NATIONAL INSTITUTE OF TECHNOLOGY-TRICHY

CSPC 62 - COMPILER DESIGN ASSIGNMENT – 01

Lexical Analyzer

TEAM MEMBERS

106120015: Ashnu
106120063: Jagadeesh
106120073: Haneel Kumar
106120083: Gopi Reddy

DATE: 16/02/2023

Lex Code:

```
%{
        #include <stdio.h>
 #include <string.h>



        struct symboltable{
                char name[100];
        char type[100];
 int length;
        }ST[1001];

        struct constanttable{
                char name[100];
        char type[100];
 int length;
}CT[1001];

        int hash(char *str){
                int value = 0;
                for(int i = 0 ; i < strlen(str) ; i++){
        value = 10*value + (str[i] - 'A');
value = value % 1001;
 while(value < 0)
                        value = value + 1001;
                }
                return value;
        }

        int lookupST1(char *str){
        int value = hash(str);
                if(ST[value].length == 0)
                        return 0;
                else if(strcmp(ST[value].name,str)==0)
                        return 1;

                return 0;
          }
        }

        int lookupST2(char *str){
                        int value = hash(str);

                        if(ST[value].length != 0 && strcmp(ST[value].name,str)!=0){
                                for(int i = value + 1 ; i!=value ; i = (i+1)%1001){
```

```
                              if(strcmp(ST[i].name,str)==0)
                         return 1;
                         }
                         return 0;
                     }
            }


            int lookupCT1(char *str){
            int value = hash(str);
                 if(CT[value].length == 0)
                         return 0;
                 else if(strcmp(CT[value].name,str)==0)
                         return 1;

                         return 0;
                 }
            }

            int lookupCT2(char *str){
                         int value = hash(str);

                         if(CT[value].length != 0 && strcmp(CT[value].name,str)!=0  ){
                              for(int i = value + 1 ; i!=value ; i = (i+1)%1001){
                 if(strcmp(CT[i].name,str)==0)
                                        return 1;
                              }
                              return 0;
                         }
                 }


            void insertST(char *str1, char *str2){
       if(lookupST1(str1) && lookupST2(str1))          return;
                 else{
                         int value = hash(str1);
       if(ST[value].length == 0){
       strcpy(ST[value].name,str1);
       strcpy(ST[value].type,str2);
        ST[value].length = strlen(str1);
                              return;
                         }
                         int pos = 0;
                         for (int i = value + 1 ; i!=value ; i = (i+1)%1001){
            if(ST[i].length == 0){
                                   pos = i;
                              break;
                              }
```

```
                                }
                                strcpy(ST[pos].name,str1);
 strcpy(ST[pos].type,str2);                    ST[pos].length = strlen(str1);
                        }
                }

                void insertCT(char *str1, char *str2){
                        if(lookupCT1(str1) && lookupCT2(str1))
                                return;
                        else{
                                int value = hash(str1);
if(CT[value].length == 0){
strcpy(CT[value].name,str1);
 strcpy(CT[value].type,str2);                          CT[value].length = strlen(str1);
                                        return;
                                }
                                int pos = 0;
                                for (int i = value + 1 ; i!=value ; i = (i+1)%1001){
                if(CT[i].length == 0){
                                                pos = i;
                                        break;
                                        }
                                }
                                strcpy(CT[pos].name,str1);
 strcpy(CT[pos].type,str2);                    CT[pos].length = strlen(str1);
                        }
                }

                void printST(){                 for(int i
= 0 ; i < 1001 ; i++){
                                if(ST[i].length == 0)
                continue;
                                printf("%s\t%s\n",ST[i].name, ST[i].type);
                        }
}

                void printCT(){                 for(int i
= 0 ; i < 1001 ; i++){
                                if(CT[i].length == 0)
                                        continue;
                                printf("%s\t%s\n",CT[i].name, CT[i].type);
                        }
                }

%}

DE "define"
IN "include"
```

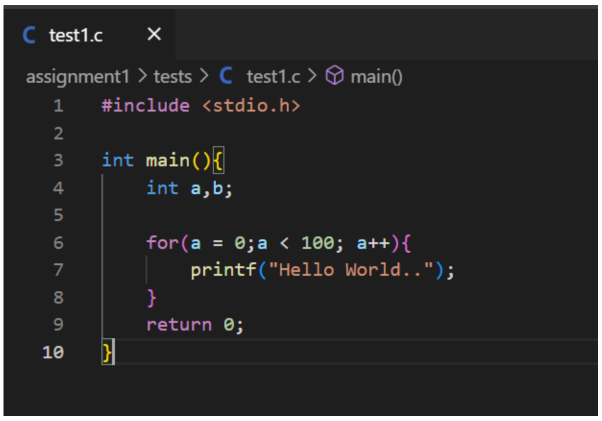
```
operator
[[<][=]|[>][=]|[=][=]|[!][=]|[>]|[<]|[\|][\|]|[&][&]|[\!]|[=]|[\^]|[\+][=]|[\-][=]|[\*][=]|[\/][
=]|[\%][=]|[\+][\+]|[\-][\-]|[\+]|[\-]|[\*]|[\/]|[\%]|[&]|[\|]|[~]|[<][<]|[>][>]]


%%
\n   {yylineno++;}
([#][" "]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|" "|"\t"] {printf("%s \t-Pre
Processor directive\n",yytext);}             //Matches #include<stdio.h>
([#][" "]*({DE})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"] {printf("%s
\tMacro\n",yytext);} //Matches macro
\/\/(.*) {printf("%s \t- SINGLE LINE COMMENT\n", yytext);}
\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/  {printf("%s \t- MULTI LINE COMMENT\n",
yytext);}
[ \n\t] ;
; {printf("%s \t- SEMICOLON DELIMITER\n", yytext);}
, {printf("%s \t- COMMA DELIMITER\n", yytext);}
\{ {printf("%s \t- OPENING BRACES\n", yytext);}
\} {printf("%s \t- CLOSING BRACES\n", yytext);}
\( {printf("%s \t- OPENING BRACKETS\n", yytext);}
\) {printf("%s \t- CLOSING BRACKETS\n", yytext);}
\[ {printf("%s \t- SQUARE OPENING BRACKETS\n", yytext);}
\] {printf("%s \t- SQUARE CLOSING BRACKETS\n", yytext);}
\: {printf("%s \t- COLON DELIMITER\n", yytext);}
\\ {printf("%s \t- FSLASH\n", yytext);} \. {printf("%s \t- DOT DELIMITER\n", yytext);}
auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|l
ong|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volati
le|while|main/[\(|" "|\{|;|:|"\n"|"\t"] {printf("%s \t- KEYWORD\n", yytext); insertST(yytext,
"KEYWORD");}
\"[^\n]*\"/[;|,|\)] {printf("%s \t- STRING CONSTANT\n", yytext); insertCT(yytext,"STRING
CONSTANT");}
\'[A-Z|a-z]\'/[;|,|\)|:] {printf("%s \t- Character CONSTANT\n", yytext);
insertCT(yytext,"Character CONSTANT");}
[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[ {printf("%s \t- ARRAY IDENTIFIER\n", yytext); insertST(yytext,
"IDENTIFIER");}

{operator}/[a-z]|[0-9]|;|" "|[A-Z]|\(|\"|\'|\)|\n|\t {printf("%s \t- OPERATOR\n", yytext);}

[1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^] {printf("%s \t-
NUMBER CONSTANT\n", yytext); insertCT(yytext, "NUMBER CONSTANT");}
([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^] {printf("%s \t- Floating
CONSTANT\n", yytext); insertCT(yytext, "Floating CONSTANT");}
[A-Za-z_][A-Za-z_0-9]*/[" "|;|,|\(|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\.|\{|\^|\t] {printf("%s
\t- IDENTIFIER\n", yytext); insertST(yytext, "IDENTIFIER");}


(.?) {
                if(yytext[0]=='#')
```

```
                        printf("Error in Pre-Processor directive at line no. %d\n",yylineno);
   else if(yytext[0]=='/')                     printf("ERR_UNMATCHED_COMMENT at line
no. %d\n",yylineno);                else if(yytext[0]=='"')
printf("ERR_INCOMPLETE_STRING at line no. %d\n",yylineno);                    else
                        printf("ERROR at line no. %d\n",yylineno);
printf("%s\n", yytext);            return 0;
}

%%

int main(int argc , char **argv){

printf("\n=====================================================
===================\n\n");
int i;   for (i=0;i<1001;i++){
ST[i].length=0;
                        CT[i].length=0;
            }
            yyin = fopen(argv[1],"r");
yylex();
            printf("\n\nSYMBOL TABLE\n\n");
printST();
            printf("\n\nCONSTANT TABLE\n\n");
printCT();
}
int yywrap(){
return 1;
}
```

Explanation:

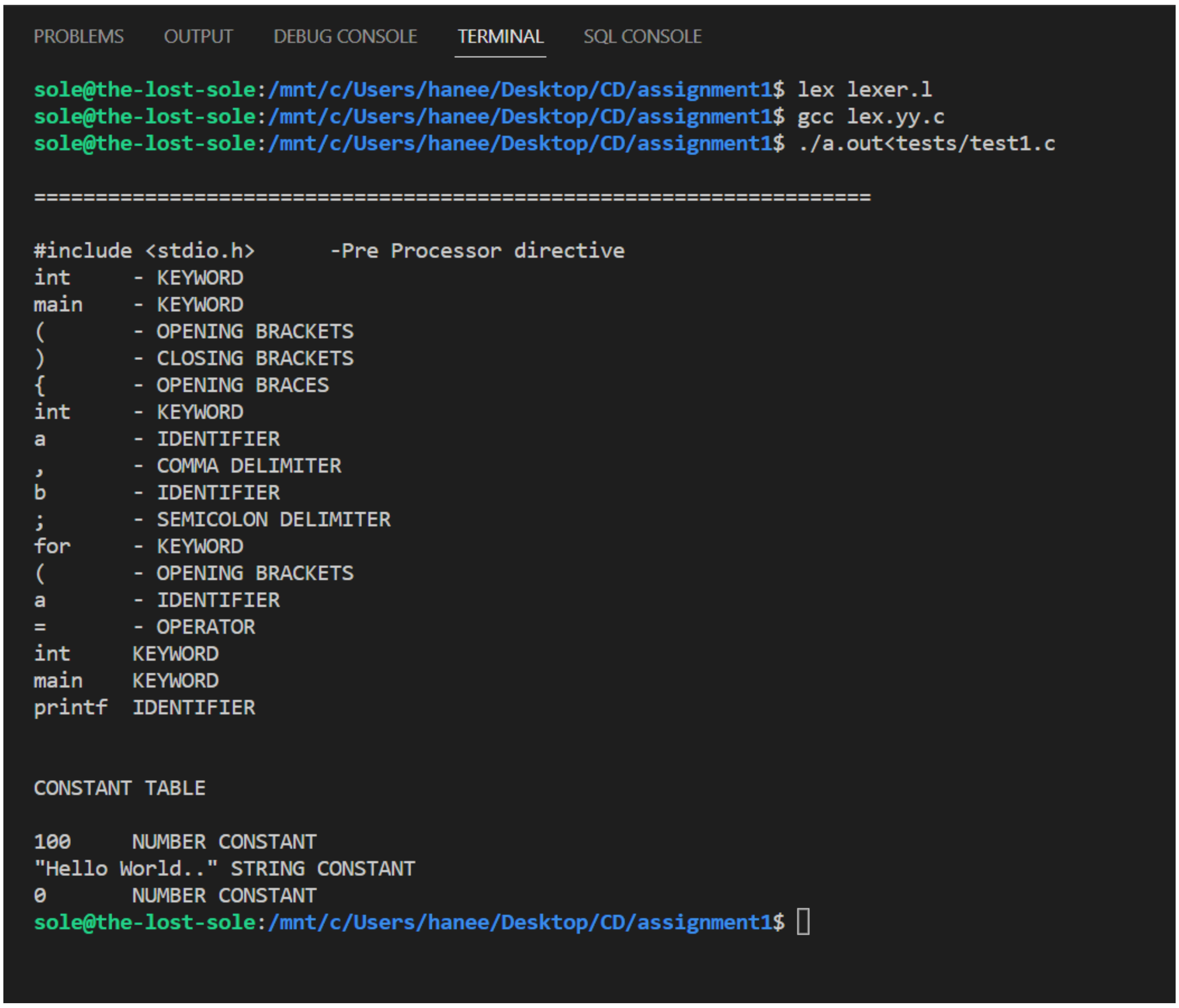
The initial phase of the compiler is the Lexical Anaylzer. In this case, we're going through the input programme character by character, identifying Lexemes and categorising them as Tokens. These tokens have been represented as a symbol table, which will be used as input in the following phase, Parser.

