CS304- COMPILER DESIGN LAB

A REPORT ON THE PROJECT ENTITLED

SEMANTIC ANALYZER FOR THE C LANGUAGE

Group Members:

B Anagha	Gouri M R	P Devi Deepika
221CS114	221CS125	221CS138

V SEMESTER B-TECH CSE- S1

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL

2024 – 2025

Abstract

A compiler is a special program that processes statements written in a particular programming language (high-level language) and turns them into machine language (low-level language) that a computer's processors use. Apart from this, the compiler is also responsible for detecting and reporting any errors in the source program during the translation process.
The file used for writing code in a specific language contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code.
This report specifies the details related to the second stage of the compiler, the parsing stage. We have developed a parser for the C programming language using the lex and yacc tools. The parser makes use of the tokens outputted by the lexer developed in the previous stage to parse the C input file. The lexical analyzer can detect only lexical errors like unmatched comments etc. but cannot detect syntactical errors like missing semi-colon etc. These syntactical errors are identified by the parser i.e. the syntax analysis phase is done by the parser.

Introduction

Semantic Analysis

Semantic analysis is a crucial phase in the process of compiler construction, following lexical analysis and parsing. While lexical analysis deals with tokens and syntax analysis focuses on the structural arrangement of these tokens, semantic analysis ensures that the program's components interact meaningfully and according to the language's rules. This phase is essential for ensuring that the source code adheres to the semantic rules of the programming language, in this case, C.
Semantic Analyzer
It uses a syntax tree and symbol table to check whether the given program is semantically consistent with the language definition. It gathers and stores type information in either a syntax tree or a symbol table. This type of information is subsequently used by the compiler during intermediate code generation.
Purpose of Semantic Analysis
The primary purpose of semantic analysis is to verify the semantic correctness of a program after its syntactic structure has been verified. This phase examines whether a syntactically correct program makes logical sense within the context of the programming language.

This includes:

- **Type Checking:** Verify that an operator's operands are compatible and that the operation is allowed for those types.
- **Scope Resolution:** Determining the scope of variables and ensuring they are used correctly within their defined scopes.
- **Type Coercion:** Implicitly converting operands to a common type when necessary and allowed by the language.
- **Function and Procedure Calls:** Checking that function calls have the correct number and types of arguments.
- **Array Index Bounds:** Verifying that array indices are within the declared bounds (where possible at compile-time).
- **Semantic Consistency:** Ensuring that language constructs' overall program structure and usage are semantically valid.
- **Symbol Table Management:** Maintaining and updating the symbol table with information about identifiers, their types, and scopes.
- **Error Detection:** Identifying semantic errors that couldn't be caught during lexical or syntactic analysis.

Implementation in a C Compiler

In the context of a C compiler built using Lex and Yacc:
Lex (or Flex) is typically used for lexical analysis, breaking down the source code into tokens. Yacc (or Bison) is used for parsing, which builds the abstract syntax tree (AST) based on the grammar of C. Semantic analysis is then performed on this AST, often by traversing the tree and applying semantic checks at each node.

The semantic analyzer for a C compiler needs to handle various language-specific features, such as:

- C's type system, including structs, unions, and pointers
- Function prototypes and definitions
- Variable declarations and initializations
- Type qualifiers (const, volatile, etc.)
- Implicit type conversions
- Function calls and parameter matching
- Type conversions and promotions
- Array bounds and pointer operations
- Structure and union member access
- Consistency of declarations and definitions

By implementing thorough semantic analysis, the compiler can catch a wide range of errors that syntactically correct code might contain, improving the overall reliability and correctness of the compiled programs.

CODE

LEX FILE

```
%option yylineno
%{
 #include <stdio.h>
 #include <string.h>
 #include<stdlib.h>
 #include "p12.tab.h"

 struct ConstantTable{
   char constant_name[100];
   char constant_type[100];
   int exist;
```

```
}CT[1000];

struct SymbolTable{
  char symbol_name[100];
  char symbol_type[100];
  char array_dimensions[100];
  char class[100];
  char value[100];
  char parameters[100];
  int line_number;
  int exist;
  int define;
  int nested_val;
  int curr_nested_val;
  int params_count;
}ST[1000];

int current_nested_val = 0;
int params_count = 0;


unsigned long hash(unsigned char *str)
{
  unsigned long hash = 5381;
  int c;

  while (c = *str++)
    hash = ((hash << 5) + hash) + c;

  return hash;
}

int search_ConstantTable(char* str){
  unsigned long temp_val = hash(str);
  int val = temp_val%1000;

  if(CT[val].exist == 0){
    return 0;
  }

  else if(strcmp(CT[val].constant_name, str) == 0)
  {
    return 1;
  }
  else
  {
    for(int i = val+1 ; i!=val ; i = (i+1)%1000)
    {
      if(strcmp(CT[i].constant_name,str)==0)
      {
        return 1;
      }
    }
    return 0;
  }
}


int search_SymbolTable(char* str){
  unsigned long temp_val = hash(str);
  int val = temp_val%1000;

  if(ST[val].exist == 0){
    return 0;
  }

  else if(strcmp(ST[val].symbol_name, str) == 0)
  {
    return val;
  }
  else
  {
    for(int i = val+1 ; i!=val ; i = (i+1)%1000)
    {
      if(strcmp(ST[i].symbol_name,str)==0)
      {
        return i;
      }
    }
    return 0;
  }
}
```

```
void insert_ConstantTable(char* name, char* type){
  int index = 0;
   if(search_ConstantTable(name)){
    return;
  }
  else{
    unsigned long temp_val = hash(name);
    int val = temp_val%1000;
    if(CT[val].exist == 0){
      strcpy(CT[val].constant_name, name);
      strcpy(CT[val].constant_type, type);
      CT[val].exist = 1;
      return;
    }

    for(int i = val+1; i != val; i = (i+1)%1000){
      if(CT[i].exist == 0){
        index = i;
        break;
      }
    }
    strcpy(CT[index].constant_name, name);
    strcpy(CT[index].constant_type, type);
    CT[index].exist = 1;
  }
}

void insert_SymbolTable(char* name, char* class){
  int index = 0;
   if(search_SymbolTable(name)){
    return;
  }
  else{
    unsigned long temp_val = hash(name);
    int val = temp_val%1000;
    if(ST[val].exist == 0){
      strcpy(ST[val].symbol_name, name);
      strcpy(ST[val].class, class);
        ST[val].nested_val = 100;
        ST[val].curr_nested_val = current_nested_val;
       //ST[val].params_count = -1;
             ST[val].line_number = yylineno;
       ST[val].exist = 1;
       return;
      }

      for(int i = val+1; i != val; i = (i+1)%1000){
       if(ST[i].exist == 0){
        index = i;
        break;
       }
      }

      strcpy(ST[index].symbol_name, name);
      strcpy(ST[val].class, class);
      ST[index].nested_val = 100;
ST[index].curr_nested_val = current_nested_val;
      //ST[index].params_count = -1;
      ST[index].exist = 1;
            if(strcmp("Function", class) == 0)ST[index].define = 1;
    }
  }

  void insert_SymbolTable_type(char *str1, char *str2)
  {
    for(int i = 0 ; i < 1000 ; i++)
    {
      if(strcmp(ST[i].symbol_name,str1)==0)
      {
        strcpy(ST[i].symbol_type,str2);
      }
    }
    }

  void insert_SymbolTable_value(char *str1, char *str2)
  {
    for(int i = 0 ; i < 1001 ; i++)
    {
      if(strcmp(ST[i].symbol_name,str1)==0 && ST[i].nested_val != current_nested_val)
      {
```

```
      strcpy(ST[i].value,str2);
    }
  }
}

  void insert_SymbolTable_arraydim(char *str1, char *dim)
{
  for(int i = 0 ; i < 1000 ; i++)
  {
    if(strcmp(ST[i].symbol_name,str1)==0)
    {
      strcpy(ST[i].array_dimensions,dim);
    }
  }
}

  void insert_SymbolTable_funcparam(char *str1, char *param)
{
  for(int i = 0 ; i < 1000 ; i++)
  {
    if(strcmp(ST[i].symbol_name,str1)==0)
    {
      strcat(ST[i].parameters," ");
              strcat(ST[i].parameters,param);
    }
  }
}

void insert_SymbolTable_line(char *str1, int line)
{
  for(int i = 0 ; i < 1000 ; i++)
  {
    if(strcmp(ST[i].symbol_name,str1)==0)
    {
      ST[i].line_number = yylineno;
    }
  }
}

void insert_SymbolTable_nest(char *s, int nest)
{
  if(search_SymbolTable(s) && ST[search_SymbolTable(s)].nested_val != 100)
  {
         int pos = 0;
         int value = hash(s);
    value = value%1001;
   for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
   {
    if(ST[i].exist == 0)
    {
      pos = i;
      break;
    }
   }

   strcpy(ST[pos].symbol_name,s);
   strcpy(ST[pos].class,"Identifier");
   ST[pos].nested_val = nest;
   ST[pos].curr_nested_val = current_nested_val;
   ST[pos].line_number = yylineno;
   ST[pos].exist = 1;
  }
  else
  {
   for(int i = 0 ; i < 1001 ; i++)
   {
    if(strcmp(ST[i].symbol_name,s)==0 )
    {
      ST[i].nested_val = nest;
      ST[i].curr_nested_val = current_nested_val;
    }
   }
  }
}

int check_scope(char *s)
{
  int flag = 0;
  for(int i = 0 ; i < 1000 ; i++)
  {
   if(strcmp(ST[i].symbol_name,s)==0)
   {
    if(ST[i].nested_val > current_nested_val)
    {
      flag = 1;
    }
    else
    {
      flag = 0;
      break;
    }
   }
  }
  if(!flag)
  {
   return 1;
  }
  else
  {
   return 0;
  }
}

void remove_scope (int nesting)
{
  for(int i = 0 ; i < 1000 ; i++)
  {
   if(ST[i].nested_val == nesting)
   {
    ST[i].nested_val = 100;
    ST[i].curr_nested_val = current_nested_val;
   }
  }
}

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

}

void updateDefine(char *s)
          {
            for(int i = 0 ; i < 1001 ; i++)
            {
              if(strcmp(ST[i].symbol_name,s)==0 )
              {

                    ST[i].define=1;
                return;
              }
            }

          }

void insert_SymbolTable_function2(char *s)
          {
            for(int i = 0 ; i < 1001 ; i++)
            {
              if(strcmp(ST[i].symbol_name,s)==0 )
              {
                strcpy(ST[i].class,"Function");
                    ST[i].define=1;
                return;
              }
            }

          }
          int isNotPointer(char *s)
                    {

                      for(int i = 0 ; i < 1001 ; i++)
                      {
                        if(strcmp(ST[i].symbol_name,s)==0 )
                        {
                              int j=0;
                        for(;ST[i].symbol_type[j+1]!='\0';j++);


                            char ch=ST[i].symbol_type[j];
                                if(ch=='*') return 0;

                            return 1;
                        }
                      }

                    }

int check_function(char *s)
{
  for(int i = 0 ; i < 1000 ; i++)
  {
    if(strcmp(ST[i].symbol_name,s)==0)
    {
      if(strcmp(ST[i].class,"Function")==0)
        return 1;
    }
  }
  return 0;
}

int check_array(char *s)
{
  for(int i = 0 ; i < 1000 ; i++)
  {
    if(strcmp(ST[i].symbol_name,s)==0)
    {
      if(strcmp(ST[i].class,"Array Identifier")==0)
      {
        return 0;
      }
    }
  }
  return 1;
}

int duplicate(char *s)
```

```
{
  for(int i = 0 ; i < 1000 ; i++)
  {
    if(strcmp(ST[i].symbol_name,s)==0)
    {
      if(ST[i].nested_val == current_nested_val)
      {
            return 1;
      }
    }
  }

  return 0;
}

int check_duplicate(char* str)
{
  for(int i=0; i<1000; i++)
  {
    if(strcmp(ST[i].symbol_name, str) == 0 && strcmp(ST[i].class, "Function") == 0)
    {
      if(ST[i].define==0){
      printf("ERROR: Cannot Redeclare same function!\n");
      printf("\nUNSUCCESSFUL: INVALID PARSE\n");
      exit(0);}
          else ST[i].define=1;
    }
  }
}

int check_declaration(char* str, char *check_type)
{
  for(int i=0; i<1000; i++)
  {
    if(strcmp(ST[i].symbol_name, str) == 0 && strcmp(ST[i].class, "Function") == 0
&&ST[i].define==1|| strcmp(ST[i].symbol_name,"printf")==0 )
    {
      return 1;
    }
  }
  return 0;
```

```
}

int check_params(char* type_specifier)
{
  if(!strcmp(type_specifier, "void"))
  {
    printf("ERROR: Here, Parameter cannot be of void type\n");
    printf("\nUNSUCCESSFUL: INVALID PARSE\n");
    exit(0);
  }
  return 0;
}

void insert_SymbolTable_paramscount(char *s, int count)
{
  for(int i = 0 ; i < 1000 ; i++)
  {
    if(strcmp(ST[i].symbol_name,s)==0 )
    {
      ST[i].params_count = count;
    }
  }
}

int getSTparamscount(char *s)
{
  for(int i = 0 ; i < 1000 ; i++)
  {
    if(strcmp(ST[i].symbol_name,s)==0 )
    {
      return ST[i].params_count;
    }
  }
  return -2;
}
int getSTdimensioncount(char *s) {
  for (int i = 0; i < 1000; i++) {
      if (strcmp(ST[i].symbol_name, s) == 0) {
          return atoi(ST[i].array_dimensions);
      }
  }
```

```
   return -2;
  }
 char gettype(char *s, int flag)
 {
    for(int i = 0 ; i < 1001 ; i++ )
    {
     if(strcmp(ST[i].symbol_name,s)==0)
     {
       return ST[i].symbol_type[0];
     }
    }

 }

 void printConstantTable(){
   printf("%20s | %20s\n", "CONSTANT","TYPE");
   for(int i = 0; i < 1000; ++i){
     if(CT[i].exist == 0)
       continue;

     printf("%20s | %20s\n", CT[i].constant_name, CT[i].constant_type);
   }
 }

 void printSymbolTable(){
 printf("%10s | %18s | %10s | %10s | %10s | %10s | %10s | %10s | %10s\n","SYMBOL",
"CLASS", "TYPE","VALUE","DIMENSIONS","PARAMETERS","PARAMETER COUNT",
"NESTING", "LINE NO");
   for(int i = 0; i < 1000; ++i){
     if(ST[i].exist == 0)
       continue;
     printf("%10s | %18s | %10s | %10s | %10s | %10s | %15d | %10d | %d\n",
ST[i].symbol_name, ST[i].class, ST[i].symbol_type,
ST[i].value,ST[i].array_dimensions,ST[i].parameters, ST[i].params_count,
ST[i].curr_nested_val,ST[i].line_number);
   }
 }
 char current_identifier[20];
 char current_type[20];
 char current_value[20];
   char current_function[20];
  char previous_operator[20];
  int flag;

%}

num              [0-9]
alpha                    [a-zA-Z]
alphanum          {alpha}|{num}
escape_sequences    0|a|b|f|n|r|t|v|"\\"|"\""|"\'"
ws                                  [ \t\r\f\v]+
%x MLCOMMENT
DE "define"
IN "include"

%%

  int nested_count = 0;
  int check_nested = 0;

\n                                                                  { }
"#include"[ ]*"<"{alpha}({alphanum})*".h>"                          { }
"#define"[ ]+(_|{alpha})({alphanum})*[ ]*(.)+                       { }
"//".*                                                              { }
"/*"                                                        { BEGIN MLCOMMENT; }
<MLCOMMENT>"/*"                                             { ++nested_count;
                                                              check_nested = 1;
                                                                     }
<MLCOMMENT>"*"+"/"                                  { if (nested_count) --nested_count;
                                                     else{ if(check_nested){
                                                      check_nested = 0;
                                                              BEGIN INITIAL;
                                                               }
                                                               else{
                                                                       BEGIN INITIAL;
                                                                                }
                                                                           }
                                                                      }

<MLCOMMENT>"*"+                                                                 ;

<MLCOMMENT>[^/*\n]+                                                              ;
```

```
<MLCOMMENT>[/]                                                          ;

<MLCOMMENT>\n                                                           ;

<MLCOMMENT><<EOF>>                                                      { printf("Line No.
%d ERROR: MULTI LINE COMMENT NOT CLOSED\n", yylineno); return 0;}

"["                         {return *yytext;}
"]"                         {return *yytext;}
"("                         {return *yytext;}
")"                         {return *yytext;}
"{"                         {return *yytext;}
"}"                         {return *yytext;}
","                         {return *yytext;}
";"                         {return *yytext;}



"char"                  { strcpy(current_type,yytext); insert_SymbolTable(yytext,
"Keyword"); return CHAR;}

"double"                { strcpy(current_type,yytext); insert_SymbolTable(yytext,
"Keyword"); return DOUBLE;}

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

"float"                 { strcpy(current_type,yytext); insert_SymbolTable(yytext,
"Keyword");return FLOAT;}

"NULL"                  {return NULL_CONST;}
"while"                 { insert_SymbolTable(yytext, "Keyword"); return WHILE;}
"do"                    { insert_SymbolTable(yytext, "Keyword"); return DO;}
"for"                   { insert_SymbolTable(yytext, "Keyword"); return FOR;}
"if"                    { insert_SymbolTable(yytext, "Keyword"); return IF;}
"int"                   { strcpy(current_type,yytext); insert_SymbolTable(yytext,
"Keyword");return INT;}
"long"                  { strcpy(current_type,yytext); insert_SymbolTable(yytext,
"Keyword");  return LONG;}
"return"                { insert_SymbolTable(yytext, "Keyword");  return RETURN;}
```

```
"short"                 { strcpy(current_type,yytext); insert_SymbolTable(yytext,
"Keyword");  return SHORT;}
"signed"                { strcpy(current_type,yytext); insert_SymbolTable(yytext,
"Keyword");  return SIGNED;}
"sizeof"                { insert_SymbolTable(yytext, "Keyword");  return SIZEOF;}
"struct"                { strcpy(current_type,yytext); insert_SymbolTable(yytext,
"Keyword");  return STRUCT;}
"unsigned"              { insert_SymbolTable(yytext, "Keyword");  return UNSIGNED;}
"void"                  { strcpy(current_type,yytext); insert_SymbolTable(yytext,
"Keyword");  return VOID;}

"break"                 { insert_SymbolTable(yytext, "Keyword");  return BREAK;}
"continue"              { insert_SymbolTable(yytext, "Keyword");  return CONTINUE;}
"goto"                  { insert_SymbolTable(yytext, "Keyword");  return GOTO;}
"switch"                { insert_SymbolTable(yytext, "Keyword");  return SWITCH;}
"case"                  { insert_SymbolTable(yytext, "Keyword");  return CASE;}
"default"               { insert_SymbolTable(yytext, "Keyword");  return DEFAULT;}

("\"")[^\n\"]*("\"")                    {strcpy(current_value,yytext);
insert_ConstantTable(yytext,"String Constant"); return string_constant;}

("\"")[^\n\"]*                {printf("Line No. %d ERROR: UNCLOSED STRING - %s\n",
yylineno, yytext); return 0;}

("\'")(("\\"({escape_sequences}))|.)("\'")   {strcpy(current_value,yytext);
insert_ConstantTable(yytext,"Character Constant"); return character_constant;}

("\'")(((("\\")[^0abfnrtv\\\"\'][^\n\']*))|[^\n\"][^\n\"]+)("\'")  {printf("Line No. %d
ERROR: NOT A CHARACTER - %s\n", yylineno, yytext); return 0; }

{num}+(\.{num}+)?e{num}+           {strcpy(current_value,yytext);
insert_ConstantTable(yytext, "Floating Constant"); return float_constant;}

{num}+\.{num}+                      {strcpy(current_value,yytext);
insert_ConstantTable(yytext, "Floating Constant"); return float_constant;}

{num}+                              {strcpy(current_value,yytext);
insert_ConstantTable(yytext, "Number Constant"); yylval = atoi(yytext); return
integer_constant;}
```

```
(_|{alpha})({alpha}|{num}|_)*
{strcpy(current_identifier,yytext);insert_SymbolTable(yytext,"Identifier");  return
identifier;}

(_|{alpha})({alpha}|{alpha}|_)*/\[
{strcpy(current_identifier,yytext);insert_SymbolTable(yytext,"Array Identifier");  return
array_identifier;}

{ws}                                                    ;
"+"                                                     {return *yytext;}
"-"                                                     {return *yytext;}
"*"                                                    {return *yytext;}
"/"                                                     {return *yytext;}
"="                                                     {return *yytext;}
"%"                                                     {return *yytext;}
"&"                                                     {return *yytext; }
"^"                                                     {return *yytext; }
"++"                                                    {return INCREMENT;}
"--"                                                    {return DECREMENT;}
"!"                                                     {return NOT;}
"+="                                                    {return ADD_EQUAL;}
"-="                                                    {return SUBTRACT_EQUAL;}
"*="                                                    {return MULTIPLY_EQUAL;}
"/="                                                    {return DIVIDE_EQUAL;}
"%="                                                   {return MOD_EQUAL;}
"&&"                                                   {return AND_AND;}
"||"                                                   {return OR_OR;}
">"                                                    {return GREAT;}
"<"                                     {return LESS;}
">="                                     {return GREAT_EQUAL;}
"<="                                      {return LESS_EQUAL;}
"=="                                    {return EQUAL;}
"!="                                    {return NOT_EQUAL;}
.        { flag = 1;
           if(yytext[0] == '#')
            printf("Line No. %d PREPROCESSOR ERROR - %s\n", yylineno, yytext);
             else
             printf("Line No. %d ERROR: ILLEGAL CHARACTER - %s\n", yylineno, yytext);
              return 0;}

%%
```

