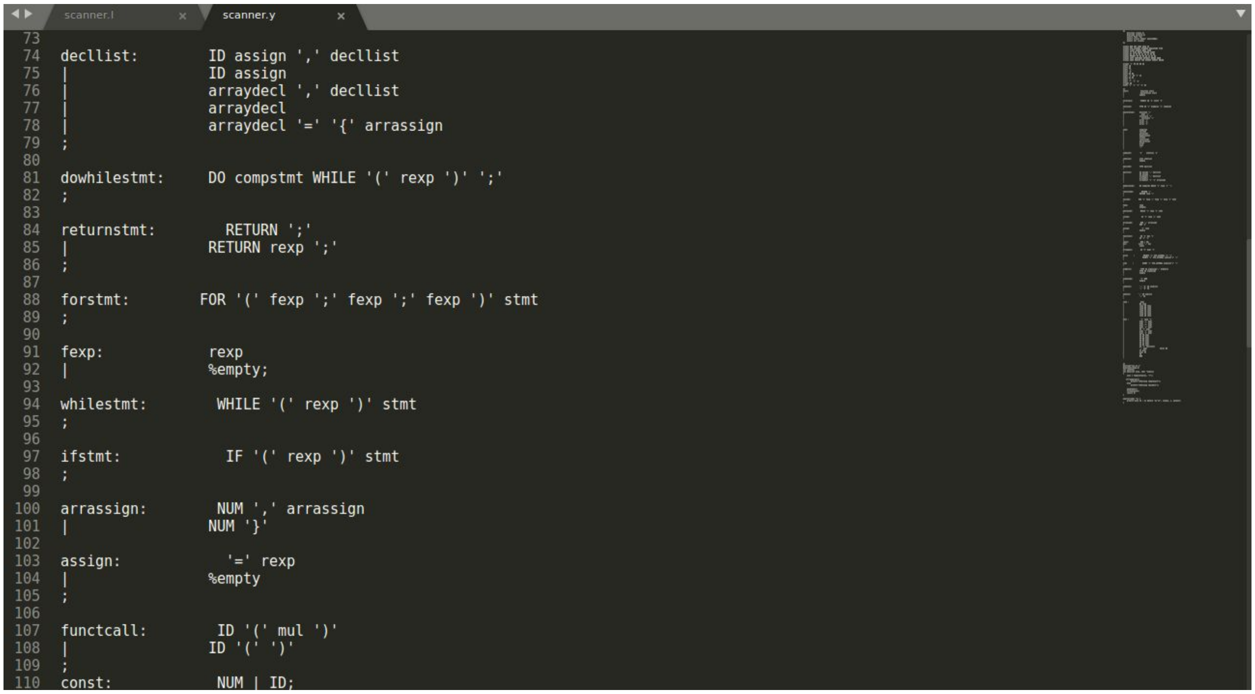
# Code For Yacc



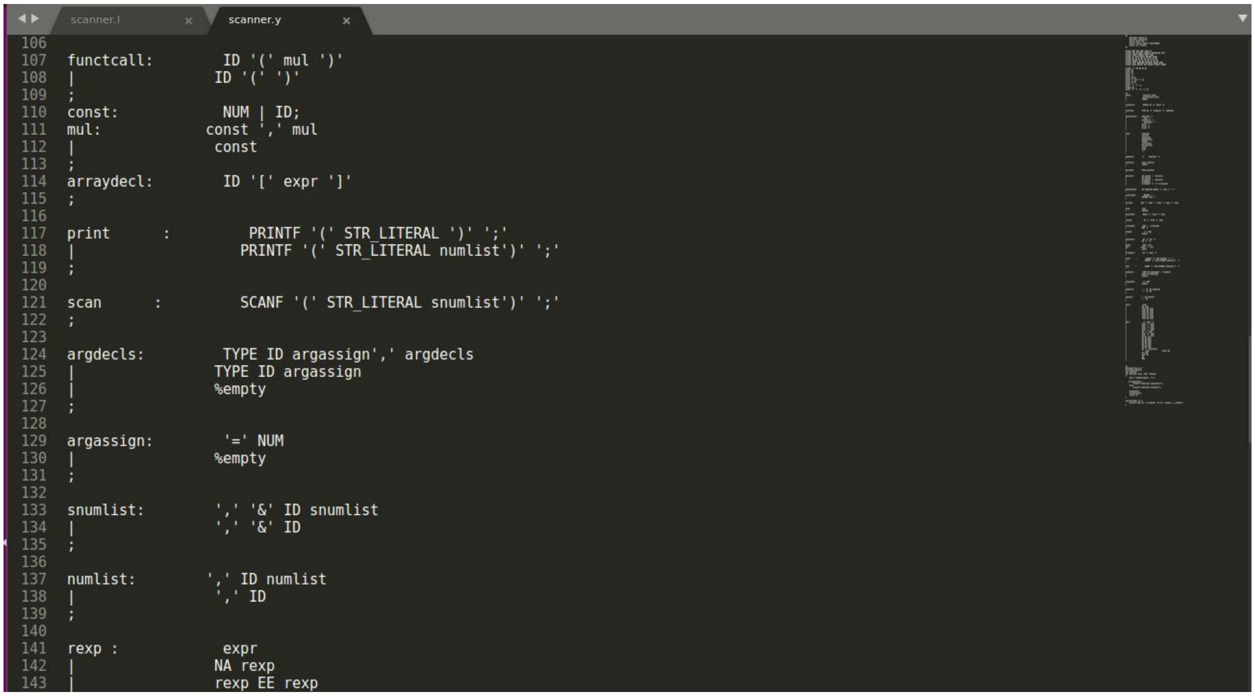
```
%{
    #include <stdio.h>
    #include <stdlib.h>
    extern FILE *fp;
    extern struct token* hash[1000];
    extern int lineno;
%}

%token NUM INT LONG VOID ID
%token STR_LITERAL STORAGE QUALIFIER TYPE
%token FLOAT DOUBLE BOOL CHAR
%token IF ELSE WHILE RETURN SCANF
%token GE LE EE NE LO LA II SH NA
%token PRINTF PE MI ME DE LS RS DO
%token GOTO CONTINUE DEFAULT BREAK ENUM
%token CASE SWITCH FOR SIZEOF STRUCT UNION

%right '=' PE MI ME DE
%left LO
%left LA
%left '|'
%left '&'
%left EE NE
%left '<' GE '>' LE
%left LS RS
%left '+' '-'
%left '*' '/' '%'
%right NA
%left '(' ')' '[' ']' II

%%
start:            function start
|                 declaration start
|                %empty
;

structure:        STRUCT ID '{' start '}'
;
```



