Intermediate Code Generation for the C Language

National Institute of Technology Karnataka Surathkal

Date: 4/11/2020

Submitted To: Dr. P. Santhi Thilagam

Submitted By:	1. Pranav Vigneshwar Kumar	181CO239
	2. Mohammed Rushad	181CO232
	3. Ankush Chandrashekar	181CO206
	4. Akshat Nambiar	181CO204

Contents Page No

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 third 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. After parser checks if the code is structured correctly, semantic analysis phase checks if that syntax structure constructed in the source program derives any meaning. The output of the syntax analysis phase is parse tree whereas that of Semantic phase is annotated parse tree. Semantic analysis is done by modifications in the parser code only. Some of the tasks performed during semantic analysis likr Scope Resolution, Type Checking, Array Bounds Checking etc. Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Thus, the annotated parse tree obtained from the previous semantic phase is converted in machine independent linear representation like three address code and is given as output in this project

Introduction

Intermediate Code Generation Phase

Intermediate Code Generation phase is the connection between the frontend and the theoretical code behind the design phases of the compiler. The end goal of a compiler is to convert programs into high-level language to run on a computer. Eventually, the program will have to be expressed as machine code which can run on the computer. Many compilers use a medium-level language as a stepping-stone between the high-level language and the very low-level machine code. Such stepping-stone languages are called intermediate code.

It provides lower abstraction from source level and maintain some high level information. Intermediate Code can be represented in many different formats depending whether it is language-specific or language-independent like 3-address code. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code. A three-address code has at most three address locations to calculate the expression. Hence, the intermediate code generator will divide any expression into sub-expressions and then generate the corresponding code. Most common independent intermediate representations are:
1. Postfix notation
2. Three Address Code
3. Syntax tree

Three Address Code

Three-address code is an intermediate code. It is used by the optimizing compilers. In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language. Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator. The compiler decides the order of operation given by three address code.
General representation: x = y op z
An address can be a name, constant or temporary.
1. Assignments x = y op z; x = op y.
2. Copy x = y.
3. Unconditional jump goto L.
4. Conditional jumps if x relop y goto L.
5. Parameters param x.
6. Function call y = call p

Yacc Script

Yacc stands for Yet Another Compiler-Compiler. Yacc is essentially a parser generator. Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. A function is then generated by Yacc to control the input process. This function is called the parser which calls the lexical analyzer to get a stream of tokens from the input. Based on the input structure rules, called grammar rules, the tokens are organized. When one of these rules has been recognized, then user code supplied for this rule, an action, is invoked. Actions have the ability to return values and make use of the values of other actions. Yacc is written in portable C. The class of specifications accepted is a very general one, LALR(1) grammars with disambiguating rules. The structure of our yacc script is divided into three sections, separated by lines that contain only two percent signs, as follows:

DECLARATIONS

%%

RULES

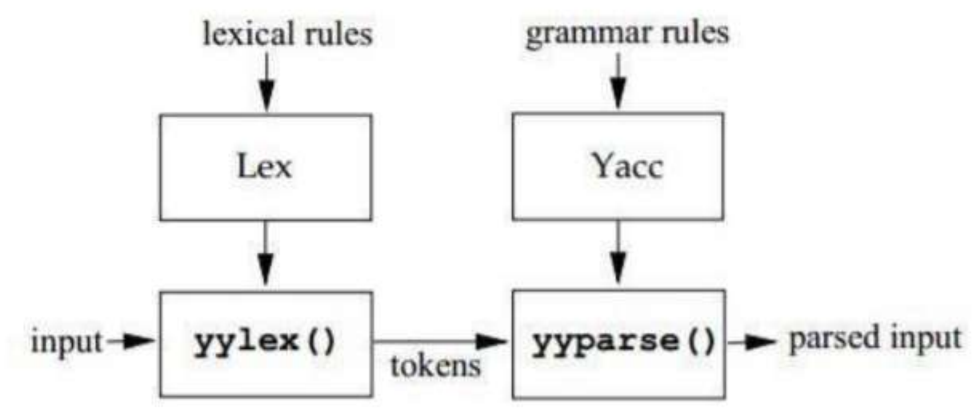
%%

AUXILIARY FUNCTIONS

The **Declarations Section** defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied directly into the generated source file. We also define all parameters related to the parser here, specifications like using leftmost derivations or rightmost derivations, precedence, left and right associativity are declared here, data types and tokens which will be used by the lexical analyzer are also declared at this stage.

The **Rules Section** contains the entire grammar which is used for deciding if the input text is legally correct according to the specifications of the language. Yacc uses these rules for reducing the token stream received from the lexical analysis stage. All rules are linked to each other from the start state.

Yacc generates C code for the rules specified in the Rules section and places this code into a single function called yyparse(). The **Auxiliary Functions Section** contains C statements and functions that are copied directly to the generated source file. These statements usually contain code called by the different rules. This section essentially allows the programmer to add to the generated source code.

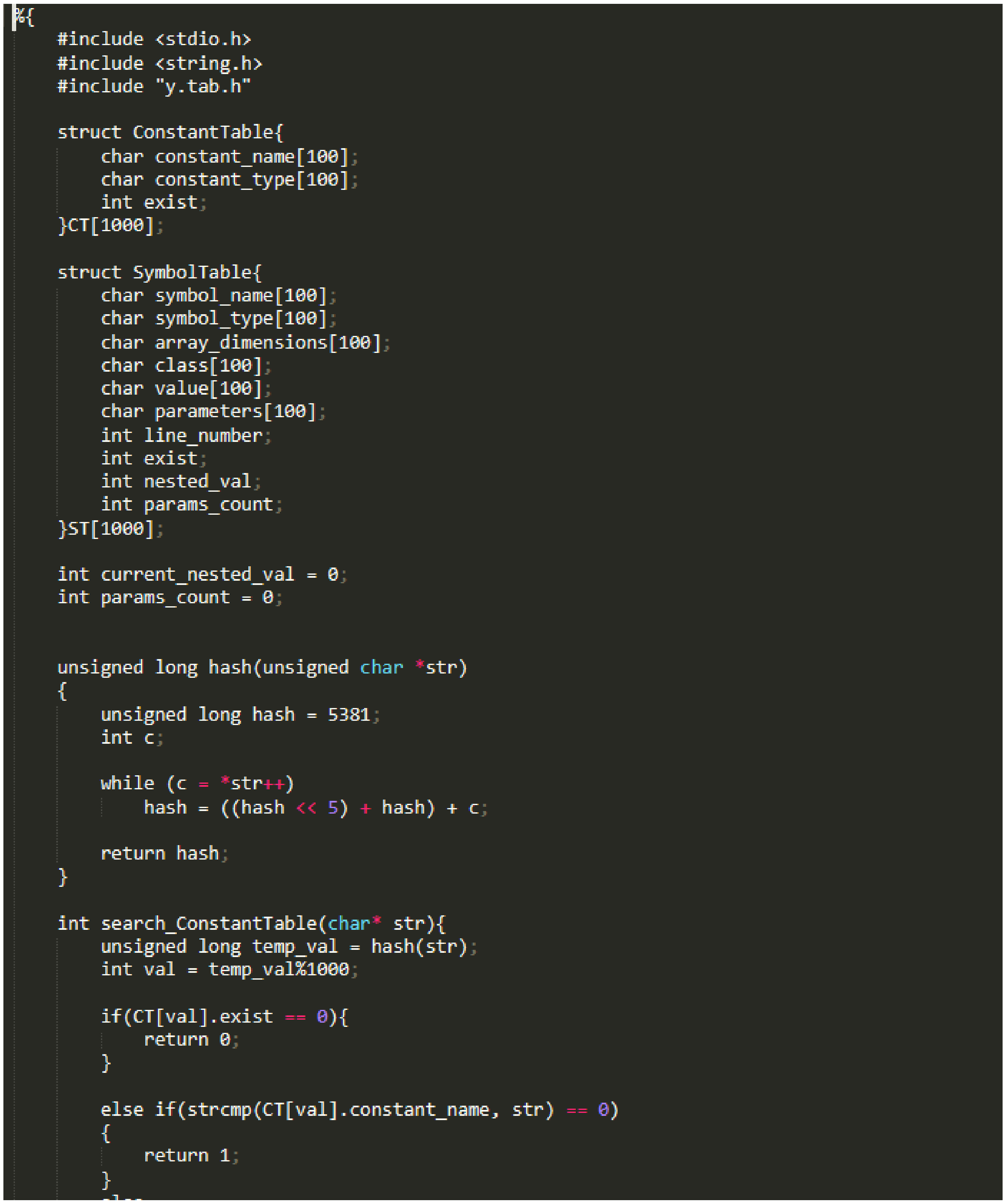


C Program

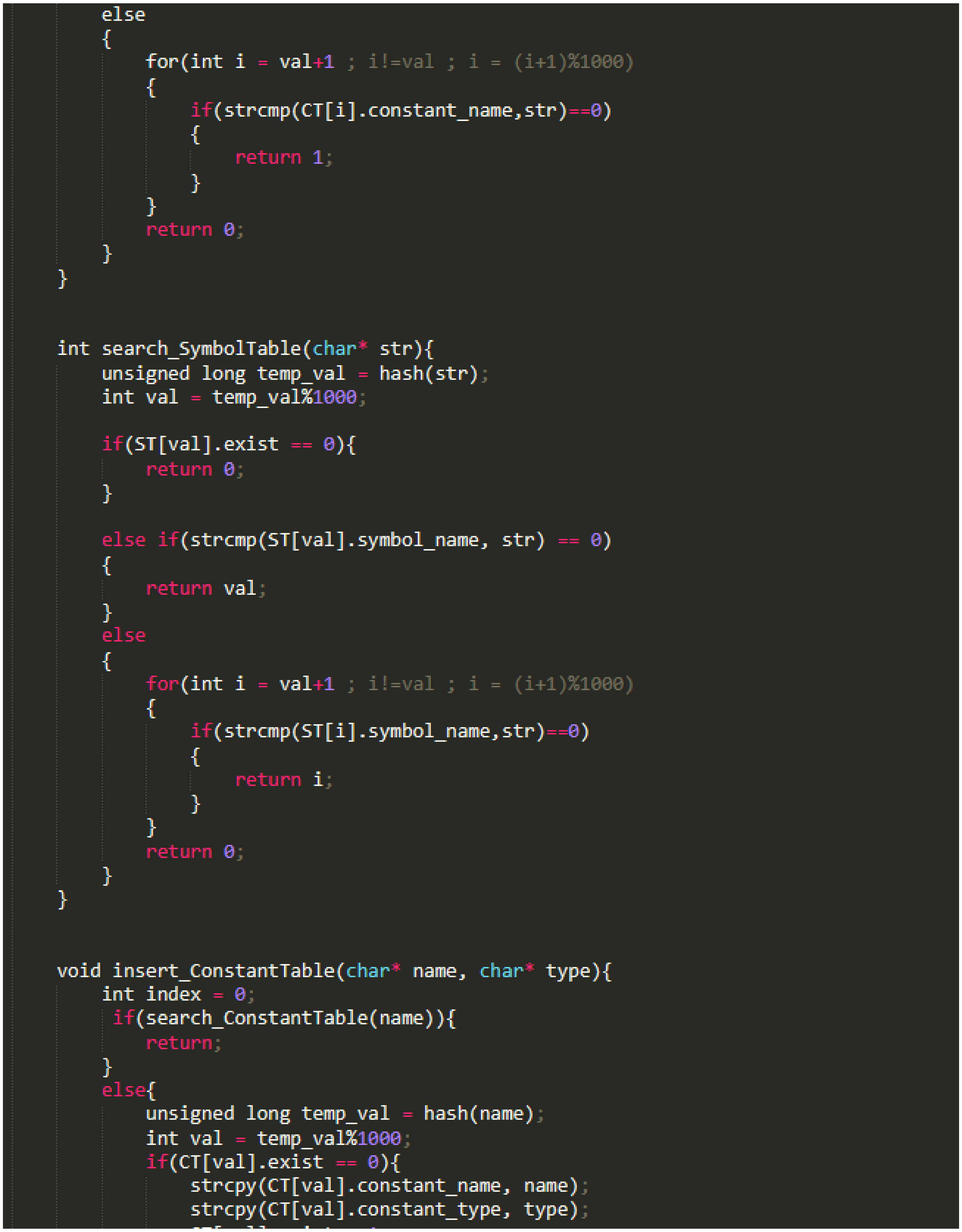
The parser takes C source files as input for parsing. The input file is specified in the auxiliary functions section of the yacc script. The workflow for testing the parser is as follows:

1. Compile the yacc script using the yacc tool
   ```
   $ yacc -d semantic.y
   ```
2. Compile the flex script using the flex tool
   ```
   $ lex lexer.l
   ```
3. The first two steps generate lex.yy.c, y.tab.c, and y.tab.h. The header file is included in lexer.l file. Then, lex.yy.c and y.tab.c are compiled together.
   ```
   $ gcc lex.yy.c y.tab.c
   ```
4. Run the generated executable file
   ```
   $ ./a.out
   ```
5. Run the generated executable file
   ```
   $ ./a.out
   ```