PARSE FILE

```
%{
   void yyerror(char *s);
   int yylex();
   #include "stdio.h"
   #include "stdlib.h"
   #include "ctype.h"
   #include "string.h"
  #include "semantic.h"
   void insert_type();
   void insert_value();
   void insert_dimensions();
   void insert_parameters();
   void remove_scope(int);
   int check_scope(char *);
   int check_function(char *);
   void insert_SymbolTable_nest(char *, int);
   void insert_SymbolTable_paramscount(char *, int);
   int isNotPointer(char *);
   int getSTparamscount(char *);
   int getSTdimensioncount(char *);
   int check_duplicate(char *);
   int check_declaration(char *, char *);
   int check_params(char *);
    void updateDefine(char *);
   int duplicate(char *s);
   int check_array(char *);
   void insert_SymbolTable_function(char *);
   char gettype(char *, int);
   extern int flag;
   int insert_flag = 0;
   extern char current_identifier[20];
   extern char current_type[20];
   extern char current_value[20];
   extern char current_function[20];
   extern char previous_operator[20];
   extern int current_nested_val;
   extern int switch_index;
```

```
    extern char switch_array[20][20];
    char currfunctype[100];
    char currfunccall[100];
    extern int params_count;
    int call_params_count;
    int current_array_dimensions;
    int actual_dimensions;
    char* getVariableType(char* identifier);
    int checkTypeCompatibility(char* expectedType, char* actualType);
    char* getParameterType(char* funcName, int paramIndex);
    char* getArgumentType(int argIndex);
    char current_call_argument_types[10][20];
    int current_exp_type = -1;

    int switch_index=0;
char switch_array[20][20];
extern FILE *yyin;
extern int yylineno;
extern char *yytext;
void insert_SymbolTable_type(char *,char *);
void insert_SymbolTable_value(char *, char *);
void insert_ConstantTable(char *, char *);
void insert_SymbolTable_arraydim(char *, char *);
void insert_SymbolTable_funcparam(char *, char *);
void printSymbolTable();
void printConstantTable();
int newCheckCompatibility(int a, int b);

struct SymbolTable{
    char symbol_name[100];
    char symbol_type[100];
        char array_dimensions[100];
    char class[100];
        char value[100];
        char parameters[100];
        int line_number;
    int exist;
    int define;
    int nested_val;
    int curr_nest_val;
    int params_count;
```

```
 };

 extern struct SymbolTable ST[1000];

 int newGetType(char* s);
 int current_exp_type2 = -1;
 char* identifier2;

%}

%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
STRUCT_ASSIGN
%token RETURN MAIN VOID WHILE FOR DO BREAK CONTINUE GOTO ENDIF
NULL_CONST SWITCH CASE DEFAULT
%token identifier array_identifier integer_constant string_constant float_constant
character_constant
%nonassoc ELSE

%right MOD_EQUAL
%right MULTIPLY_EQUAL DIVIDE_EQUAL
%right ADD_EQUAL SUBTRACT_EQUAL
%right '='

%left OR_OR
%left AND_AND
%left '^'
%left EQUAL NOT_EQUAL
%left LESS_EQUAL LESS GREAT_EQUAL GREAT
%left '+' '-'
%left '*' '/' '%'

%right SIZEOF
%right NOT
%left INCREMENT DECREMENT

%start begin_parse

%%

begin_parse
```

```
    : declarations
    ;

declarations
    : declaration declarations
    |
    ;

declaration
    : variable_dec
    | function_dec
    | structure_dec
    | fun_decl
    ;

structure_dec
    : STRUCT identifier { insert_type(); } '{' structure_content '}' struct_name ';'
    ;

structure_content
    : variable_dec structure_content
    | structure_dec structure_content
    |
    ;

struct_name
    : variables
    |
    ;

variable_dec
    : structure_initialize ';'
    | datatype variables ';'
    ;

structure_initialize
    : STRUCT struct_init_identifier struct_star variables
    ;

struct_init_identifier
```

```
    : identifier { if (gettype(current_identifier, 0) != 's') {   yyerror("Identifier undeclared
or out of scope\n"); }}
    ;

struct_star
    : '*' struct_star
    |
    ;

variables
    : identifier_name multiple_variables
    ;

multiple_variables
    : ',' variables
    |
    ;

identifier_name
    : identifier {
        identifier2 = strdup(current_identifier);
        if (duplicate(current_identifier)) { yyerror("Duplicate value!\n");  }
        insert_SymbolTable_nest(current_identifier, current_nested_val);
        insert_type();
    } extended_identifier { if (($3<= 5) &&
(newCheckCompatibility(newGetType(identifier2), $3) != 1)) {yyerror("ERROR: Type
Mismatch\n"); exit(0);}}
    | array_identifier {
        if (duplicate(current_identifier)) { yyerror("Duplicate value!\n");  }
        insert_SymbolTable_nest(current_identifier, current_nested_val);
        insert_type();
    } extended_identifier
    ;

extended_identifier
    : array_iden {$$ = 100;}
    | '=' NULL_CONST {if(isNotPointer(current_identifier) )yyerror("NULL ERROR!!!!"); $$
= 6;}
    | '=' { strcpy(previous_operator, "="); } condition_ternary{ if($3 <1) {yyerror("Invalid
expression.\n");} $$ = $3;}
    ;
```

```
array_iden
    : '[' array_dims
    |
    ;

array_dims
    : integer_constant { insert_dimensions(); } ']' initialization{if($$ < 1) {yyerror("Array
must have size greater than 1!\n");}}
    | ']' string_initialization
    | float_constant { insert_dimensions(); } ']' initialization{yyerror("Array size must be
an integer!\n");}
    | string_constant { insert_dimensions(); } ']' initialization{yyerror("Array size must be
an integer!\n");}
    | expression { insert_dimensions(); } ']' initialization{if($1 != 1 && $1 != 2)
{yyerror("Array size must be an integer!\n");}else if($$ < 1) {yyerror("Array must have
size greater than 1!\n");} current_exp_type = -1;}
    ;

initialization
    : string_initialization
    | array_initialization
    |
    ;

string_initialization
    : '=' { strcpy(previous_operator, "="); } string_constant { insert_value(); }
    ;

array_initialization
    : '=' { strcpy(previous_operator, "="); } multi_dim
    ;

multi_dim
    : '{' arr_elements '}'
    ;

arr_elements
    : array_values
    | '{' array_values '}'
    | multi_dim
    | multi_dim ',' '{' arr_elements '}'
    | '{' array_values '}' ',' arr_elements
    ;

array_values
    : constant multiple_array_values
    ;

multiple_array_values
    : ',' array_values
    |
    ;

datatype
    : INT | CHAR | FLOAT | DOUBLE
    | LONG long_grammar
    | SHORT short_grammar
    | UNSIGNED unsigned_grammar
    | SIGNED signed_grammar
    | VOID
    | datatype star
    ;

star
    : star '*' { strcat(current_type, "*"); }
    | '*' { strcat(current_type, "*"); }
    ;

unsigned_grammar
    : INT | LONG long_grammar | SHORT short_grammar |
    ;

signed_grammar
    : INT | LONG long_grammar | SHORT short_grammar |
    ;

long_grammar
    : INT
    |
    ;
```

```
short_grammar
   : INT
   |
   ;

function_dec
   : function_datatype function_parameters
   ;

fun_decl : ';';


function_datatype
   : datatype identifier '(' {
      strcpy(currfunctype, current_type);
      check_duplicate(current_identifier);
      insert_SymbolTable_function(current_identifier);
      strcpy(current_function, current_identifier);
      insert_type();
   }
   ;

func_param : func_param ',' datatype | datatype ;


function_parameters
   : parameters ')' { params_count = 0; } body
   ;
body : {updateDefine(current_function);}statements {updateDefine(current_function);
}
| ';' ;
parameters
   : datatype { check_params(current_type); } all_parameter_identifiers {
insert_SymbolTable_paramscount(current_function, params_count); }
   |
   ;

all_parameter_identifiers
   : parameter_identifier multiple_parameters
   ;
```

```
multiple_parameters
    : ',' parameters
    |
    ;

parameter_identifier
    : identifier {
        insert_parameters();
        insert_type();
        insert_SymbolTable_nest(current_identifier, 1);
        params_count++;
    } extended_parameter
    ;

extended_parameter
    : '[' ']'
    |
    ;

statement
    : expression_statement {$$ = $1;}
    | multiple_statement
    | conditional_statements
    | iterative_statements
    | return_statement
    | break_statement
    | switch_statement
    | variable_dec
    | function_dec
    ;

switch_statement
    : SWITCH '(' simple_expression ')' { if ($3 != 1) { yyerror("ERROR: Condition must have
integer value!\n"); exit(0); }}
      '{' case_statement DEFAULT ':' statement '}'
    ;

case_statement
    : CASE int_char_const {
        if (switch_check(current_value, switch_array, switch_index)) {
            yyerror("ERROR: Duplicate case value");
        } else {
            switch_insert(current_value, switch_array, &switch_index);
        }
    } ':' case_body case_break case_statement
    |
    ;

case_body
    : statement {$$ = $1;}
    |
    ;

case_break
    : break_statement
    |
    ;

int_char_const
    : integer_constant {$$ = 1;}
    | character_constant {$$ = 2;}
    ;

multiple_statement
    : { current_nested_val++; } '{' statements '}' { remove_scope(current_nested_val);
current_nested_val--; }
    ;

statements
    : statement statements {$$ = $2>$2? $1 : $2;}
    |
    ;

expression_statement
    : expression ';'{if($1 == -1) {yyerror("Cannot perform this operation\n."); $$ = -1;} else
{$$ = 1;}current_exp_type = -1;}
    | expression {current_exp_type = -1;} ',' expression_statement {$$ = $1>$4? $1 : $4;}
    | ';'
    ;

conditional_statements
```

```
    : IF '(' simple_expression ')' {if ($3 != 1) { yyerror("ERROR: Condition must have integer
value!\n"); $$ = -1;} else $$ = 1;} statement extended_conditional_statements
    | condition_ternary {$$ = $1;}
    ;

extended_conditional_statements
    : ELSE statement {$$ = $2;}
    |
    ;

condition_ternary
    : '(' expression {current_exp_type = -1;} ')' '?' condition_ternary ':' condition_ternary
{if($2>0 && $2<4 && $6 > 0 && $6<4 && $8>0 && $8<4) $$ = 1; else $$ = -1;}
    | '(' conditional_statements ')' {$$ = $2;}
    | expression { $$ = $1; current_exp_type = -1;}
    ;

iterative_statements
    : WHILE '(' simple_expression ')' { if ($3 < 1 || $3 > 3) { yyerror("ERROR: Condition must
have integer value!\n");  } $$ = 1;} statement
    | FOR '(' for_initialization simple_expression ';' {if($4<1 || $4 > 3){yyerror("Here,
condition must have integer value!\n");}} expression {current_exp_type = -1; $$ = 1;} ')'
    | DO statement WHILE '(' simple_expression ')' { if ($5 < 1 || $5>3) { yyerror("ERROR:
Condition must have integer value!\n");  } $$ = 1;} ';'
    ;

for_initialization
    : variable_dec
    | mutable ',' for_initialization
    | expression ';' {current_exp_type = -1;}
    | expression ','{current_exp_type = -1;} for_initialization
    | variable_dec ',' for_initialization
    | ';'
    ;

simple_expression_for
    : simple_expression
    | simple_expression ',' simple_expression_for
    | ';'
    ;
```

```
expression_for
   : expression {current_exp_type = -1;}
   | expression ','{current_exp_type = -1;} expression_for
   |
   ;

return_statement
   : RETURN ';' { if (strcmp(currfunctype, "void")) { yyerror("ERROR: Cannot have void
return for non-void function!\n"); }}
   | RETURN expression ';' { if (!strcmp(currfunctype, "void")) {
      yyerror("Non-void return for void function!"); current_exp_type = -1;
   }

   //printSymbolTable();
    if (((strcmp(currfunctype,"int") == 0|| strcmp(currfunctype,"char") == 0) && ($2 != 1)
&& ($2 != 2))) {
      yyerror("Expression doesn't match return type of function\n");
   }
   else if (strcmp(currfunctype,"char*") == 0 && ($2 != 0)) {
      yyerror("Expression doesn't match return type of function\n");
   }
   else if (strcmp(currfunctype,"int*") == 0 && ($2 != 4)) {
      yyerror("Expression doesn't match return type of function\n");
   }
   else if (strcmp(currfunctype,"float*") == 0 && ($2 != 5)) {
      yyerror("Expression doesn't match return type of function\n");
   }

   }
   ;

break_statement
   : BREAK ';'
   ;

expression
   :
   {}NULL_CONST {if(isNotPointer(current_identifier) )yyerror("NULL ERROR!!!!"); else
$$=6; }
  | expression '+' term {
      if (!($1 >= 1 && $1 <4 && $3 >= 1 && $3 <4)) {
            yyerror("Cannot perform arithmetic operations on strings/pointers");
            $$ = -1;
        } else {
            $$ = $1>$3? $1 : $3;
        }
    }
    |mutable '=' expression { strcpy(previous_operator, "=");
        if (($1 >= 1 && $1 <4 && $3 >= 1 && $3 <4) || ($1 == $3) || $3 == 6) { $$ = $1>$3? $1 :
$3; }
        else { $$ = -1; yyerror("Type Mismatch\n"); }
    }
    | mutable ADD_EQUAL expression { strcpy(previous_operator, "+=");
        if ($1 >= 1 && $1 <4 && $3 >= 1 && $3 <4) { $$ = $1>$3? $1 : $3; }
        else { $$ = -1; yyerror("Type Mismatch\n"); }
    }
    | mutable SUBTRACT_EQUAL expression { strcpy(previous_operator, "-=");
        if ($1 >= 1 && $1 <4 && $3 >= 1 && $3 <4) { $$ = $1>$3? $1 : $3; }
        else { $$ = -1; yyerror("Type Mismatch\n"); }
    }
    | mutable MULTIPLY_EQUAL expression { strcpy(previous_operator, "*=");
        if ($1 >= 1 && $1 <4 && $3 >= 1 && $3 <4) { $$ = $1>$3? $1 : $3; }
        else { $$ = -1; yyerror("Type Mismatch\n");  }
    }
    | mutable DIVIDE_EQUAL expression { strcpy(previous_operator, "/=");
        if ($1 >= 1 && $1 <4 && $3 >= 1 && $3 <4) { $$ = $1>$3? $1 : $3; }
        else { $$ = -1; yyerror("Type Mismatch\n");  }
    }
    | mutable MOD_EQUAL expression { strcpy(previous_operator, "%=");
        if ($1 >= 1 && $1 <4 && $3 >= 1 && $3 <4) { $$ = $1>$3? $1 : $3; }
        else { $$ = -1; yyerror("Type Mismatch\n"); }
    }
    | mutable INCREMENT { if ($1 >= 1) $$ = $1; else $$ = -1; }