Input:



```
#include <stdio.h>

int main(){
    int a,b;

    for(a = 0;a < 100; a++){
        printf("Hello World..");
    }
    return 0;
}
```

Output:



```
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment1$ lex lexer.l
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment1$ gcc lex.yy.c
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment1$ ./a.out<tests/test1.c

====================================================================

#include <stdio.h>      -Pre Processor directive
int      - KEYWORD
main     - KEYWORD
(        - OPENING BRACKETS
)        - CLOSING BRACKETS
{        - OPENING BRACES
int      - KEYWORD
a        - IDENTIFIER
,        - COMMA DELIMITER
b        - IDENTIFIER
;        - SEMICOLON DELIMITER
for      - KEYWORD
(        - OPENING BRACKETS
a        - IDENTIFIER
=        - OPERATOR
int      KEYWORD
main     KEYWORD
printf   IDENTIFIER


CONSTANT TABLE

100      NUMBER CONSTANT
"Hello World.." STRING CONSTANT
0        NUMBER CONSTANT
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment1$
```

NATIONAL INSTITUTE OF TECHNOLOGY-TRICHY

CSPC 62 - COMPILER DESIGN ASSIGNMENT – 02

Parser

TEAM MEMBERS

106120015: Ashnu
106120063: Jagadeesh
106120073: Haneel Kumar
106120083: Gopi Reddy

DATE: 30/03/2023

Lex Code:

```
%{
            #include <stdio.h>
 #include            <string.h>
#include "y.tab.h"  struct symboltable
            {
char
name[100];                  char
class[100];                 char
type[100];                  char
value[100];          int  lineno;
int
length;             }ST[1001];
struct constanttable
            {
                        char name[100];
char type[100];
 int length;        }CT[1001];
int hash(char *str){
int value = 0;
                        for(int i = 0 ; i < strlen(str) ; i++){
            value = 10*value + (str[i] - 'A');
value = value % 1001;                          while(value < 0)

            value = value + 1001;
                        }
                        return value;

            int lookupST1(char *str){
            int value = hash(str);
            if(ST[value].length == 0)
                                    return 0;
```

```
                    else if(strcmp(ST[value].name,str)==0)
                              return 1;

                    return 0;
             }
          }

          int lookupST2(char *str){
                              int value = hash(str);

                              if(ST[value].length != 0 && strcmp(ST[value].name,str)!=0){
                                        for(int i = value + 1 ; i!=value ; i = (i+1)%1001){
                                        if(strcmp(ST[i].name,str)==0)
                                        return 1;
}
                              return 0;
                      }
          }

          int lookupCT1(char *str){
 int value = hash(str);
if(CT[value].length == 0)
                              return 0;
                    else if(strcmp(CT[value].name,str)==0)
                              return 1;


                              return 0;

                    }

          }
          int lookupCT2(char *str){
int value = hash(str);


                    if(CT[value].length != 0 &&  strcmp(CT[value].name,str)!=0){
for(int i = value + 1 ; i!=value ; i = (i+1)%1001){
                                        if(strcmp(CT[i].name,str)==0)
                                                  return 1;
                              }
```

```
                        return 0;
                }
        }


        void insertST(char *str1, char *str2){
if(lookupST1(str1) && lookpuST2(str2)){
                        return;
                }
                else{                   int
value = hash(str1);
 if(ST[value].length == 0){
strcpy(ST[value].name,str1);
strcpy(ST[value].class,str2);
ST[value].length = strlen(str1);
 insertSTline(str1,yylineno);
                                return;
                        }
                        int pos = 0;
                        for (int i = value + 1 ; i!=value ; i = (i+1)%1001){
                                if(ST[i].length == 0){
                                        pos = i;
                        break;
        }
                        }
                        strcpy(ST[pos].name,str1);
 strcpy(ST[pos].class,str2);
                        ST[pos].length = strlen(str1);
                }
        }
        void insertSTtype(char *str1, char *str2){
                for(int i = 0 ; i < 1001 ; i++){
```

```
if(strcmp(ST[i].name,str1)==0)
 strcpy(ST[i].type,str2);

                         }
   }
            void insertSTvalue(char *str1, char *str2){
for(int i = 0 ; i < 1001 ; i++){
                                     if(strcmp(ST[i].name,str1)==0)
 strcpy(ST[i].value,str2);
                         }
            }
            void insertSTline(char *str1, int line){
                         for(int i = 0 ; i < 1001 ; i++){
 if(strcmp(ST[i].name,str1)==0)
                                                 ST[i].lineno = line;
                         }
            }
            void insertCT(char *str1, char *str2){
if(lookupCT1(str1) && lookupCT2(str1))
                                     return;
            else{                                    int value =
hash(str1);
 if(CT[value].length == 0){
strcpy(CT[value].name,str1);
strcpy(CT[value].type,str2);
 CT[value].length = strlen(str1);
                                                 return;
                                     }
                                     int pos = 0;
                                     for (int i = value + 1 ; i!=value ; i = (i+1)%1001){
                                                 if(CT[i].length == 0){
```

```
                                        pos = i;
                                break;
                                }
                        }
                strcpy(CT[pos].name,str1);
strcpy(CT[pos].type,str2);
                        CT[pos].length = strlen(str1);
                }
        }
        void printST(){
                printf("%10s | %15s | %10s | %10s | %10s\n","SYMBOL", "CLASS",
"TYPE","VALUE", "LINE NO");
                for(int i=0;i<81;i++)
        printf("-");                    printf("\n");
                for(int i = 0 ; i < 1001 ; i++){
                        if(ST[i].length == 0)
                                continue;
                        printf("%10s | %15s | %10s | %10s | %10d\n",ST[i].name, ST[i].class,
ST[i].type, ST[i].value, ST[i].lineno);
                }
   }
 void printCT(){   printf("%10s | %15s\n","NAME",
"TYPE");
                for(int i=0;i<81;i++)
        printf("-");                    printf("\n");
                for(int i = 0 ; i < 1001 ; i++){
                        if(CT[i].length == 0)
                                continue;
                        printf("%10s | %15s\n",CT[i].name, CT[i].type);
                }
        }
```

```
            char curid[20];
char curtype[20];        char
curval[20]; %}


DE "define"
IN "include"
%%
\n        {yylineno++;}
([#][" "]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|" "|"\t"] { }
([#][" "]*({DE})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"]                    { }
\/\/(.*)                    { }
\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/
                          { }
[ \n\t] ;
";"                               { return(';'); }
","                               { return(','); }
("{")                     { return('{'); }
("}")                     { return('}'); }
"("                               { return('('); }
")"                               { return(')'); }
("["|"<:")                { return('['); }
("]"|":>")                { return(']'); }
":"                               { return(':'); }
"."                               { return('.'); }

"char"                    { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return CHAR;}
"double"                    { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return DOUBLE;}
"else"            { insertSTline(yytext, yylineno); insertST(yytext, "Keyword"); return ELSE;}
"float"                   { strcpy(curtype,yytext); insertST(yytext, "Keyword");return FLOAT;}
"while"                   { insertST(yytext, "Keyword"); return WHILE;}
"do"                      { insertST(yytext, "Keyword"); return DO;}
```

```
"for"                   { insertST(yytext, "Keyword"); return FOR;}

"if"                    { insertST(yytext, "Keyword"); return IF;}

"int"                   { strcpy(curtype,yytext); insertST(yytext, "Keyword");return INT;}

"return"                { insertST(yytext, "Keyword");  return RETURN;}

"void"            { strcpy(curtype,yytext); insertST(yytext, "Keyword");  return VOID;} "break"
            { insertST(yytext, "Keyword");  return BREAK;}



"<="                    { return lessthan_assignment_operator; }

"<"                         { return lessthan_operator; }

">="                    { return greaterthan_assignment_operator; }

">"                         { return greaterthan_operator; }

"=="                    { return equality_operator; }

"!="                    { return inequality_operator; }

"&&"                    { return AND_operator; }

"||"                    { return OR_operator; }

"&"                         { return amp_operator; }

"!"                         { return exclamation_operator; }

"-"                         { return subtract_operator; }

"+"                         { return add_operator; }

"*"                         { return multiplication_operator; }

"/"                         { return division_operator; }

"%"                         { return modulo_operator; }

\=                          { return assignment_operator;}


\"[^\n]*\"/[;|,|\)]                 {strcpy(curval,yytext); insertCT(yytext,"String Constant");
return string_constant;}

\'[A-Z|a-z]\'/[;|,|\)|:]    {strcpy(curval,yytext); insertCT(yytext,"Character Constant"); return
character_constant;}

[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[  {strcpy(curid,yytext); insertST(yytext, "Array Identifier");  return
identifier;}

[1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^]
            {strcpy(curval,yytext); insertCT(yytext, "Number Constant"); return integer_constant;}
```

```
([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^]
            {strcpy(curval,yytext); insertCT(yytext, "Floating Constant"); return float_constant;}

[A-Za-z_][A-Za-z_0-9]* {strcpy(curid,yytext);insertST(yytext,"Identifier");  return identifier;}


(.?) {

                        if(yytext[0]=='#')

                                    printf("Error in Pre-Processor directive at line no. %d\n",yylineno);

                        else if(yytext[0]=='/')

 printf("ERR_UNMATCHED_COMMENT at line no. %d\n",yylineno);                else

if(yytext[0]=='"')                                printf("ERR_INCOMPLETE_STRING at line no.

%d\n",yylineno);

                        else

                                    printf("ERROR at line no. %d\n",yylineno);

 printf("%s\n", yytext);

                        return 0;

}

%%
```

Yacc Code:

```
%{        void yyerror(char*

s);       int yylex();

            #include "stdio.h"

            #include "stdlib.h"

            #include "ctype.h"

 #include "string.h"   void

ins();    void insV();      int

flag=0;  extern char

curid[20];   extern char

curtype[20];   extern char

curval[20];


%}
```

```
%nonassoc IF

%token INT CHAR FLOAT DOUBLE RETURN MAIN VOID WHILE FOR DO BREAK ENDIF identifier
integer_constant string_constant float_constant character_constant

%nonassoc ELSE

%right assignment_operator

%left OR_operator AND_operator amp_operator equality_operator inequality_operator
lessthan_assignment_operator lessthan_operator greaterthan_assignment_operator
greaterthan_operator leftshift_operator rightshift_operator add_operator subtract_operator
multiplication_operator division_operator modulo_operator


%right exclamation_operator

%start program


%%
program
                        : declaration_list;

declaration_list
                        : declaration D

D
                        : declaration_list

                        | ;

declaration
                        : variable_declaration

                        | function_declaration;
variable_declaration
                        : type_specifier variable_declaration_list ';' ;

variable_declaration_list
                        : variable_declaration_identifier V;

V
                        : ',' variable_declaration_list

                        | ;

variable_declaration_identifier
                        : identifier { ins(); } vdi;
```

```
vdi : identifier_array_type | assignment_operator expression ;  identifier_array_type
                            : '[' initilization_params

                            | ;

initilization_params
                            : integer_constant ']' initilization
                            | ']' string_initilization;

initilization

                            : string_initilization

                            | array_initialization

                            | ;

type_specifier
                            : INT | CHAR | FLOAT | DOUBLE

                            | VOID ;
function_declaration

                            : function_declaration_type function_declaration_param_statement;

function_declaration_type

                            : type_specifier identifier '('  { ins();};

function_declaration_param_statement

                            : params ')' statement;

params

                            : parameters_list | ;

parameters_list

                            : type_specifier parameters_identifier_list;

parameters_identifier_list

                            : param_identifier parameters_identifier_list_breakup;

parameters_identifier_list_breakup

                            : ',' parameters_list

                            | ;

param_identifier

                            : identifier { ins(); } param_identifier_breakup;  param_identifier_breakup

                            : '[' ']'

                            | ;
```

```
statement
                                  : expression_statment | compound_statement
                                  | conditional_statements | iterative_statements
                                  | return_statement | break_statement
                                  | variable_declaration;
compound_statement
                                  : '{' statment_list '}' ;
statment_list
                                  : statement statment_list
                                  | ;
expression_statment
                                  : expression ';'
                                  | ';' ;
conditional_statements
     : IF '(' simple_expression ')' statement conditional_statements_breakup;
conditional_statements_breakup
                                  : ELSE statement
                                  | ;
iterative_statements
                                  : WHILE '(' simple_expression ')' statement
                                  | FOR '(' expression ';' simple_expression ';' expression ')'
                                  | DO statement WHILE '(' simple_expression ')' ';';


return_statement
                                  : RETURN return_statement_breakup;
return_statement_breakup
                                  : ';'
                                  | expression ';' ;
break_statement
                                  : BREAK ';' ;
string_initilization
```