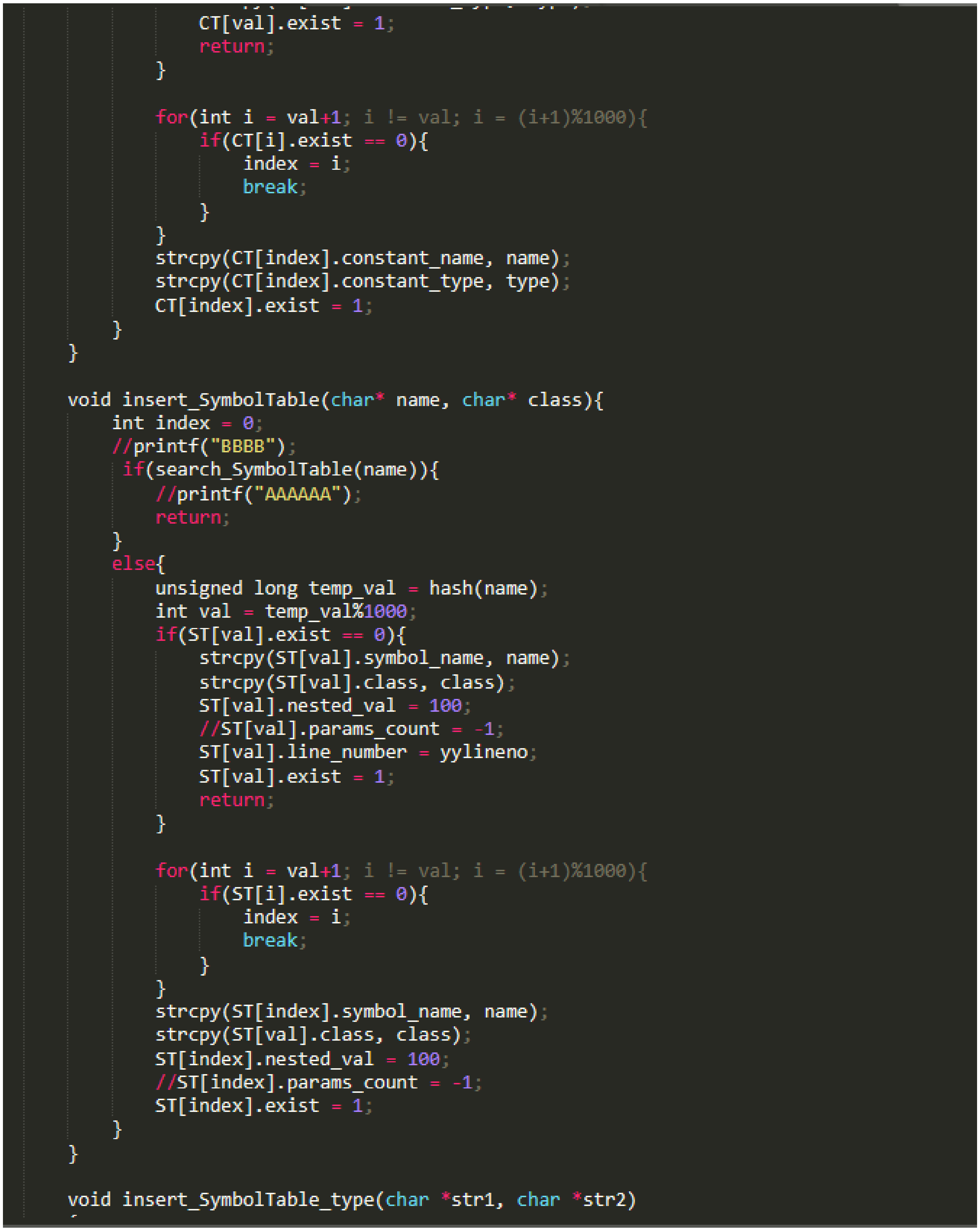
Design of Programs
Updated Lexical Analyzer Code



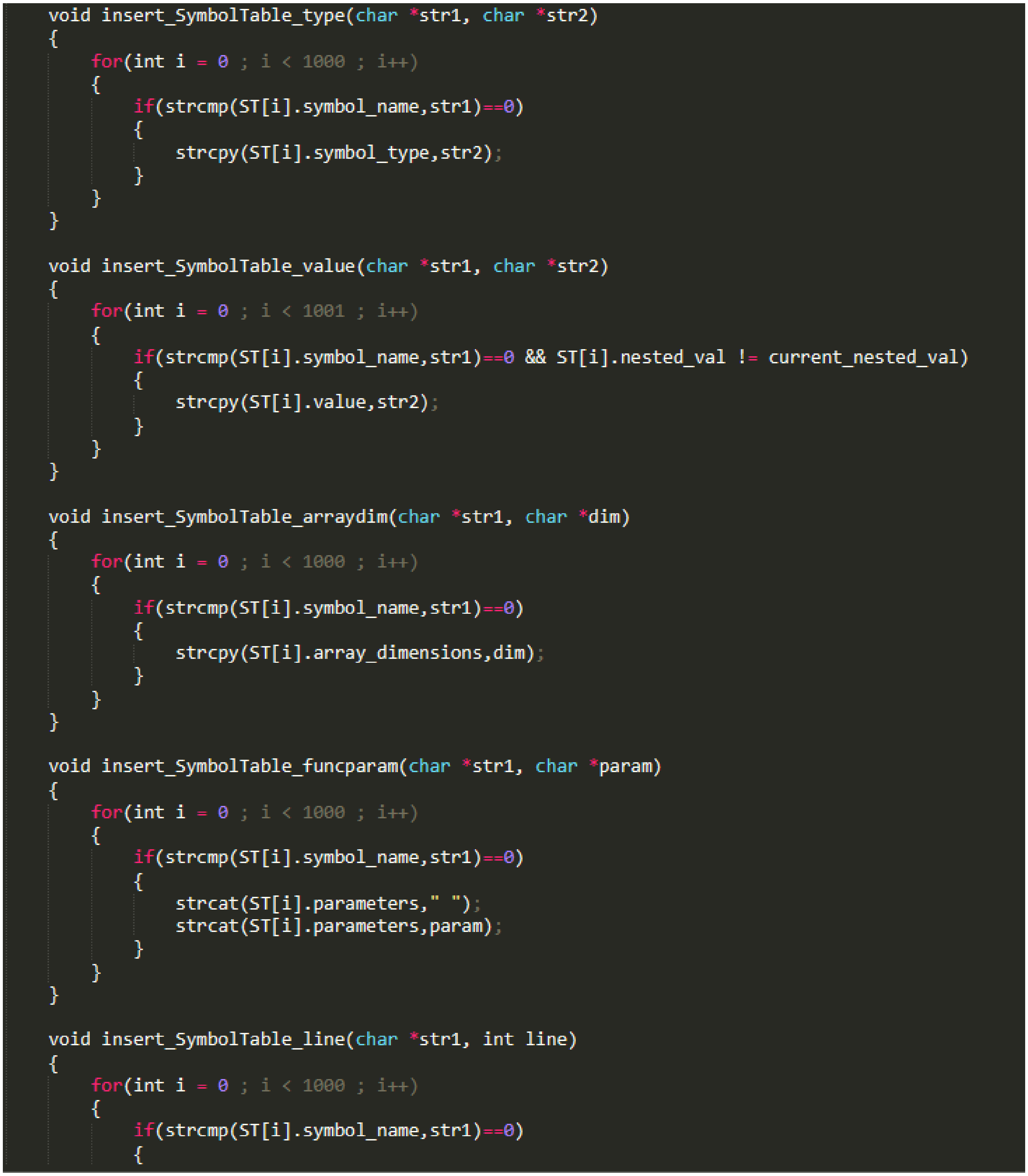
```
%{
    #include <stdio.h>
    #include <string.h>
    #include "y.tab.h"

    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 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;
        }
```

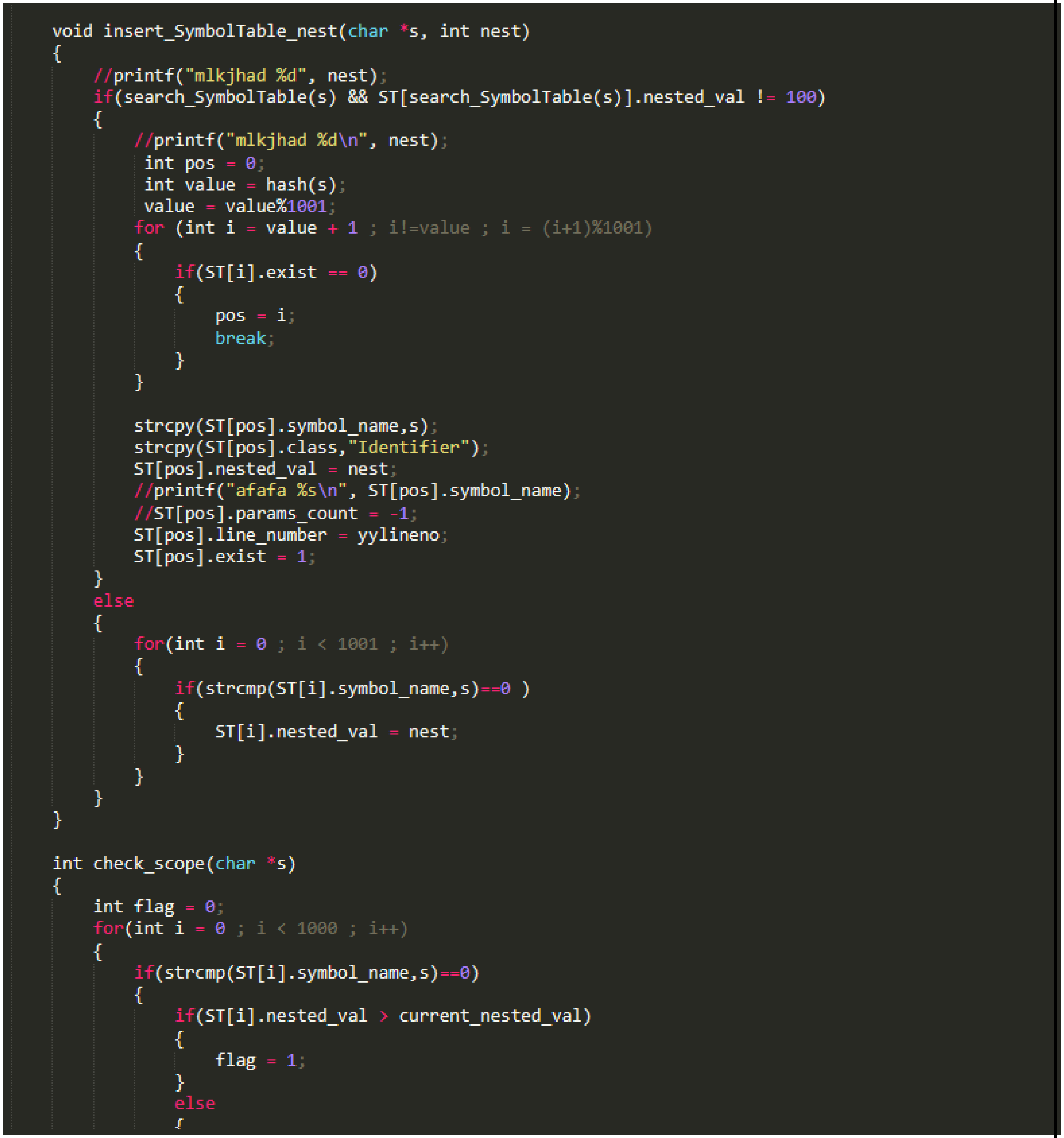


```
        else
        {
            for(int i = val+1 ; i!=val ; i = (i+1)%1000)
            {
                if(strcmp(CT[i].constant_name,str)==0)
                {
                    return 1;
                }
            }
            return 0;
        }
    }


    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);
```

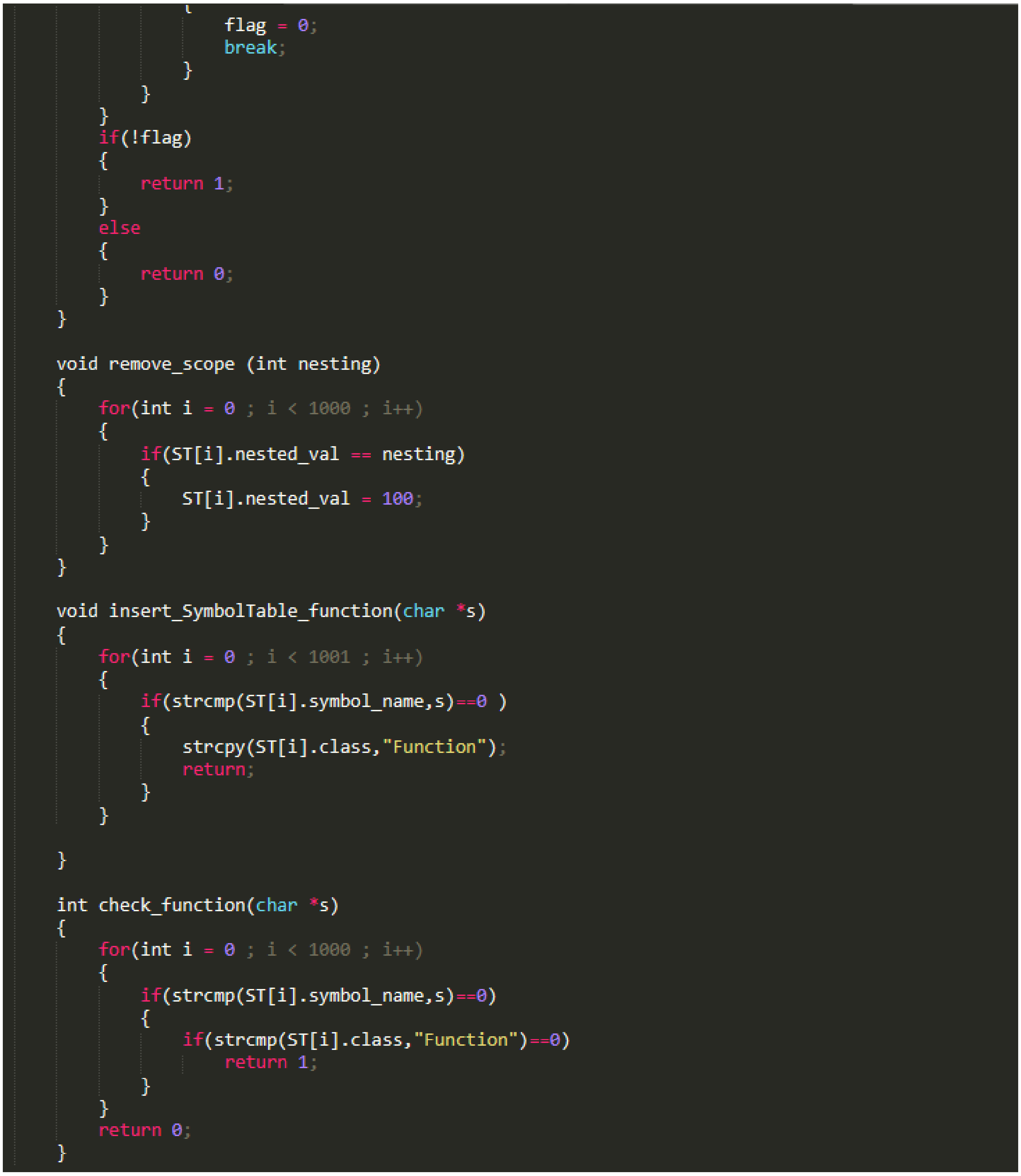


```
            CT[val].exist = 1;
            return;
        }

        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;
    //printf("BBBB");
     if(search_SymbolTable(name)){
        //printf("AAAAAA");
        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].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].params_count = -1;
        ST[index].exist = 1;
    }
}

void insert_SymbolTable_type(char *str1, char *str2)
```