    | mutable DECREMENT { if ($1 >= 1) $$ = $1; else $$ = -1; }
    | simple_expression { if ($1 >= 0) $$ = $1; else $$ = -1; }
    ;

simple_expression
    : simple_expression OR_OR and_expression { if ($1 >= 1 && $1 <4 && $3 >= 1 && $3
<4) {$$ = $1>$3? $1 : $3;} else {$$ = -1; }}
    | and_expression { if ($1 >= 0) {$$ = $1;} else {$$ = -1; }}
    ;
```

```
and_expression
    : and_expression AND_AND unary_relation_expression { if ($1 >= 0 &&  $3 >= 0) $$ =
$1>$3? $1 : $3; else $$ = -1; }
    | unary_relation_expression { if ($1 >= 0) $$ = $1; else $$ = -1; }
    ;

unary_relation_expression
    : NOT unary_relation_expression { if ($2 >= 0) $$ = $2; else $$ = -1; }
    | regular_expression { if ($1 >= 0) $$ = $1; else $$ = -1; }
    ;

regular_expression
    : regular_expression relational_operators sum_expression { if ($1 >= 0  && $3 >=0) $$
= $1>$3? $1 : $3; else $$ = -1; }
    | sum_expression {  $$ = $1;  }
    ;

relational_operators
    : GREAT_EQUAL { strcpy(previous_operator, ">="); }
    | LESS_EQUAL { strcpy(previous_operator, "<="); }
    | GREAT { strcpy(previous_operator, ">"); }
    | LESS { strcpy(previous_operator, "<"); }
    | EQUAL { strcpy(previous_operator, "=="); }
    | NOT_EQUAL { strcpy(previous_operator, "!="); }
    ;

sum_expression
    : sum_expression sum_operators term { if ($1 >= 1 && $3 >= 1) $$ = $1>$3? $1 : $3;
else $$ = -1; }
    | term {  if ($1 >= 0) $$ = $1; else $$ = -1; }
    ;

sum_operators
    : '+'
    | '-'
    | '='{yyerror("Line no. Error: Invalid lhs value\n"); exit(0);}
    ;

term
```

```
   : term MULOP factor {  if ($1 >= 1 && $1 <4 && $3 >= 1 && $3 <4) $$ = $1>$3? $1 : $3;
else $$ = -1; }
   | factor { if ($1 >= 0) $$ = $1; else $$ = -1; }
   ;

MULOP
   : '*' | '/' | '%'
   ;

factor
   : immutable {if ($1 >= 0) $$ = $1; else $$ = -1; }
   | mutable { if ($1 >= 0) $$ = $1; else $$ = -1; }
   ;

mutable
   : identifier {
      if (!check_scope(current_identifier)) {
         yyerror("Identifier undeclared or out of scope");

      }

       int typ = newGetType(current_identifier);


      $$ = typ;
      current_exp_type = current_exp_type<$$? $$: current_exp_type ;
   }
   | '&' identifier {
      if (!check_scope(current_identifier)) {   yyerror("Identifier undeclared or out of
scope\n");  }
      if (!check_array(current_identifier)) {   yyerror("Array Identifier has No
Subscript\n");  }
       int typ = newGetType(current_identifier);

      $$ = typ;
      current_exp_type = current_exp_type<$$? $$: current_exp_type ;
   }
   | identifier INCREMENT {
      if (!check_scope(current_identifier)) {   yyerror("Identifier undeclared or out of
scope\n");  }
```

```
        if (!check_array(current_identifier)) {   yyerror("Array Identifier has No
Subscript\n");  }
         int typ = newGetType(current_identifier);

        $$ = typ;
        current_exp_type = current_exp_type<$$? $$: current_exp_type ;
    }
    | identifier DECREMENT {
        if (!check_scope(current_identifier)) {   yyerror("Identifier undeclared or out of
scope\n");  }
        if (!check_array(current_identifier)) {  yyerror("Array Identifier has No
Subscript\n");  }
         int typ = newGetType(current_identifier);

        $$ = typ;
        current_exp_type = current_exp_type<$$? $$: current_exp_type ;
    }
    | INCREMENT identifier {
        if (!check_scope(current_identifier)) {  yyerror("Identifier undeclared or out of
scope\n");  }
        if (!check_array(current_identifier)) {  yyerror("Array Identifier has No
Subscript\n"); }
        int typ = newGetType(current_identifier);

        $$ = typ;
        current_exp_type = current_exp_type<$$? $$: current_exp_type ;
    }
    | DECREMENT identifier {
        if (!check_scope(current_identifier)) {  yyerror("Identifier undeclared or out of
scope\n");  }
        if (!check_array(current_identifier)) {  yyerror("Array Identifier has No
Subscript\n");  }
         int typ = newGetType(current_identifier);

        $$ = typ;
        current_exp_type = current_exp_type<$$? $$: current_exp_type ;
    }
    | '+' identifier {
        if (!check_scope(current_identifier)) {  yyerror("Identifier undeclared or out of
scope\n");  }
```

```
        if (!check_array(current_identifier)) {  yyerror("Array Identifier has No
Subscript\n"); }
         int typ = newGetType(current_identifier);

        $$ = typ;
        current_exp_type = current_exp_type<$$? $$: current_exp_type ;
    }
    | '-' identifier {
        if (!check_scope(current_identifier)) {  yyerror("Identifier undeclared or out of
scope\n");  }
        if (!check_array(current_identifier)) {  yyerror("Array Identifier has No
Subscript\n"); }
         int typ = newGetType(current_identifier);

        $$ = typ;
        current_exp_type = current_exp_type<$$? $$: current_exp_type ;
    }
    | array_identifier {
        actual_dimensions = getSTdimensioncount(current_identifier);
        current_array_dimensions = 0;
        if (!check_scope(current_identifier)) {  yyerror("Identifier undeclared or out of
scope\n"); }
    } '[' { current_array_dimensions++; } expression ']' extended_array_dimension {
         int typ = newGetType(current_identifier);

        $$ = typ;
        current_exp_type = current_exp_type<$$? $$: current_exp_type ;
    }
    | struct_identifier STRUCT_ASSIGN mutable {
        if (!check_scope(current_identifier)) {  yyerror("Identifier undeclared or out of
scope\n"); }
        if (!check_array(current_identifier)) {  yyerror("Array Identifier has No
Subscript\n");  }
         int typ = newGetType(current_identifier);

        $$ = typ;
        current_exp_type = current_exp_type<$$? $$: current_exp_type ;
    }
    ;

struct_identifier
```

```
    : identifier {
        if (!check_scope(current_identifier)) {  yyerror("Identifier undeclared or out of
scope\n");  }
        if (!check_array(current_identifier)) {  yyerror("Array Identifier has No
Subscript\n"); }
        if (gettype(current_identifier, 0) == 's') $$ = 0;
        else { yyerror("Identifier not of struct type\n");  $$ = -1; }
    }
    ;

extended_array_dimension
    : '[' { current_array_dimensions++; } expression ']' extended_array_dimension
    | { if (current_array_dimensions != actual_dimensions) { yyerror("Dimensions
mismatch of the array\n"); }}
    ;

immutable
    : '(' expression ')' { if ($2 >= 1) $$ = $2; else $$ = -1; }
    | call {$$ = $1;}
    | constant_noString { if ($1 >= 0) $$ = $1; else $$ = -1; }
    ;
constant_noString : integer_constant { insert_type(); $$ = 1; }
    | string_constant { insert_type(); $$ = 0;}
    | float_constant { insert_type(); $$ = 3; }
    | character_constant { insert_type(); $$ = 2; }
    ;

call
    : identifier '(' {
        strcpy(previous_operator, "(");
        if (!check_declaration(current_identifier, "Function")) {
            yyerror("Function not defined");
        }
        char tp = gettype(current_identifier,0);
        int typ = newGetType(current_identifier);

            if (strcmp(current_identifier, "printf") == 0)
                    typ = 1;

        $$ = typ;
        current_exp_type = current_exp_type<$$? $$: current_exp_type ;
        insert_SymbolTable_function(current_identifier);
        strcpy(currfunccall, current_identifier);
        call_params_count = 0;
        current_exp_type2 = $$;
    } arguments ')' {
        if (strcmp(currfunccall, "printf")) {
            if (getSTparamscount(currfunccall) != call_params_count) {
                yyerror("Number of parameters not same as number of arguments during
function call!");

            }

            for (int i = 0; i < call_params_count; i++) {
                char* expectedType = getParameterType(currfunccall, i);
                char* actualType = getArgumentType(i);

                if (!checkTypeCompatibility(expectedType, actualType)) {
                    char error_message[100];
                    sprintf(error_message, "Type mismatch in function call: expected %s but
got %s for argument %d",
                            expectedType, actualType, i+1);
                    yyerror(error_message);

                }
            }
        }
        call_params_count = 0;
        $$ = current_exp_type2;

    }
    ;

arguments
    : arguments_list
    |
    ;

arguments_list
    : expression {
        strcpy(current_call_argument_types[call_params_count],
getVariableType(current_identifier));
        call_params_count++;
    } A
    ;

A
    : ',' expression {
        strcpy(current_call_argument_types[call_params_count],
getVariableType(current_identifier));
        call_params_count++;
    } A
    |
    ;

constant
    : integer_constant { insert_type(); $$ = 1; }
    | string_constant { insert_type(); $$ = 0; }
    | float_constant { insert_type(); $$ = 3; }
    | character_constant { insert_type(); $$ = 2; }
    ;

%%


void yyerror(char *s) {
    printf("Line No. : %d %s %s\n", yylineno, s, yytext);
    flag = 1;
    printf("\nUNSUCCESSFUL: INVALID PARSE\n");
}

void insert_type() {
    insert_SymbolTable_type(current_identifier, current_type);
}

void insert_value() {
    if (strcmp(previous_operator, "=") == 0) {
        insert_SymbolTable_value(current_identifier, current_value);
    }
}
```

```
void insert_dimensions() {
   insert_SymbolTable_arraydim(current_identifier, current_value);
}

void insert_parameters() {
   insert_SymbolTable_funcparam(current_function, current_type);
}

//int yywrap() {
//    return 1;
//}*/
char* getVariableType(char* id) {

    for(int i = 0 ; i < 1001 ; i++ )
     {
       if(strcmp(ST[i].symbol_name,id)==0)
       {
         return (ST[i].symbol_type);
         break;
       }
     }
     return "error";
}

char* getParameterType(char* funcName, int paramIndex) {
   char* p = NULL;
   int pc = 0;
   for(int i = 0 ; i < 1000 ; i++)
   {
     if(strcmp(ST[i].symbol_name,funcName)==0 )
     {
       p = ST[i].parameters;
       pc = ST[i].params_count;
       break;
     }
   }
   if(p == NULL)
   return "void";

   if(paramIndex >= pc)
   return "error";
```

```
    char res[100] = "";
    int count = -1;
    for(int i = 0;i<100;i++){
        if(p[i] == '\0')
        break;
        if(p[i] == ' '){
            count++;
            continue;
        }


        if(count == paramIndex){
            int len = strlen(res);  // Find the current length of the string.
            res[len] = p[i];          // Add the new character at the end of the string.
            res[len + 1] = '\0';
        }
        else if(count>paramIndex){
            break;
        }

    }

    return strdup(res);
}

char* getArgumentType(int argIndex) {

    return current_call_argument_types[argIndex];
}

int checkTypeCompatibility(char* expectedType, char* actualType) {
    if (strcmp(expectedType, actualType) == 0) {
        return 1;
    }
    if (strcmp(expectedType, "float") == 0 && strcmp(actualType, "int") == 0) {
        return 1;
    }
    return 0;
}
```

```
int main() {
    yyin = fopen("input.c", "r");
    yyparse();

    if (flag == 0) {
        printf("VALID PARSE\n");
        printf("%30s SYMBOL TABLE \n", " ");
        printf("%30s %s\n", " ", "------------");
        printSymbolTable();

        printf("\n\n%30s CONSTANT TABLE \n", " ");
        printf("%30s %s\n", " ", "--------------");
        printConstantTable();
    }
}
/*void yyerror(char *s) {
    printf("Line No.: %d %s %s\n", yylineno, s, yytext);
    flag = 1;
    printf("\nUNSUCCESSFUL: INVALID PARSE\n");
}

void insert_type() {
    insert_SymbolTable_type(current_identifier, current_type);
}

void insert_value() {
    if (strcmp(previous_operator, "=") == 0) {
        insert_SymbolTable_value(current_identifier, current_value);
    }
}

void insert_dimensions() {
    insert_SymbolTable_arraydim(current_identifier, current_value);
}

void insert_parameters() {
    insert_SymbolTable_funcparam(current_function, current_type);
}*/

int yywrap() {
    return 1;
```

```
}

int newGetType(char* s){
    char* res = "";
    int t = -1;

        for(int i = 0 ; i < 1001 ; i++ )
      {
        if(strcmp(ST[i].symbol_name,s)==0)
        {
          res = ST[i].symbol_type;
          break;
        }
      }

        if(strcmp(res,"char*") == 0 ){
            t = 0;
        }
        else if (res[0] == 's') {
            t = 0;  // Indicate it's a string
        }
        else if (strcmp(res,"int") == 0) {
            t = 1;  // Indicate it's a numeric type
            //current_exp_type = current_exp_type<1? 1: current_exp_type ;
        }
        else if (strcmp(res,"char") == 0) {
            t = 2;  // Indicate it's a numeric type
            //current_exp_type = current_exp_type<2? 2: current_exp_type ;
        }
        else if (strcmp(res,"float") == 0) {
            t = 3;  // Indicate it's a numeric type
            //current_exp_type = current_exp_type<3? 3: current_exp_type ;
        }
        else if (strcmp(res,"int*") == 0) {
            t = 4;  // Indicate it's a numeric type
            //current_exp_type = current_exp_type<3? 3: current_exp_type ;
        }
        else if (strcmp(res,"float*") == 0) {
            t = 5;  // Indicate it's a numeric type
            //current_exp_type = current_exp_type<3? 3: current_exp_type ;
        }
        else {
           t = -1;
       }

       return t;
}


int newCheckCompatibility(int a, int b){
    if(a == 1 || a== 2){
       if(b == 1||b == 2)
       return 1;
    }
    if(a == 3)
    if(b == 1 || b == 2 || b == 3){
       return 1;
    }
    return (a == b || b == 6);
 }
```