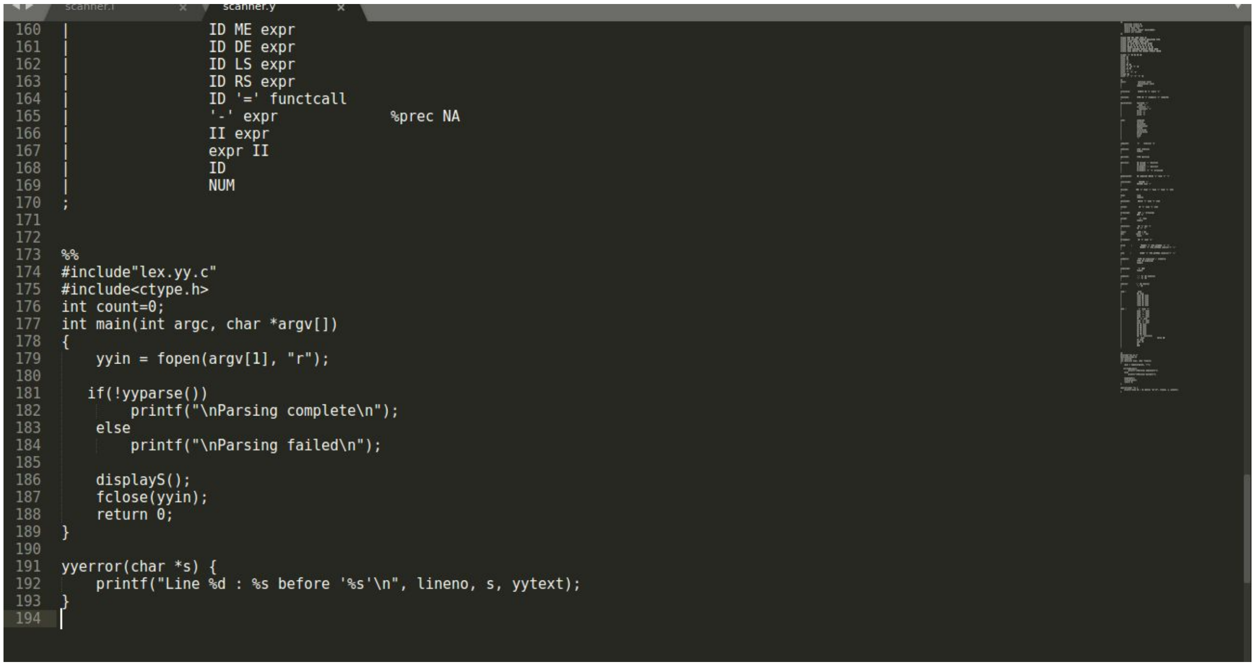
```
structure:         STRUCT ID '{' start '}'
;

function:         TYPE ID '(' argdecls ')' compstmt
;

declaration:      declstmt ';'
|                   expr ';'
|                 structure ';'
|                   functcall ';'
|                 error ';'
|                 error '}'
|                 error ')'
;

stmt:             compstmt
|                 forstmt
|                 whilestmt
|                 dowhilestmt
|                 ifstmt
|                 returnstmt
|                 declaration
|                 print
|                 scan
|                 ';'
;

compstmt:         '{'    stmtlist '}'
;

stmtlist:         stmt stmtlist
|                 %empty
;

declstmt:         TYPE decllist
;

decllist:         ID assign ',' decllist
```



```

decllist:         ID assign ',' decllist
|                 ID assign
|                 arraydecl ',' decllist
|                 arraydecl
|                 arraydecl '=' '{' arrassign
;

dowhilestmt:      DO compstmt WHILE '(' rexp ')' ';'
;

returnstmt:         RETURN ';'
|                 RETURN rexp ';'
;

forstmt:         FOR '(' fexp ';' fexp ';' fexp ')' stmt
;

fexp:             rexp
|                 %empty;

whilestmt:         WHILE '(' rexp ')' stmt
;

ifstmt:             IF '(' rexp ')' stmt
;

arrassign:         NUM ',' arrassign
|                 NUM '}'

assign:             '=' rexp
|                 %empty
;

functcall:         ID '(' mul ')'
|                 ID '(' ')'
;
const:             NUM | ID;
```

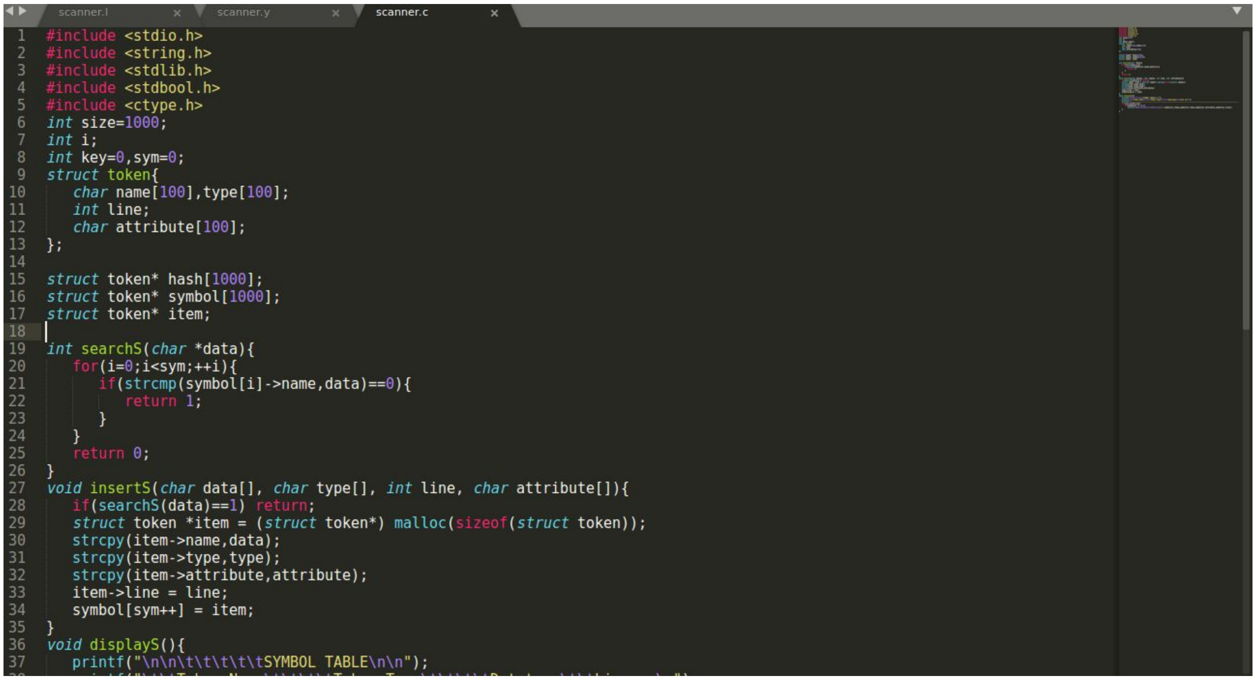


```

functcall:        ID '(' mul ')'
|                ID '(' ')'
;
const:            NUM | ID;
mul:             const ',' mul
|                 const
;
arraydecl:        ID '[' expr ']'
;

print      :         PRINTF '(' STR_LITERAL ')' ';'
|                   PRINTF '(' STR_LITERAL numlist')' ';'
;

scan      :         SCANF '(' STR_LITERAL snumlist')' ';'
;

argdecls:         TYPE ID argassign',' argdecls
|                TYPE ID argassign
|                %empty
;

argassign:        '=' NUM
|                %empty
;

snumlist:        ',' '&' ID snumlist
|                 ',' '&' ID
;

numlist:        ',' ID numlist
|                ',' ID
;

rexp :            expr
|                NA rexp
|                rexp EE rexp
```

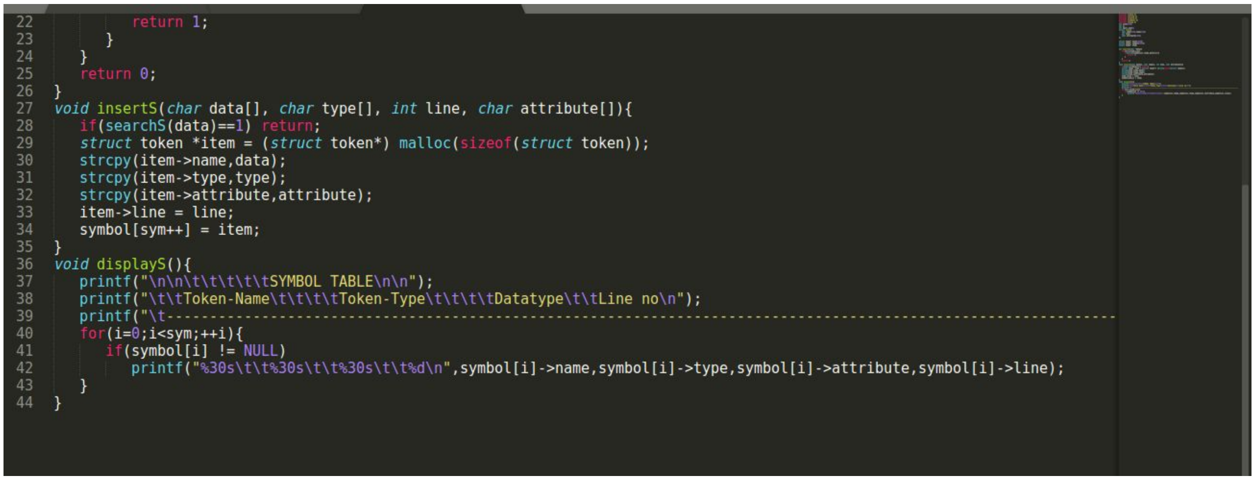


```
|                ID ME expr
|                ID DE expr
|                ID LS expr
|                ID RS expr
|                ID '=' functcall
|                '-' expr                %prec NA
|                II expr
|                expr II
|                ID
|                NUM
;


%%
#include"lex.yy.c"
#include<ctype.h>
int count=0;
int main(int argc, char *argv[])
{
    yyin = fopen(argv[1], "r");

   if(!yyparse())
        printf("\nParsing complete\n");
    else
        printf("\nParsing failed\n");

    displayS();
    fclose(yyin);
    return 0;
}

yyerror(char *s) {
    printf("Line %d : %s before '%s'\n", lineno, s, yytext);
}