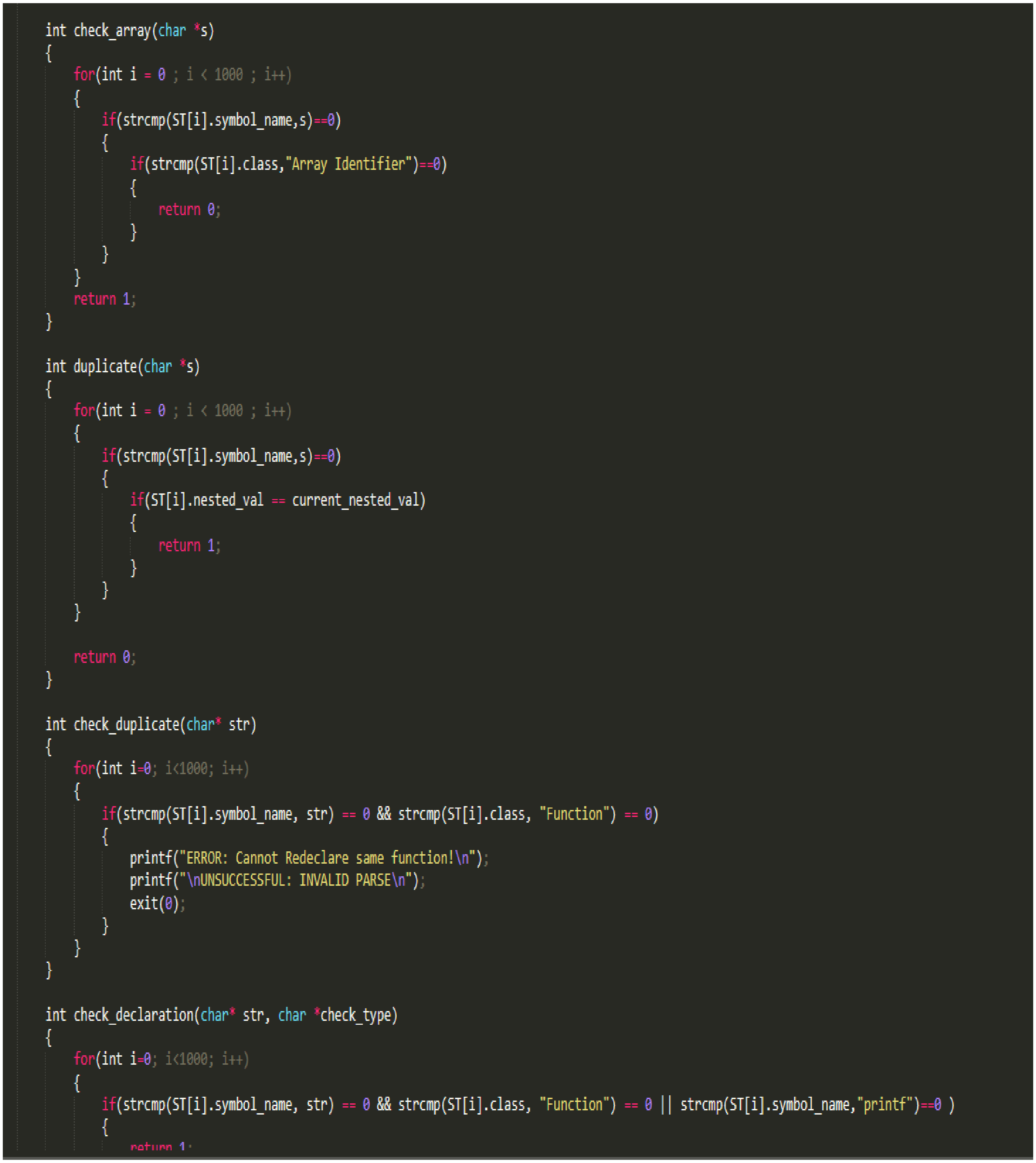

```
void insert_SymbolTable_type(char *str1, char *str2)
{
    for(int i = 0 ; i < 1000 ; i++)
    {
        if(strcmp(ST[i].symbol_name,str1)==0)
        {
            strcpy(ST[i].symbol_type,str2);
        }
    }
}

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)
        {
```



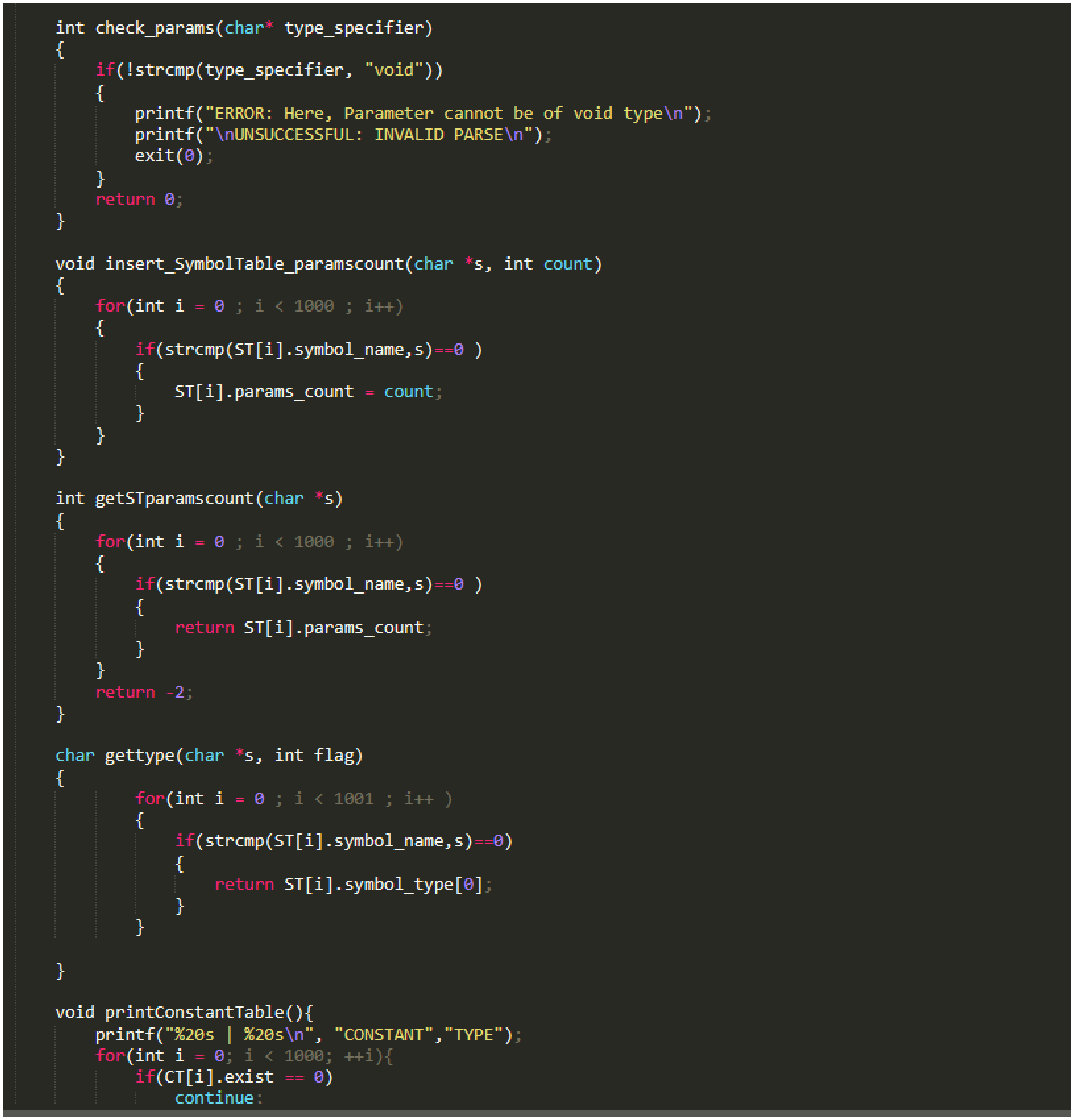
```
void insert_SymbolTable_nest(char *s, int nest)
{
    //printf("mlkjhad %d", nest);
    if(search_SymbolTable(s) && ST[search_SymbolTable(s)].nested_val != 100)
    {
        //printf("mlkjhad %d\n", nest);
         int pos = 0;
         int value = hash(s);
         value = value%1001;
        for (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;
        //printf("afafa %s\n", ST[pos].symbol_name);
        //ST[pos].params_count = -1;
        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;
            }
        }
    }
}

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
```



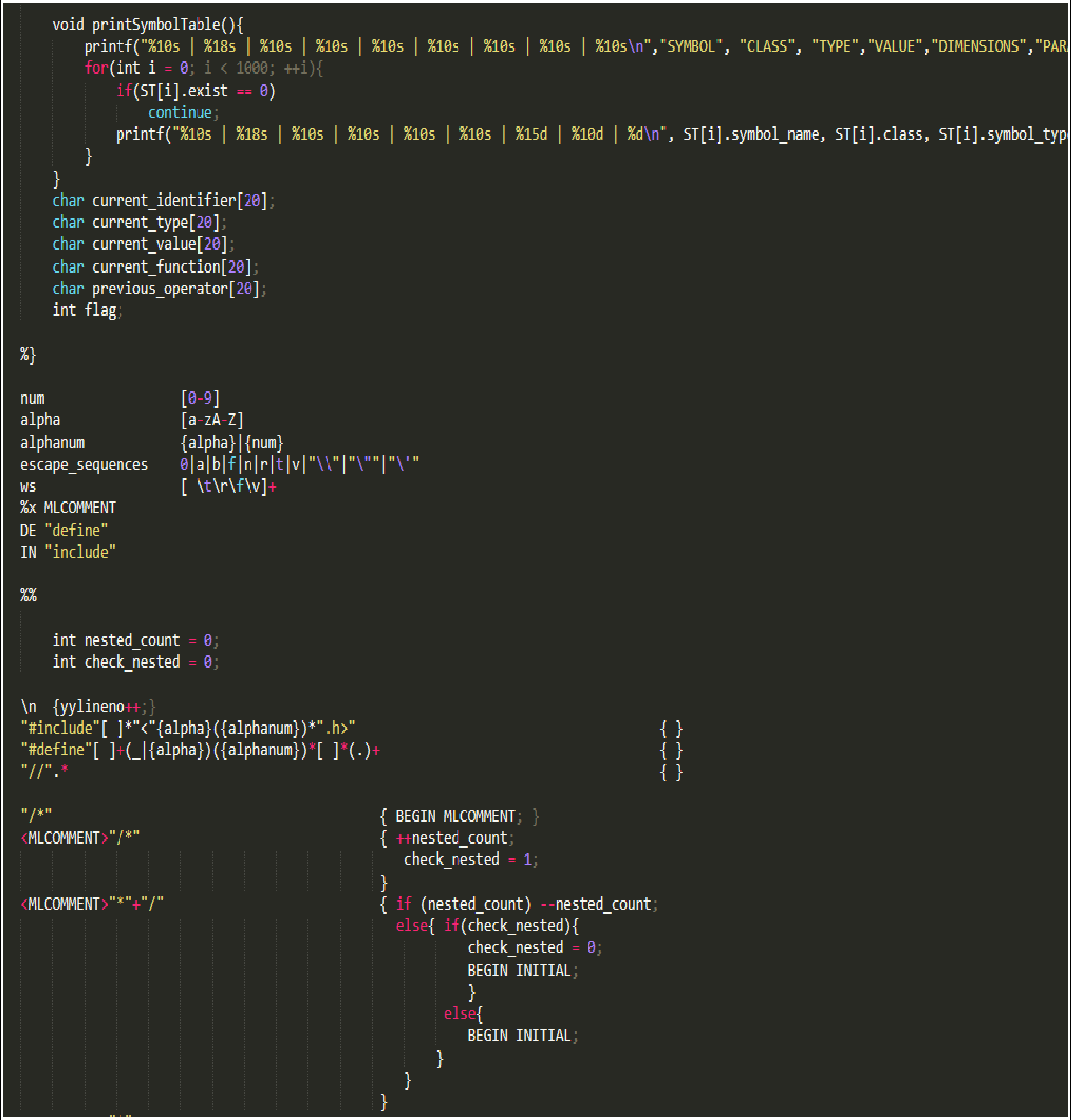
```
                    flag = 0;
                    break;
                }
            }
        }
        if(!flag)
        {
            return 1;
        }
        else
        {
            return 0;
        }
    }

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

    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");
                return;
            }
        }

    }

    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;
    }
```