OVERVIEW OF CODE

LEX CODE

The lexer works in conjunction with a parser (indicated by the inclusion of "y.tab.h") and provides comprehensive token recognition along with sophisticated symbol table management. It's designed to handle both the basic lexical analysis tasks like tokenization and the more complex requirements of tracking symbols, scopes, and program structure.

Overview:

1. Data Structures:
    - Implements two main tables:
        - ConstantTable: Stores constants and their types
        - SymbolTable: A more complex table storing identifiers, functions, and various attributes like scope, parameters, line numbers etc.
2. Core Functionality:
    - Handles symbol table management for a C-like programming language
    - Uses hash-based storage for efficient lookup and insertion
    - Maintains scoping information through nesting levels

    - Tracks function declarations, definitions and parameters
    - Manages array declarations and dimensions
    - Keeps track of identifiers' line numbers and nested levels
3. Key Features:
    - Symbol management with scope handling
    - Function declaration and definition tracking
    - Support for arrays
    - Parameter counting and tracking
    - Type checking capabilities
    - Multi-level nested scope support
    - Line number tracking
    - Duplicate declaration checking
4. Pattern Recognition:
    - Recognizes C language keywords
    - Handles string and character constants
    - Processes numeric constants (integers and floating point)
    - Identifies identifiers and array identifiers
    - Recognizes operators and special symbols
    - Handles multi-line and single-line comments
    - Processes preprocessor directives
5. Error Handling:
    - Detects unclosed strings
    - Identifies illegal characters
    - Catches invalid character constants
    - Checks for function redeclarations
    - Validates parameter types

This is the foundation for a compiler, providing comprehensive token recognition and symbol management capabilities while maintaining detailed information about the program structure and catching various syntax and semantic errors at the lexical analysis stage.

YACC CODE

This code sets up a parser with symbol table management, type checking, and semantic analysis capabilities for the C programming language. It includes necessary headers, function declarations, data structures, and token definitions required for parsing C code. The key components and features of this code are:

1. Parser Structure:
    - This is a YACC/Bison grammar file that defines the syntax rules for parsing a C-like programming language

- It includes support for basic C language constructs including variables, functions, control structures, and expressions

2. Key Features Supported:
    - Data types: int, char, float, double, long, short, signed, unsigned, struct, void
    - Control structures: if-else, while, for, do-while, switch-case
    - Symbol management with scope handling
    - Function declaration and definition tracking
    - Function argument and Return type tracking
    - Variable Declaration
    - Support for arrays
    - Parameter counting and tracking
    - Type checking capabilities
    - Multi-level nested scope support
    - Line number tracking
    - Duplicate declaration checking
    - NULL handling
    - Expression Handling
3. Symbol Table Management:
    - Maintains a symbol table to track variables, functions, and their properties
    - Store information about:
        - Variable/function names
        - Data types
        - Array dimensions
        - Function parameters
        - Scope information
        - Values
4. Type Checking:
    - Implements comprehensive type checking for:
        - Assignment compatibility
        - Function parameter matching
        - Array operations
        - Expression evaluation
        - Return type validation
5. Error Handling:
    - Detects and reports various errors, including:
        - Variable declarations
        - Duplicate declaration of ID
        - Array size less than 1
        - Function declarations

        - Params of type void
        - No functions defined
        - Function main is not last function
        - Int function has void return
        - Void function has int return
        - Call expressions
        - Argument expression not of type integer
        - ID is not a function
        - ID undeclared in current scope
        - Type of paramaters does not match type of argument
        - Number of parameters and arguments do not match
        - Select/while statements
        - Expression in test not of type int
        - Expressions
        - ID undeclared in current scope
        - Array ID has no subscript
        - Single variable ID or function has subscript
        - Expression in subscript not of type int
        - Expression on rhs of assigment and in arithmetic ops must be int
        - Expression on lhs of assignment not single variable
        - Return statement
        - Expression has type void, function or array

6. Additional Features:
    - Scope management for variables
    - Parameter counting for function calls
    - Support for pointer operations
    - Constant table management
    - Array dimension validation
    - Struct member access

This parser is designed to validate C code for correctness while performing semantic analysis during compilation. It's a significant part of a compiler's front end, handling both syntactic and semantic analysis of the input code.

CHECKS PERFORMED

1. Type Checking:
    - Compatibility checks between operands in expressions
    - Return type matching for functions

    - Array dimension checks
    - Pointer and NULL assignment checks
2. Scope and Declaration Checks:
    - Checking if identifiers are declared before use
    - Checking if identifiers are within the correct scope
    - Checking for duplicate declarations
3. Function Call Checks:
    - Verifying that functions are declared before being called
    - Checking the number of arguments matches the function's parameter count
    - Type checking for function arguments against parameter types
4. Array Checks:
    - Verifying correct array subscripting
    - Checking that array identifiers are used with subscripts
5. Struct Checks:
    - Verifying that struct members are accessed correctly
6. Control Flow Checks:
    - Ensuring that conditions in if statements and loops are of integer type
7. Operator Checks:
    - Verifying that operators are used with compatible types
8. Constant and Value Insertion:
    - Inserting types and values into the symbol table
9. Parameter Checks:
    - Inserting and checking function parameters
10. Semantic Checks:
    - Various semantic checks throughout the parsing process
11. Error Reporting:
    - Generating error messages for various issues detected during parsing

SYMBOL TABLE DESCRIPTION

Data Structure: Array with Hashing The symbol table is implemented using an array, likely of a struct type, with a hashing mechanism for efficient lookup and insertion. The hashing function implements the DJB2 hash algorithm, a simple but effective non-cryptographic hash function. It takes a string (str) as input and produces an unsigned long integer as the hash value. The algorithm initialises the hash with a prime number (5381) and then iterates through each input string character.

Entries: Each entry in the symbol table contains the following information:

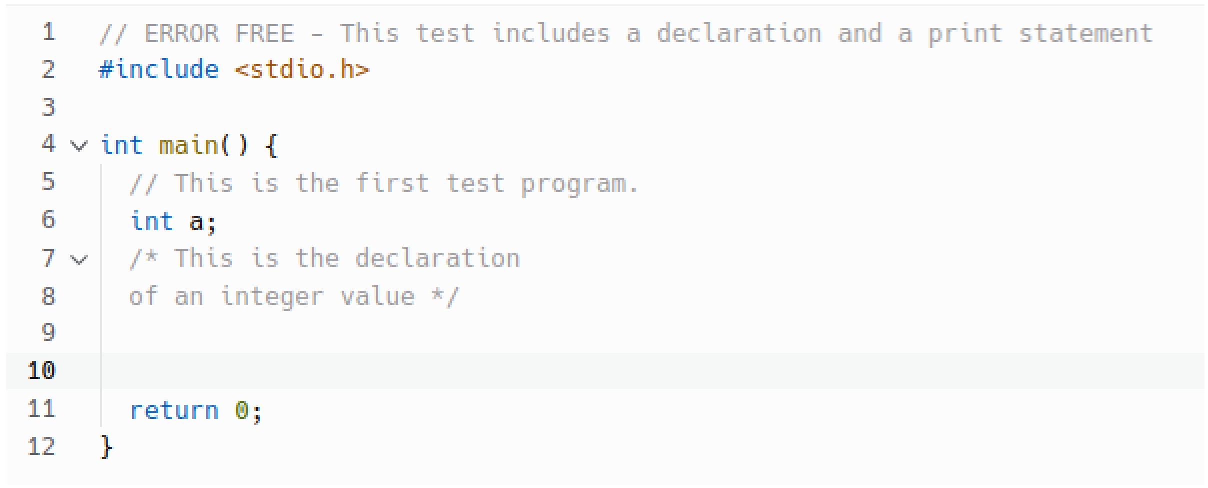
1. Symbol Name: The variable, function, or other program entity..
2. Type: The data type of the symbol (e.g., int, float, char, etc.).
3. Class: Likely indicates the category of the symbol (e.g., variable, function, parameter).
4. Value: Stores the value of the symbol if applicable.
5. Array Dimensions: For array variables, stores the dimension information.
6. Parameters List: For functions, stores the list of parameters.
7. Parameter Count: For functions, stores the number of parameters.
8. Nesting Level: Indicates the scope depth of the symbol.
9. Line Number: The line in the source code where the symbol is declared or defined.

TESTCASES

VALID CASES:

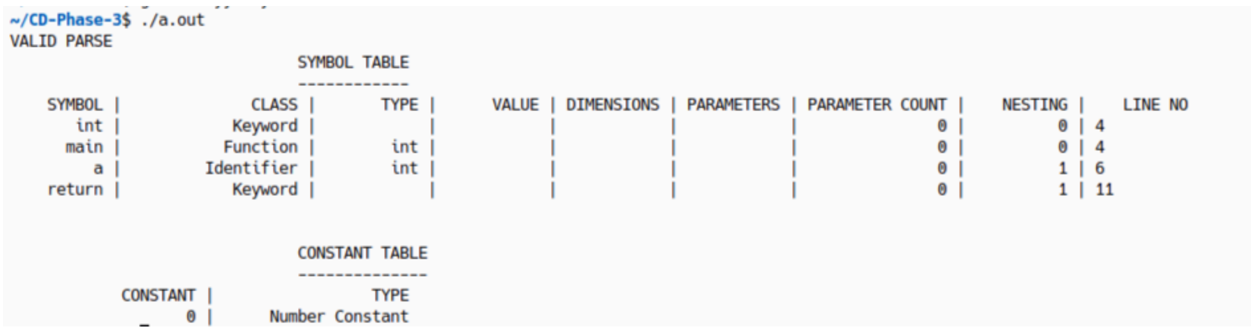
1.Includes declaration

Input:



```c
// ERROR FREE - This test includes a declaration and a print statement
#include <stdio.h>

int main() {
  // This is the first test program.
  int a;
  /* This is the declaration
  of an integer value */


  return 0;
}
```

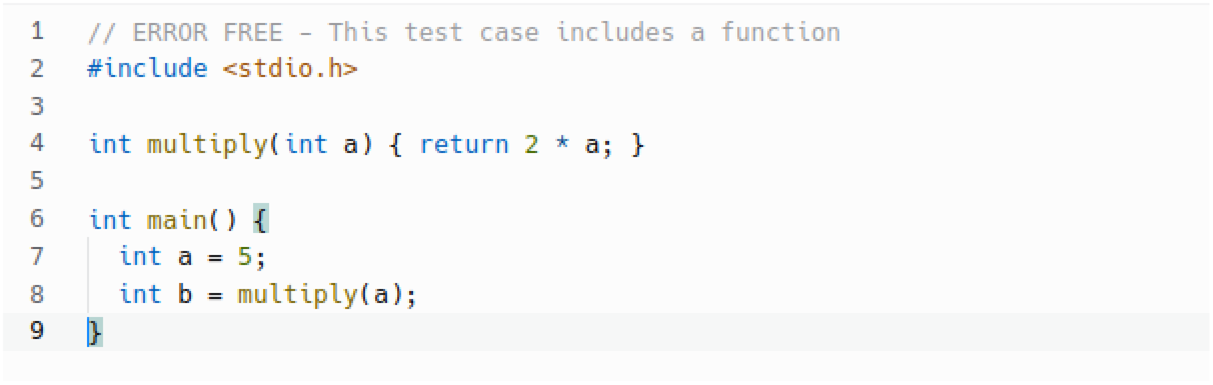
Output:



```
~/CD-Phase-3$ ./a.out
VALID PARSE
                                        SYMBOL TABLE
                                        ------------

     SYMBOL |              CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |     NESTING |     LINE NO
        int |            Keyword |            |            |            |            |               0 |           0 | 4
       main |           Function |        int |            |            |            |               0 |           0 | 4
          a |         Identifier |        int |            |            |            |               0 |           1 | 6
     return |            Keyword |            |            |            |            |               0 |           1 | 11


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

               CONSTANT |                TYPE
                      0 |     Number Constant
```

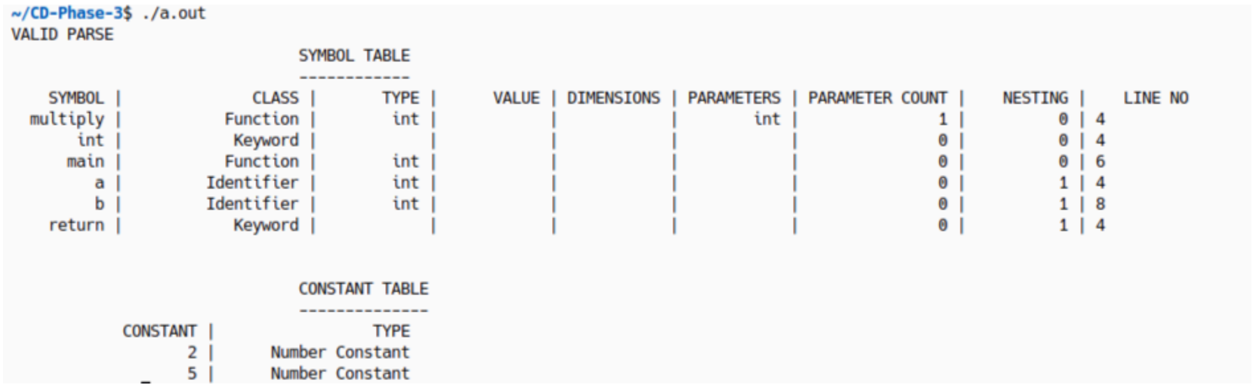
2.Includes a function

Input:



```
// ERROR FREE - This test case includes a function
#include <stdio.h>

int multiply(int a) { return 2 * a; }

int main() {
  int a = 5;
  int b = multiply(a);
}
```

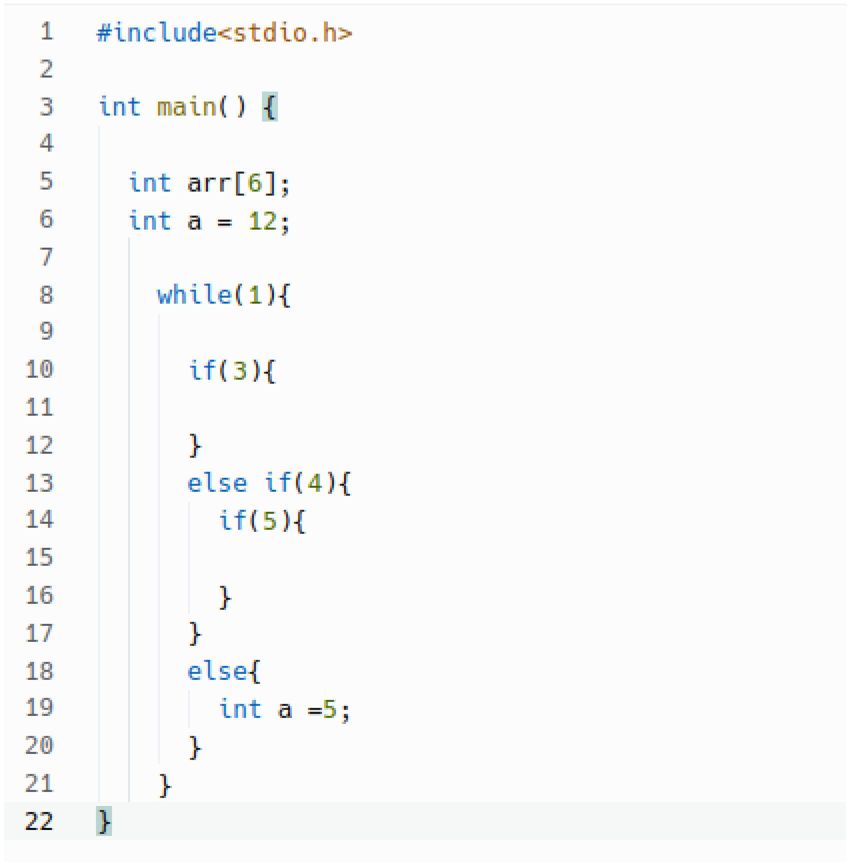
Output:



```
~/CD-Phase-3$ ./a.out
VALID PARSE
                                        SYMBOL TABLE
                                        ------------

     SYMBOL |              CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |     NESTING |     LINE NO
   multiply |           Function |        int |            |            |        int |               1 |           0 | 4
        int |            Keyword |            |            |            |            |               0 |           0 | 4
       main |           Function |        int |            |            |            |               0 |           0 | 6
          a |         Identifier |        int |            |            |            |               0 |           1 | 4
          b |         Identifier |        int |            |            |            |               0 |           1 | 8
     return |            Keyword |            |            |            |            |               0 |           1 | 4


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

               CONSTANT |                TYPE
                      2 |     Number Constant
                      5 |     Number Constant
```