```

# Code for Hash Table



```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <ctype.h>
int size=1000;
int i;
int key=0,sym=0;
struct token{
   char name[100],type[100];
   int line;
   char attribute[100];
};

struct token* hash[1000];
struct token* symbol[1000];
struct token* item;

int searchS(char *data){
   for(i=0;i<sym;++i){
      if(strcmp(symbol[i]->name,data)==0){
         return 1;
      }
   }
   return 0;
}
void insertS(char data[], char type[], int line, char attribute[]){
   if(searchS(data)==1) return;
   struct token *item = (struct token*) malloc(sizeof(struct token));
   strcpy(item->name,data);
   strcpy(item->type,type);
   strcpy(item->attribute,attribute);
   item->line = line;
   symbol[sym++] = item;
}
void displayS(){
   printf("\n\n\t\t\t\t\tSYMBOL TABLE\n\n");
```



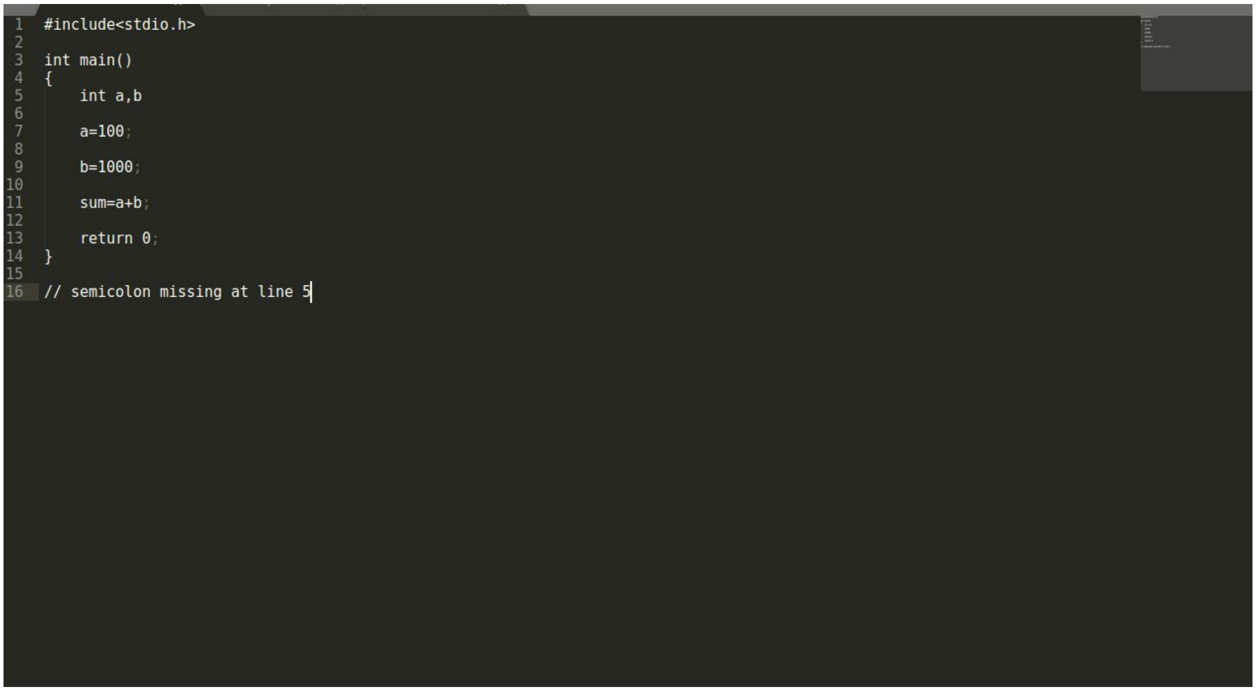
```c
         return 1;
      }
   }
   return 0;
}
void insertS(char data[], char type[], int line, char attribute[]){
   if(searchS(data)==1) return;
   struct token *item = (struct token*) malloc(sizeof(struct token));
   strcpy(item->name,data);
   strcpy(item->type,type);
   strcpy(item->attribute,attribute);
   item->line = line;
   symbol[sym++] = item;
}
void displayS(){
   printf("\n\n\t\t\t\t\tSYMBOL TABLE\n\n");
```

```c
   printf("\t\tToken-Name\t\t\t\tToken-Type\t\t\t\tDatatype\t\tLine no\n");
   printf("\t---------------------------------------------------------------------------------------------------------
   for(i=0;i<sym;++i){
      if(symbol[i] != NULL)
         printf("%30s\t\t%30s\t\t%30s\t\t%d\n",symbol[i]->name,symbol[i]->type,symbol[i]->attribute,symbol[i]->line);
   }
}
```

# Test Cases

**Test case # 1 :**

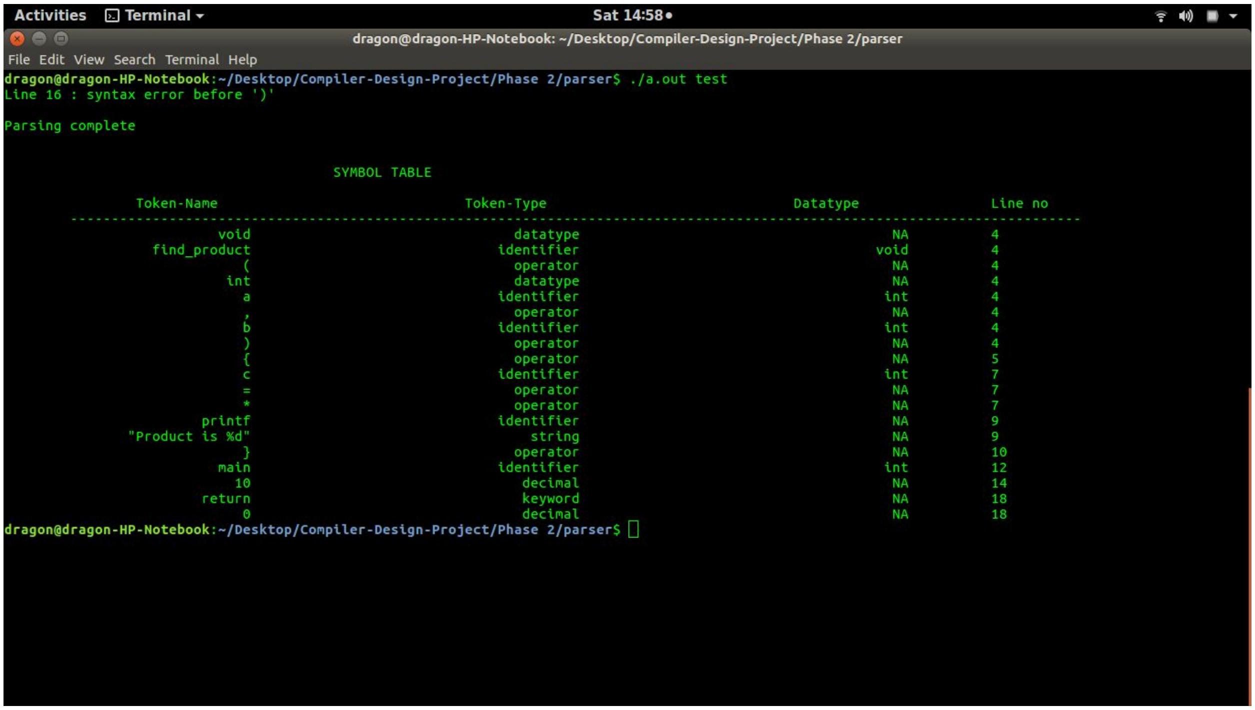
Input :



```
1  #include<stdio.h>
2
3  int main()
4  {
5      int a,b
6
7      a=100;
8
9      b=1000;
10
11     sum=a+b;
12
13     return 0;
14 }
15
16 // semicolon missing at line 5
```