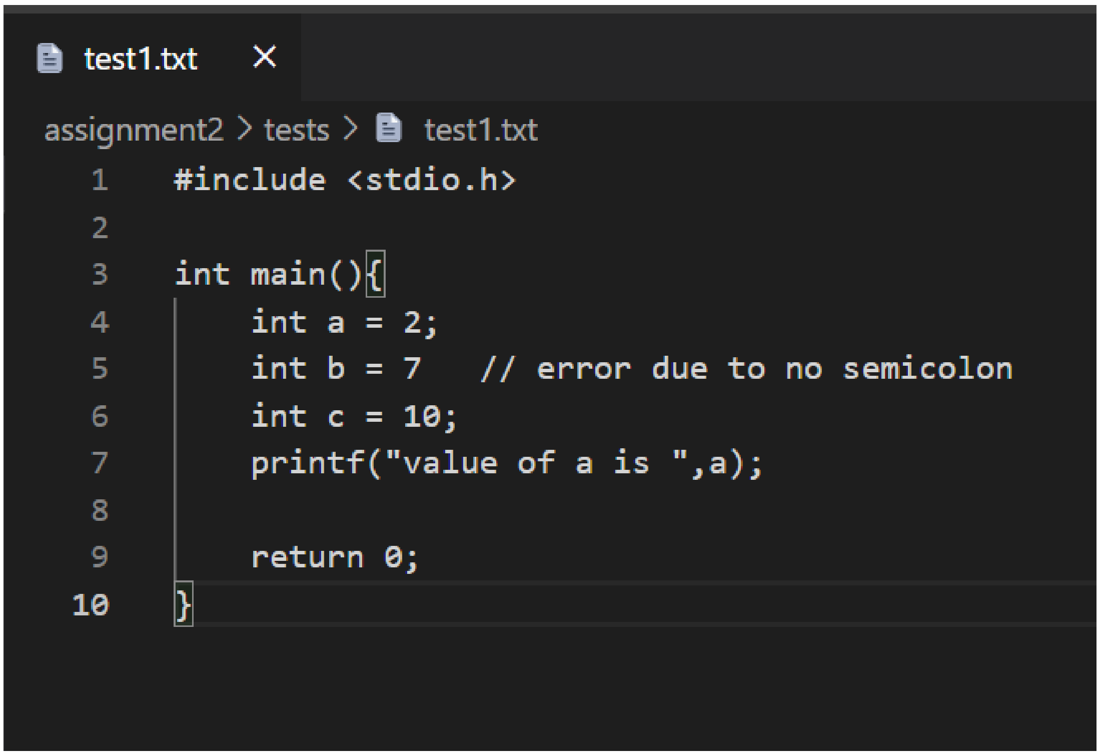
```
                                    : assignment_operator string_constant { insV(); };
array_initialization
                                    : assignment_operator '{' array_int_declarations '}';
array_int_declarations
                                    : integer_constant array_int_declarations_breakup;
array_int_declarations_breakup
                                    : ',' array_int_declarations | ;
expression
                                    : mutable expression_breakup
                                    | simple_expression ;
expression_breakup
                                    : assignment_operator expression;
simple_expression
                                    : and_expression simple_expression_breakup;
simple_expression_breakup
                                    : OR_operator and_expression simple_expression_breakup | ;
and_expression
                                    : unary_relation_expression and_expression_breakup;
and_expression_breakup
                                    : AND_operator unary_relation_expression and_expression_breakup
                                    | ;
unary_relation_expression
                                    : exclamation_operator unary_relation_expression
                                    | regular_expression ;
regular_expression
                                    : sum_expression regular_expression_breakup;
regular_expression_breakup
                                    : relational_operators sum_expression
                                    | ;
relational_operators
              :       greaterthan_assignment_operator       |       lessthan_assignment_operator       |
greaterthan_operator
     | lessthan_operator | equality_operator | inequality_operator ;  sum_expression
                                    : sum_expression sum_operators term
                                    | term ;

sum_operators
                                    : add_operator
                         | subtract_operator ;

term
                                    : term MULOP factor
                         | factor ;

MULOP
                                    : multiplication_operator | division_operator | modulo_operator ;

factor
                                    : immutable | mutable ;

mutable
                                    : identifier
                                    | mutable mutable_breakup;

mutable_breakup
                                    : '[' expression ']'
                                    | '.' identifier;

immutable
                                    : '(' expression ')'
                                    | call | constant;

call
                                    : identifier '(' arguments ')';
arguments
                                    : arguments_list | ;
arguments_list
                                    : expression A;
A
```

```
                                    : ',' expression A
                                    | ;

constant
                                    : integer_constant        { insV(); }
                                    | string_constant         { insV(); }
                                    | float_constant { insV(); }
                                    | character_constant{ insV(); };
%%
extern FILE *yyin; extern int
yylineno; extern char *yytext;
void insertSTtype(char *,char *);
void insertSTvalue(char *, char *);
void incertCT(char *, char *); void
printST(); void printCT();


int main(int argc , char **argv){           yyin =
fopen(argv[1], "r");      yyparse();        if(flag == 0){
printf("Status: Parsing Complete - Valid\n");
printf("%30s SYMBOL
TABLE\n", " ");                          printf("%30s %s\n", "
", "------------");
                        printST();
                        printf("\n\n%30s CONSTANT TABLE\n", " ");
printf("%30s %s\n", " ", "--------------");
                        printCT();
            }
}
void yyerror(char *s){              printf("%d %s
%s\n", yylineno, s, yytext); flag=1;
printf("Status: Parsing Failed - Invalid\n");
}
void ins(){
        insertSTtype(curid,curtype);
}
void insV(){
insertSTvalue(curid,curval);
}
int yywrap(){
        return 1;
}
```

Explanation:

After the lexical analysis stage, we get the stream of tokens from source C code which is given as input to the parser. Parser verifies that a string of token names can be generated by the grammar of the source language. Here we are expecting the parser to report any syntax errors in an intelligible manner and to recover from the commonly occurring errors to continue processing the remainder of the program

Input1:

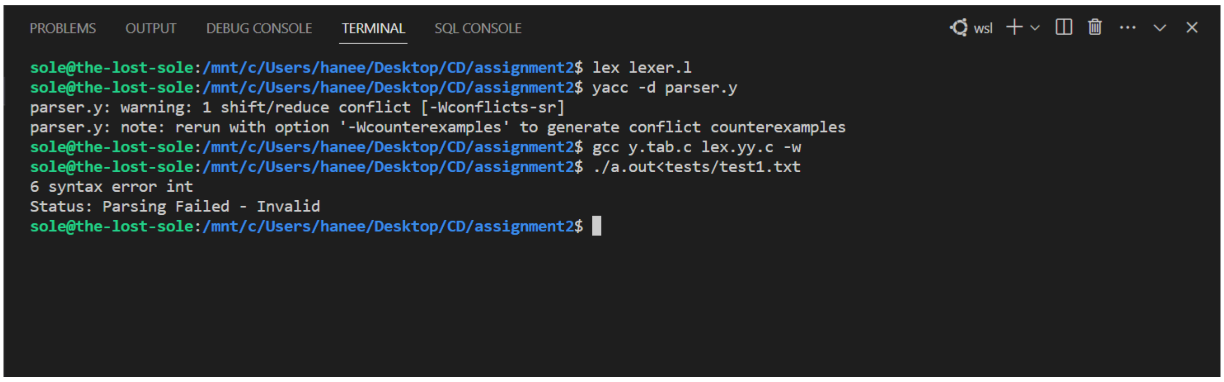


test1.txt

assignment2 > tests > test1.txt

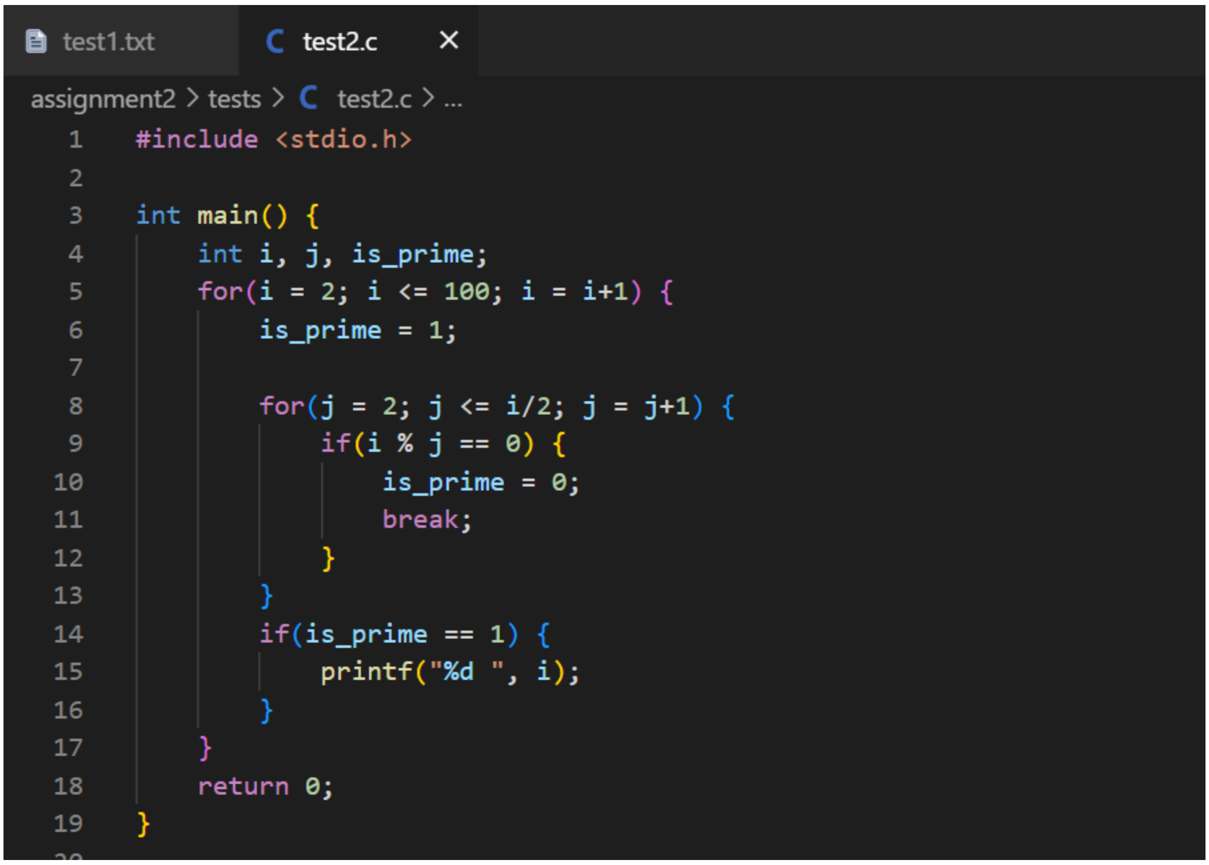
```
#include <stdio.h>

int main(){
    int a = 2;
    int b = 7   // error due to no semicolon
    int c = 10;
    printf("value of a is ",a);

    return 0;
}
```

Output1:



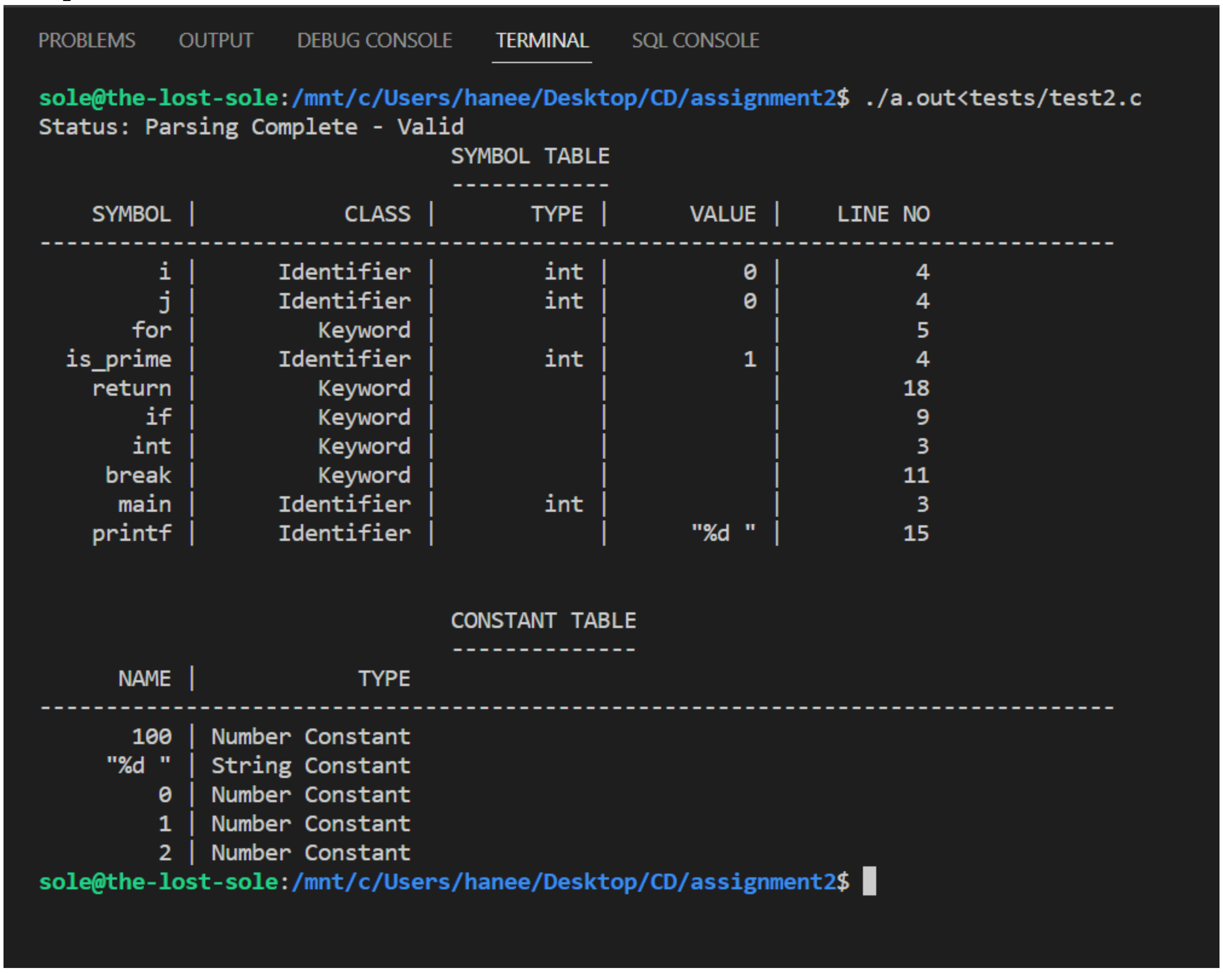
```
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment2$ lex lexer.l
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment2$ yacc -d parser.y
parser.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
parser.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment2$ gcc y.tab.c lex.yy.c -w
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment2$ ./a.out<tests/test1.txt
6 syntax error int
Status: Parsing Failed - Invalid
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment2$
```

Input2:



```c
#include <stdio.h>

int main() {
    int i, j, is_prime;
    for(i = 2; i <= 100; i = i+1) {
        is_prime = 1;

        for(j = 2; j <= i/2; j = j+1) {
            if(i % j == 0) {
                is_prime = 0;
                break;
            }
        }
        if(is_prime == 1) {
            printf("%d ", i);
        }
    }
    return 0;
}
```

Output2:



```
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment2$ ./a.out<tests/test2.c
Status: Parsing Complete - Valid
                                SYMBOL TABLE
                                ------------
    SYMBOL |             CLASS |        TYPE |       VALUE |      LINE NO
--------------------------------------------------------------------------------
         i |        Identifier |         int |           0 |            4
         j |        Identifier |         int |           0 |            4
       for |           Keyword |             |             |            5
  is_prime |        Identifier |         int |           1 |            4
    return |           Keyword |             |             |           18
        if |           Keyword |             |             |            9
       int |           Keyword |             |             |            3
     break |           Keyword |             |             |           11
      main |        Identifier |         int |             |            3
    printf |        Identifier |             |       "%d " |           15


                                CONSTANT TABLE
                                --------------
      NAME |             TYPE
--------------------------------------------------------------------------------
       100 | Number Constant
     "%d " | String Constant
         0 | Number Constant
         1 | Number Constant
         2 | Number Constant
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment2$ 
```