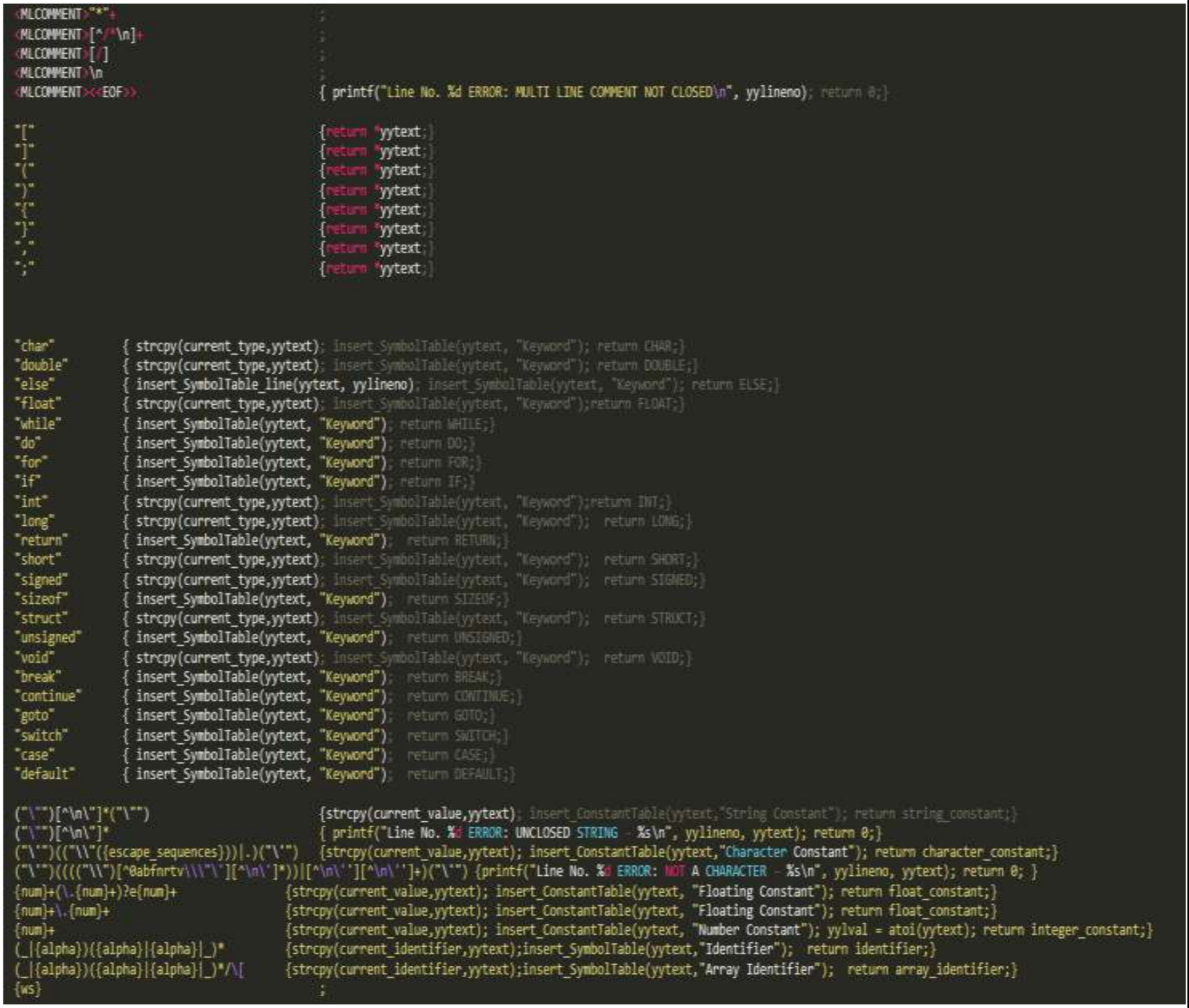
```c
int check_array(char *s)
{
    for(int i = 0 ; i < 1000 ; i++)
    {
        if(strcmp(ST[i].symbol_name,s)==0)
        {
            if(strcmp(ST[i].class,"Array Identifier")==0)
            {
                return 0;
            }
        }
    }
    return 1;
}

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)
        {
            printf("ERROR: Cannot Redeclare same function!\n");
            printf("\nUNSUCCESSFUL: INVALID PARSE\n");
            exit(0);
        }
    }
}

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 || strcmp(ST[i].symbol_name,"printf")==0 )
        {
            return 1;
```



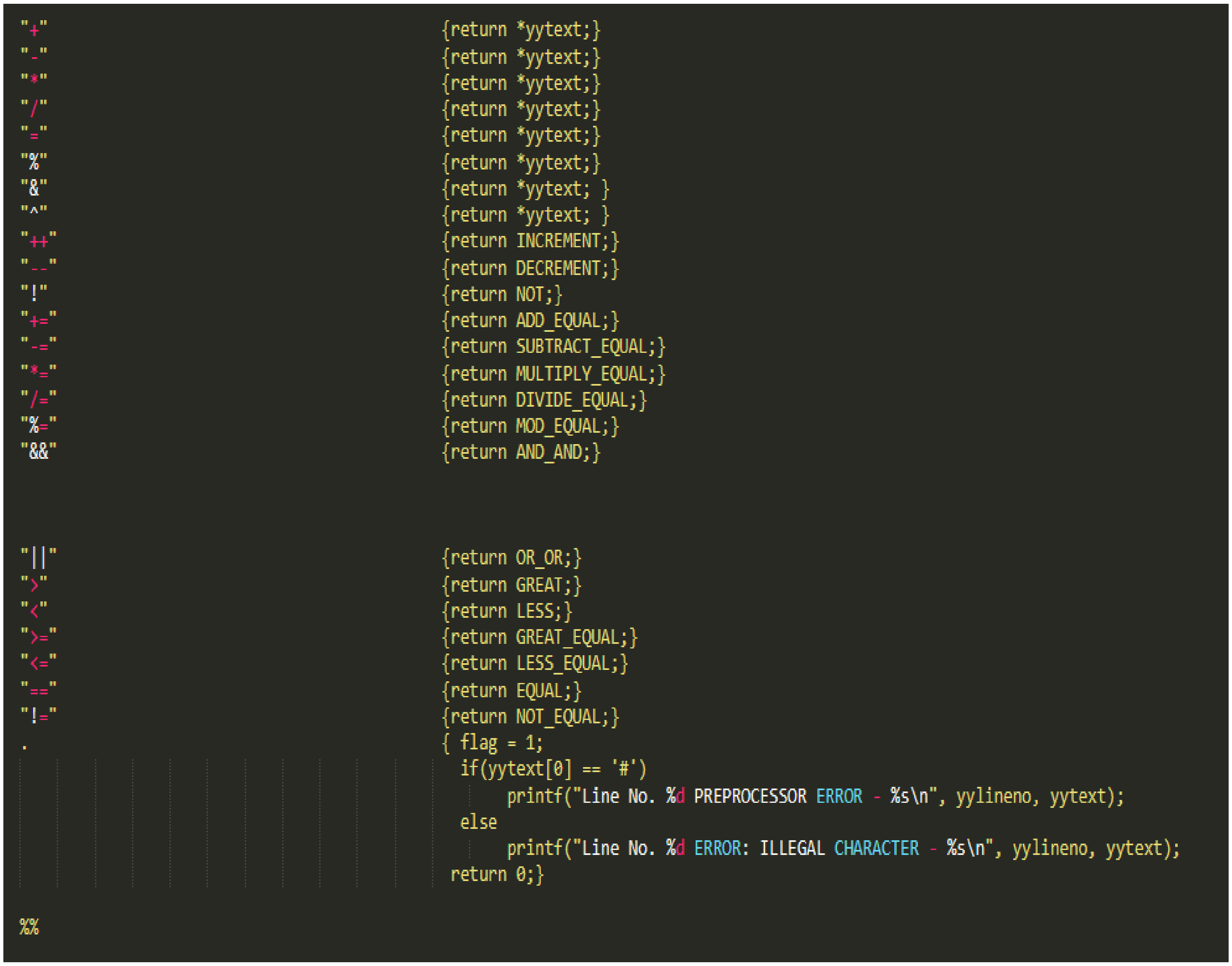
```
int check_params(char* type_specifier)
{
    if(!strcmp(type_specifier, "void"))
    {
        printf("ERROR: Here, Parameter cannot be of void type\n");
        printf("\nUNSUCCESSFUL: INVALID PARSE\n");
        exit(0);
    }
    return 0;
}

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;
}

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;
```



```
    void printSymbolTable(){
        printf("%10s | %18s | %10s | %10s | %10s | %10s | %10s | %10s | %10s\n","SYMBOL", "CLASS", "TYPE","VALUE","DIMENSIONS","PAR
        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_typ
        }
    }
    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  {yylineno++;}
"#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;
                                                    }
                                                }
                                             }
```

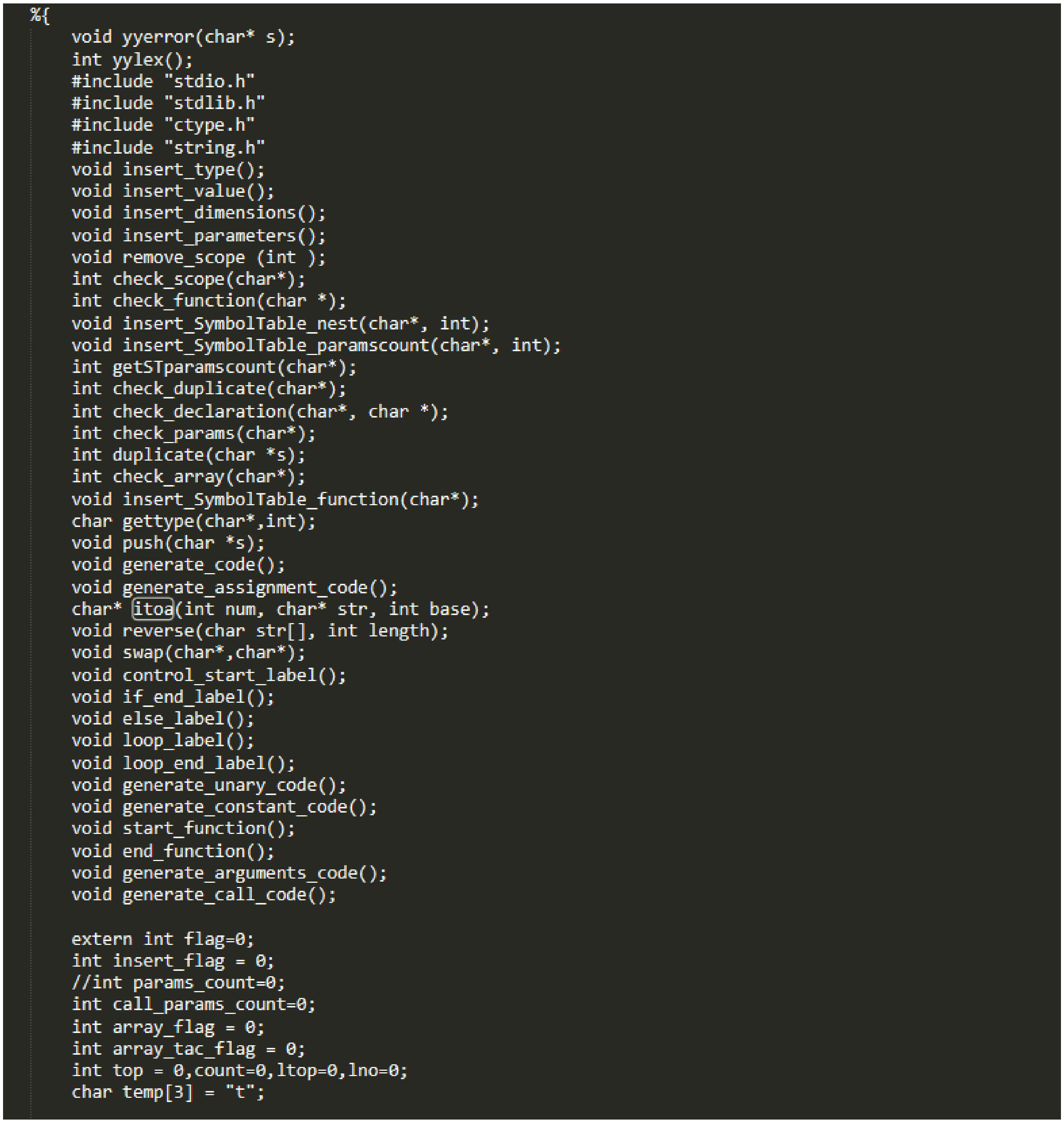


```
<MLCOMMENT>"*"+                          ;
<MLCOMMENT>[^/*\n]+                      ;
<MLCOMMENT>[/]                           ;
<MLCOMMENT>\n                            ;
<MLCOMMENT><<EOF>>                       { printf("Line No. %d ERROR: MULTI LINE COMMENT NOT CLOSED\n", yylineno); return 0;}

"["                                      {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_line(yytext, yylineno); insert_SymbolTable(yytext, "Keyword"); return ELSE;}
"float"         { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");return FLOAT;}
"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}|{alpha}|_)*       {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}                                     ;
```

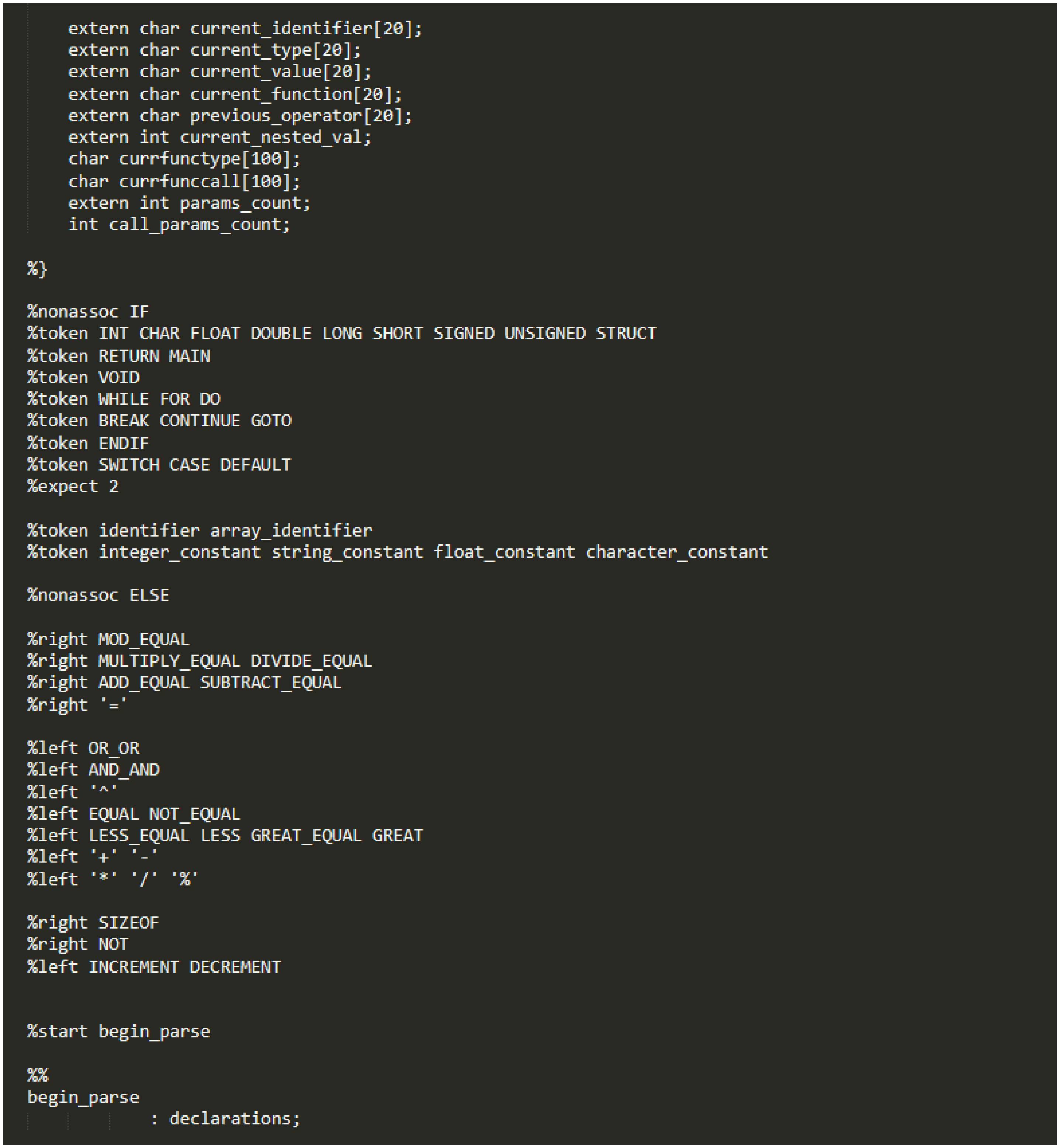


```
"+"                                         {return *yytext;}
"-"                                         {return *yytext;}
"*"                                         {return *yytext;}
"/"                                         {return *yytext;}
"="                                         {return *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;}

%%
```