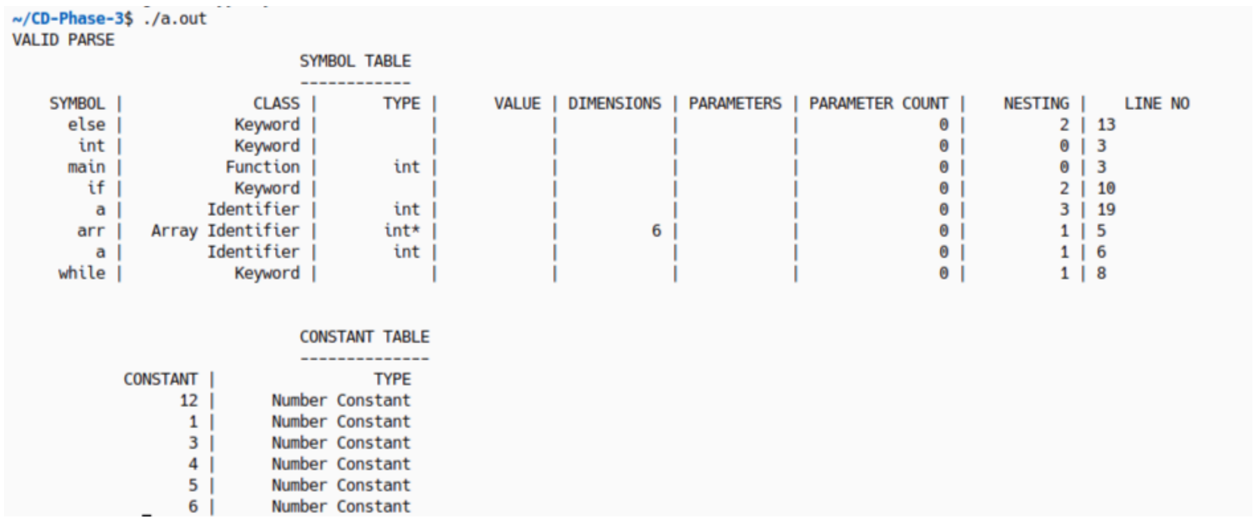
3. Includes array declarations and conditional statements

Input:



```
1   #include<stdio.h>
2
3   int main() {
4
5     int arr[6];
6     int a = 12;
7
8       while(1){
9
10        if(3){
11
12        }
13        else if(4){
14          if(5){
15
16          }
17        }
18        else{
19          int a =5;
20        }
21      }
22  }
```

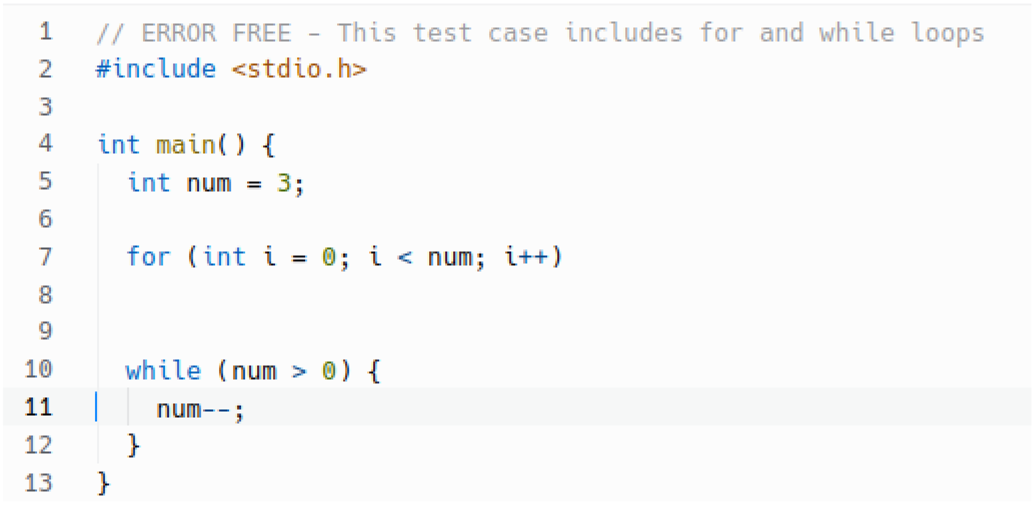
Output:



```
~/CD-Phase-3$ ./a.out
VALID PARSE
                                    SYMBOL TABLE
                                    ------------

    SYMBOL |              CLASS |       TYPE |       VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |      NESTING |      LINE NO
      else |            Keyword |            |             |            |            |               0 |            2 | 13
       int |            Keyword |            |             |            |            |               0 |            0 | 3
      main |           Function |        int |             |            |            |               0 |            0 | 3
        if |            Keyword |            |             |            |            |               0 |            2 | 10
         a |         Identifier |        int |             |            |            |               0 |            3 | 19
       arr |   Array Identifier |       int* |             |          6 |            |               0 |            1 | 5
         a |         Identifier |        int |             |            |            |               0 |            1 | 6
     while |            Keyword |            |             |            |            |               0 |            1 | 8


                                    CONSTANT TABLE
                                    --------------
           CONSTANT |           TYPE
                 12 |   Number Constant
                  1 |   Number Constant
                  3 |   Number Constant
                  4 |   Number Constant
                  5 |   Number Constant
                  6 |   Number Constant
```

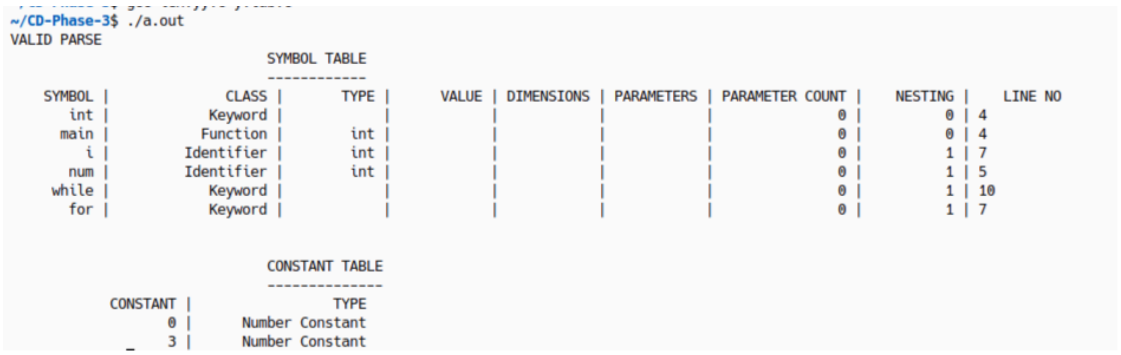
4. Includes for and while loops

Input:



```c
// ERROR FREE - This test case includes for and while loops
#include <stdio.h>

int main() {
  int num = 3;

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


  while (num > 0) {
    num--;
  }
}
```

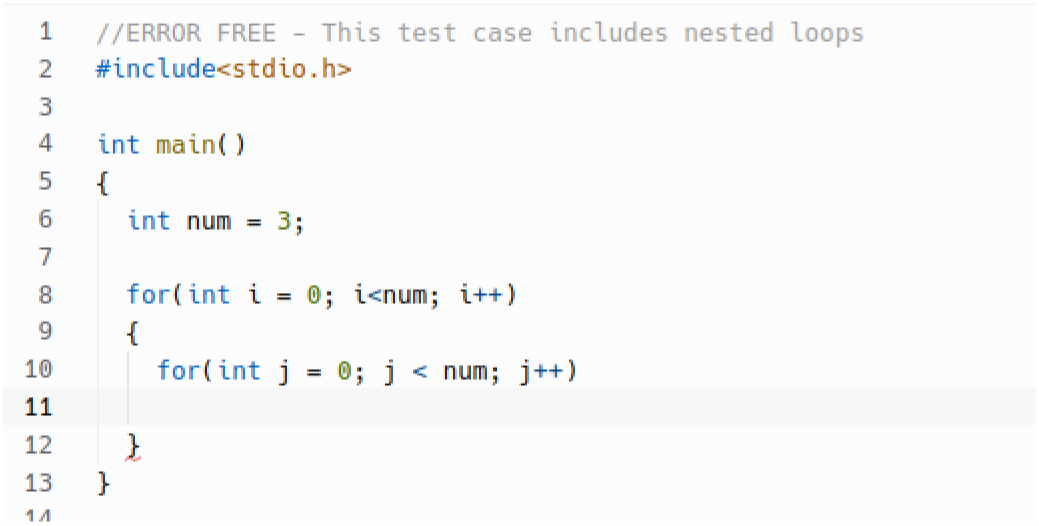
Output:



```
~/CD-Phase-3$ ./a.out
VALID PARSE
                         SYMBOL TABLE
                         ------------

   SYMBOL |            CLASS |      TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |     NESTING |    LINE NO
      int |          Keyword |           |            |            |            |               0 |           0 | 4
     main |         Function |       int |            |            |            |               0 |           0 | 4
        i |       Identifier |       int |            |            |            |               0 |           1 | 7
      num |       Identifier |       int |            |            |            |               0 |           1 | 5
    while |          Keyword |           |            |            |            |               0 |           1 | 10
      for |          Keyword |           |            |            |            |               0 |           1 | 7


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

         CONSTANT |                 TYPE
                0 |      Number Constant
                3 |      Number Constant
```

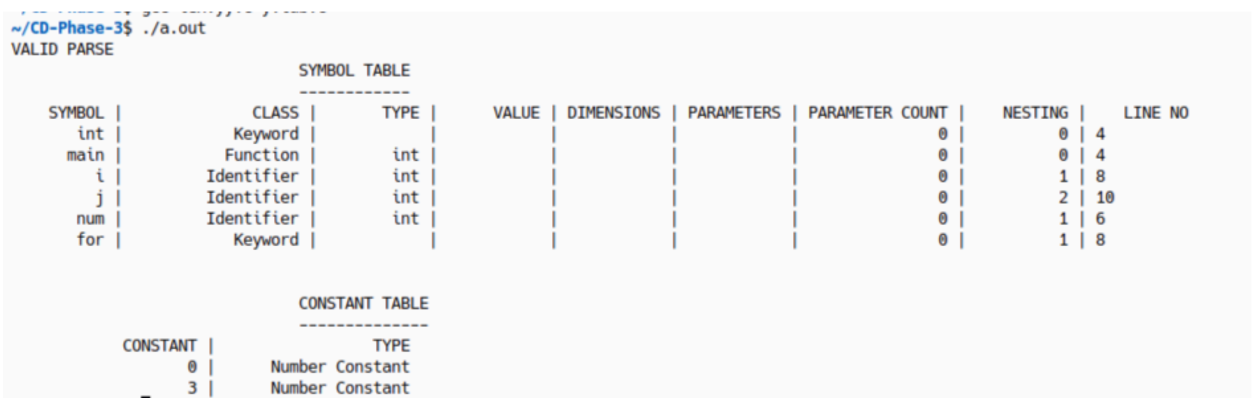
5. Includes nested loops

Input:



```c
//ERROR FREE - This test case includes nested loops
#include<stdio.h>

int main()
{
  int num = 3;

  for(int i = 0; i<num; i++)
  {
    for(int j = 0; j < num; j++)

  }
}
```

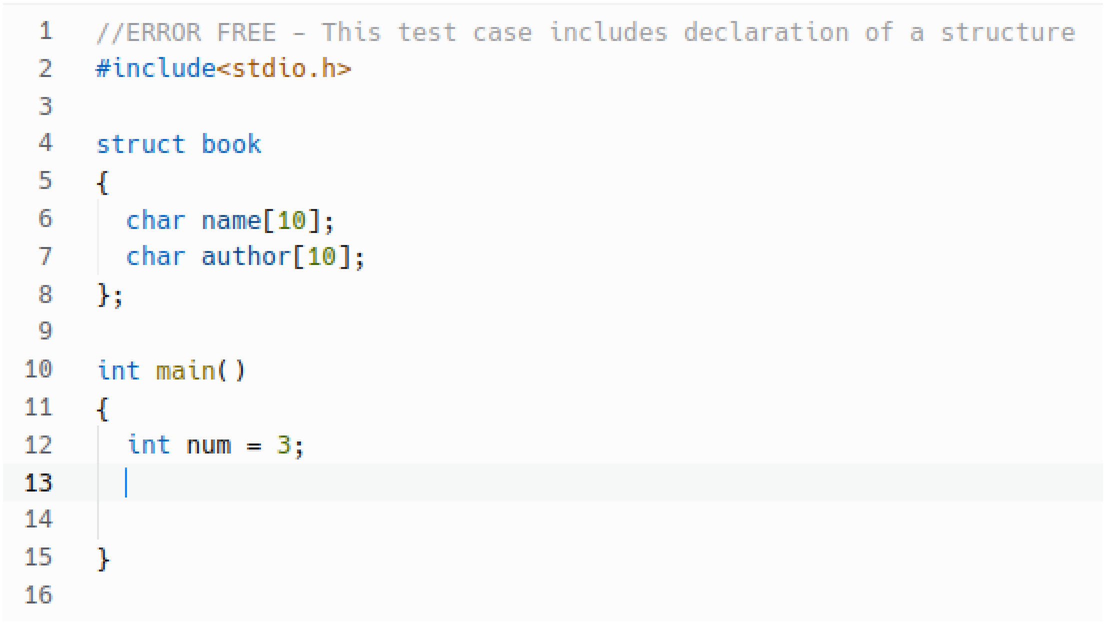
Output:



```
~/CD-Phase-3$ ./a.out
VALID PARSE
                                    SYMBOL TABLE
                                    ------------

     SYMBOL |              CLASS |        TYPE |        VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |     NESTING |     LINE NO
        int |            Keyword |             |              |            |            |               0 |           0 | 4
       main |           Function |         int |              |            |            |               0 |           0 | 4
          i |         Identifier |         int |              |            |            |               0 |           1 | 8
          j |         Identifier |         int |              |            |            |               0 |           2 | 10
        num |         Identifier |         int |              |            |            |               0 |           1 | 6
        for |            Keyword |             |              |            |            |               0 |           1 | 8

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

          CONSTANT |                 TYPE
                 0 |      Number Constant
                 3 |      Number Constant
```

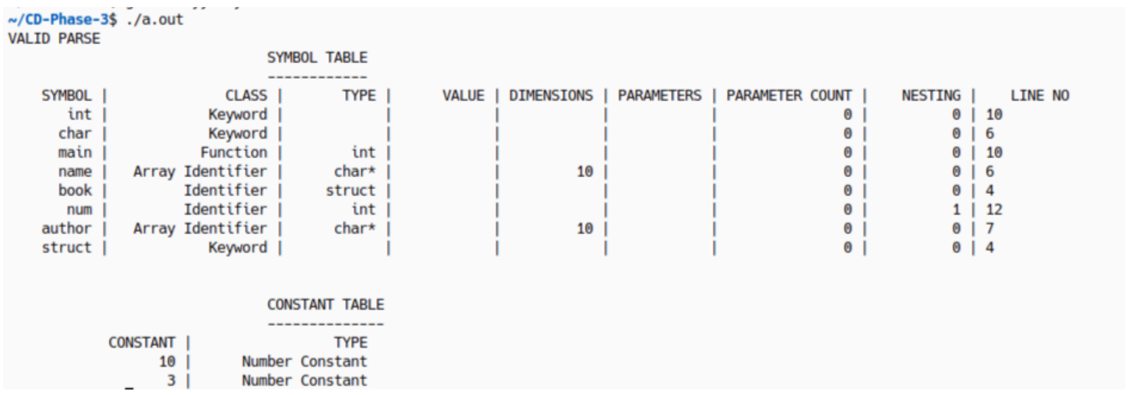
6. Includes declaration of structure

Input:



```
//ERROR FREE - This test case includes declaration of a structure
#include<stdio.h>

struct book
{
    char name[10];
    char author[10];
};

int main()
{
    int num = 3;


}