NATIONAL INSTITUTE OF TECHNOLOGY-TRICHY

CSPC 62 - COMPILER DESIGN ASSIGNMENT – 03

Intermediate Code Generation

TEAM MEMBERS

106120015: Ashnu
106120063: Jagadeesh
106120073: Haneel Kumar
106120083: Gopi Reddy

DATE: 06/04/2023

Modified Lex Code:

```
%{
            #include <stdio.h>
            #include <string.h>
            #include <stdlib.h>
            #include "y.tab.h"

            struct symboltable{
   char name[100];    char class[100];
 char type[100];            char value[100];
int nestval;       int lineno;          int
length;
                        int params_count;
            }ST[1001];

            struct constanttable{
                        char name[100];
char type[100];
 int length;        }CT[1001];
int currnest = 0;   extern int
yylval;                 int hash(char
*str){
                        int value = 0;
                        for(int i = 0 ; i < strlen(str) ; i++){
            value = 10*value + (str[i] - 'A');
value = value % 1001;                          while(value < 0)

            value = value + 1001;
                        }
                        return value;
```

```
        }
        int lookupST1(char *str){
                int value = hash(str);
if(ST[value].length == 0)
                        return 0;
                else if(strcmp(ST[value].name,str)==0)
                        return value;


        }
        int lookupST2(char *str){


                        if(ST[value].length!=0 && strcmp(ST[value].name,str)!=0 ){
                                for(int i = value + 1 ; i!=value ; i = (i+1)%1001){
                                        if(strcmp(ST[i].name,str)==0)
                                                return i;
                                }
                                return 0;
                        }
                }

        int lookupCT1(char *str){
        int value = hash(str);
if(CT[value].length == 0)
                        return 0;
                else if(strcmp(CT[value].name,str)==0)
                        return 1;


                        return 0;
                }
        }
```

```
            int lookupCT2(char *str){  int
            value = hash(str);


                        if(CT[value].length != 0 &&  strcmp(CT[value].name,str)!=0){
for(int i = value + 1 ; i!=value ; i = (i+1)%1001){
                                                if(strcmp(CT[i].name,str)==0)
                                                            return 1;
                                    }
                                    return 0;
                        }
            }
            void insertSTline(char *str1, int line){
                        for(int i = 0 ; i < 1001 ; i++){
if(strcmp(ST[i].name,str1)==0)
                                                ST[i].lineno = line;
                        }
            }
            void insertST(char *str1, char *str2){
if(lookupST1(str1) || lookupST2(str1){
     (if(strcmp(ST1[lookupST(str1)].class,"Identifier")==0                              ||
if(strcmp(ST2[lookupST(str1)].class,"Identifier")==0   )&&   strcmp(str2,"Array
Identifier")==0){
                                                printf("Error use of array\n");
                                                exit(0);
                                    }
                                    return;
                        }
                        else{                              int
value = hash(str1);
 if(ST[value].length == 0){
strcpy(ST[value].name,str1);
 strcpy(ST[value].class,str2);
```

```
                                        ST[value].length = strlen(str1);
                                        ST[value].nestval = 9999;
ST[value].params_count = -1;
 insertSTline(str1,yylineno);
                                        return;
                              }
                              int pos = 0;
                              for (int i = value + 1 ; i!=value ; i = (i+1)%1001){
                                        if(ST[i].length == 0){
                                                  pos = i;
                                                  break;
                                        }
                              }
                              strcpy(ST[pos].name,str1);
strcpy(ST[pos].class,str2);
                              ST[pos].length = strlen(str1);
                              ST[pos].nestval = 9999;
                              ST[pos].params_count = -1;
                    }
          }
          void insertSTtype(char *str1, char *str2){
for(int i = 0 ; i < 1001 ; i++){
                              if(strcmp(ST[i].name,str1)==0)
 strcpy(ST[i].type,str2);
                    }
   }
          void insertSTvalue(char *str1, char *str2){
                    for(int i = 0 ; i < 1001 ; i++){
 if(strcmp(ST[i].name,str1)==0 && ST[i].nestval == currnest)
                                        strcpy(ST[i].value,str2);
                    }
```

```
        } void insertSTnest(char *s, int
        nest){
                if(lookupST1(s) && ST[lookupST1(s)].nestval != 9999 || lookupST2(s) &&
ST[lookupST2(s)].nestval != 9999){
       int pos = 0;
int value = hash(s);
                        for (int i = value + 1 ; i!=value ; i = (i+1)%1001){
                                if(ST[i].length == 0){
                                        pos = i;
                        break;
                                }
                        }
                        strcpy(ST[pos].name,s);
                        strcpy(ST[pos].class,"Identifier");
                        ST[pos].length = strlen(s);
                        ST[pos].nestval = nest;
                        ST[pos].params_count = -1;
                        ST[pos].lineno = yylineno;
                }
                else{
                        for(int i = 0 ; i < 1001 ; i++){
if(strcmp(ST[i].name,s)==0)
                                        ST[i].nestval = nest;
                        }
                }
        }
 void insertSTparamscount(char *s, int count1){   for(int i = 0
; i < 1001 ; i++){
                        if(strcmp(ST[i].name,s)==0 )
                                ST[i].params_count = count1;
                }
```

```
            }
            int getSTparamscount(char *s){
for(int i = 0 ; i < 1001 ; i++){
if(strcmp(ST[i].name,s)==0 )     return
ST[i].params_count;
                        }
                        return -1;
            }
            void insertSTF(char *s){
 for(int i = 0 ; i < 1001 ; i++){
                                    if(strcmp(ST[i].name,s)==0){
 strcpy(ST[i].class,"Function");              return;

                                    }
                        }
            }


            void insertCT(char *str1, char *str2){
if(lookupCT1(str1) && lookupCT2(str1))
                                    return;
            else{                               int value =
hash(str1);
 if(CT[value].length == 0){
strcpy(CT[value].name,str1);
strcpy(CT[value].type,str2);
 CT[value].length = strlen(str1);
                                                return;
                                    }
                                    int pos = 0;
                                    for (int i = value + 1 ; i!=value ; i = (i+1)%1001){
                                                if(CT[i].length == 0){
```

```
                                        pos = i;
                        break;

                                }
                        }
                        strcpy(CT[pos].name,str1);
strcpy(CT[pos].type,str2);
                        CT[pos].length = strlen(str1);
                }
        }
        int check_id_is_func(char *s){
for(int i = 0 ; i < 1000 ; i++){
                        if(strcmp(ST[i].name,s)==0){
if(strcmp(ST[i].class,"Function")==0)                   return 1;

                        }}
                return 0;
        }
        char gettype(char *s, int flag){
for(int i = 0 ; i < 1001 ; i++ ){
                        if(strcmp(ST[i].name,s)==0)
                        return ST[i].type[0];
                }
        }
        void printST(){
                printf("%10s | %15s | %10s | %10s | %10s | %15s |\n","SYMBOL", "CLASS",
"TYPE","VALUE", "LINE NO", "PARAMS COUNT");
                for(int i=0;i<87;i++)
        printf("-");                    printf("\n");
                for(int i = 0 ; i < 1001 ; i++){
                        if(ST[i].length == 0)
                                continue;
```

```
                              printf("%10s | %15s | %10s | %10s | %10d | %15d |\n",ST[i].name,
ST[i].class, ST[i].type, ST[i].value, ST[i].lineno, ST[i].params_count);

                    }

  }
 void printCT(){   printf("%10s | %15s\n","NAME",
"TYPE");

                    for(int i=0;i<81;i++)

          printf("-");                  printf("\n");

                    for(int i = 0 ; i < 1001 ; i++){

                              if(CT[i].length == 0)

                                        continue;

                              printf("%10s | %15s\n",CT[i].name, CT[i].type);

                    }

          }

          char curid[20];  char

          curtype[20];

          char curval[20];


%}

DE "define"

IN "include"


%%

\n        {yylineno++;}

([#][" "]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|" "|"\t"] { }

([#][" "]*({DE})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"]                                   { }

\/\/(.*)                        { }

\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/
                              { }

[ \n\t] ;

";"                                       { return(';'); }

","                                       { return(','); }
```

```
("{")                    { return('{'); }
("}")                    { return('}'); }
"("                              { return('('); }
")"                              { return(')'); }
("["|"<:")               { return('['); }
("]"|":>")               { return(']'); }
":"                              { return(':'); }
"."                              { return('.'); }

"char"             { strcpy(curtype,yytext); insertST(yytext, "Keyword");return CHAR;} "double"
 { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return DOUBLE;}
"else"                   { insertST(yytext, "Keyword"); return ELSE;}
"float"                  { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return FLOAT;}
"while"                  { insertST(yytext, "Keyword"); return WHILE;}
"do"                     { insertST(yytext, "Keyword"); return DO;}
"for"                    { insertST(yytext, "Keyword"); return FOR;}
"if"                     { insertST(yytext, "Keyword"); return IF;}
"int"                    { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return INT;}
"return"                 { insertST(yytext, "Keyword"); return RETURN;}
"void"   { strcpy(curtype,yytext);   insertST(yytext, "Keyword");  return VOID;} "break"  {
insertST(yytext, "Keyword");  return BREAK;}

"<="                     { return lessthan_assignment_operator; }
"<"                              { return lessthan_operator; }
">="                     { return greaterthan_assignment_operator; }
">"                              { return greaterthan_operator; }
"=="                     { return equality_operator; }
"!="                     { return inequality_operator; }
"&&"                     { return AND_operator; }
"||"                     { return OR_operator; }
"&"                              { return amp_operator; }
```

```
"!"                         { return exclamation_operator; }

"-"                         { return subtract_operator; }

"+"                         { return add_operator; }

"*"                         { return multiplication_operator; }

"/"                         { return division_operator; }

"%"                         { return modulo_operator; }

"\="                  { return assignment_operator;}


\"[^\n]*\"/[;|,|\)]                    {strcpy(curval,yytext); insertCT(yytext,"String Constant");
return string_constant;}

\'[A-Z|a-z]\'/[;|,|\)|:]   {strcpy(curval,yytext); insertCT(yytext,"Character Constant"); return
character_constant;}

[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[  {strcpy(curid,yytext); insertST(yytext, "Array Identifier");  return
array_identifier;}

[1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^]
        {strcpy(curval,yytext); insertCT(yytext, "Number Constant"); yylval = atoi(yytext);
return integer_constant;}

([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^]
        {strcpy(curval,yytext); insertCT(yytext, "Floating Constant"); return float_constant;}

[A-Za-z_][A-Za-z_0-9]* {strcpy(curid,yytext); insertST(curid,"Identifier"); return identifier;}


(.?) {

        if(yytext[0]=='#')           printf("Error in Pre-Processor

directive at line no. %d\n",yylineno);        else if(yytext[0]=='/')

  printf("ERR_UNMATCHED_COMMENT at line no. %d\n",yylineno);   else

if(yytext[0]=='"')           printf("ERR_INCOMPLETE_STRING at line no.

%d\n",yylineno);

        else

                printf("ERROR at line no. %d\n",yylineno);

 printf("%s\n", yytext);

        return 0;

}

%%
```