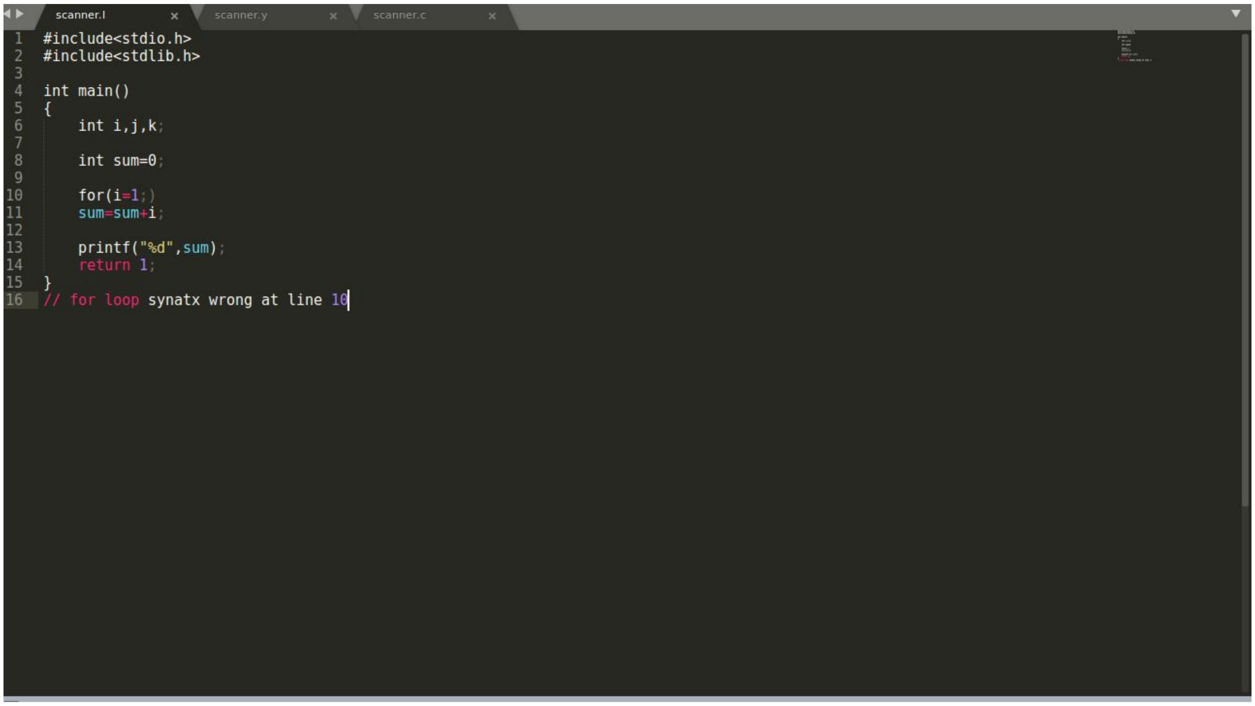
Expected output :  syntax error before ‘)’

Actual output :


Activities
Terminal
Sat 14:58
dragon@dragon-HP-Notebook: ~/Desktop/Compiler-Design-Project/Phase 2/parser
File Edit View Search Terminal Help

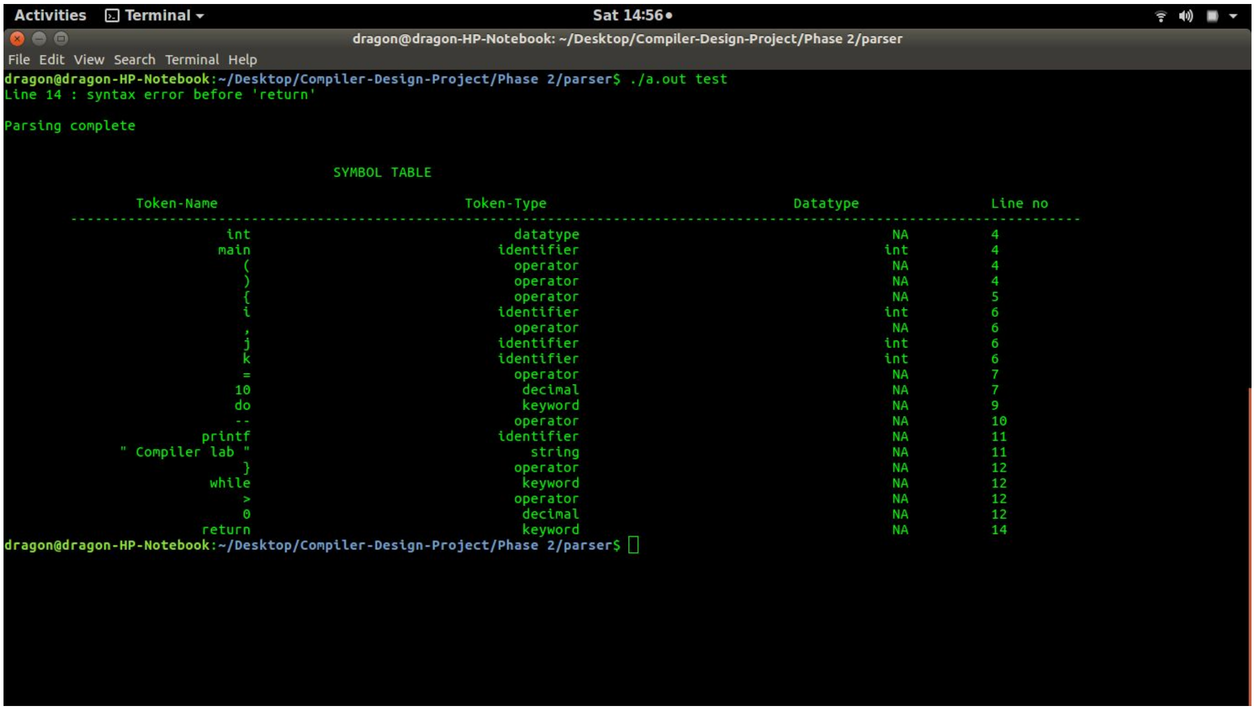
```
dragon@dragon-HP-Notebook:~/Desktop/Compiler-Design-Project/Phase 2/parser$ ./a.out test
Line 16 : syntax error before ')'

Parsing complete


                                        SYMBOL TABLE

                Token-Name                                      Token-Type                                      Datatype                Line no
        --------------------------------------------------------------------------------------------------------------------------------
                          void                                    datatype                                          NA             4
                  find_product                                  identifier                                        void             4
                             (                                    operator                                          NA             4
                           int                                    datatype                                          NA             4
                             a                                  identifier                                         int             4
                             ,                                    operator                                          NA             4
                             b                                  identifier                                         int             4
                             )                                    operator                                          NA             4
                             {                                    operator                                          NA             5
                             c                                  identifier                                         int             7
                             =                                    operator                                          NA             7
                             *                                    operator                                          NA             7
                        printf                                  identifier                                          NA             9
               "Product is %d"                                      string                                          NA             9
                             }                                    operator                                          NA             10
                          main                                  identifier                                         int             12
                            10                                     decimal                                          NA             14
                        return                                     keyword                                          NA             18
                             0                                     decimal                                          NA             18
dragon@dragon-HP-Notebook:~/Desktop/Compiler-Design-Project/Phase 2/parser$ 
```