```

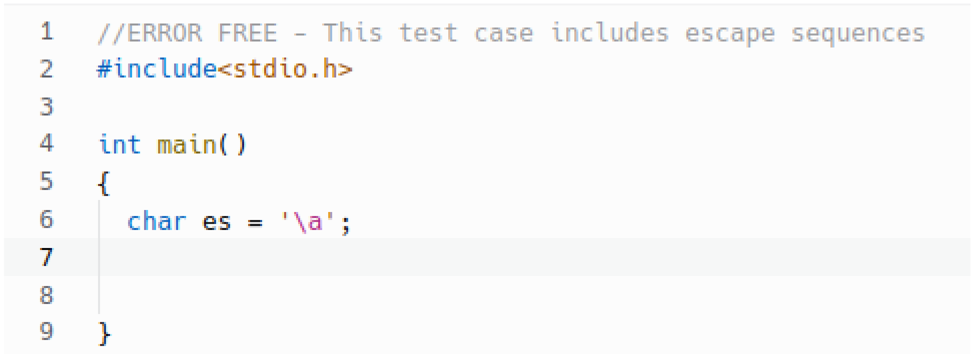
Output:



```
~/CD-Phase-3$ ./a.out
VALID PARSE
                                    SYMBOL TABLE
                                    ------------

     SYMBOL |              CLASS |        TYPE |        VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |     NESTING |     LINE NO
        int |            Keyword |             |              |            |            |               0 |           0 | 10
       char |            Keyword |             |              |            |            |               0 |           0 | 6
       main |           Function |         int |              |            |            |               0 |           0 | 10
       name |   Array Identifier |       char* |              |         10 |            |               0 |           0 | 6
       book |         Identifier |      struct |              |            |            |               0 |           0 | 4
        num |         Identifier |         int |              |            |            |               0 |           1 | 12
     author |   Array Identifier |       char* |              |         10 |            |               0 |           0 | 7
     struct |            Keyword |             |              |            |            |               0 |           0 | 4

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

          CONSTANT |                 TYPE
                10 |      Number Constant
                 3 |      Number Constant
```

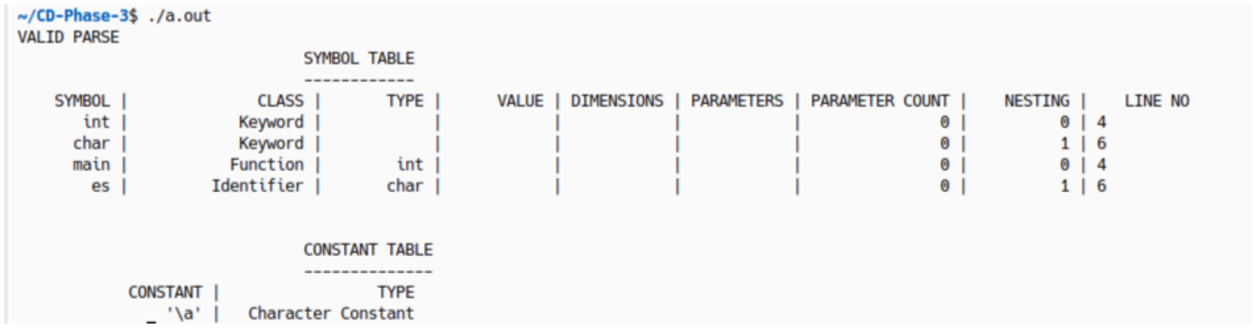
7. Includes escape sequences

Input:



```c
//ERROR FREE - This test case includes escape sequences
#include<stdio.h>

int main()
{
  char es = '\a';


}
```

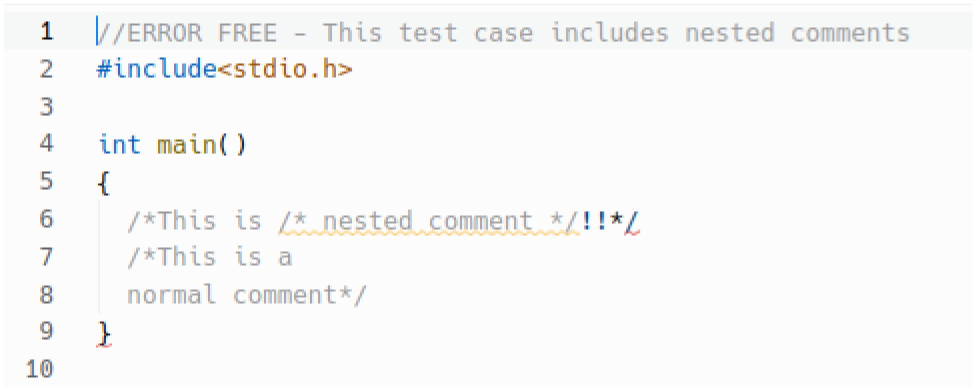
Output:



```
~/CD-Phase-3$ ./a.out
VALID PARSE
                              SYMBOL TABLE
                              ------------
    SYMBOL |            CLASS |      TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |     NESTING |     LINE NO
       int |          Keyword |           |            |            |            |               0 |           0 | 4
      char |          Keyword |           |            |            |            |               0 |           1 | 6
      main |         Function |       int |            |            |            |               0 |           0 | 4
        es |       Identifier |      char |            |            |            |               0 |           1 | 6


                              CONSTANT TABLE
                              --------------
          CONSTANT |                 TYPE
             '\a' |   Character Constant
```

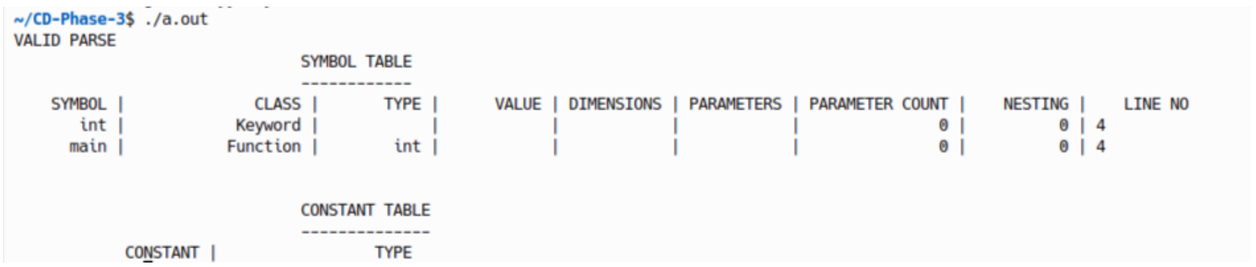
8. Includes nested comments

Input:



```c
//ERROR FREE - This test case includes nested comments
#include<stdio.h>

int main()
{
  /*This is /* nested comment */!!*/
  /*This is a
  normal comment*/
}

```

Output:

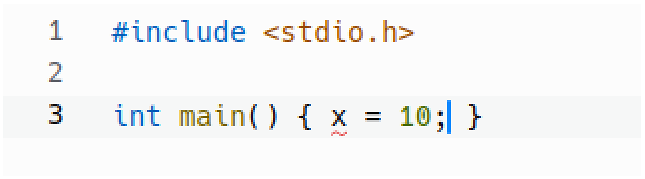


```
~/CD-Phase-3$ ./a.out
VALID PARSE
                               SYMBOL TABLE
                               ------------
    SYMBOL |                CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |              Keyword |            |            |            |            |               0 |          0 | 4
      main |             Function |        int |            |            |            |               0 |          0 | 4


                               CONSTANT TABLE
                               --------------
            CONSTANT |                TYPE
```

ERROR CASES:

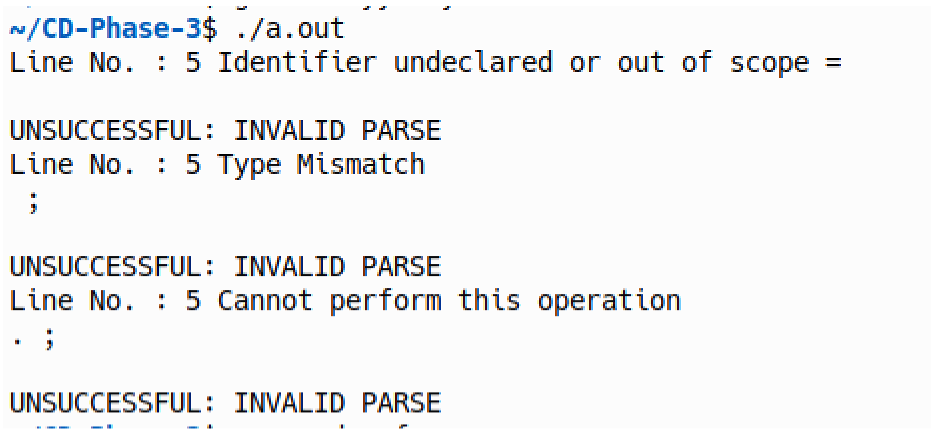
1. Variable declarations

Input:



```
#include <stdio.h>

int main() { x = 10; }
```

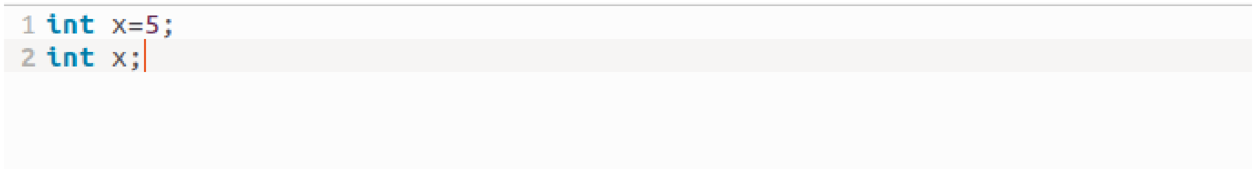
Output:



```
~/CD-Phase-3$ ./a.out
Line No. : 5 Identifier undeclared or out of scope =

UNSUCCESSFUL: INVALID PARSE
Line No. : 5 Type Mismatch
 ;

UNSUCCESSFUL: INVALID PARSE
Line No. : 5 Cannot perform this operation
. ;

UNSUCCESSFUL: INVALID PARSE
```

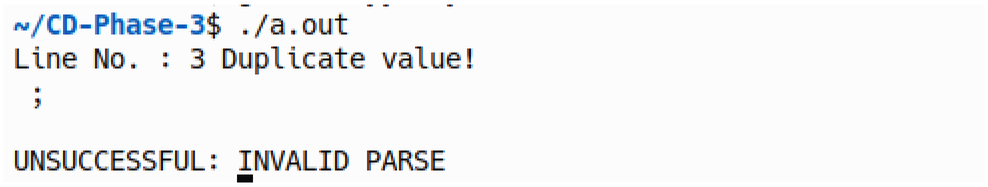
2. Duplicate declaration of ID

Input:



```
int x=5;
int x;
```

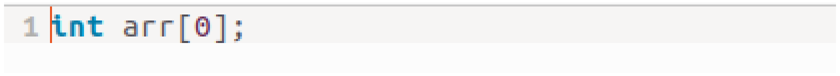
Output:



```
~/CD-Phase-3$ ./a.out
Line No. : 3 Duplicate value!
 ;

UNSUCCESSFUL: INVALID PARSE
```

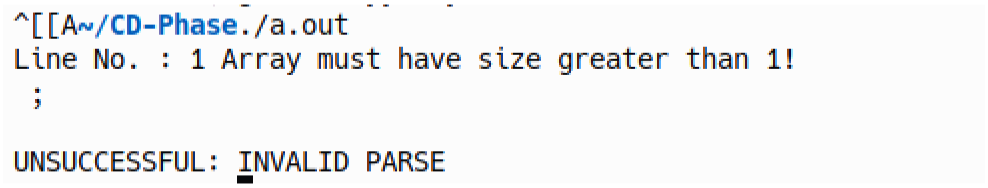
3. Array size less than1

Input:



```
int arr[0];
```

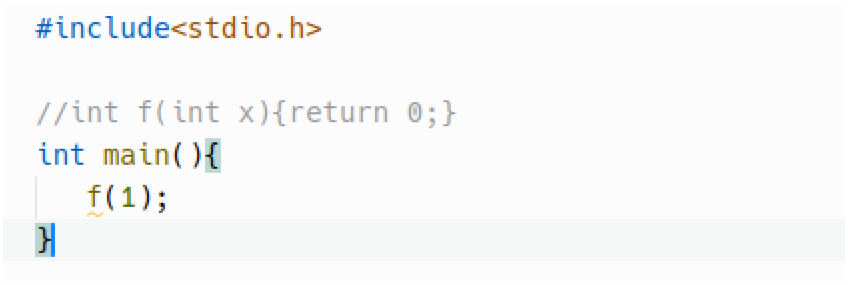
Output:



```
^[[A~/CD-Phase./a.out
Line No. : 1 Array must have size greater than 1!
 ;

UNSUCCESSFUL: INVALID PARSE
```

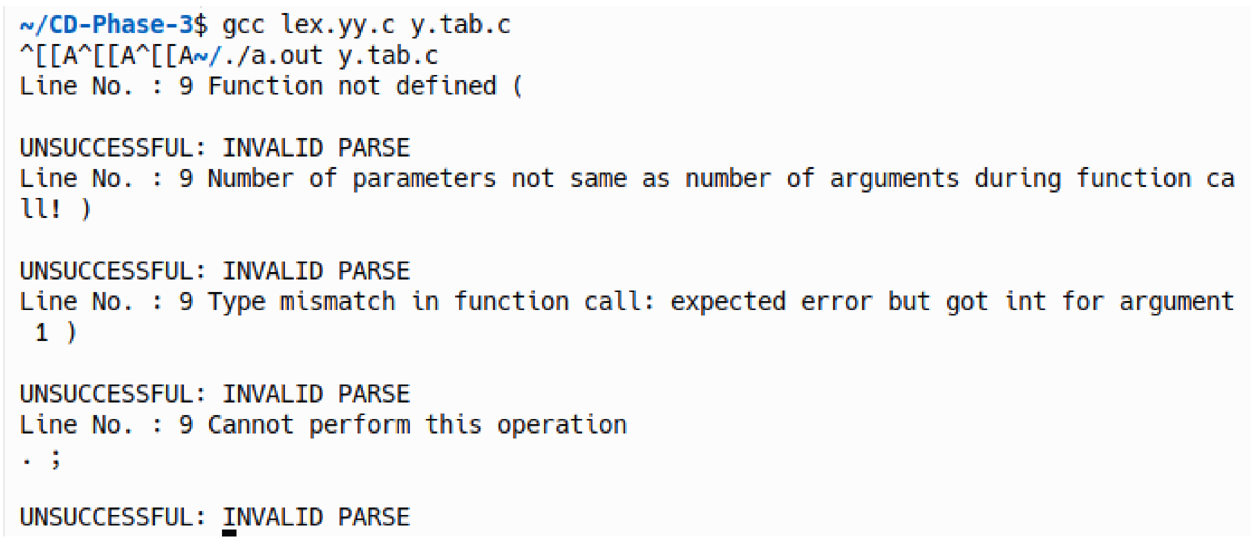
4. Function declerations

Input:



```
#include<stdio.h>

//int f(int x){return 0;}
int main(){
    f(1);
}
```

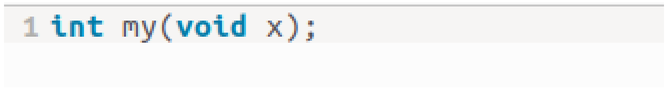
Output:



```
~/CD-Phase-3$ gcc lex.yy.c y.tab.c
^[[A^[[A^[[A~/./a.out y.tab.c
Line No. : 9 Function not defined (

UNSUCCESSFUL: INVALID PARSE
Line No. : 9 Number of parameters not same as number of arguments during function ca
ll! )

UNSUCCESSFUL: INVALID PARSE
Line No. : 9 Type mismatch in function call: expected error but got int for argument
 1 )

UNSUCCESSFUL: INVALID PARSE
Line No. : 9 Cannot perform this operation
. ;

UNSUCCESSFUL: INVALID PARSE
```