YACC Code:

```
%{

            #include <stdio.h>

            #include <string.h>

            #include <stdlib.h>


            void yyerror(char* s);

 int yylex();

 void ins();  void

insV();  int flag=0;
            extern char curid[20];   char     extern
curtype[20];     extern char

                                                            int
curval[20];      extern  int  currnest;
check_id_is_func(char   *);         void
insertST(char*,    char*);          void
insertSTnest(char*, int);   void


insertSTparamscount(char*, int);         int
getSTparamscount(char*);       char

currfunctype[100];       char currfunc[100];   char

currfunccall[100];        void insertSTF(char*);

char gettype(char*,int);            char

getfirst(char*);   void push(char *s);    void

codegen();   void codeassign();            char*

itoa1(int num, char* str, int base); char* itoa2(int

num, char* str, int base);           void reverse(char

str[], int length);

            void

swap(char*,char*);       void

label1();           void label2();
```

```
 void label3();   void label4();
void label5();   void label6();
void codegencon();

              void funcgen();
 void  funcgenend();
void arggen();   void
callgen();


              int params_count=0;   int
call_params_count=0;   int top =
0,count=0,ltop=0,lno=0;          char
temp[3] = "t";
%}


%nonassoc IF
%token INT CHAR FLOAT DOUBLE RETURN MAIN VOID WHILE FOR DO BREAK identifier
array_identifier func_identifier integer_constant string_constant float_constant
character_constant
%nonassoc ELSE
%right assignment_operator
%left OR_operator AND_operator amp_operator equality_operator inequality_operator
lessthan_assignment_operator lessthan_operator greaterthan_assignment_operator
greaterthan_operator leftshift_operator rightshift_operator add_operator subtract_operator
multiplication_operator division_operator modulo_operator
%right exclamation_operator
%left increment_operator decrement_operator
%start program
%%
program : declaration_list; declaration_list
: declaration D
D   : declaration_list
              | ;
declaration : variable_declaration
```

```
            | function_declaration

variable_declaration    : type_specifier variable_declaration_list ';'

variable_declaration_list            : variable_declaration_list ',' variable_declaration_identifier |
variable_declaration_identifier;  variable_declaration_identifier : identifier

{insertSTnest(curid,currnest); ins();  } vdi

 | array_identifier {insertSTnest(curid,currnest); ins();  } vdi;     vdi : identifier_array_type |

assignment_operator simple_expression  ; identifier_array_type  : '[' initilization_params

            | ;

initilization_params      : integer_constant ']' initilization {if($$ < 1) {printf("Wrong array
size\n"); exit(0);} }

            | ']' string_initilization; initilization

: string_initilization

            | array_initialization

            | ;

type_specifier : INT | CHAR | FLOAT  | DOUBLE

            | VOID  ;

function_declaration     : function_declaration_type function_declaration_param_statement;

function_declaration_type         : type_specifier identifier '('  { strcpy(currfunctype, curtype);
strcpy(currfunc, curid); insertSTF(curid); ins(); };

function_declaration_param_statement             : {params_count=0;}params ')' {funcgen();}
statement {funcgenend();};  params : parameters_list {

insertSTparamscount(currfunc, params_count); }| { insertSTparamscount(currfunc,

params_count); }; parameters_list         : type_specifier parameters_identifier_list ;

parameters_identifier_list         : param_identifier

parameters_identifier_list_breakup; parameters_identifier_list_breakup           : ','

parameters_list

            | ;

param_identifier           : identifier { ins();insertSTnest(curid,1); params_count++; }
param_identifier_breakup; param_identifier_breakup              : '[' ']'

            | ;

statement          : expression_statment | compound_statement

            | conditional_statements | iterative_statements

            | return_statement | break_statement
 | variable_declaration;  compound_statement :
{currnest++;} '{'  statment_list  '}' ;
statment_list       :       statement       statment_list       |       ;
expression_statment  : expression ';'  | ';' ;

conditional_statements              : IF '(' simple_expression ')' {label1();if($3!=1){printf("Condition
checking is not of type int\n");exit(0);}} statement {label2();}  conditional_statements_breakup;
conditional_statements_breakup          : ELSE statement {label3();}
            | {label3();};
iterative_statements   : WHILE '(' {label4();} simple_expression ')'
{label1();if($4!=1){printf("Condition checking is not of type int\n");exit(0);}} statement
{label5();}
            | FOR '(' expression ';' {label4();} simple_expression ';'
{label1();if($6!=1){printf("Condition checking is not of type int\n");exit(0);}} expression
')'statement {label5();}
            | {label4();}DO statement WHILE '(' simple_expression
')'{label1();label5();if($6!=1){printf("Condition checking is not of type int\n");exit(0);}} ';';
return_statement
 : RETURN ';' {if(strcmp(currfunctype,"void")) {printf("Returning void of a non-void
function\n"); exit(0);}}
 | RETURN expression ';' {   if(!strcmp(currfunctype, "void")){       yyerror("Function is void");
            }
   if((currfunctype[0]=='i' || currfunctype[0]=='c') && $2!=1){     printf("Expression
doesn't match return type of function\n"); exit(0);
            }
   };


break_statement          : BREAK ';' ; string_initilization             :
assignment_operator string_constant {insV();} ; array_initialization  : assignment_operator
'{' array_int_declarations '}'; array_int_declarations : integer_constant
array_int_declarations_breakup; array_int_declarations_breakup         : ','
array_int_declarations
            | ;
expression        : mutable assignment_operator {push("=");} expression   {
```

```
            if($1==1 && $4==1)
               $$=1;
 else{        $$=-1;
printf("Type mismatch\n");
exit(0);}

                     codeassign();

            }
            | simple_expression {if($1 == 1) $$=1; else $$=-1;} ;



simple_expression : simple_expression OR_operator and_expression {push("||");} {if($1 == 1 &&
$3==1) $$=1; else $$=-1; codegen();}

            | and_expression {if($1 == 1) $$=1; else $$=-1;};


and_expression : and_expression AND_operator {push("&&");} unary_relation_expression  {if($1
== 1 && $3==1) $$=1; else $$=-1; codegen();}

            |unary_relation_expression {if($1 == 1) $$=1; else $$=-1;} ;


unary_relation_expression       : exclamation_operator {push("!");} unary_relation_expression
{if($2==1) $$=1; else $$=-1; codegen();}

            | regular_expression {if($1 == 1) $$=1; else $$=-1;} ;


regular_expression  : regular_expression relational_operators sum_expression {if($1 == 1 &&
$3==1) $$=1; else $$=-1; codegen();}

            | sum_expression {if($1 == 1) $$=1; else $$=-1;} ;


relational_operators   : greaterthan_assignment_operator {push(">=");} |
lessthan_assignment_operator {push("<=");} | greaterthan_operator {push(">");}|
lessthan_operator {push("<");}| equality_operator {push("==");}| inequality_operator
{push("!=");} ;


sum_expression            : sum_expression sum_operators term  {if($1 == 1 && $3==1) $$=1;
else $$=-1; codegen();}

            | term {if($1 == 1) $$=1; else $$=-1;};
```

```
sum_operators              : add_operator {push("+");}

            | subtract_operator {push("-");} ;


term     : term MULOP factor {if($1 == 1 && $3==1) $$=1; else $$=-1; codegen();}

| factor {if($1 == 1) $$=1; else $$=-1;} ;


MULOP             : multiplication_operator {push("*");}| division_operator {push("/");} |
modulo_operator {push("%");} ;


factor  : immutable {if($1 == 1) $$=1; else $$=-1;}

 | mutable {if($1 == 1) $$=1; else $$=-1;} ; mutable

 : identifier {

            push(curid);

if(check_id_is_func(curid))

               {printf("Function name used as Identifier\n"); exit(8);}

if(gettype(curid,0)=='i' || gettype(curid,1)== 'c')

               $$ = 1;

            else

               $$ = -1;

            }

            | array_identifier '[' expression ']'

            {if(gettype(curid,0)=='i' || gettype(curid,1)== 'c')

               $$ = 1;

            else

               $$ = -1;

            };


immutable        : '(' expression ')' {if($2==1) $$=1; else $$=-1;}

            | call {if($1==-1) $$=-1; else $$=1;}

            | constant {if($1==1) $$=1; else $$=-1;};
```

```
call    : identifier '('{
insertSTF(curid);
            strcpy(currfunccall,curid);
if(gettype(curid,0)=='i' || gettype(curid,1)== 'c')
                $$ = 1;
            else
                $$ = -1;
   call_params_count=0;
            }
            arguments ')'
            { if(strcmp(currfunccall,"printf"))
            {
            if(getSTparamscount(currfunccall)!=call_params_count){
yyerror("Number of arguments in function call doesn't match number of parameters");
   exit(8);
            }
            }
            callgen();
            };

arguments        : arguments_list | ;  arguments_list  :
arguments_list ',' exp { call_params_count++; }
            | exp { call_params_count++; };

exp : identifier {arggen(1);} | integer_constant {arggen(2);} | string_constant {arggen(3);} |
float_constant {arggen(4);} | character_constant {arggen(5);} ;

constant            : integer_constant         {  insV(); codegencon(); $$=1; }
            | string_constant           {  insV(); codegencon();$$=-1;}
            | float_constant {  insV(); codegencon();}
```

```
              | character_constant{  insV(); codegencon();$$=1; };


%%
extern FILE *yyin; extern
int yylineno; extern char
*yytext; void
insertSTtype(char *,char
*); void
insertSTvalue(char *, char
*); void incertCT(char *,
char *); void printST();
void printCT();


struct  stack{      char
value[100];           int
labelvalue;
}s[100],label[100];


void push(char *x){
strcpy(s[++top].value,x);
}
void swap(char *x, char *y){
              char temp = *x;
 *x = *y;
              *y = temp;
}
void reverse(char str[], int length){
int start = 0;     int end = length -1;
while (start < end){
swap((str+start), (str+end));
       start++;
end--;
```

```
   }
}
char* itoa1(int num, char* str, int
base){  int i = 0; int isNegative = 0; if
(num == 0){
str[i++] = '0';
str[i] = '\0'; return
str;
}
if (num < 0 && base == 10){ isNegative
= 1; num = -num;
}
while (num != 0){ int rem = num % base;str[i++] = (rem > 9)? (rem-
10) + 'a' : rem + '0'; num = num/base;
}
if (isNegative)
str[i++] = '-'; str[i] =
'\0'; reverse(str,
i); return str;
}

char* itoa2(int num, char* str, int base){  int
i = 0; int
isNegative = 0;

if (num < 0 && base == 10){ isNegative
= 1;
num = -num;
}
while (num != 0){ int rem = num % base;str[i++] = (rem > 9)? (rem-
10) + 'a' : rem + '0';
num = num/base;
```

```
}
if (isNegative)
str[i++] = '-'; str[i] =
'\0'; reverse(str,
i); return str;
}
void codegen(){
strcpy(temp,"t");
 char buffer[100];   itoa1(count,buffer,10);   itoa2(count,buffer,10);
strcat(temp,buffer);     printf("%s = %s %s %s\n",temp,s[top-2].value,s[top-
1].value,s[top].value);               top = top - 2;
 strcpy(s[top].value,temp);
            count++;
}
void codegencon(){
            strcpy(temp,"t");          char
buffer[100];     itoa1(count,buffer,10);
itoa2(count,buffer,10);
strcat(temp,buffer);   printf("%s =
%s\n",temp,curval);
            push(temp);  count++;
}
void codeassign(){       printf("%s = %s\n",s[top-
2].value,s[top].value);
            top = top - 2;
}
void label1(){
            strcpy(temp,"L");          char buffer[100];
itoa1(lno,buffer,10);
```

```
 itoa2(lno,buffer,10);   strcat(temp,buffer);   printf("IF not
%s GoTo %s\n",s[top].value,temp);
label[++ltop].labelvalue = lno++;
}
void label2(){
             strcpy(temp,"L");
 char buffer[100];   itoa1(lno,buffer,10);
itoa2(lno,buffer,10);   strcat(temp,buffer);
printf("GoTo %s\n",temp);   strcpy(temp,"L");
itoa1(label[ltop].labelvalue,buffer,10);
itoa2(label[ltop].labelvalue,buffer,10);
             strcat(temp,buffer);
 printf("%s:\n",temp);
             ltop--;
label[++ltop].labelvalue=lno++;
}

void label3(){
             strcpy(temp,"L");  char buffer[100];
itoa1(label[ltop].labelvalue,buffer,10);
itoa2(label[ltop].labelvalue,buffer,10);
             strcat(temp,buffer);
 printf("%s:\n",temp);   ltop--;
}
void label4(){
             strcpy(temp,"L");           char
buffer[100];      itoa1(lno,buffer,10);
itoa2(lno,buffer,10);
strcat(temp,buffer);
printf("%s:\n",temp);
label[++ltop].labelvalue = lno++;
```

```
}
void label5(){
            strcpy(temp,"L");
 char        buffer[100];                            itoa1(label[ltop-
1].labelvalue,buffer,10);                            itoa2(label[ltop-
1].labelvalue,buffer,10);    strcat(temp,buffer);
 printf("GoTo %s:\n",temp);   strcpy(temp,"L");
itoa1(label[ltop].labelvalue,buffer,10);
itoa2(label[ltop].labelvalue,buffer,10);
            strcat(temp,buffer);
 printf("%s:\n",temp);   ltop = ltop
- 2;
}

void funcgen(){
            printf("func begin %s\n",currfunc);
} void funcgenend(){
printf("func end\n\n");
}
void arggen(int i){
if(i==1)
printf("refparam %s\n",
curid);

            else
               printf("refparam %s\n", curval);
}
void callgen(){
printf("refparam result\n");
 push("result");            printf("call %s,
%d\n",currfunccall,call_params_count);
}
```

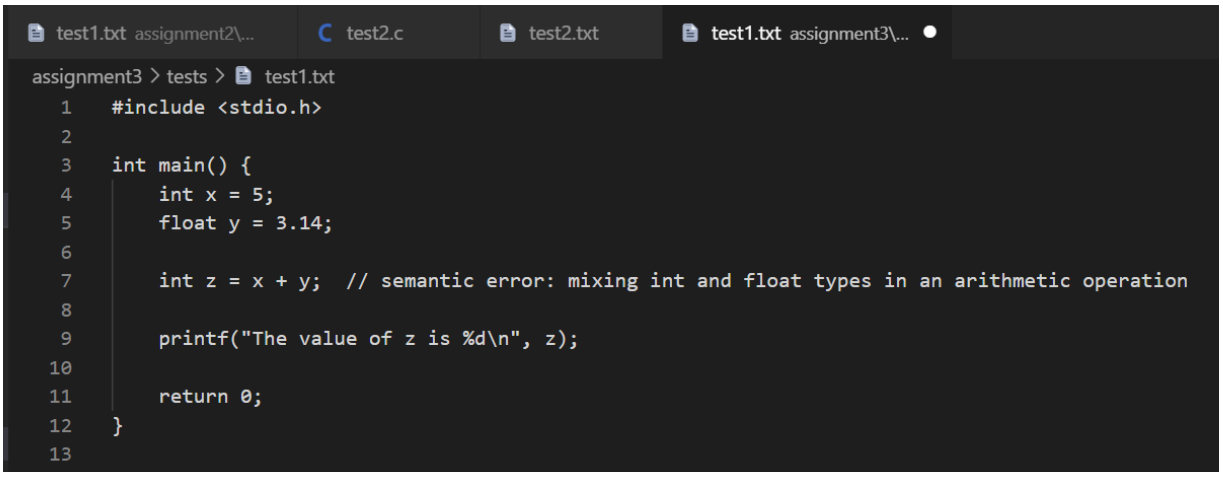
```
int main(int argc , char **argv){           yyin =
fopen(argv[1], "r");      yyparse();        if(flag == 0){
printf("\nStatus: Parsing
Complete - Valid\n");                printf("%30s SYMBOL
TABLE\n", " ");                          printf("%30s %s\n", "
", "------------");
                        printST();


                        printf("\n\n%30s CONSTANT TABLE\n", " ");
printf("%30s %s\n", " ", "--------------");
                        printCT();
            }
}
void yyerror(char *s){              printf("%d %s
%s\n", yylineno, s, yytext);
            flag=1;
            printf("\nStatus: Parsing Failed - Invalid\n");
            exit(7);
}
void ins(){
            insertSTtype(curid,curtype);
}
void insV(){
insertSTvalue(curid,curval);
}
int yywrap(){
            return 1;
}
```

Explanation:

The final frontend phase of a compiler design stage is Intermediate Code Generation. A compiler's ultimate purpose is to get programmes written in a high-level language to run on a computer. This is because the programme will eventually have to be written as machine code that can execute on the computer. Many compilers utilise a medium-level language as a bridge between high-level and very low-level machine code. These stepping-stone languages are known as intermediate code. It offers reduced abstraction from the source level while retaining certain high level information. Depending on whether it is ByteCode for Java or three-address-code (language independent), intermediate code can be represented in a variety of forms.