## Test case #2:



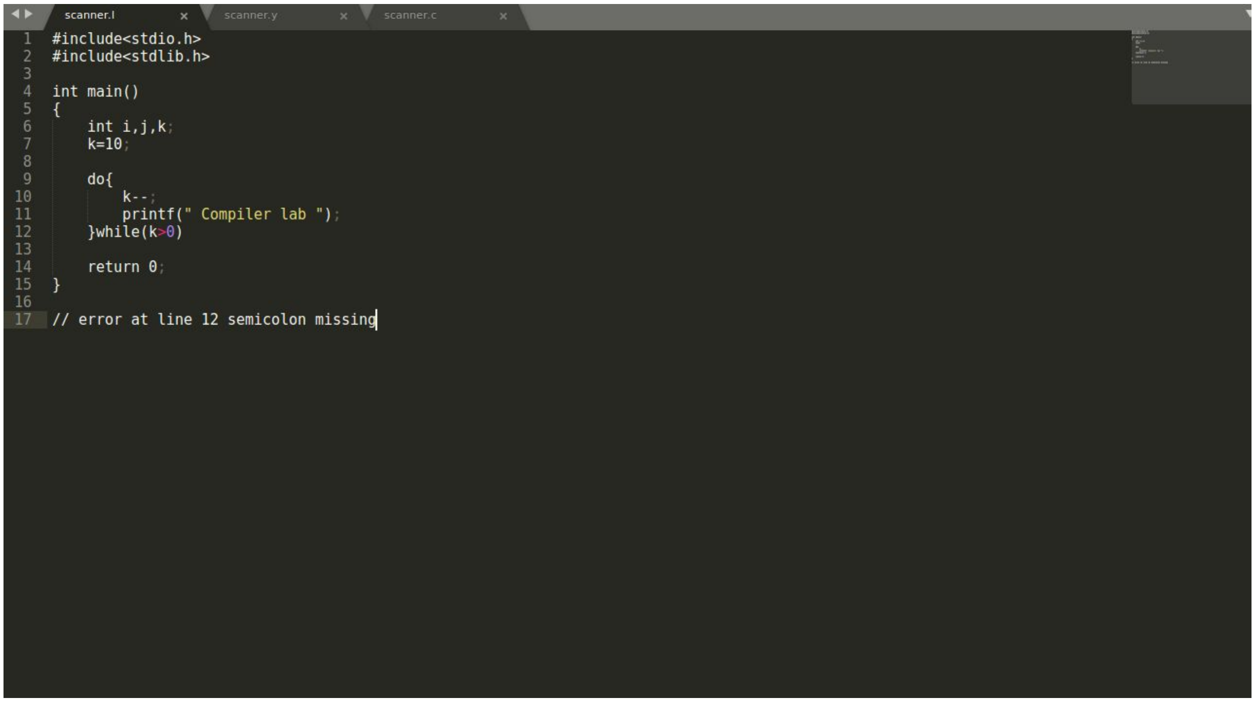
```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6      int i,j,k;
7
8      int sum=0;
9
10     for(i=1;)
11     sum=sum+i;
12
13     printf("%d",sum);
14     return 1;
15 }
16 // for loop synatx wrong at line 10
```

Output :


Activities
Terminal
Sat 14:56
dragon@dragon-HP-Notebook: ~/Desktop/Compiler-Design-Project/Phase 2/parser
File Edit View Search Terminal Help

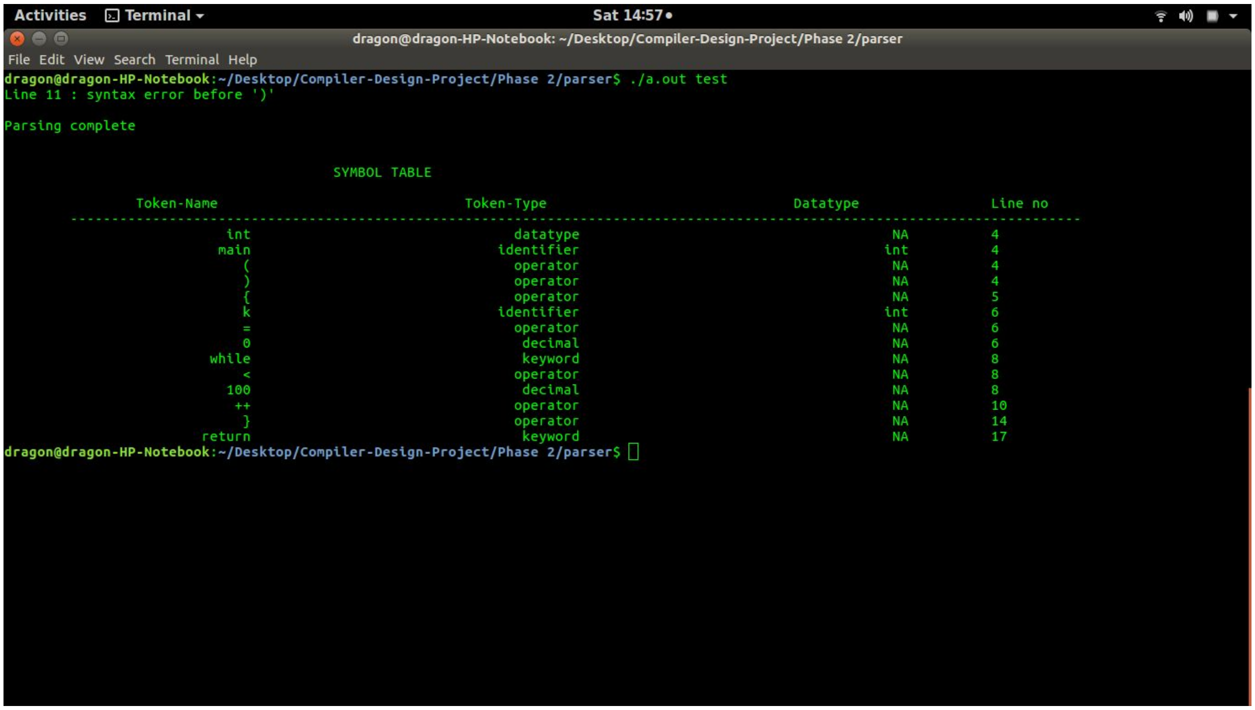
```
dragon@dragon-HP-Notebook:~/Desktop/Compiler-Design-Project/Phase 2/parser$ ./a.out test
Line 14 : syntax error before 'return'

Parsing complete


                                        SYMBOL TABLE

                Token-Name                              Token-Type                              Datatype                Line no
        ----------------------------------------------------------------------------------------------------------------------------
                           int                                 datatype                                 NA              4
                          main                               identifier                                int              4
                             (                                 operator                                 NA              4
                             )                                 operator                                 NA              4
                             {                                 operator                                 NA              5
                             i                               identifier                                int              6
                             ,                                 operator                                 NA              6
                             j                               identifier                                int              6
                             k                               identifier                                int              6
                             =                                 operator                                 NA              7
                            10                                  decimal                                 NA              7
                            do                                  keyword                                 NA              9
                            --                                 operator                                 NA              10
                        printf                               identifier                                 NA              11
              " Compiler lab "                                   string                                 NA              11
                             }                                 operator                                 NA              12
                         while                                  keyword                                 NA              12
                             >                                 operator                                 NA              12
                             0                                  decimal                                 NA              12
                        return                                  keyword                                 NA              14
dragon@dragon-HP-Notebook:~/Desktop/Compiler-Design-Project/Phase 2/parser$
```

## Test case # 3



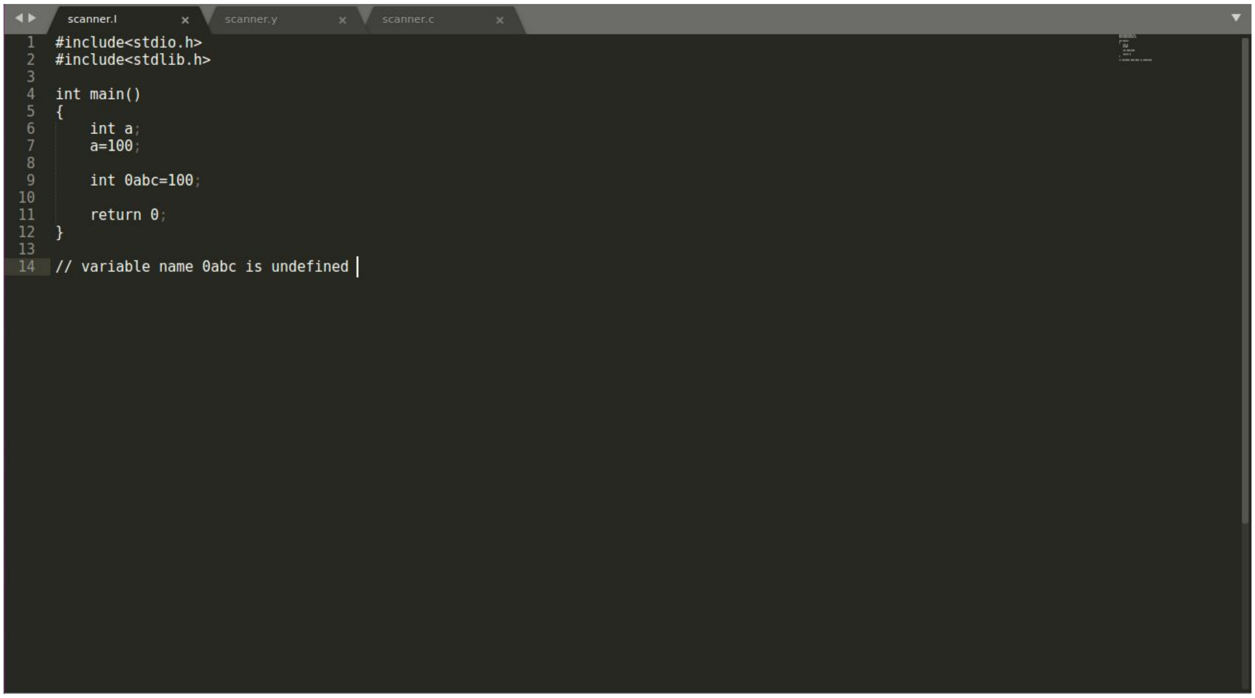
```
1   #include<stdio.h>
2   #include<stdlib.h>
3
4   int main()
5   {
6       int i,j,k;
7       k=10;
8
9       do{
10          k--;
11          printf(" Compiler lab ");
12      }while(k>0)
13
14      return 0;
15  }
16
17  // error at line 12 semicolon missing
```