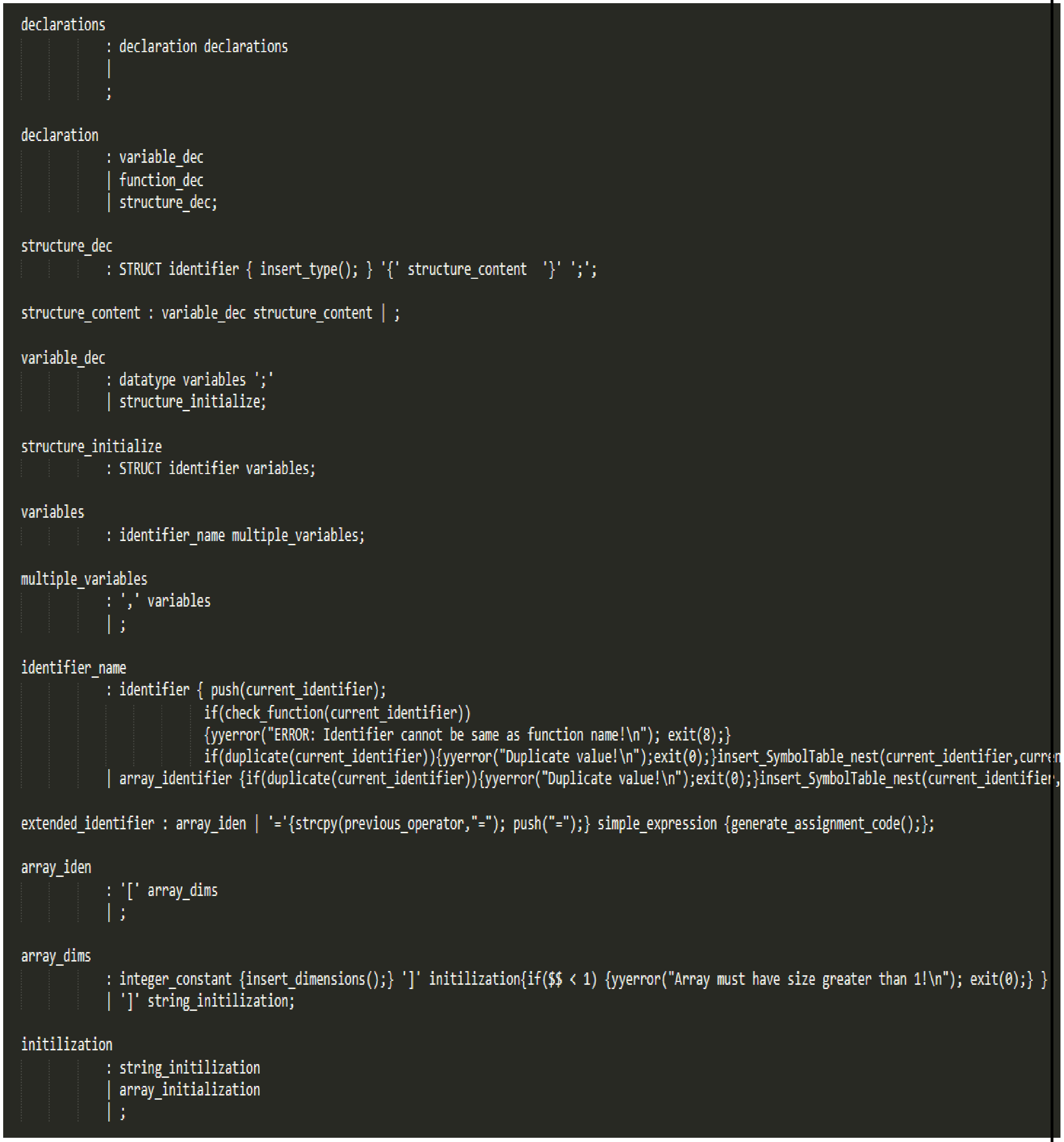
Updated Parser Code



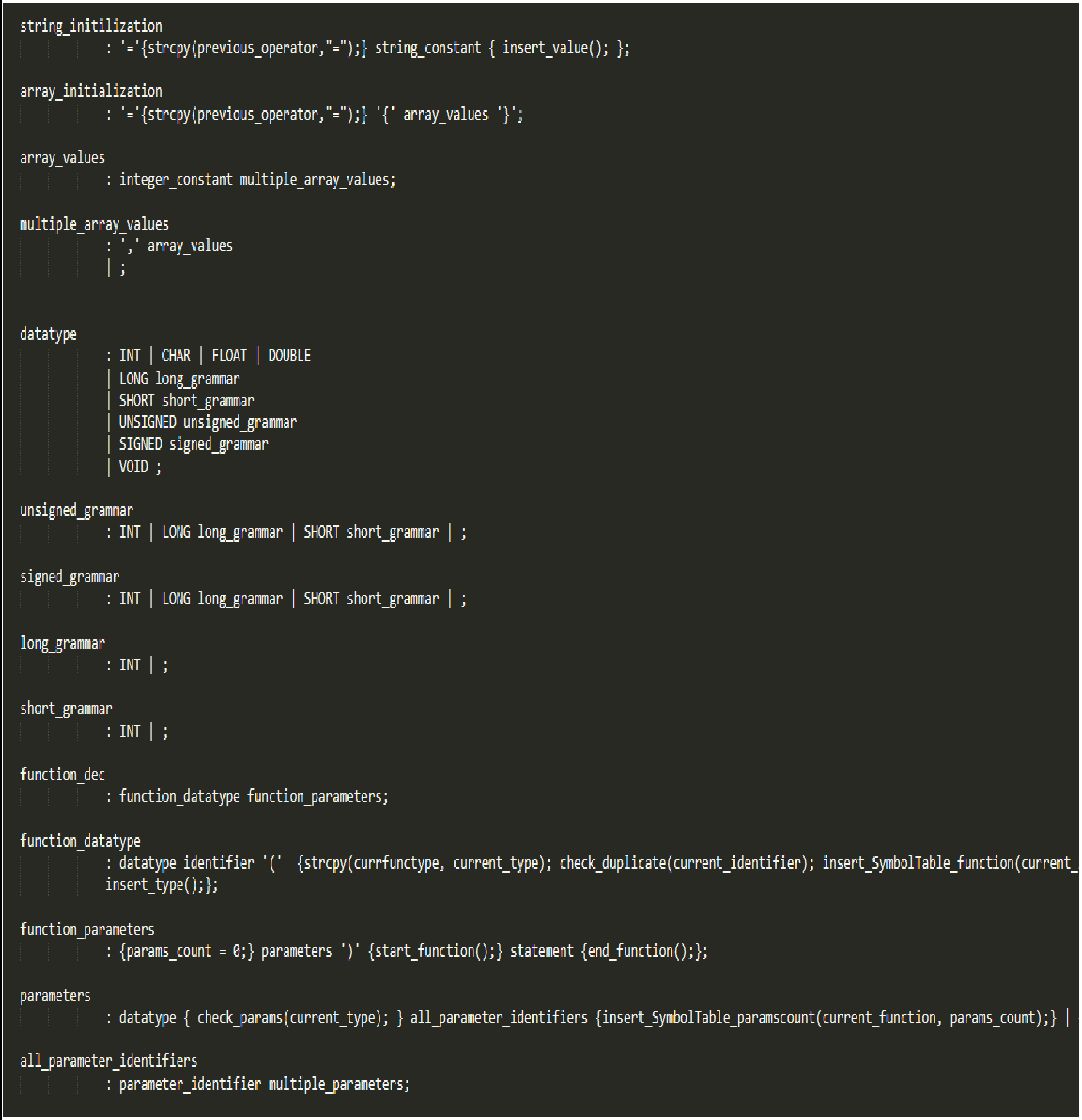
```
%{
    void yyerror(char* s);
    int yylex();
    #include "stdio.h"
    #include "stdlib.h"
    #include "ctype.h"
    #include "string.h"
    void insert_type();
    void insert_value();
    void insert_dimensions();
    void insert_parameters();
    void remove_scope (int );
    int check_scope(char*);
    int check_function(char *);
    void insert_SymbolTable_nest(char*, int);
    void insert_SymbolTable_paramscount(char*, int);
    int getSTparamscount(char*);
    int check_duplicate(char*);
    int check_declaration(char*, char *);
    int check_params(char*);
    int duplicate(char *s);
    int check_array(char*);
    void insert_SymbolTable_function(char*);
    char gettype(char*,int);
    void push(char *s);
    void generate_code();
    void generate_assignment_code();
    char* itoa(int num, char* str, int base);
    void reverse(char str[], int length);
    void swap(char*,char*);
    void control_start_label();
    void if_end_label();
    void else_label();
    void loop_label();
    void loop_end_label();
    void generate_unary_code();
    void generate_constant_code();
    void start_function();
    void end_function();
    void generate_arguments_code();
    void generate_call_code();

    extern int flag=0;
    int insert_flag = 0;
    //int params_count=0;
    int call_params_count=0;
    int array_flag = 0;
    int array_tac_flag = 0;
    int top = 0,count=0,ltop=0,lno=0;
    char temp[3] = "t";
```



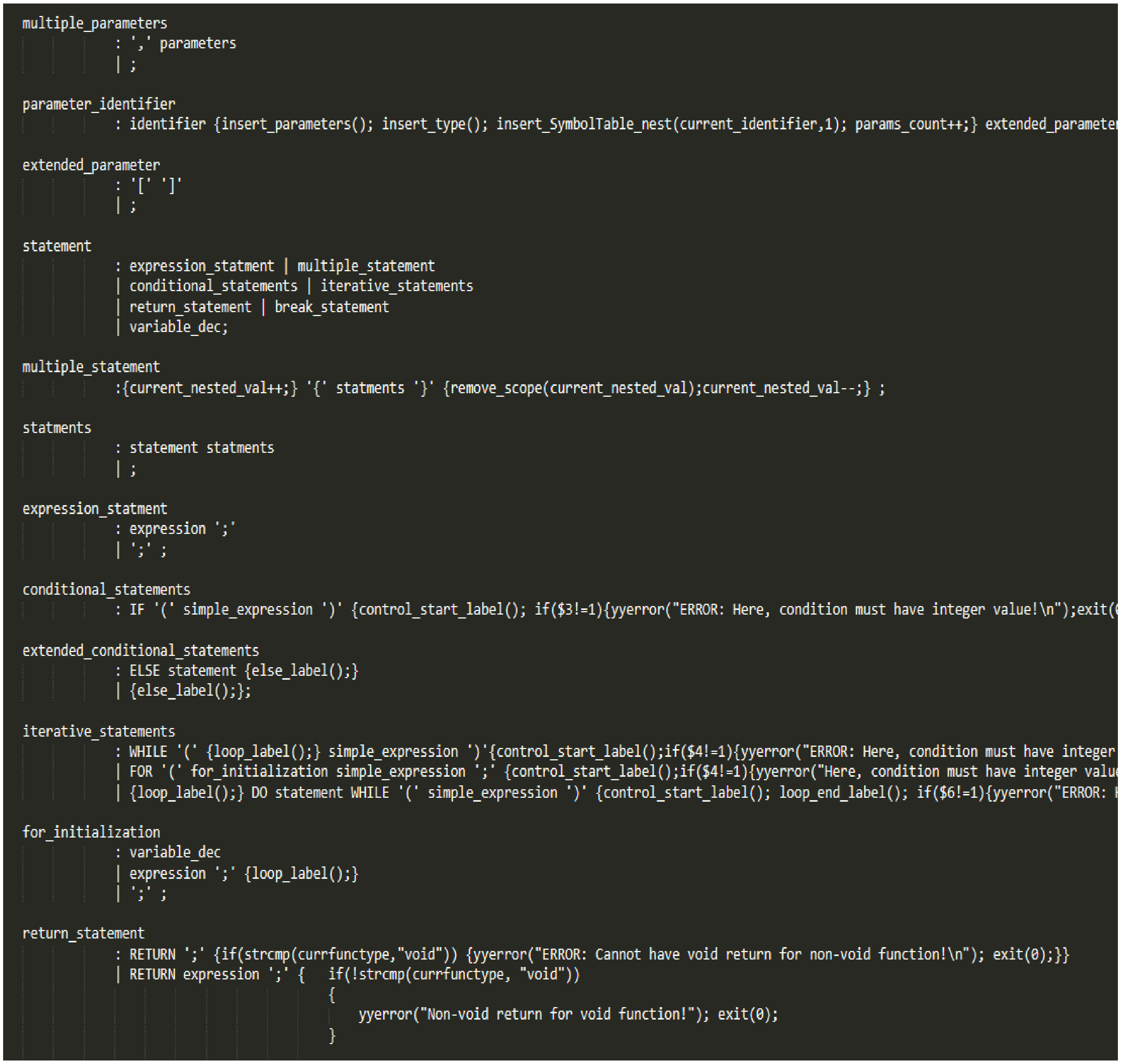
```
    extern char current_identifier[20];
    extern char current_type[20];
    extern char current_value[20];
    extern char current_function[20];
    extern char previous_operator[20];
    extern int current_nested_val;
    char currfunctype[100];
    char currfunccall[100];
    extern int params_count;
    int call_params_count;

%}

%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
%token RETURN MAIN
%token VOID
%token WHILE FOR DO
%token BREAK CONTINUE GOTO
%token ENDIF
%token SWITCH CASE DEFAULT
%expect 2

%token identifier array_identifier
%token 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;
```