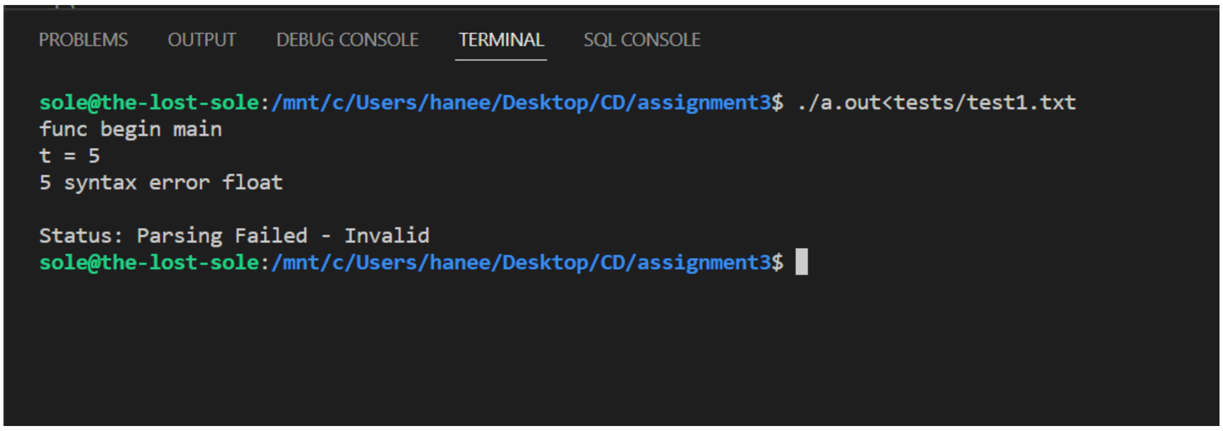
We used three-address-code here.

Input1:


test1.txt assignment2\...
test2.c
test2.txt
test1.txt assignment3\...
assignment3 > tests > test1.txt

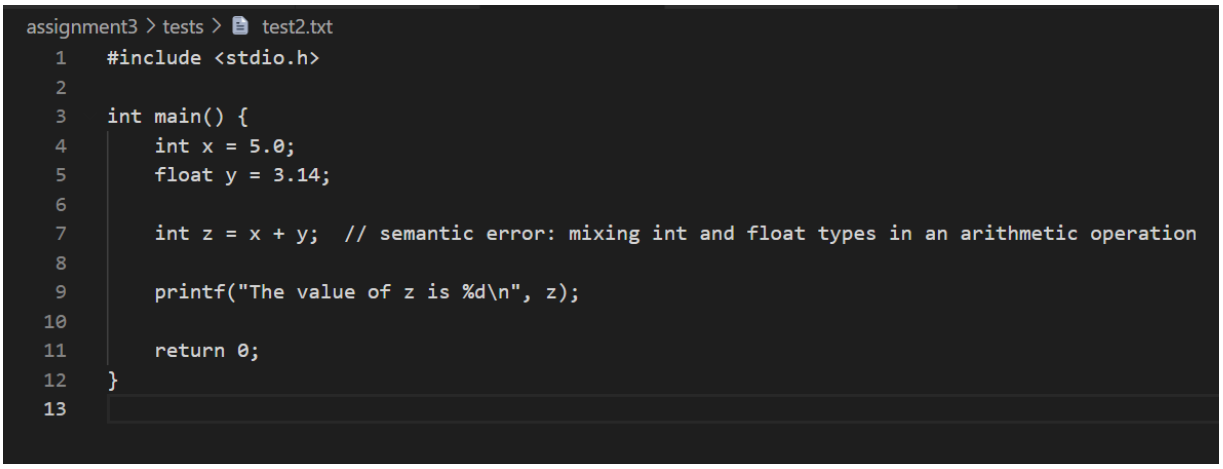
```c
#include <stdio.h>

int main() {
    int x = 5;
    float y = 3.14;

    int z = x + y;  // semantic error: mixing int and float types in an arithmetic operation

    printf("The value of z is %d\n", z);

    return 0;
}

```

Output1:



```
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment3$ ./a.out<tests/test1.txt
func begin main
t = 5
5 syntax error float

Status: Parsing Failed - Invalid
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment3$ 
```

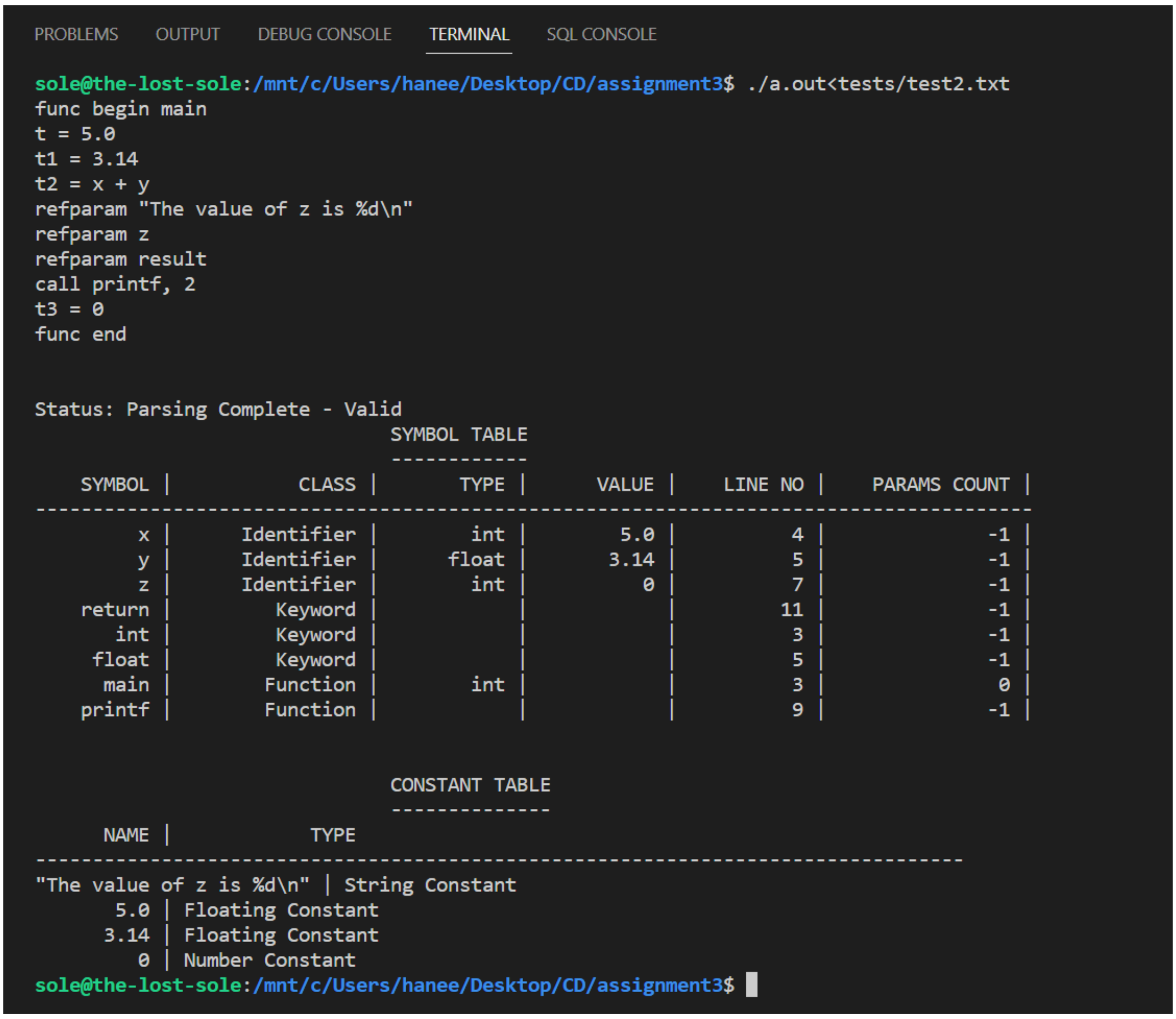
Input2:



assignment3 > tests > test2.txt

```
#include <stdio.h>

int main() {
    int x = 5.0;
    float y = 3.14;

    int z = x + y;  // semantic error: mixing int and float types in an arithmetic operation

    printf("The value of z is %d\n", z);

    return 0;
}

```

Output2:



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL SQL CONSOLE

```
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment3$ ./a.out<tests/test2.txt
func begin main
t = 5.0
t1 = 3.14
t2 = x + y
refparam "The value of z is %d\n"
refparam z
refparam result
call printf, 2
t3 = 0
func end


Status: Parsing Complete - Valid
                               SYMBOL TABLE
                               ------------

    SYMBOL |            CLASS |         TYPE |         VALUE |        LINE NO |         PARAMS COUNT |
---------------------------------------------------------------------------------------
         x |       Identifier |          int |           5.0 |              4 |                   -1 |
         y |       Identifier |        float |          3.14 |              5 |                   -1 |
         z |       Identifier |          int |             0 |              7 |                   -1 |
    return |          Keyword |              |               |             11 |                   -1 |
       int |          Keyword |              |               |              3 |                   -1 |
     float |          Keyword |              |               |              5 |                   -1 |
      main |         Function |          int |               |              3 |                    0 |
    printf |         Function |              |               |              9 |                   -1 |


                               CONSTANT TABLE
                               --------------

      NAME |             TYPE
---------------------------------------------------------------------------------
"The value of z is %d\n" | String Constant
       5.0 | Floating Constant
      3.14 | Floating Constant
         0 | Number Constant
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment3$ 
```

NATIONAL INSTITUTE OF TECHNOLOGY-TRICHY

CSPC 62 - COMPILER DESIGN ASSIGNMENT – 04

Intermediate Code Optimisation

TEAM MEMBERS

106120015: Ashnu
106120063: Jagadeesh
106120073: Haneel Kumar
106120083: Gopi Reddy