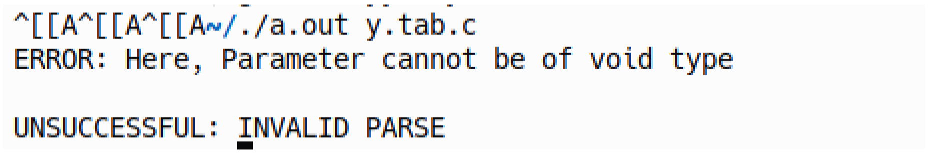
5. Params of type void

Input:



```c
int my(void x);
```

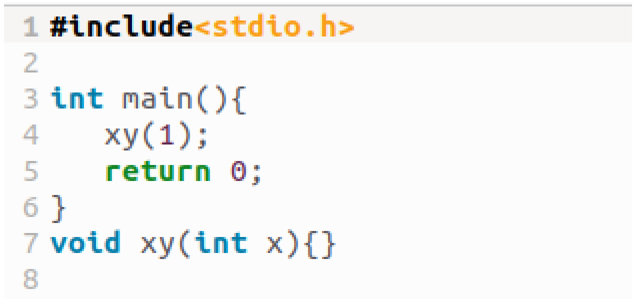
Output:



```
^[[A^[[A^[[A~/./a.out y.tab.c
ERROR: Here, Parameter cannot be of void type

UNSUCCESSFUL: INVALID PARSE
```

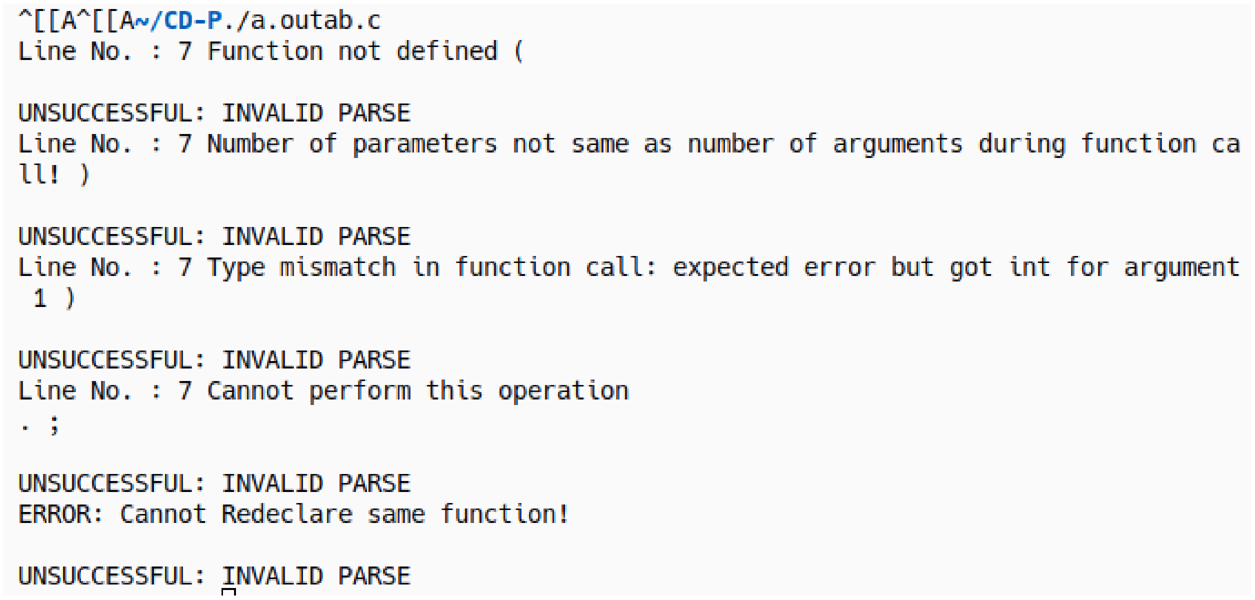
7. Function main is not last function

Input:



```c
#include<stdio.h>

int main(){
    xy(1);
    return 0;
}
void xy(int x){}

```

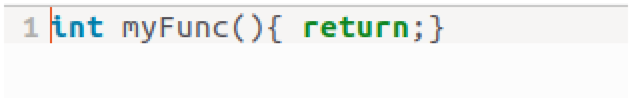
Output:



```
^[[A^[[A~/CD-P./a.outab.c
Line No. : 7 Function not defined (

UNSUCCESSFUL: INVALID PARSE
Line No. : 7 Number of parameters not same as number of arguments during function ca
ll! )

UNSUCCESSFUL: INVALID PARSE
Line No. : 7 Type mismatch in function call: expected error but got int for argument
 1 )

UNSUCCESSFUL: INVALID PARSE
Line No. : 7 Cannot perform this operation
. ;

UNSUCCESSFUL: INVALID PARSE
ERROR: Cannot Redeclare same function!

UNSUCCESSFUL: INVALID PARSE
```

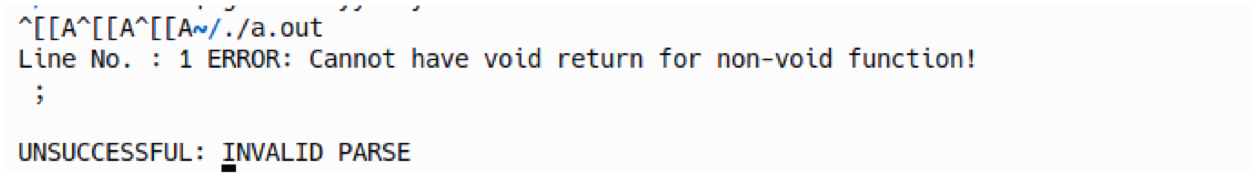
8. Int function h a s void return

Input:



```
int myFunc(){ return;}
```

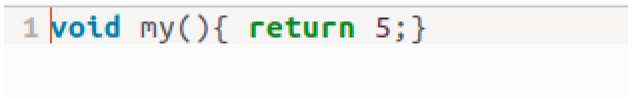
Output:



```
^[[A^[[A^[[A~/./a.out
Line No. : 1 ERROR: Cannot have void return for non-void function!
 ;

UNSUCCESSFUL: INVALID PARSE
```

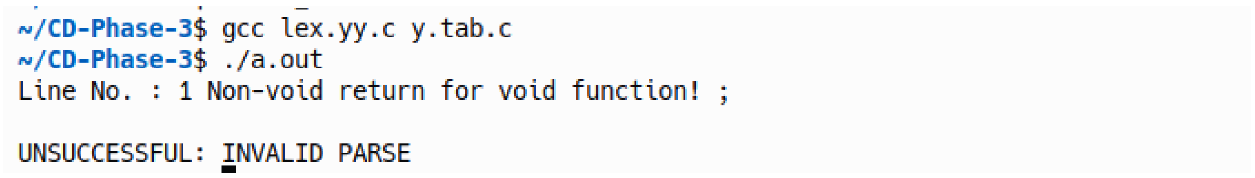
9. Void function h a s int return

Input:



```
void my(){ return 5;}
```

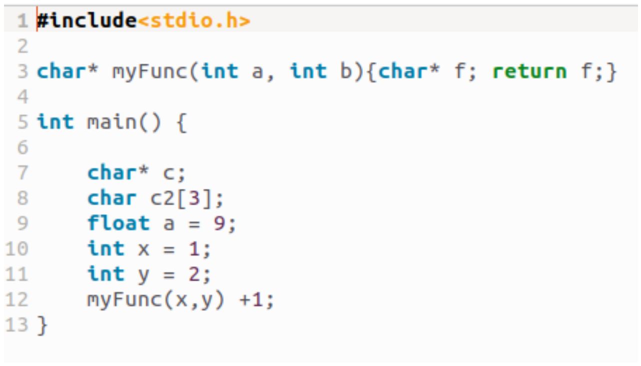
Output:



```
~/CD-Phase-3$ gcc lex.yy.c y.tab.c
~/CD-Phase-3$ ./a.out
Line No. : 1 Non-void return for void function! ;

UNSUCCESSFUL: INVALID PARSE
```

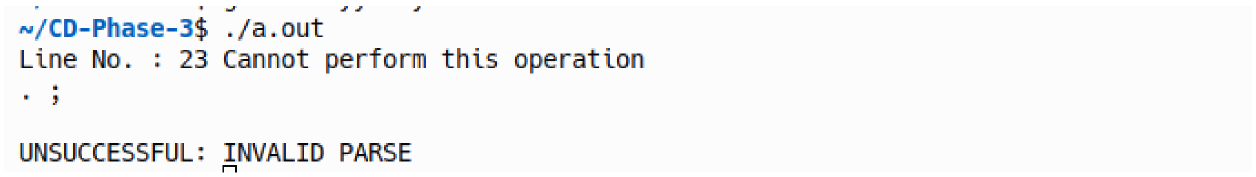
10. Call expressions

Input:



```c
#include<stdio.h>

char* myFunc(int a, int b){char* f; return f;}

int main() {

    char* c;
    char c2[3];
    float a = 9;
    int x = 1;
    int y = 2;
    myFunc(x,y) +1;
}
```

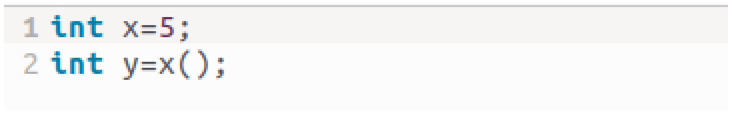
Output:



```
~/CD-Phase-3$ ./a.out
Line No. : 23 Cannot perform this operation
. ;

UNSUCCESSFUL: INVALID PARSE
```

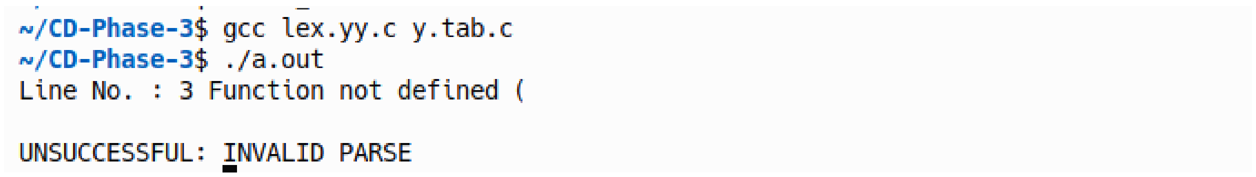
12. ID is not a function

Input:



```c
int x=5;
int y=x();
```

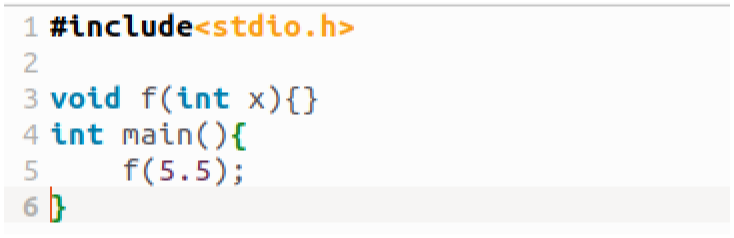
Output:



```
~/CD-Phase-3$ gcc lex.yy.c y.tab.c
~/CD-Phase-3$ ./a.out
Line No. : 3 Function not defined (

UNSUCCESSFUL: INVALID PARSE
```

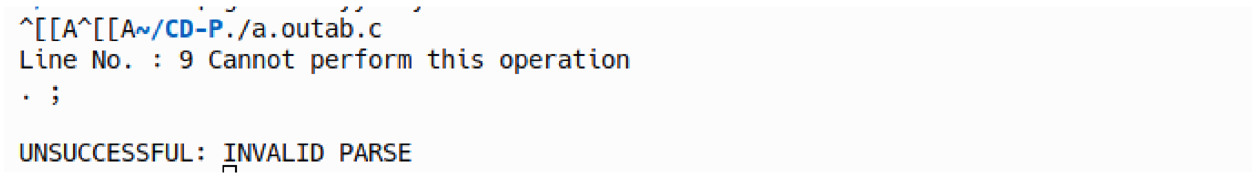
13. ID undeclared in current scope

Input:



```c
#include<stdio.h>

void f(int x){}
int main(){
    f(5.5);
}
```

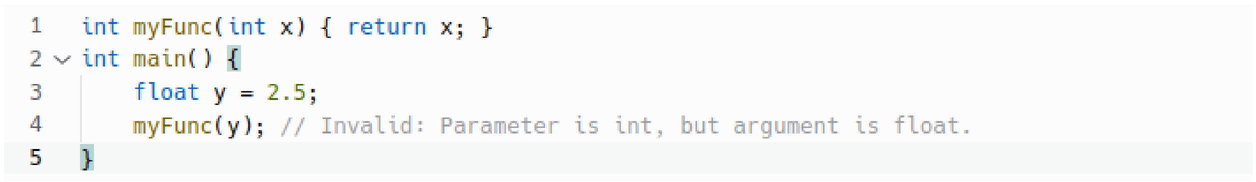
Output:



```
^[[A^[[A~/CD-P./a.outab.c
Line No. : 9 Cannot perform this operation
. ;

UNSUCCESSFUL: INVALID PARSE
```

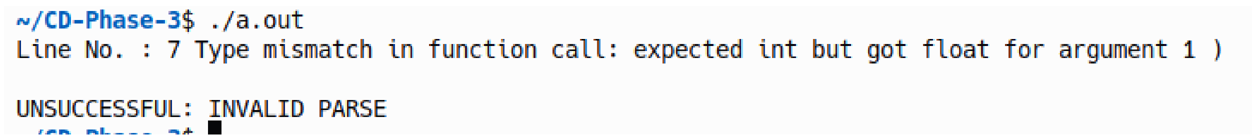
14. Type of paramaters does not match type of argument

Input:



```c
int myFunc(int x) { return x; }
int main() {
    float y = 2.5;
    myFunc(y); // Invalid: Parameter is int, but argument is float.
}
```

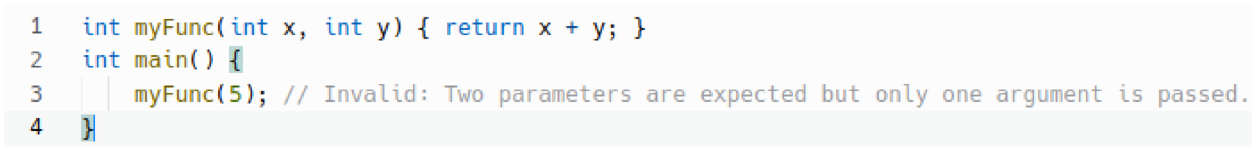
Output:



```
~/CD-Phase-3$ ./a.out
Line No. : 7 Type mismatch in function call: expected int but got float for argument 1 )

UNSUCCESSFUL: INVALID PARSE
```

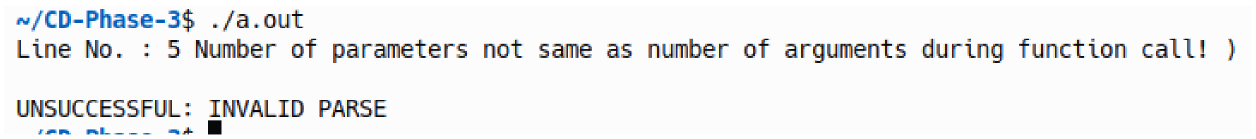
15. Number of parameters and arguments donotmatch

Input:



```c
int myFunc(int x, int y) { return x + y; }
int main() {
    myFunc(5); // Invalid: Two parameters are expected but only one argument is passed.
}
```

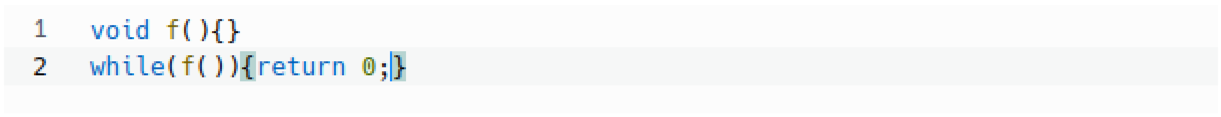
Output:



```
~/CD-Phase-3$ ./a.out
Line No. : 5 Number of parameters not same as number of arguments during function call! )

UNSUCCESSFUL: INVALID PARSE
```

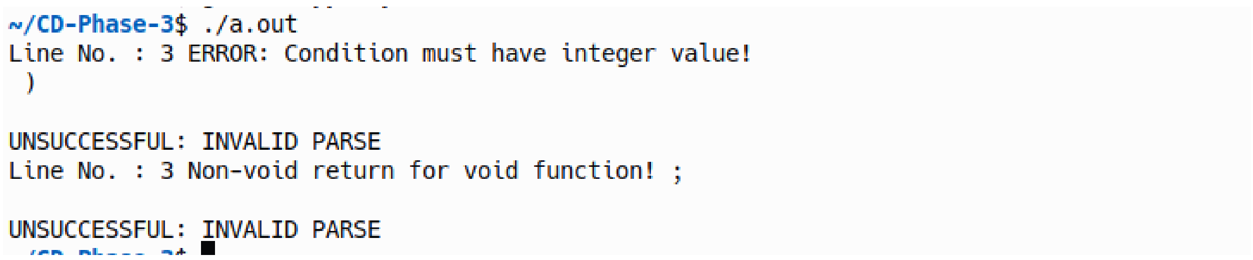
16. Select/while statements

Input:



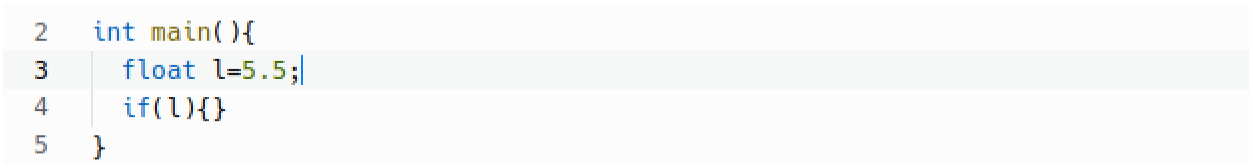
```c
void f(){}
while(f()){return 0;}
```

Output:

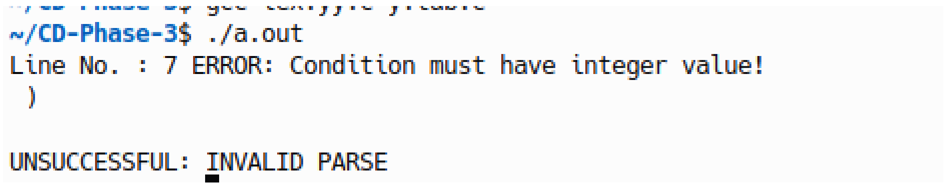

```
~/CD-Phase-3$ ./a.out
Line No. : 3 ERROR: Condition must have integer value!
 )

UNSUCCESSFUL: INVALID PARSE
Line No. : 3 Non-void return for void function! ;

UNSUCCESSFUL: INVALID PARSE
```

17. Expression in test not of type int

Input:

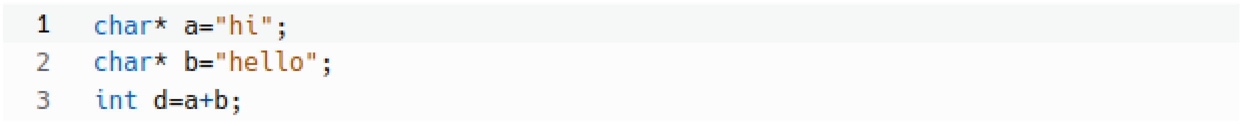

```
int main(){
  float l=5.5;
  if(l){}
}
```

Output:

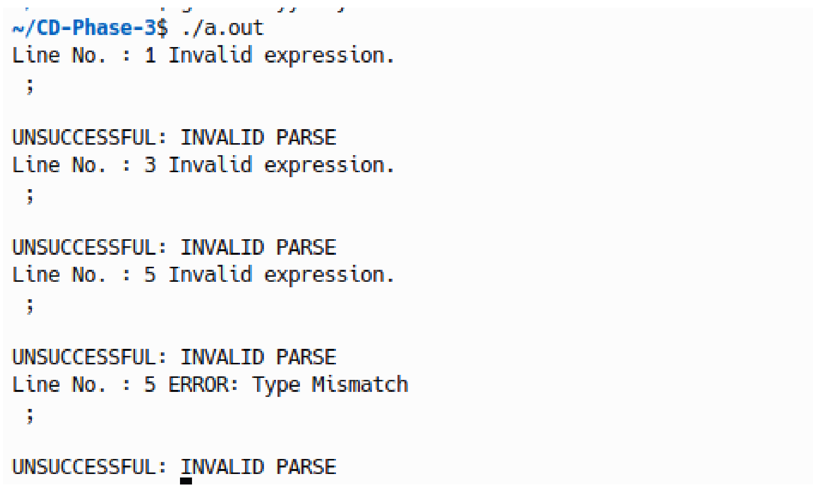

```
~/CD-Phase-3$ ./a.out
Line No. : 7 ERROR: Condition must have integer value!
 )

UNSUCCESSFUL: INVALID PARSE
```

18. Expressions

Input:


```
char* a="hi";
char* b="hello";
int d=a+b;
```

Output:

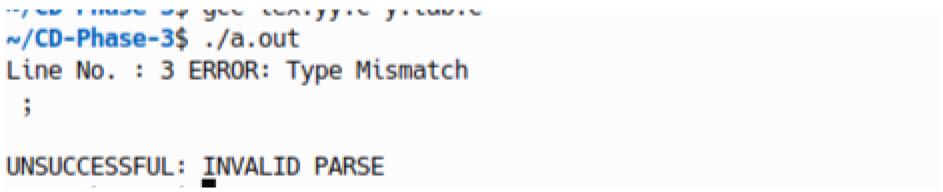

```
~/CD-Phase-3$ ./a.out
Line No. : 1 Invalid expression.
 ;

UNSUCCESSFUL: INVALID PARSE
Line No. : 3 Invalid expression.
 ;

UNSUCCESSFUL: INVALID PARSE
Line No. : 5 Invalid expression.
 ;

UNSUCCESSFUL: INVALID PARSE
Line No. : 5 ERROR: Type Mismatch
 ;

UNSUCCESSFUL: INVALID PARSE
```

19. Array ID has no subscript

Input:

```
int arr[5];
int x = arr;
```

Output:



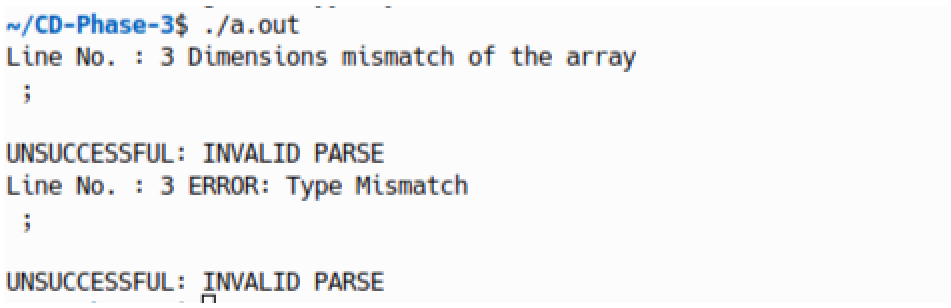
```
~/CD-Phase-3$ ./a.out
Line No. : 3 ERROR: Type Mismatch
 ;

UNSUCCESSFUL: INVALID PARSE
```

20. Single variable ID or function has subscript

Input:

```
int x;
int y = x[2];
```

Output:



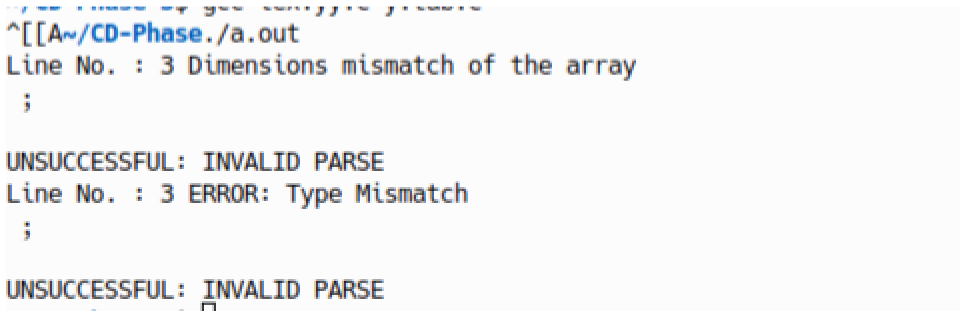
```
~/CD-Phase-3$ ./a.out
Line No. : 3 Dimensions mismatch of the array
 ;

UNSUCCESSFUL: INVALID PARSE
Line No. : 3 ERROR: Type Mismatch
 ;

UNSUCCESSFUL: INVALID PARSE
```

21. Expression in subscript not of type int

Input:

```
int arr[5];
float f = 2.5;
int x = arr[f];
```

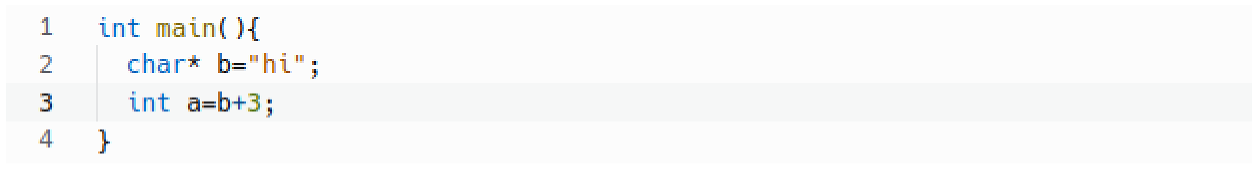
Output:



```
^[[A~/CD-Phase./a.out
Line No. : 3 Dimensions mismatch of the array
 ;

UNSUCCESSFUL: INVALID PARSE
Line No. : 3 ERROR: Type Mismatch
 ;

UNSUCCESSFUL: INVALID PARSE
```

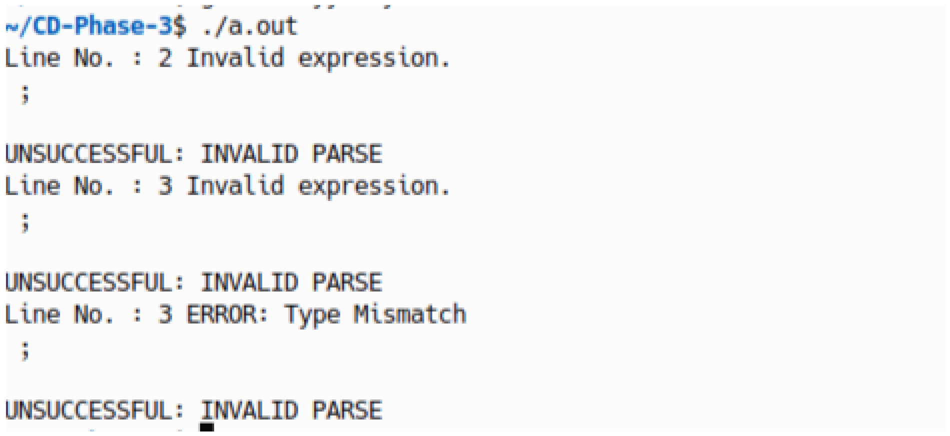
22. Expression on rhs of assigment and in arithmetic ops must be int

Input:



```
int main(){
  char* b="hi";
  int a=b+3;
}
```

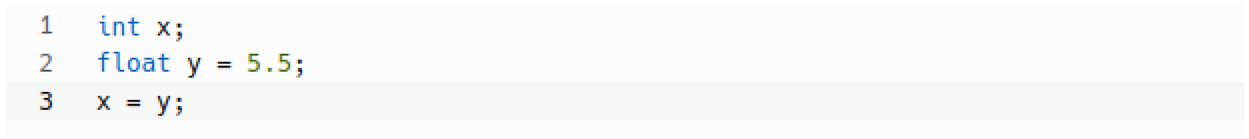
Output:



```
~/CD-Phase-3$ ./a.out
Line No. : 2 Invalid expression.
 ;

UNSUCCESSFUL: INVALID PARSE
Line No. : 3 Invalid expression.
 ;

UNSUCCESSFUL: INVALID PARSE
Line No. : 3 ERROR: Type Mismatch
 ;

UNSUCCESSFUL: INVALID PARSE
```

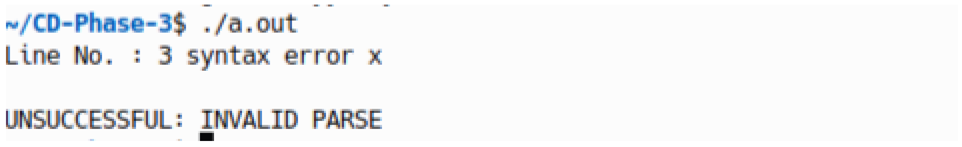
23. Expression on Ihs of assignment not single variable

Input:



```
int x;
float y = 5.5;
x = y;
```

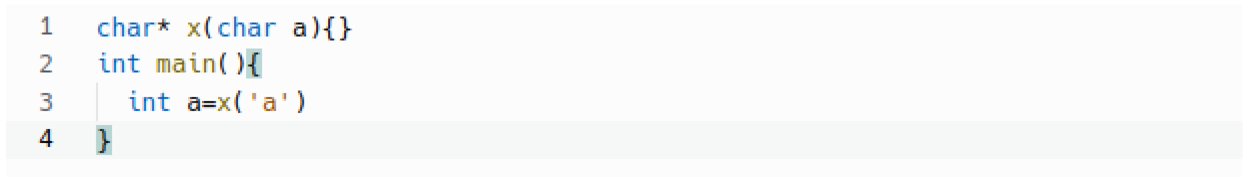
Output:



```
~/CD-Phase-3$ ./a.out
Line No. : 3 syntax error x

UNSUCCESSFUL: INVALID PARSE
```

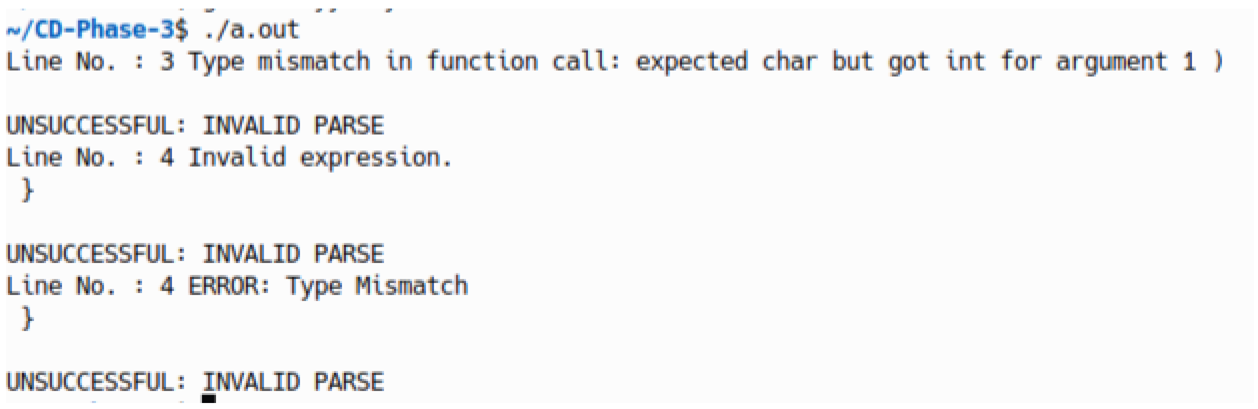
24. Return statement

Input:



```
char* x(char a){}
int main(){
  int a=x('a')
}
```

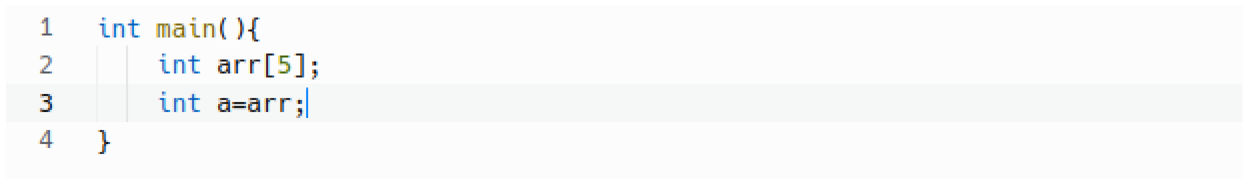
Output:



```
~/CD-Phase-3$ ./a.out
Line No. : 3 Type mismatch in function call: expected char but got int for argument 1 )

UNSUCCESSFUL: INVALID PARSE
Line No. : 4 Invalid expression.
 }

UNSUCCESSFUL: INVALID PARSE
Line No. : 4 ERROR: Type Mismatch
 }

UNSUCCESSFUL: INVALID PARSE
```

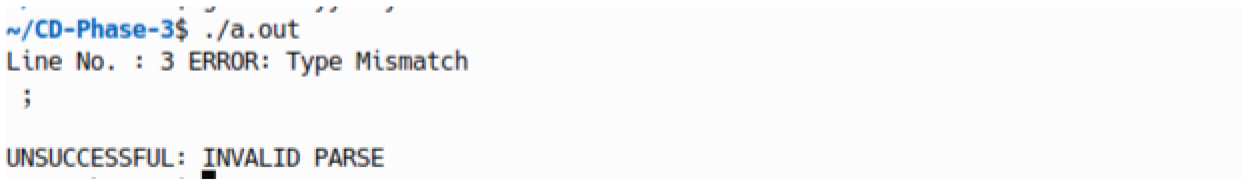
25. Expression has type void, function or array

Input:



```
int main(){
    int arr[5];
    int a=arr;
}
```

Output:



```
~/CD-Phase-3$ ./a.out
Line No. : 3 ERROR: Type Mismatch
 ;

UNSUCCESSFUL: INVALID PARSE
```