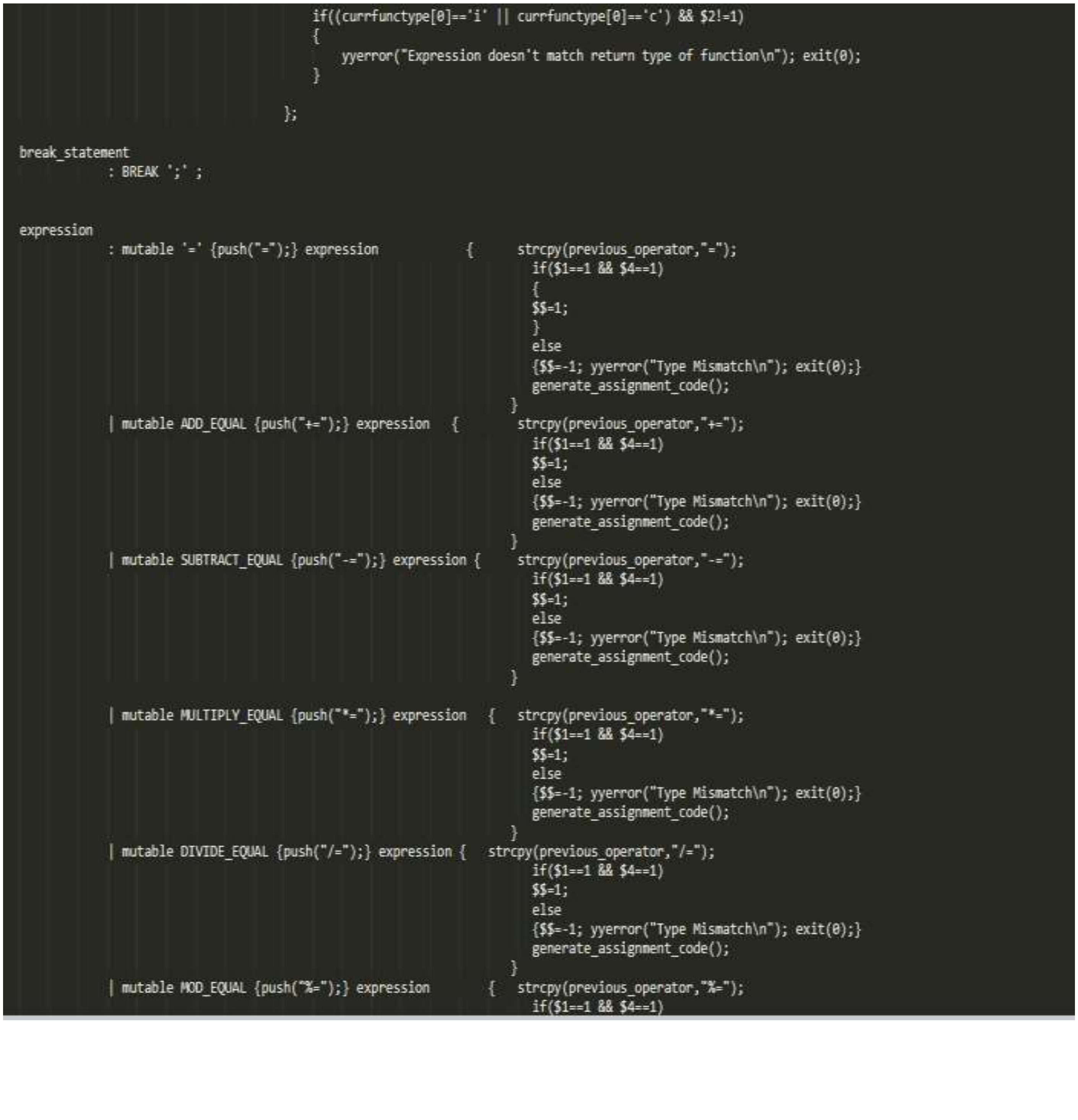
```
declarations
            : declaration declarations
            |
            ;

declaration
            : variable_dec
            | function_dec
            | structure_dec;

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

structure_content : variable_dec structure_content | ;

variable_dec
            : datatype variables ';'
            | structure_initialize;

structure_initialize
            : STRUCT identifier variables;

variables
            : identifier_name multiple_variables;

multiple_variables
            : ',' variables
            | ;

identifier_name
            : identifier { push(current_identifier);
                        if(check_function(current_identifier))
                        {yyerror("ERROR: Identifier cannot be same as function name!\n"); exit(8);}
                        if(duplicate(current_identifier)){yyerror("Duplicate value!\n");exit(0);}insert_SymbolTable_nest(current_identifier,curre
            | array_identifier {if(duplicate(current_identifier)){yyerror("Duplicate value!\n");exit(0);}insert_SymbolTable_nest(current_identifier,

extended_identifier : array_iden | '='{strcpy(previous_operator,"="); push("=");} simple_expression {generate_assignment_code();};

array_iden
            : '[' array_dims
            | ;

array_dims
            : integer_constant {insert_dimensions();} ']' initilization{if($$ < 1) {yyerror("Array must have size greater than 1!\n"); exit(0);} }
            | ']' string_initilization;

initilization
            : string_initilization
            | array_initialization
            | ;
```



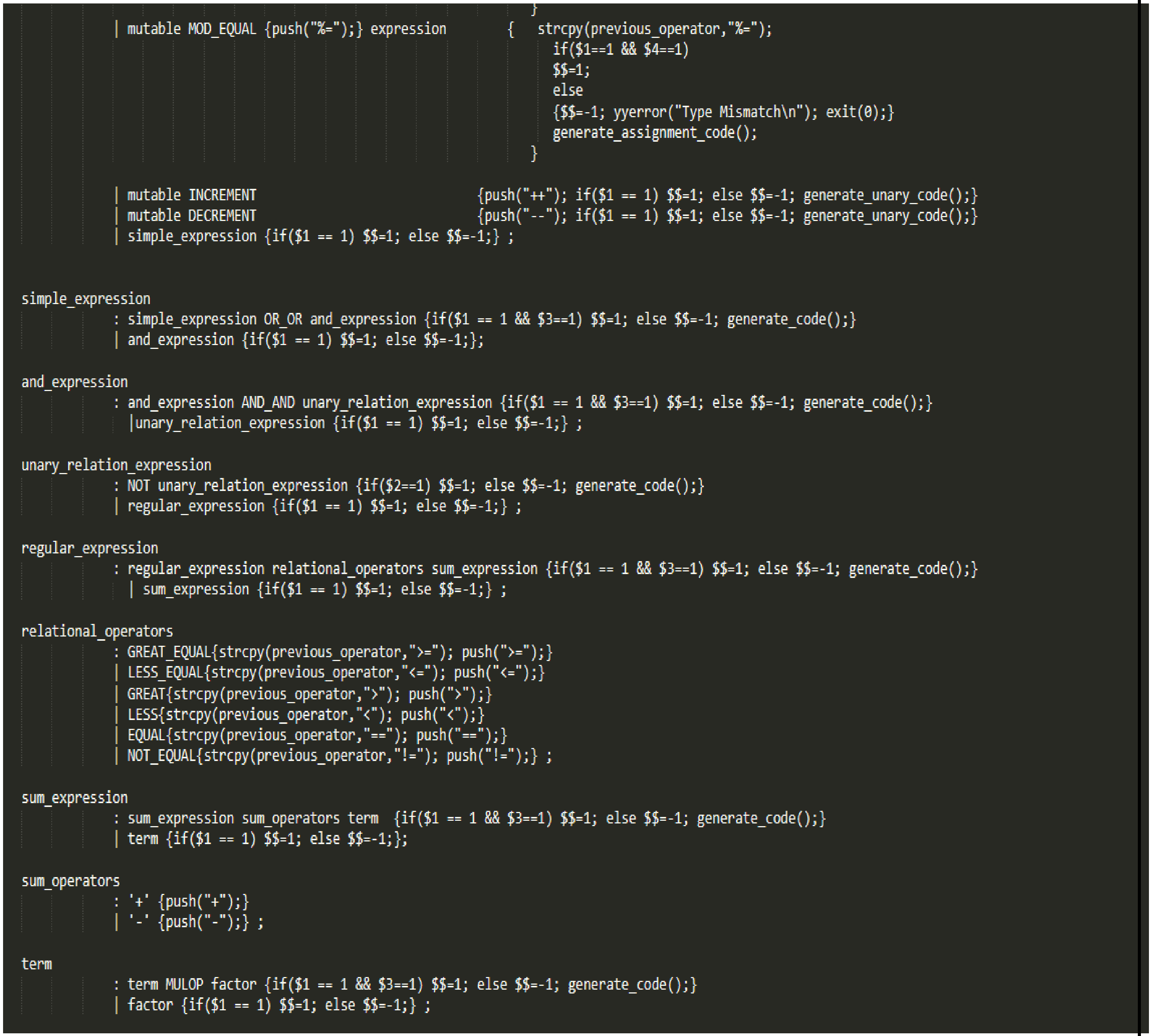
```
string_initilization
            : '='{strcpy(previous_operator,"=");} string_constant { insert_value(); };

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

array_values
            : integer_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 ;

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;

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

function_parameters
            : {params_count = 0;} parameters ')' {start_function();} statement {end_function();};

parameters
            : datatype { check_params(current_type); } all_parameter_identifiers {insert_SymbolTable_paramscount(current_function, params_count);} |

all_parameter_identifiers
            : parameter_identifier multiple_parameters;
```



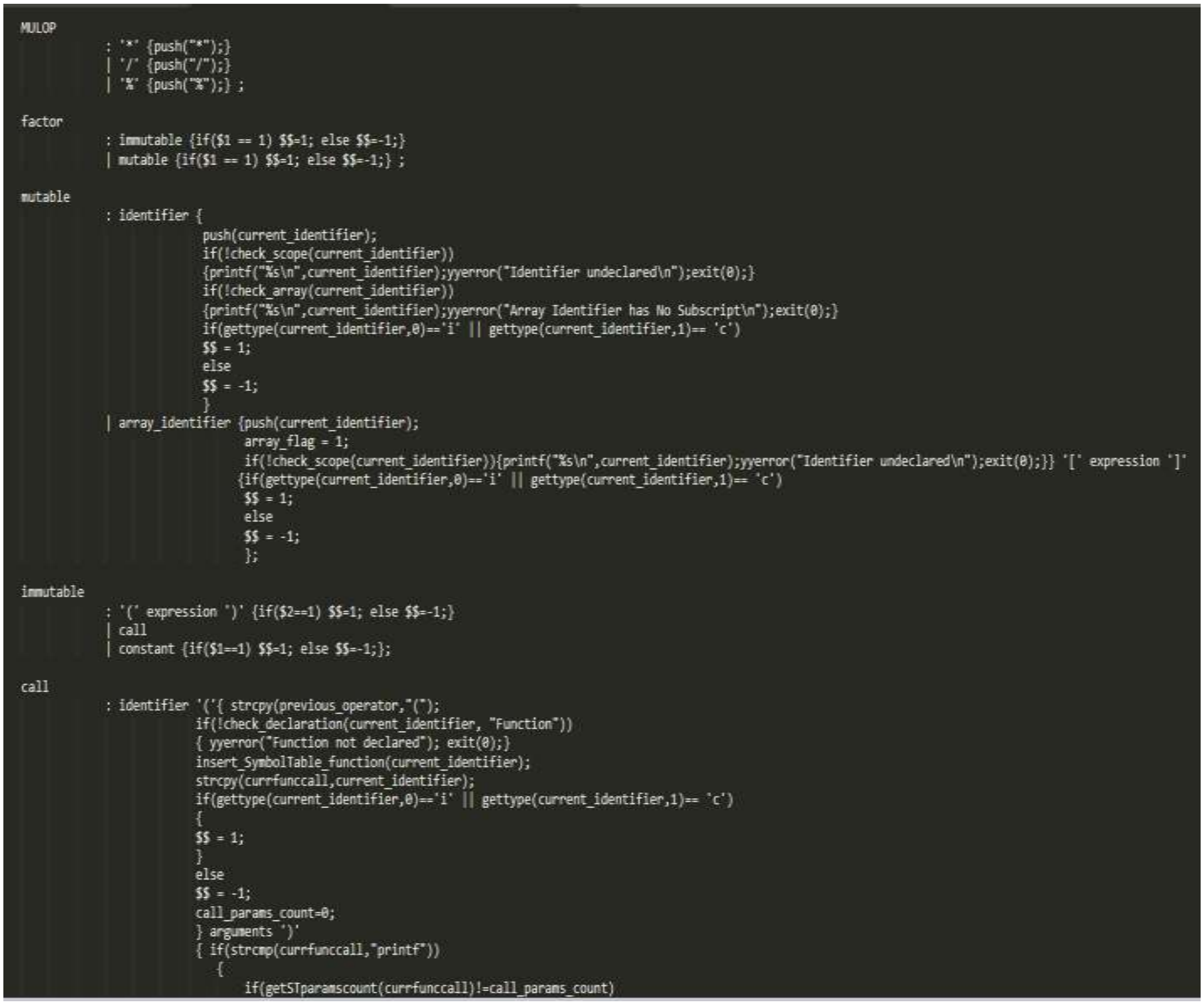
```
multiple_parameters
            : ',' parameters
            | ;

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

extended_parameter
            : '[' ']'
            | ;

statement
            : expression_statment | multiple_statement
            | conditional_statements | iterative_statements
            | return_statement | break_statement
            | variable_dec;

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

statments
            : statement statments
            | ;

expression_statment
            : expression ';'
            | ';' ;

conditional_statements
            : IF '(' simple_expression ')' {control_start_label(); if($3!=1){yyerror("ERROR: Here, condition must have integer value!\n");exit(

extended_conditional_statements
            : ELSE statement {else_label();}
            | {else_label();};

iterative_statements
            : WHILE '(' {loop_label();} simple_expression ')'{control_start_label();if($4!=1){yyerror("ERROR: Here, condition must have integer
            | FOR '(' for_initialization simple_expression ';' {control_start_label();if($4!=1){yyerror("Here, condition must have integer valu
            | {loop_label();} DO statement WHILE '(' simple_expression ')' {control_start_label(); loop_end_label(); if($6!=1){yyerror("ERROR: 

for_initialization
            : variable_dec
            | expression ';' {loop_label();}
            | ';' ;

return_statement
            : RETURN ';' {if(strcmp(currfunctype,"void")) {yyerror("ERROR: Cannot have void return for non-void function!\n"); exit(0);}}
            | RETURN expression ';' {   if(!strcmp(currfunctype, "void"))
                                        {
                                            yyerror("Non-void return for void function!"); exit(0);
                                        }
```