DATE: 01/05/2023

Modified Lex Code:

```
%{
            #include <stdio.h>
            #include <string.h>
            #include <stdlib.h> #include
            "y.tab.h"

            struct symboltable{
                        char name[100];
char class[100];            char
type[100];         char value[100];
 int nestval;                 int lineno;
int length;
 int params_count;
            }ST[1001];

            struct constanttable{
                        char name[100];
char type[100];
 int length;        }CT[1001];
int currnest = 0;   extern int
yylval;             int hash(char
*str){
                        int value = 0;
                        for(int i = 0 ; i < strlen(str) ; i++){
            value = 10*value + (str[i] - 'A');
value = value % 1001;                          while(value < 0)
```

```
          value = value + 1001;
                    }
                    return value;
          }         int
lookupST(char *str){
                    int value = hash(str);  if(ST[value].length
                    == 0)
                              return 0;
                    else if(strcmp(ST[value].name,str)==0)
                              return value;
else{
                              for(int i = value + 1 ; i!=value ; i = (i+1)%1001){
                                        if(strcmp(ST[i].name,str)==0)
                                                  return i;
                              }
                              return 0;
                    }
          }
          int lookupCT(char *str){
          int value = hash(str);
 if(CT[value].length == 0)
                              return 0;
                    else if(strcmp(CT[value].name,str)==0)
                              return 1;
          else{                         for(int i = value + 1 ; i!=value ; i =
(i+1)%1001){
                                        if(strcmp(CT[i].name,str)==0)
                                                  return 1;
                              }
                              return 0;
                    }
```

```
            }
            void insertSTline(char *str1, int line){
            for(int i = 0 ; i < 1001 ; i++){
if(strcmp(ST[i].name,str1)==0)
                                    ST[i].lineno = line;
                  }
            }
            void insertST(char *str1, char *str2){
                  if(lookupST(str1)){
   if(strcmp(ST[lookupST(str1)].class,"Identifier")==0 &&  strcmp(str2,"Array
Identifier")==0){
                                    printf("Error use of array\n");
                                    exit(0);
                              }
                              return;
                  }
                  else{                   int
value = hash(str1);
 if(ST[value].length == 0){
strcpy(ST[value].name,str1);
 strcpy(ST[value].class,str2);
                                    ST[value].length = strlen(str1);
                                    ST[value].nestval = 9999;
 ST[value].params_count = -1;
 insertSTline(str1,yylineno);
                                    return;
                              }
                              int pos = 0;
                              for (int i = value + 1 ; i!=value ; i = (i+1)%1001){
                                    if(ST[i].length == 0){
                                        pos = i;
                            break;
                }

                            }
                            strcpy(ST[pos].name,str1);
strcpy(ST[pos].class,str2);
                            ST[pos].length = strlen(str1);
                            ST[pos].nestval = 9999;
                            ST[pos].params_count = -1;
                      }
                }
                void insertSTtype(char *str1, char *str2){
                      for(int i = 0 ; i < 1001 ; i++){
                            if(strcmp(ST[i].name,str1)==0)
strcpy(ST[i].type,str2);
                      }
   }
                void insertSTvalue(char *str1, char *str2){          for(int i = 0 ; i < 1001 ; i++){
                if(strcmp(ST[i].name,str1)==0 && ST[i].nestval ==
currnest)
                                  strcpy(ST[i].value,str2);
                      }
                }
                void insertSTnest(char *s, int nest){          if(lookupST(s)
&& ST[lookupST(s)].nestval != 9999){
          int pos = 0;
int value = hash(s);
                            for (int i = value + 1 ; i!=value ; i = (i+1)%1001){
                                  if(ST[i].length == 0){
                                        pos = i;
                            break;
                }
                            }
                            strcpy(ST[pos].name,s);
                            strcpy(ST[pos].class,"Identifier");
                            ST[pos].length = strlen(s);
                            ST[pos].nestval = nest;
                            ST[pos].params_count = -1;
                            ST[pos].lineno = yylineno;
                    }
                    else{
                    for(int i = 0 ; i < 1001 ; i++){
if(strcmp(ST[i].name,s)==0)
                                        ST[i].nestval = nest;
                            }
                    }
            }
 void insertSTparamscount(char *s, int count1){   for(int i = 0
; i < 1001 ; i++){
                            if(strcmp(ST[i].name,s)==0 )
                                    ST[i].params_count = count1;
                    }
            }
            int getSTparamscount(char *s){
 for(int i = 0 ; i < 1001 ; i++){
if(strcmp(ST[i].name,s)==0 )     return
ST[i].params_count;
                    }
                    return -1;
            }
```

```
        void insertSTF(char *s){
    for(int i = 0 ; i < 1001 ; i++){
                        if(strcmp(ST[i].name,s)==0){
    strcpy(ST[i].class,"Function");
                            return;
                        }
                } }

        void insertCT(char *str1, char *str2){
                if(lookupCT(str1))
                        return;
                else{
                int value = hash(str1);
        if(CT[value].length == 0){
                strcpy(CT[value].name,str1);
                            strcpy(CT[value].type,str2);
        CT[value].length = strlen(str1);
                            return;
                        }
                        int pos = 0;
                        for (int i = value + 1 ; i!=value ; i = (i+1)%1001){
                            if(CT[i].length == 0){
                                pos = i;
                        break;
        }
                        }
                        strcpy(CT[pos].name,str1);
    strcpy(CT[pos].type,str2);
                        CT[pos].length = strlen(str1);
                }
        }
```

```
            int check_id_is_func(char *s){
    for(int i = 0 ; i < 1000 ; i++){
                            if(strcmp(ST[i].name,s)==0){
    if(strcmp(ST[i].class,"Function")==0)
                                        return 1;
                            }
                    }
                    return 0;
            }
            char gettype(char *s, int flag){ for(int
                    i = 0 ; i < 1001 ; i++ ){
                    if(strcmp(ST[i].name,s)==0)
            return ST[i].type[0];
                    }
            }
            void printST(){
                    printf("%10s | %15s | %10s | %10s | %10s | %15s |\n","SYMBOL", "CLASS",
    "TYPE","VALUE", "LINE NO", "PARAMS COUNT");
                    for(int i=0;i<87;i++)
            printf("-");            printf("\n");
                    for(int i = 0 ; i < 1001 ; i++){
                            if(ST[i].length == 0)
                                    continue;
                            printf("%10s | %15s | %10s | %10s | %10d | %15d |\n",ST[i].name,
    ST[i].class, ST[i].type, ST[i].value, ST[i].lineno, ST[i].params_count);
                    }
      }
    void printCT(){   printf("%10s | %15s\n","NAME",
    "TYPE");
                    for(int i=0;i<81;i++)
            printf("-");            printf("\n");
                    for(int i = 0 ; i < 1001 ; i++){
```

```
					if(CT[i].length == 0)
						continue;
					printf("%10s | %15s\n",CT[i].name, CT[i].type);
				}
		}
		char curid[20];
char curtype[20];		char
curval[20];

%}
DE "define"
IN "include"

%%
\n		{yylineno++;}
([#][" "]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|" "|"\t"] { }
([#][" "]*({DE})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"]					{ }
\/\/(.*)				{ }
\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/
					{ }
[ \n\t] ;
";"					{ return(';'); }
","					{ return(','); }
("{")				{ return('{'); }
("}")				{ return('}'); }
"("					{ return('('); }
")"					{ return(')'); }
("["|"<:")				{ return('['); }
("]"|":>")				{ return(']'); }
":"					{ return(':'); }
"."					{ return('.'); }
```

```
"char"                  { strcpy(curtype,yytext); insertST(yytext, "Keyword");return CHAR;}
"double"          { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return DOUBLE;}

"else"                  { insertST(yytext, "Keyword"); return ELSE;}

"float"                 { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return FLOAT;}

"while"                 { insertST(yytext, "Keyword"); return WHILE;}

"do"                    { insertST(yytext, "Keyword"); return DO;}

"for"                   { insertST(yytext, "Keyword"); return FOR;}

"if"                    { insertST(yytext, "Keyword"); return IF;}

"int"                   { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return INT;}
"return"                { insertST(yytext, "Keyword"); return RETURN;}

"void"   { strcpy(curtype,yytext);   insertST(yytext, "Keyword");  return VOID;} "break"  {
insertST(yytext, "Keyword");  return BREAK;}


"<="                    { return lessthan_assignment_operator; }

"<"                          { return lessthan_operator; }

">="                    { return greaterthan_assignment_operator; }

">"                          { return greaterthan_operator; }

"=="                    { return equality_operator; }

"!="                    { return inequality_operator; }

"&&"                    { return AND_operator; }

"||"                    { return OR_operator; }

"&"                          { return amp_operator; }

"!"                          { return exclamation_operator; }

"-"                          { return subtract_operator; }

"+"                          { return add_operator; }

"*"                          { return multiplication_operator; }

"/"                          { return division_operator; }

"%"                          { return modulo_operator; }

"\="                    { return assignment_operator;}
```

```
\"[^\n]*\"/[;|,|\)]                    {strcpy(curval,yytext); insertCT(yytext,"String Constant");
return string_constant;}

\'[A-Z|a-z]\'/[;|,|\)|:]   {strcpy(curval,yytext); insertCT(yytext,"Character Constant"); return
character_constant;}

[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[  {strcpy(curid,yytext); insertST(yytext, "Array Identifier");  return
array_identifier;}

[1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^]
        {strcpy(curval,yytext); insertCT(yytext, "Number Constant"); yylval = atoi(yytext);
return integer_constant;}

([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^]
        {strcpy(curval,yytext); insertCT(yytext, "Floating Constant"); return float_constant;}

[A-Za-z_][A-Za-z_0-9]* {strcpy(curid,yytext); insertST(curid,"Identifier"); return identifier;}


(.?) {

        if(yytext[0]=='#')                  printf("Error in Pre-Processor

directive at line no. %d\n",yylineno);              else if(yytext[0]=='/')

  printf("ERR_UNMATCHED_COMMENT at line no. %d\n",yylineno);   else

if(yytext[0]=='"')                    printf("ERR_INCOMPLETE_STRING at line no.

%d\n",yylineno);

        else                 printf("ERROR at

line no.

%d\n",yylineno);         printf("%s\n", yytext);

 return 0;

}

%%
```

YACC Code:

```
%{

        #include <stdio.h>

        #include <string.h>

        #include <stdlib.h>


        void yyerror(char*
s);         int yylex();        void

ins();   void insV();        int flag=0;

             extern char curid[20];

 extern char curtype[20];
extern char curval[20];
extern int currnest;       int
check_id_is_func(char *);           void

insertST(char*, char*);             void

insertSTnest(char*, int);           void
insertSTparamscount(char*, int);
int getSTparamscount(char*);
char currfunctype[100];           char
currfunc[100];              char

currfunccall[100];        void

insertSTF(char*);         char

gettype(char*,int);       char

getfirst(char*);            void

push(char *s);              void
codegen();        void codeassign();        char*

itoa(int num, char* str, int base);   void

reverse(char str[], int length);

             void

swap(char*,char*);       void label1();

void label2();

 void label3();   void label4();

void label5();   void label6();

void codegencon();       void

funcgen();        void

funcgenend();   void arggen();

void callgen();


             int params_count=0;   int

call_params_count=0;   int top =

0,count=0,ltop=0,lno=0;          char

temp[3] = "t";
```

```
%}

%nonassoc IF

%token INT CHAR FLOAT DOUBLE RETURN MAIN VOID WHILE FOR DO BREAK identifier
array_identifier func_identifier integer_constant string_constant float_constant
character_constant

%nonassoc ELSE

%right assignment_operator

%left OR_operator AND_operator amp_operator equality_operator inequality_operator
lessthan_assignment_operator lessthan_operator greaterthan_assignment_operator
greaterthan_operator leftshift_operator rightshift_operator add_operator subtract_operator
multiplication_operator division_operator modulo_operator

%right exclamation_operator

%left increment_operator decrement_operator

%start program

%%

program : declaration_list; declaration_list

: declaration D

D   : declaration_list

            | ;

declaration : variable_declaration

            | function_declaration

variable_declaration    : type_specifier variable_declaration_list ';'   variable_declaration_list

: variable_declaration_list ',' variable_declaration_identifier

| variable_declaration_identifier; variable_declaration_identifier : identifier

{insertSTnest(curid,currnest); ins();  } vdi      | array_identifier

{insertSTnest(curid,currnest); ins();  } vdi;       vdi : identifier_array_type |

assignment_operator simple_expression  ; identifier_array_type          : '[' initilization_params

            | ;

initilization_params      : integer_constant ']' initilization {if($$ < 1) {printf("Wrong array
size\n"); exit(0);} }

            | ']' string_initilization; initilization

: string_initilization

            | array_initialization
            | ;
type_specifier : INT | CHAR | FLOAT  | DOUBLE
            | VOID  ;
function_declaration     : function_declaration_type function_declaration_param_statement;
function_declaration_type         : type_specifier identifier '('  { strcpy(currfunctype, curtype);
strcpy(currfunc, curid); insertSTF(curid); ins(); };
function_declaration_param_statement             : {params_count=0;}params ')' {funcgen();}
statement {funcgenend();};  params : parameters_list {
insertSTparamscount(currfunc, params_count); }| { insertSTparamscount(currfunc,
params_count); }; parameters_list         : type_specifier parameters_identifier_list ;
parameters_identifier_list         : param_identifier
parameters_identifier_list_breakup; parameters_identifier_list_breakup           : ','
parameters_list
            | ;
param_identifier           : identifier { ins();insertSTnest(curid,1); params_count++; }
param_identifier_breakup; param_identifier_breakup  : '[' ']'
            | ;
statement         : expression_statment | compound_statement
            | conditional_statements | iterative_statements
            | return_statement | break_statement
  | variable_declaration;  compound_statement :
{currnest++;} '{'  statment_list  '}' ;
statment_list       :       statement       statment_list       |       ;
expression_statment : expression ';'  | ';' ;
conditional_statements              : IF '(' simple_expression ')' {label1();if($3!=1){printf("Condition
checking is not of type int\n");exit(0);}} statement {label2();}  conditional_statements_breakup;
conditional_statements_breakup           : ELSE statement {label3();}
            | {label3();};
iterative_statements   : WHILE '(' {label4();} simple_expression ')'
{label1();if($4!=1){printf("Condition checking is not of type int\n");exit(0);}} statement
{label5();}
            | FOR '(' expression ';' {label4();} simple_expression ';'
{label1();if($6!=1){printf("Condition checking is not of type int\n");exit(0);}} expression
')'statement {label5();}
```

```
            | {label4();}DO statement WHILE '(' simple_expression
')'{label1();label5();if($6!=1){printf("Condition checking is not of type int\n");exit(0);}} ';';

return_statement

 : RETURN ';' {if(strcmp(currfunctype,"void")) {printf("Returning void of a non-void
function\n"); exit(0);}}

  | RETURN expression ';' {   if(!strcmp(currfunctype, "void")){       yyerror("Function is void");

            }

   if((currfunctype[0]=='i' || currfunctype[0]=='c') && $2!=1){     printf("Expression

doesn't match return type of function\n"); exit(0);

            }

   };


break_statement           : BREAK ';' ; string_initilization             : assignment_operator

string_constant {insV();} ; array_initialization  : assignment_operator '{'

array_int_declarations '}'; array_int_declarations :

integer_constant array_int_declarations_breakup; array_int_declarations_breakup

: ',' array_int_declarations

            | ;

expression        : mutable assignment_operator {push("=");} expression   {

            if($1==1 && $4==1)

               $$=1;

else{

       $$=-1; printf("Type mismatch\n"); exit(0);}

                        codeassign();

            }

            | simple_expression {if($1 == 1) $$=1; else $$=-1;} ;


simple_expression : simple_expression OR_operator and_expression {push("||");} {if($1 == 1 &&
$3==1) $$=1; else $$=-1; codegen();}

            | and_expression {if($1 == 1) $$=1; else $$=-1;};


and_expression : and_expression AND_operator {push("&&");} unary_relation_expression  {if($1
== 1 && $3==1) $$=1; else $$=-1; codegen();}
```

```
            |unary_relation_expression {if($1 == 1) $$=1; else $$=-1;} ;


unary_relation_expression        : exclamation_operator {push("!");} unary_relation_expression
{if($2==1) $$=1; else $$=-1; codegen();}

            | regular_expression {if($1 == 1) $$=1; else $$=-1;} ;


regular_expression  : regular_expression relational_operators sum_expression {if($1 == 1 &&
$3==1) $$=1; else $$=-1; codegen();}

            | sum_expression {if($1 == 1) $$=1; else $$=-1;} ;


relational_operators   : greaterthan_assignment_operator {push(">=");} |
lessthan_assignment_operator {push("<=");} | greaterthan_operator {push(">");}|
lessthan_operator {push("<");}| equality_operator {push("==");}| inequality_operator
{push("!=");} ;


sum_expression            : sum_expression sum_operators term  {if($1 == 1 && $3==1) $$=1;
else $$=-1; codegen();}

            | term {if($1 == 1) $$=1; else $$=-1;};


sum_operators              : add_operator {push("+");}

            | subtract_operator {push("-");} ;


term     : term MULOP factor {if($1 == 1 && $3==1) $$=1; else $$=-1; codegen();}

| factor {if($1 == 1) $$=1; else $$=-1;} ;


MULOP             : multiplication_operator {push("*");}| division_operator {push("/");} |
modulo_operator {push("%");} ;

factor  : immutable {if($1 == 1) $$=1; else $$=-1;}

 | mutable {if($1 == 1) $$=1; else $$=-1;} ; mutable

 : identifier {

            push(curid);

if(check_id_is_func(curid))

              {printf("Function name used as Identifier\n"); exit(8);}

if(gettype(curid,0)=='i' || gettype(curid,1)== 'c')
```

```
                $$ = 1;
            else
                $$ = -1;
            }
            | array_identifier '[' expression ']'
            {if(gettype(curid,0)=='i' || gettype(curid,1)== 'c')
                $$ = 1;
            else
                $$ = -1;
            };

immutable       : '(' expression ')' {if($2==1) $$=1; else $$=-1;}
            | call {if($1==-1) $$=-1; else $$=1;}
            | constant {if($1==1) $$=1; else $$=-1;};

call    : identifier '('{
insertSTF(curid);
            strcpy(currfunccall,curid);
if(gettype(curid,0)=='i' || gettype(curid,1)== 'c')
                $$ = 1;
            else
                $$ = -1;
    call_params_count=0;
            }
            arguments ')'
            { if(strcmp(currfunccall,"printf"))
            {
            if(getSTparamscount(currfunccall)!=call_params_count){
yyerror("Number of arguments in function call doesn't match number of parameters");
                exit(8);
            }
            }
            callgen();
            };

arguments       : arguments_list | ;  arguments_list  :
arguments_list ',' exp { call_params_count++; }
            | exp { call_params_count++; };

exp : identifier {arggen(1);} | integer_constant {arggen(2);} | string_constant {arggen(3);} |
float_constant {arggen(4);} | character_constant {arggen(5);} ;

constant           : integer_constant        {  insV(); codegencon(); $$=1; }
            | string_constant          {  insV(); codegencon();$$=-1;}
            | float_constant {  insV(); codegencon();}
            | character_constant{  insV(); codegencon();$$=1; };

%%
extern FILE *yyin; extern int
yylineno; extern char *yytext;
void insertSTtype(char *,char *);
void insertSTvalue(char *, char *);
void incertCT(char *, char *); void
printST(); void printCT();  struct
stack{
            char value[100];
 int labelvalue;
}s[100],label[100];  void
push(char *x){
strcpy(s[++top].value,x);
}
void swap(char *x, char *y){
```

```
               char temp = *x;
 *x = *y;
               *y = temp;
}
void reverse(char str[], int length){      int
start = 0;      int end = length -1;      while
(start < end){
swap((str+start), (str+end));
        start++;
end--;
    }
}
char* itoa(int num, char* str, int base){
    int i = 0;       int
isNegative = 0;       if
(num == 0){
str[i++] = '0';
str[i] = '\0';         return
str;
    }
    if (num < 0 && base == 10){
        isNegative = 1;
num = -num;
    }
    while (num != 0){         int
rem = num % base;
str[i++] = (rem > 9)?
(rem-10) + 'a' : rem + '0';
num = num/base;
    }
    if (isNegative)
str[i++] = '-';     str[i] =
'\0';     reverse(str, i);
return str;
}
void codegen(){
strcpy(temp,"t");
 char buffer[100];   itoa(count,buffer,10);   strcat(temp,buffer);           printf("%s =
%s %s %s\n",temp,s[top-2].value,s[top-1].value,s[top].value);            top = top - 2;
 strcpy(s[top].value,temp);
            count++;
}
void codegencon(){
            strcpy(temp,"t");          char
buffer[100];   itoa(count,buffer,10);   strcat(temp,buffer);
printf("%s =
%s\n",temp,curval);
            push(temp);
 count++;
}
void    codeassign(){        printf("%s    =    %s\n",s[top-
2].value,s[top].value);  top = top - 2;
}
void label1(){
            strcpy(temp,"L");          char buffer[100];
 itoa(lno,buffer,10);      strcat(temp,buffer);   printf("IF not
%s GoTo %s\n",s[top].value,temp);
label[++ltop].labelvalue = lno++;
}
void label2(){
            strcpy(temp,"L");
 char buffer[100];   itoa(lno,buffer,10);
strcat(temp,buffer);   printf("GoTo
%s\n",temp);   strcpy(temp,"L");
itoa(label[ltop].labelvalue,buffer,10);
strcat(temp,buffer);
 printf("%s:\n",temp);   ltop--;
              label[++ltop].labelvalue=lno++;
}

void label3(){
              strcpy(temp,"L");
 char buffer[100];
itoa(label[ltop].labelvalue,buffer,10);
strcat(temp,buffer);
 printf("%s:\n",temp);
              ltop--;
}
void label4(){
              strcpy(temp,"L");
char buffer[100];
 itoa(lno,buffer,10);      strcat(temp,buffer);
printf("%s:\n",temp);
label[++ltop].labelvalue = lno++;
}
void label5(){
              strcpy(temp,"L");
 char buffer[100];
              itoa(label[ltop-
1].labelvalue,buffer,10);
strcat(temp,buffer);   printf("GoTo
%s:\n",temp);   strcpy(temp,"L");
itoa(label[ltop].labelvalue,buffer,10);
strcat(temp,buffer);
```

```
 printf("%s:\n",temp);   ltop = ltop
- 2;
}

void funcgen(){             printf("func begin
%s\n",currfunc);
}
void funcgenend(){
printf("func end\n\n");
}
void arggen(int i){     if(i==1)
printf("refparam %s\n", curid);
            else
               printf("refparam %s\n", curval);
}
void callgen(){
printf("refparam result\n");
 push("result");             printf("call %s,
%d\n",currfunccall,call_params_count);
}
int main(int argc , char **argv){           yyin =
fopen(argv[1], "r");      yyparse();        if(flag == 0){
printf("\nStatus: Parsing Complete - Valid\n");
printf("%30s SYMBOL
TABLE\n", " ");                         printf("%30s %s\n", " ", "-------
-----");
                        printST();

                        printf("\n\n%30s CONSTANT TABLE\n", " ");
printf("%30s %s\n", " ", "--------------");
                        printCT();
```