Output :


Activities
Terminal
Sat 14:57
dragon@dragon-HP-Notebook: ~/Desktop/Compiler-Design-Project/Phase 2/parser
File Edit View Search Terminal Help

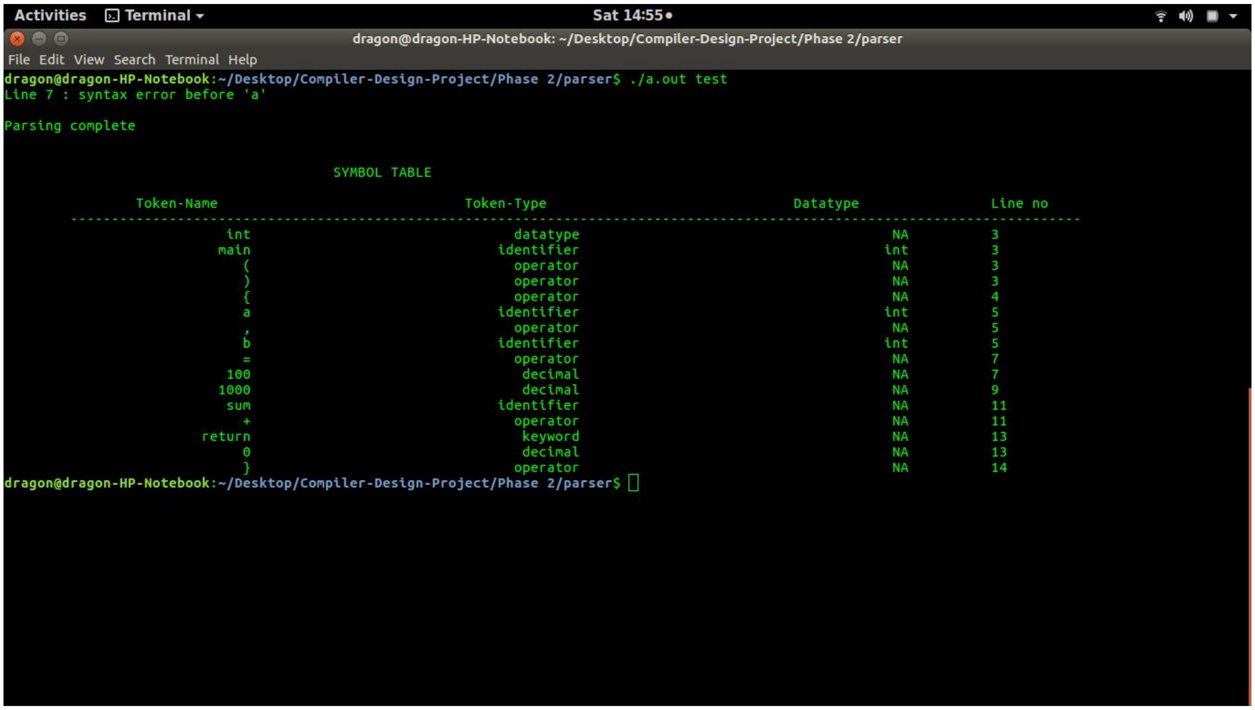
```
dragon@dragon-HP-Notebook:~/Desktop/Compiler-Design-Project/Phase 2/parser$ ./a.out test
Line 11 : syntax error before ')'

Parsing complete


                                        SYMBOL TABLE

                Token-Name                                      Token-Type                                      Datatype                        Line no
        ------------------------------------------------------------------------------------------------------------------------------------------------
                             int                                   datatype                                        NA              4
                            main                                 identifier                                       int              4
                               (                                   operator                                        NA              4
                               )                                   operator                                        NA              4
                               {                                   operator                                        NA              5
                               k                                 identifier                                       int              6
                               =                                   operator                                        NA              6
                               0                                    decimal                                        NA              6
                           while                                    keyword                                        NA              8
                               <                                   operator                                        NA              8
                             100                                    decimal                                        NA              8
                              ++                                   operator                                        NA              10
                               }                                   operator                                        NA              14
                          return                                    keyword                                        NA              17
dragon@dragon-HP-Notebook:~/Desktop/Compiler-Design-Project/Phase 2/parser$ 
```

## Test case #4



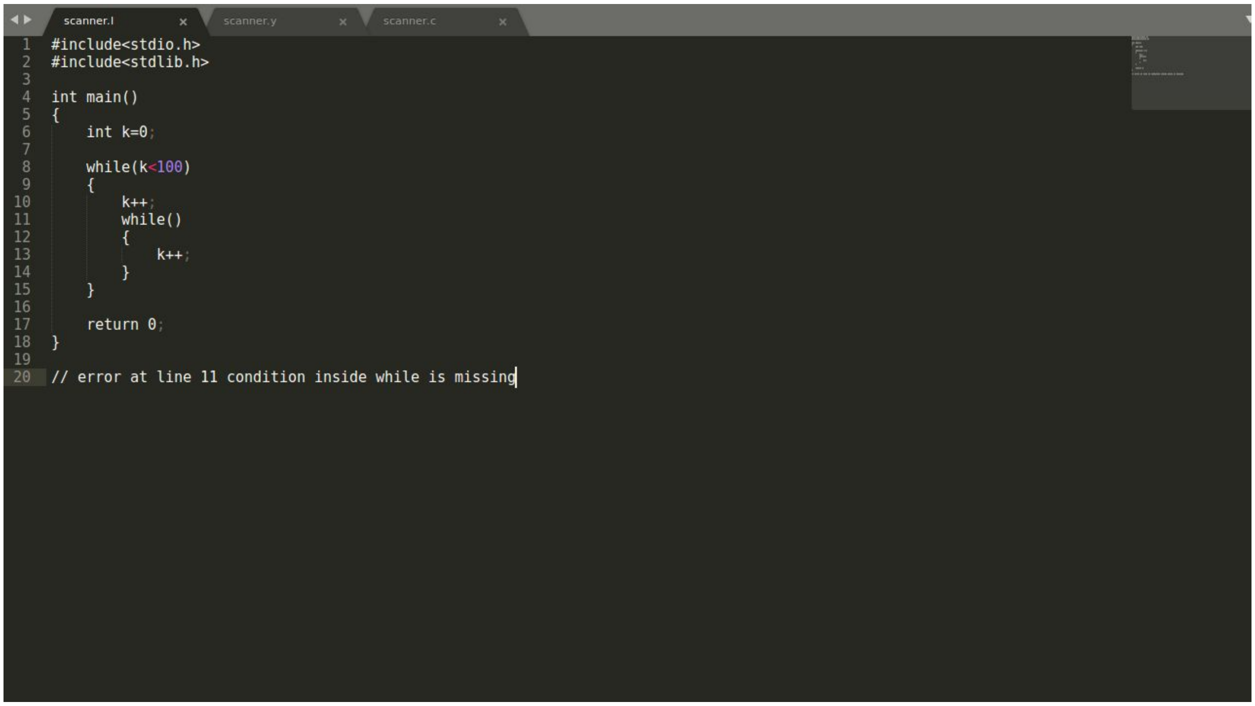
```
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int a;
    a=100;

    int 0abc=100;

    return 0;
}

// variable name 0abc is undefined
```