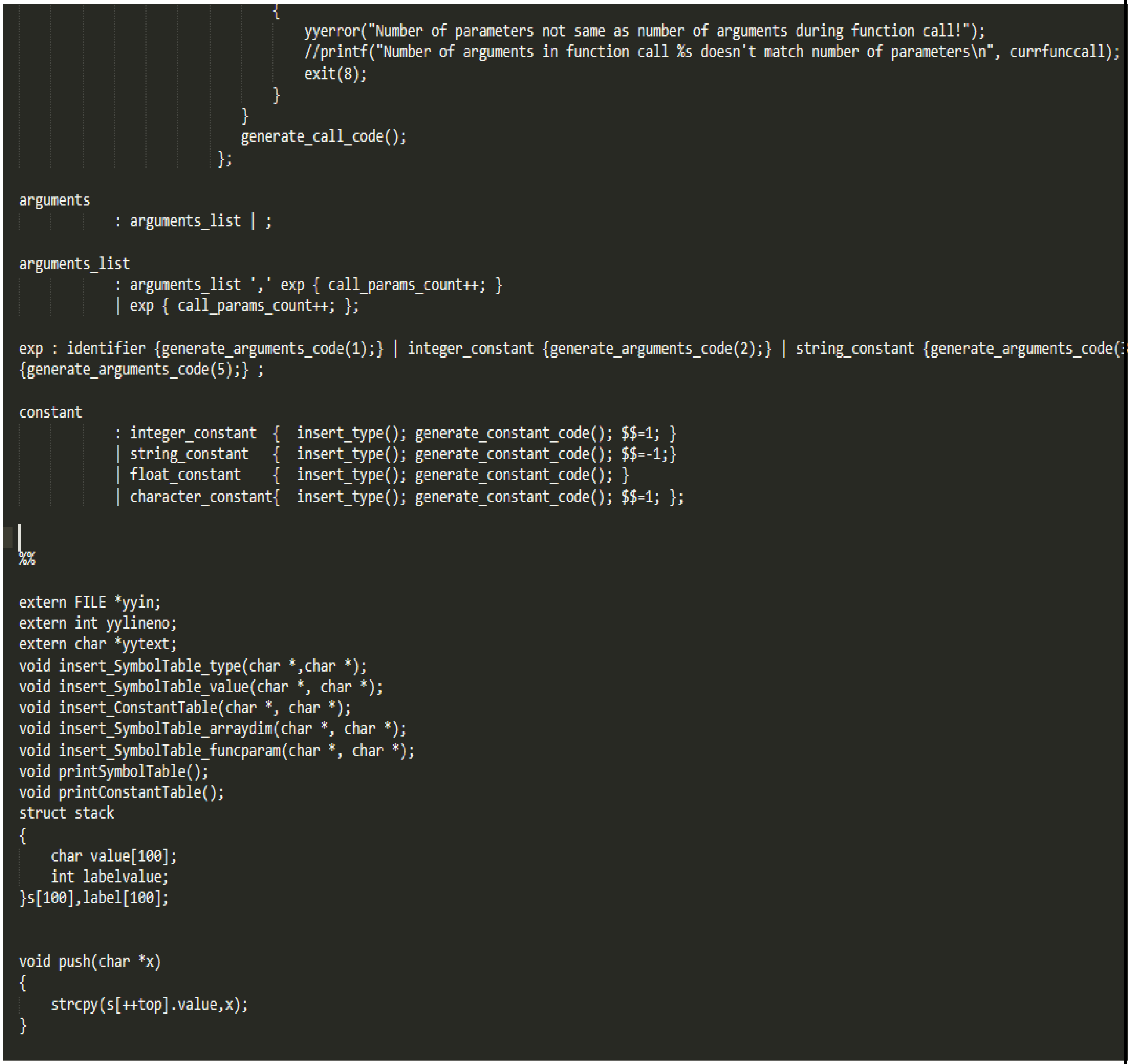
```
                                        if((currfunctype[0]=='i' || currfunctype[0]=='c') && $2!=1)
                                        {
                                            yyerror("Expression doesn't match return type of function\n"); exit(0);
                                        }

                                    };

break_statement
            : BREAK ';' ;


expression
            : mutable '=' {push("=");} expression           {      strcpy(previous_operator,"=");
                                                                     if($1==1 && $4==1)
                                                                     {
                                                                     $$=1;
                                                                     }
                                                                     else
                                                                     {$$=-1; yyerror("Type Mismatch\n"); exit(0);}
                                                                     generate_assignment_code();
                                                                   }
            | mutable ADD_EQUAL {push("+=");} expression    {       strcpy(previous_operator,"+=");
                                                                     if($1==1 && $4==1)
                                                                     $$=1;
                                                                     else
                                                                     {$$=-1; yyerror("Type Mismatch\n"); exit(0);}
                                                                     generate_assignment_code();
                                                                   }
            | mutable SUBTRACT_EQUAL {push("-=");} expression {     strcpy(previous_operator,"-=");
                                                                     if($1==1 && $4==1)
                                                                     $$=1;
                                                                     else
                                                                     {$$=-1; yyerror("Type Mismatch\n"); exit(0);}
                                                                     generate_assignment_code();
                                                                   }

            | mutable MULTIPLY_EQUAL {push("*=");} expression   {   strcpy(previous_operator,"*=");
                                                                     if($1==1 && $4==1)
                                                                     $$=1;
                                                                     else
                                                                     {$$=-1; yyerror("Type Mismatch\n"); exit(0);}
                                                                     generate_assignment_code();
                                                                   }
            | mutable DIVIDE_EQUAL {push("/=");} expression {   strcpy(previous_operator,"/=");
                                                                     if($1==1 && $4==1)
                                                                     $$=1;
                                                                     else
                                                                     {$$=-1; yyerror("Type Mismatch\n"); exit(0);}
                                                                     generate_assignment_code();
                                                                   }
            | mutable MOD_EQUAL {push("%=");} expression        {   strcpy(previous_operator,"%=");
                                                                     if($1==1 && $4==1)
```



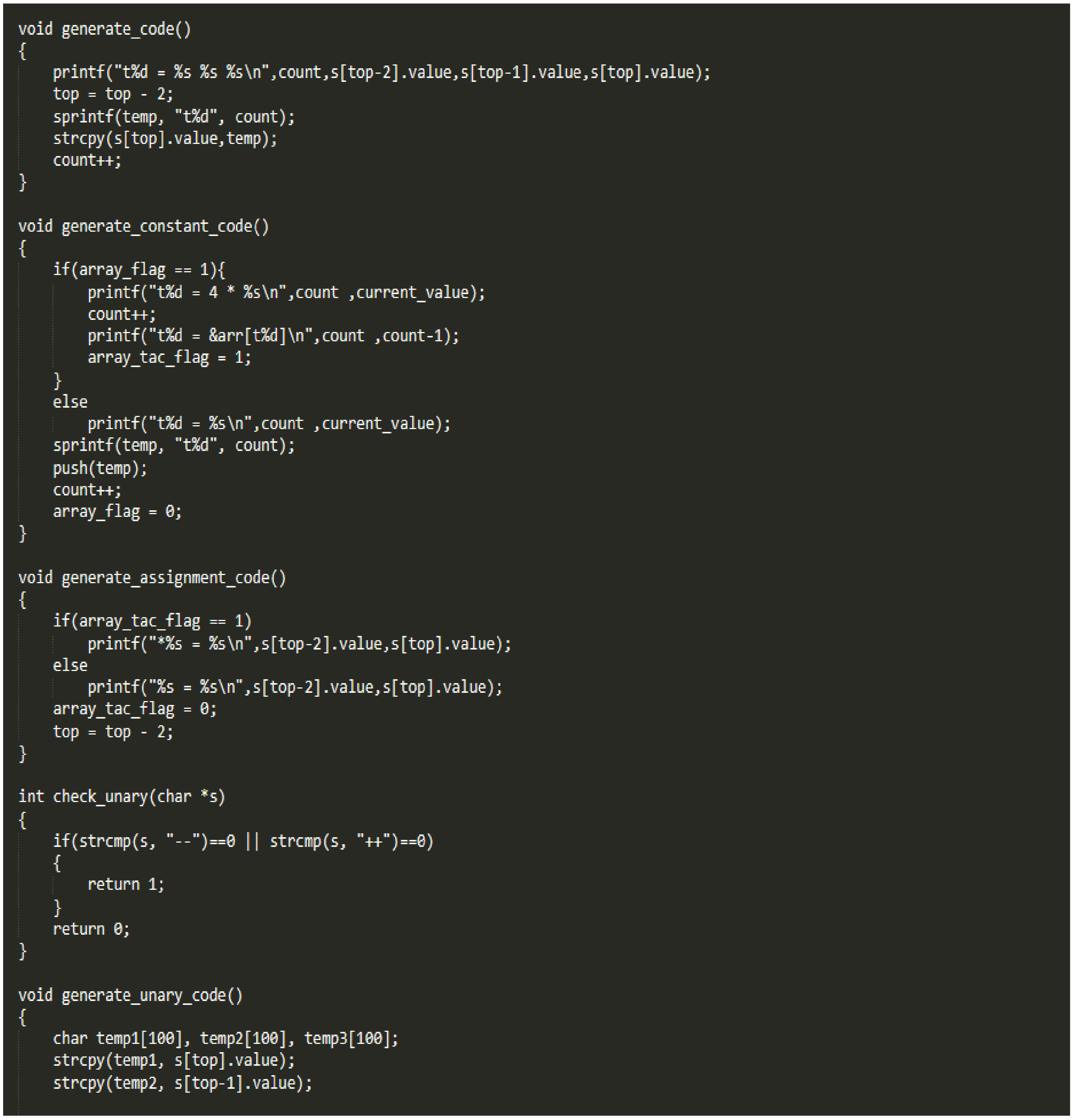
```
                                                                    }
            | mutable MOD_EQUAL {push("%=");} expression        {   strcpy(previous_operator,"%=");
                                                                      if($1==1 && $4==1)
                                                                      $$=1;
                                                                      else
                                                                      {$$=-1; yyerror("Type Mismatch\n"); exit(0);}
                                                                      generate_assignment_code();
                                                                    }

            | mutable INCREMENT                              {push("++"); if($1 == 1) $$=1; else $$=-1; generate_unary_code();}
            | mutable DECREMENT                              {push("--"); if($1 == 1) $$=1; else $$=-1; generate_unary_code();}
            | simple_expression {if($1 == 1) $$=1; else $$=-1;} ;


simple_expression
            : simple_expression OR_OR and_expression {if($1 == 1 && $3==1) $$=1; else $$=-1; generate_code();}
            | and_expression {if($1 == 1) $$=1; else $$=-1;};

and_expression
            : and_expression AND_AND unary_relation_expression {if($1 == 1 && $3==1) $$=1; else $$=-1; generate_code();}
              |unary_relation_expression {if($1 == 1) $$=1; else $$=-1;} ;

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

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

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

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

sum_operators
            : '+' {push("+");}
            | '-' {push("-");} ;

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

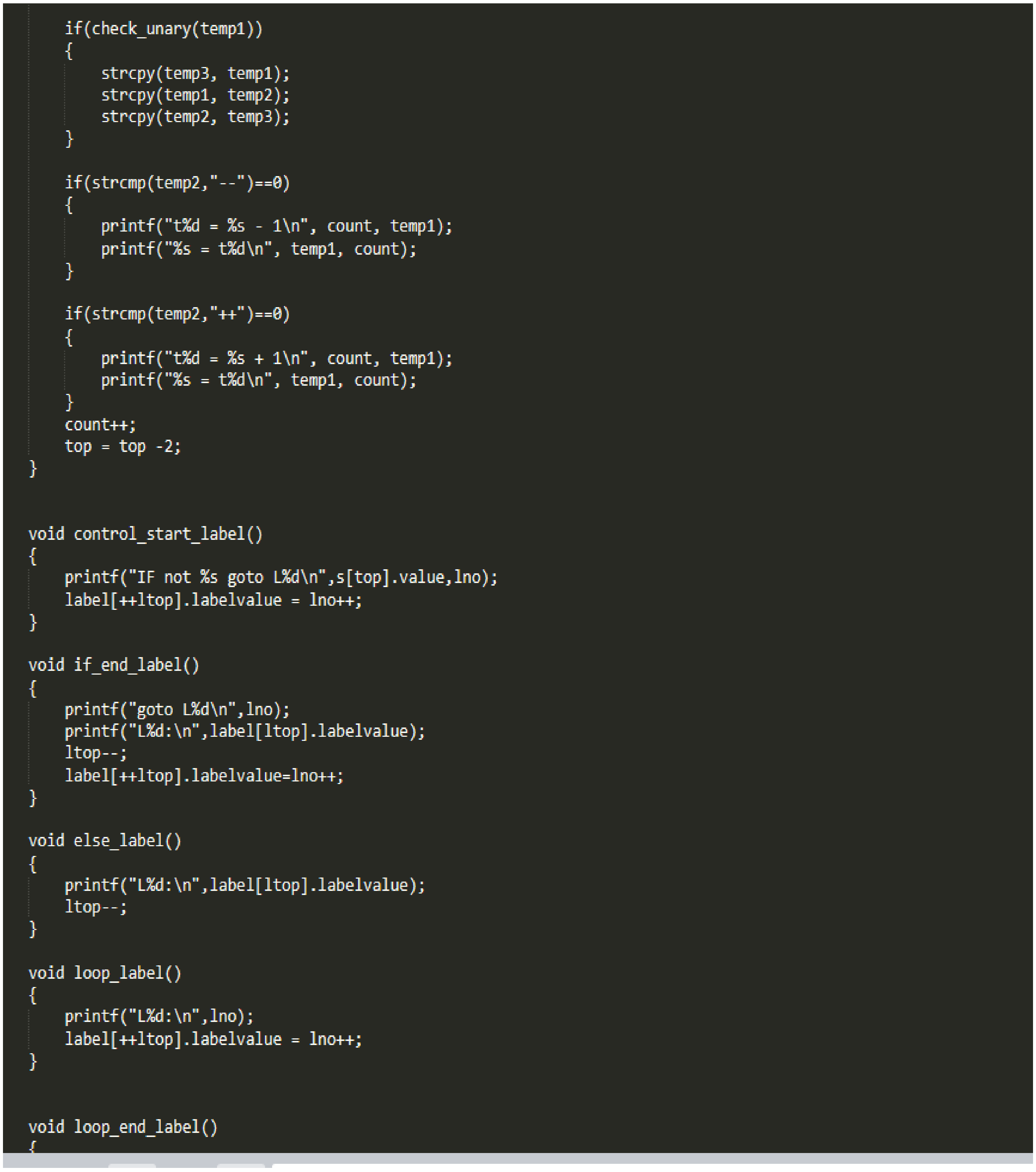


```
MULOP
            : '*' {push("*");}
            | '/' {push("/");}
            | '%' {push("%");} ;