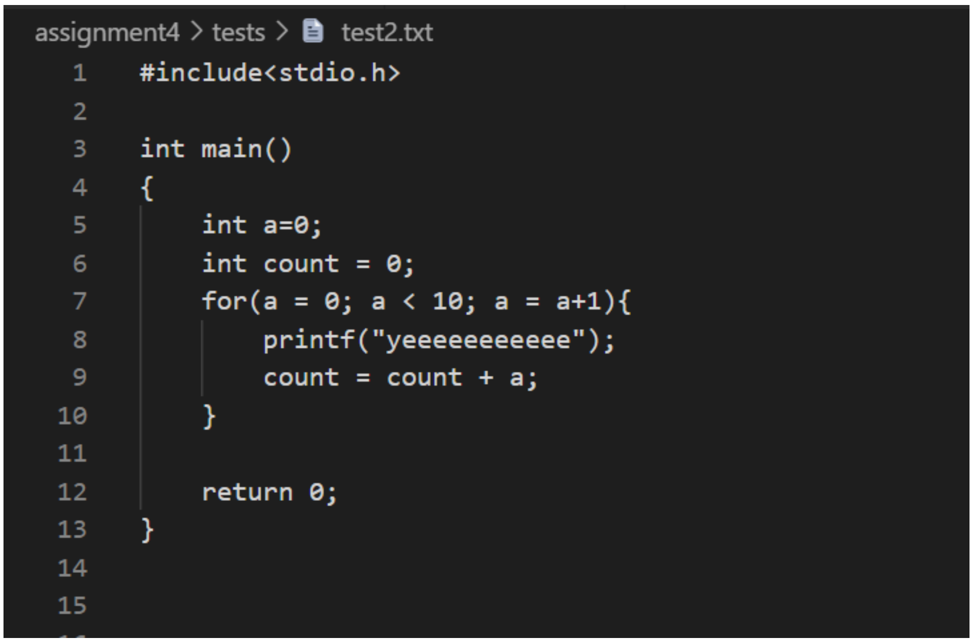
```
        }
}
void yyerror(char *s){          printf("%d %s
%s\n", yylineno, s, yytext);
        flag=1;
        printf("\nStatus: Parsing Failed - Invalid\n");
        exit(7);
}
void ins(){
        insertSTtype(curid,curtype);
}
void insV(){
insertSTvalue(curid,curval);
}
int yywrap(){
        return 1;
}
```

Explanation:

The final frontend phase of a compiler design stage is Intermediate Code Generation. A compiler's ultimate purpose is to get programmes written in a high-level language to run on a computer. This is because the programme will eventually have to be written as machine code that can execute on the computer. Many compilers utilise a medium-level language as a bridge between high-level and very low-level machine code. These stepping-stone languages are known as intermediate code. It offers reduced abstraction from the source level while retaining certain high level information. Depending on whether it is ByteCode for Java or three-address-code (language independent), intermediate code can be represented in a variety of forms.

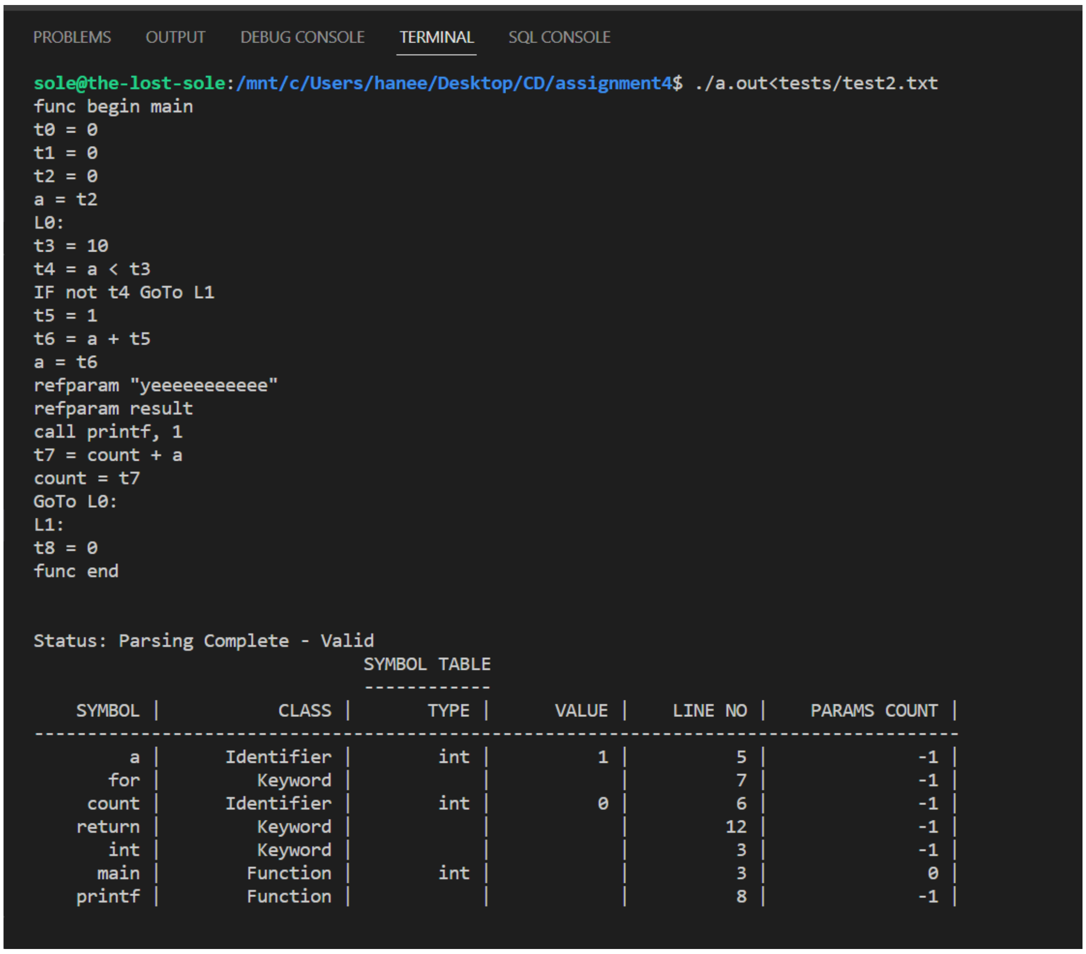
We used three-address-code here.

Input:



```c
#include<stdio.h>

int main()
{
    int a=0;
    int count = 0;
    for(a = 0; a < 10; a = a+1){
        printf("yeeeeeeeeeee");
        count = count + a;
    }

    return 0;
}


```

Output:



```
sole@the-lost-sole:/mnt/c/Users/hanee/Desktop/CD/assignment4$ ./a.out<tests/test2.txt
func begin main
t0 = 0
t1 = 0
t2 = 0
a = t2
L0:
t3 = 10
t4 = a < t3
IF not t4 GoTo L1
t5 = 1
t6 = a + t5
a = t6
refparam "yeeeeeeeeeee"
refparam result
call printf, 1
t7 = count + a
count = t7
GoTo L0:
L1:
t8 = 0
func end


Status: Parsing Complete - Valid
                               SYMBOL TABLE
                               ------------

    SYMBOL |           CLASS |        TYPE |        VALUE |      LINE NO |      PARAMS COUNT |
---------------------------------------------------------------------------------------------
         a |      Identifier |         int |            1 |            5 |                -1 |
       for |         Keyword |             |              |            7 |                -1 |
     count |      Identifier |         int |            0 |            6 |                -1 |
    return |         Keyword |             |              |           12 |                -1 |
       int |         Keyword |             |              |            3 |                -1 |
      main |        Function |         int |              |            3 |                 0 |
    printf |        Function |             |              |            8 |                -1 |
```