Output :


Activities
Terminal
Sat 14:55
dragon@dragon-HP-Notebook: ~/Desktop/Compiler-Design-Project/Phase 2/parser
File Edit View Search Terminal Help

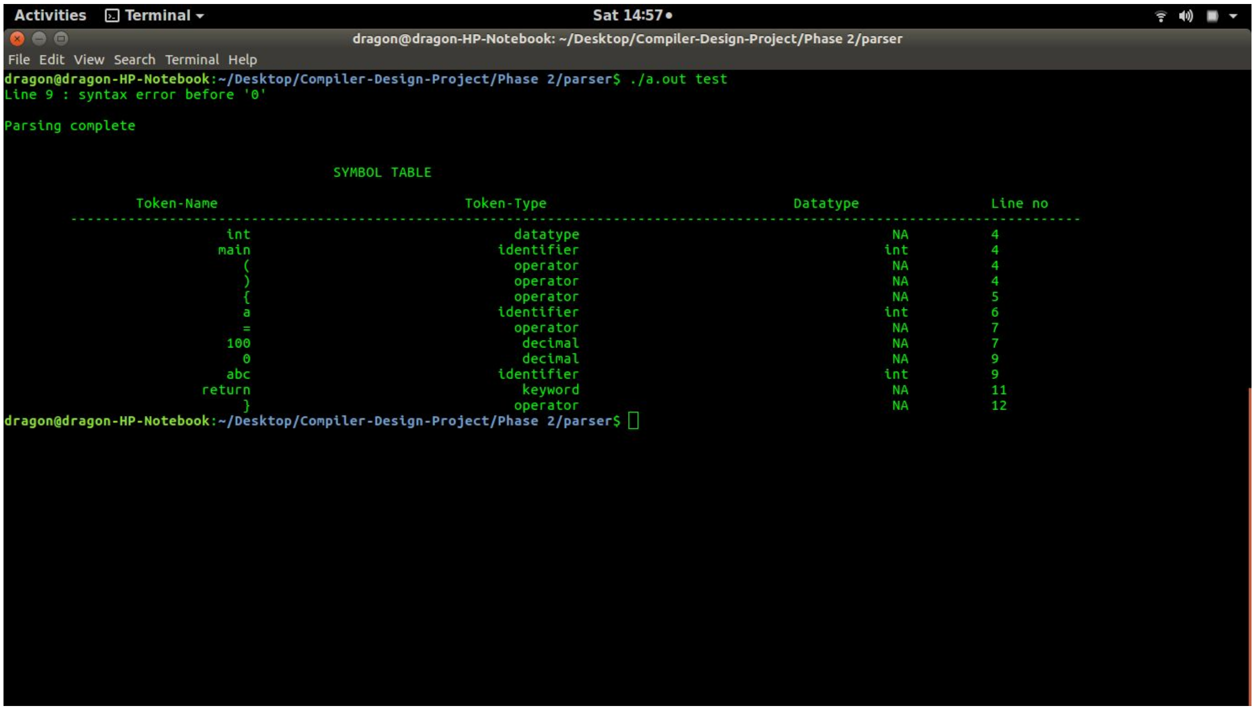
```
dragon@dragon-HP-Notebook:~/Desktop/Compiler-Design-Project/Phase 2/parser$ ./a.out test
Line 7 : syntax error before 'a'

Parsing complete


                                         SYMBOL TABLE

                Token-Name                               Token-Type                              Datatype                Line no
        ----------------------------------------------------------------------------------------------------------------------------
                            int                                datatype                                      NA          3
                           main                              identifier                                     int          3
                              (                                operator                                      NA          3
                              )                                operator                                      NA          3
                              {                                operator                                      NA          4
                              a                              identifier                                     int          5
                              ,                                operator                                      NA          5
                              b                              identifier                                     int          5
                              =                                operator                                      NA          7
                            100                                 decimal                                      NA          7
                           1000                                 decimal                                      NA          9
                            sum                              identifier                                      NA          11
                              +                                operator                                      NA          11
                         return                                 keyword                                      NA          13
                              0                                 decimal                                      NA          13
                              }                                operator                                      NA          14
dragon@dragon-HP-Notebook:~/Desktop/Compiler-Design-Project/Phase 2/parser$ 
```

## Test case #5:



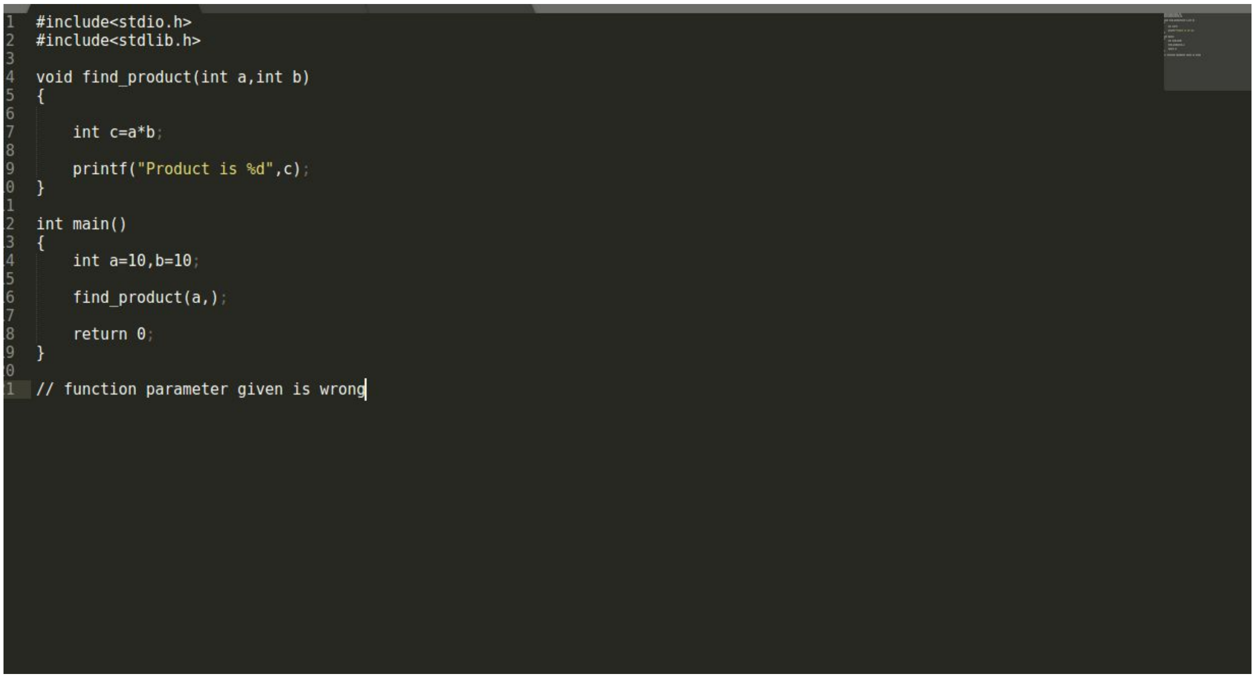
```
1   #include<stdio.h>
2   #include<stdlib.h>
3
4   int main()
5   {
6       int k=0;
7
8       while(k<100)
9       {
10          k++;
11          while()
12          {
13              k++;
14          }
15      }
16
17      return 0;
18  }
19
20  // error at line 11 condition inside while is missing
```

Output :


Activities
Terminal
Sat 14:57
dragon@dragon-HP-Notebook: ~/Desktop/Compiler-Design-Project/Phase 2/parser
File Edit View Search Terminal Help

```
dragon@dragon-HP-Notebook:~/Desktop/Compiler-Design-Project/Phase 2/parser$ ./a.out test
Line 9 : syntax error before '0'

Parsing complete


                                        SYMBOL TABLE

                Token-Name                               Token-Type                              Datatype                 Line no
        ----------------------------------------------------------------------------------------------------------------------------
                           int                                 datatype                                      NA          4
                          main                               identifier                                     int          4
                             (                                 operator                                      NA          4
                             )                                 operator                                      NA          4
                             {                                 operator                                      NA          5
                             a                               identifier                                     int          6
                             =                                 operator                                      NA          7
                           100                                  decimal                                      NA          7
                             0                                  decimal                                      NA          9
                           abc                               identifier                                     int          9
                        return                                  keyword                                      NA          11
                             }                                 operator                                      NA          12
dragon@dragon-HP-Notebook:~/Desktop/Compiler-Design-Project/Phase 2/parser$ 
```

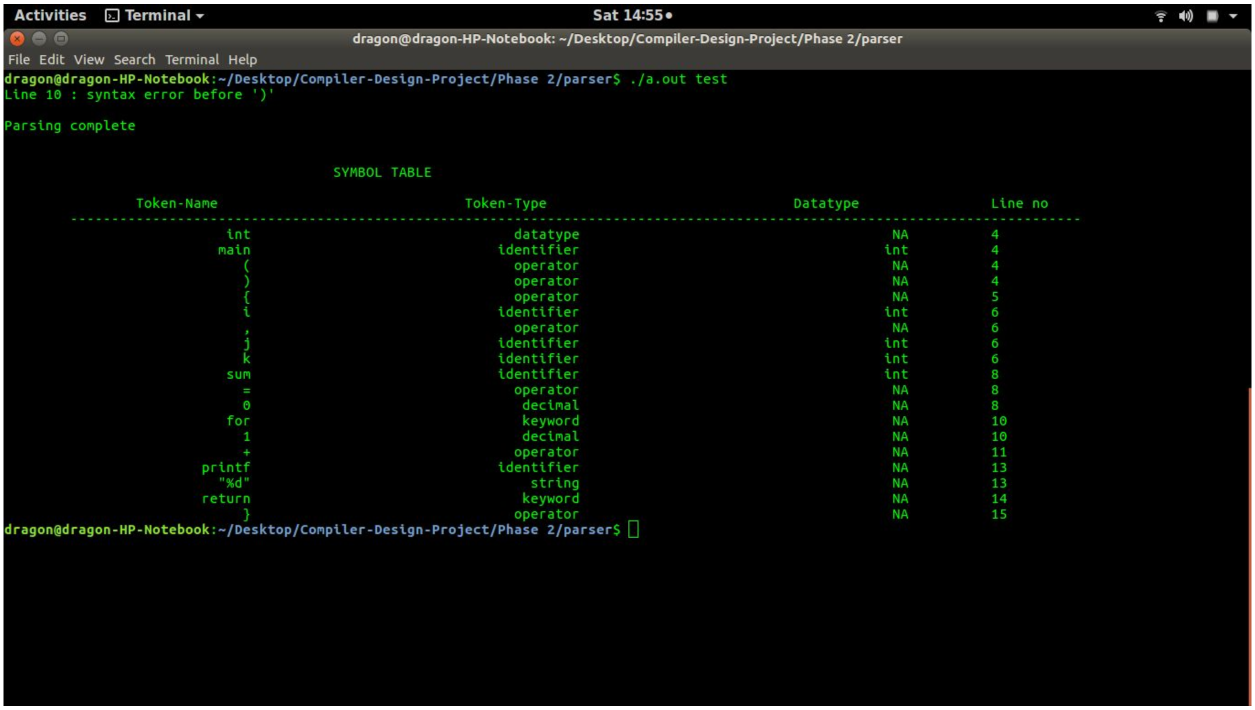
## Test case #6 :



```
#include<stdio.h>
#include<stdlib.h>

void find_product(int a,int b)
{

    int c=a*b;

    printf("Product is %d",c);
}

int main()
{
    int a=10,b=10;

    find_product(a,);

    return 0;
}

// function parameter given is wrong
```

Expected output : Syntax error at line number 10

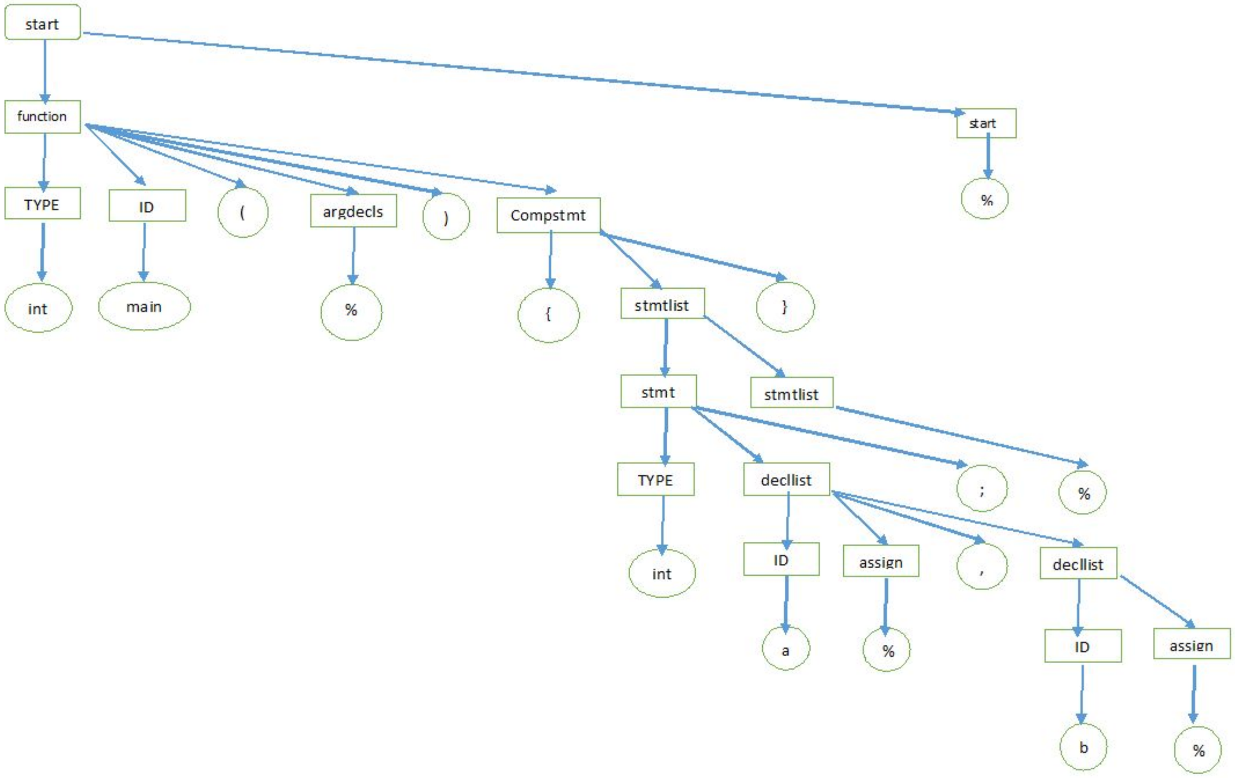
Actual Output :


Activities
Terminal
Sat 14:55
dragon@dragon-HP-Notebook: ~/Desktop/Compiler-Design-Project/Phase 2/parser
File Edit View Search Terminal Help

```
dragon@dragon-HP-Notebook:~/Desktop/Compiler-Design-Project/Phase 2/parser$ ./a.out test
Line 10 : syntax error before ')'

Parsing complete


                                        SYMBOL TABLE

                Token-Name                              Token-Type                              Datatype                Line no
        ---------------------------------------------------------------------------------------------------------------------------
                           int                                datatype                                      NA          4
                          main                              identifier                                     int          4
                             (                                operator                                      NA          4
                             )                                operator                                      NA          4
                             {                                operator                                      NA          5
                             i                              identifier                                     int          6
                             ,                                operator                                      NA          6
                             j                              identifier                                     int          6
                             k                              identifier                                     int          6
                           sum                              identifier                                     int          8
                             =                                operator                                      NA          8
                             0                                 decimal                                      NA          8
                           for                                 keyword                                      NA          10
                             1                                 decimal                                      NA          10
                             +                                operator                                      NA          11
                        printf                              identifier                                      NA          13
                          "%d"                                  string                                      NA          13
                        return                                 keyword                                      NA          14
                             }                                operator                                      NA          15
dragon@dragon-HP-Notebook:~/Desktop/Compiler-Design-Project/Phase 2/parser$ 
```

Parse Tree:

```
#include<stdio.h>
Int main(){
        int a,b;
}
```



# Conclusion

So far we have finished the two stages of the C compiler lexical phase and syntax phase.Lexical phase will generate all tokens and show lexical errors whenever it is found.While the syntax phase will generate
Syntax related errors and the parse the Code.All the cases are handled very carefully.It will also handle multi-line errors and print on different lines.