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

mutable
            : identifier {
                          push(current_identifier);
                          if(!check_scope(current_identifier))
                          {printf("%s\n",current_identifier);yyerror("Identifier undeclared\n");exit(0);}
                          if(!check_array(current_identifier))
                          {printf("%s\n",current_identifier);yyerror("Array Identifier has No Subscript\n");exit(0);}
                          if(gettype(current_identifier,0)=='i' || gettype(current_identifier,1)== 'c')
                          $$ = 1;
                          else
                          $$ = -1;
                          }
            | array_identifier {push(current_identifier);
                                array_flag = 1;
                                if(!check_scope(current_identifier)){printf("%s\n",current_identifier);yyerror("Identifier undeclared\n");exit(0);}} '[' expression ']'
                               {if(gettype(current_identifier,0)=='i' || gettype(current_identifier,1)== 'c')
                                $$ = 1;
                                else
                                $$ = -1;
                                };

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

call
            : identifier '('{ strcpy(previous_operator,"(");
                         if(!check_declaration(current_identifier, "Function"))
                         { yyerror("Function not declared"); exit(0);}
                         insert_SymbolTable_function(current_identifier);
                         strcpy(currfunccall,current_identifier);
                         if(gettype(current_identifier,0)=='i' || gettype(current_identifier,1)== 'c')
                         {
                         $$ = 1;
                         }
                         else
                         $$ = -1;
                         call_params_count=0;
                         } arguments ')'
                         { if(strcmp(currfunccall,"printf"))
                            {
                                if(getSTparamscount(currfunccall)!=call_params_count)
```

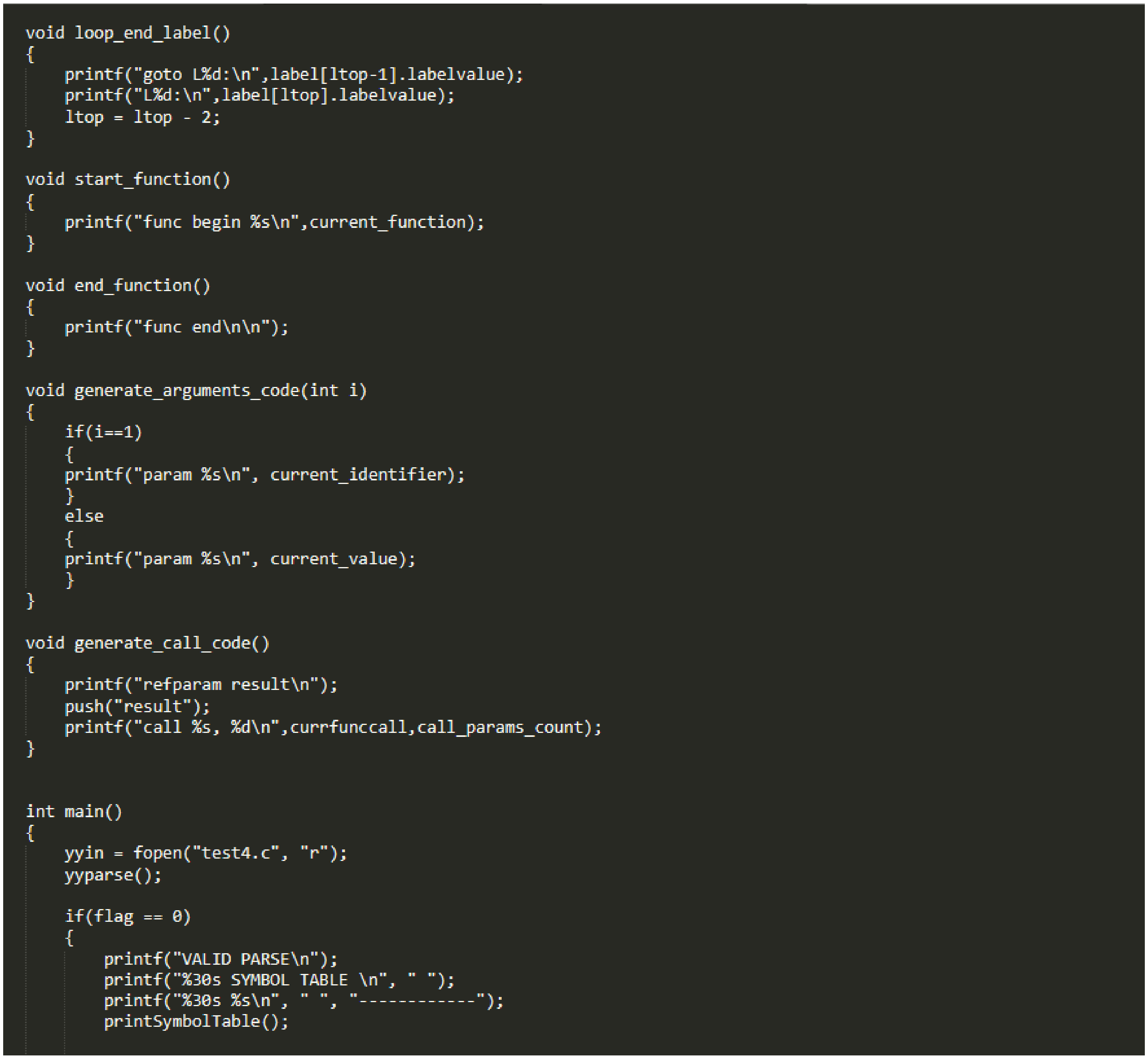


```
                                    {
                                        yyerror("Number of parameters not same as number of arguments during function call!");
                                        //printf("Number of arguments in function call %s doesn't match number of parameters\n", currfunccall);
                                        exit(8);
                                    }
                                }
                                generate_call_code();
                            };

arguments
            : arguments_list | ;

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

exp : identifier {generate_arguments_code(1);} | integer_constant {generate_arguments_code(2);} | string_constant {generate_arguments_code(3
{generate_arguments_code(5);} ;

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


%%

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


void push(char *x)
{
    strcpy(s[++top].value,x);
}
```

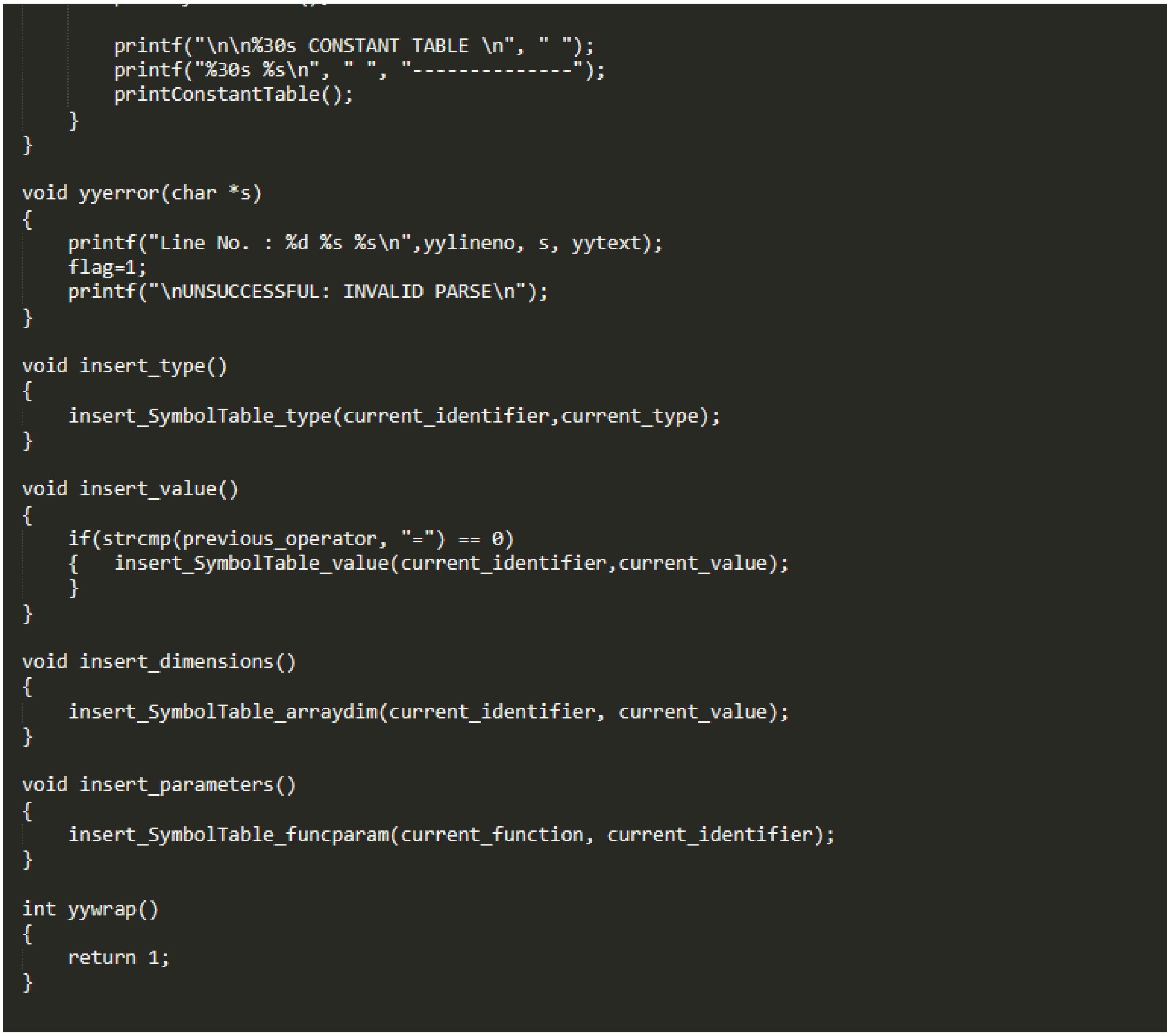

```
void generate_code()
{
    printf("t%d = %s %s %s\n",count,s[top-2].value,s[top-1].value,s[top].value);
    top = top - 2;
    sprintf(temp, "t%d", count);
    strcpy(s[top].value,temp);
    count++;
}

void generate_constant_code()
{
    if(array_flag == 1){
        printf("t%d = 4 * %s\n",count ,current_value);
        count++;
        printf("t%d = &arr[t%d]\n",count ,count-1);
        array_tac_flag = 1;
    }
    else
        printf("t%d = %s\n",count ,current_value);
    sprintf(temp, "t%d", count);
    push(temp);
    count++;
    array_flag = 0;
}

void generate_assignment_code()
{
    if(array_tac_flag == 1)
        printf("*%s = %s\n",s[top-2].value,s[top].value);
    else
        printf("%s = %s\n",s[top-2].value,s[top].value);
    array_tac_flag = 0;
    top = top - 2;
}

int check_unary(char *s)
{
    if(strcmp(s, "--")==0 || strcmp(s, "++")==0)
    {
        return 1;
    }
    return 0;
}

void generate_unary_code()
{
    char temp1[100], temp2[100], temp3[100];
    strcpy(temp1, s[top].value);
    strcpy(temp2, s[top-1].value);
```



```
    if(check_unary(temp1))
    {
        strcpy(temp3, temp1);
        strcpy(temp1, temp2);
        strcpy(temp2, temp3);
    }

    if(strcmp(temp2,"--")==0)
    {
        printf("t%d = %s - 1\n", count, temp1);
        printf("%s = t%d\n", temp1, count);
    }

    if(strcmp(temp2,"++")==0)
    {
        printf("t%d = %s + 1\n", count, temp1);
        printf("%s = t%d\n", temp1, count);
    }
    count++;
    top = top -2;
}


void control_start_label()
{
    printf("IF not %s goto L%d\n",s[top].value,lno);
    label[++ltop].labelvalue = lno++;
}

void if_end_label()
{
    printf("goto L%d\n",lno);
    printf("L%d:\n",label[ltop].labelvalue);
    ltop--;
    label[++ltop].labelvalue=lno++;
}

void else_label()
{
    printf("L%d:\n",label[ltop].labelvalue);
    ltop--;
}

void loop_label()
{
    printf("L%d:\n",lno);
    label[++ltop].labelvalue = lno++;
}


void loop_end_label()
{
```



```
void loop_end_label()
{
    printf("goto L%d:\n",label[ltop-1].labelvalue);
    printf("L%d:\n",label[ltop].labelvalue);
    ltop = ltop - 2;
}

void start_function()
{
    printf("func begin %s\n",current_function);
}

void end_function()
{
    printf("func end\n\n");
}

void generate_arguments_code(int i)
{
    if(i==1)
    {
    printf("param %s\n", current_identifier);
    }
    else
    {
    printf("param %s\n", current_value);
    }
}

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


int main()
{
    yyin = fopen("test4.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_identifier);
}

int yywrap()
{
    return 1;
}
```

Explanation

The lex code is used to detect tokens and generate a stream of tokens from the input C source code. In the first phase of the project, we only stored the different symbols and constants their respective tables and printed out the different tokens with their corresponding line numbers. For the second stage, we return the tokens identified by the lexer to the parser so that the parser is able to use it for further computation. In addition to the functions used in the previous stage, we added functions to help the parser insert the type, value, function parameter, and array dimensions into the symbol table. In the 3rd stage we are using the symbol table and constant table of the previous phase only. We added functions to insert the nesting value of a variable which is essential to check for the scope of a variable, to insert the number of function parameters, to check the scope matches that of the variable etc., in order to check the semantics. For the last stage along with semantic actions SDT also included function to generate the 3 address code.

Definition Section

In the definition section of the yacc program, we include all the required header files, function definitions and other variables. All the tokens which are returned by the lexical analyzer are also listed in the order of their precedence in this section. Operators are also declared here according to their associativity and precedence. This helps ensure that the grammar given to the parser is unambiguous.

Rules Section

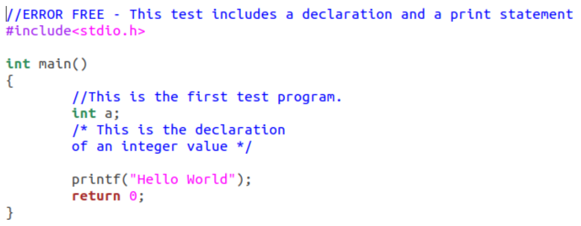
In this section, grammar rules for the C Programming Language is written. The grammar rules are written in such a way that there is no left recursion and the grammar is also deterministic. Non deterministic grammar is converted by applying left factoring. The grammar productions does the syntax analysis of the source code. When the complete statement with proper syntax is matched by the parser, the parser recognizes that it is a valid parse and prints the symbol and constant table. If the statement is not matched, the parser recognizes that there is an error and outputs the error along with the line number. Then code generation function was also associated with the production so that we can get the desired 3 address code for the given input program.

C Code Section

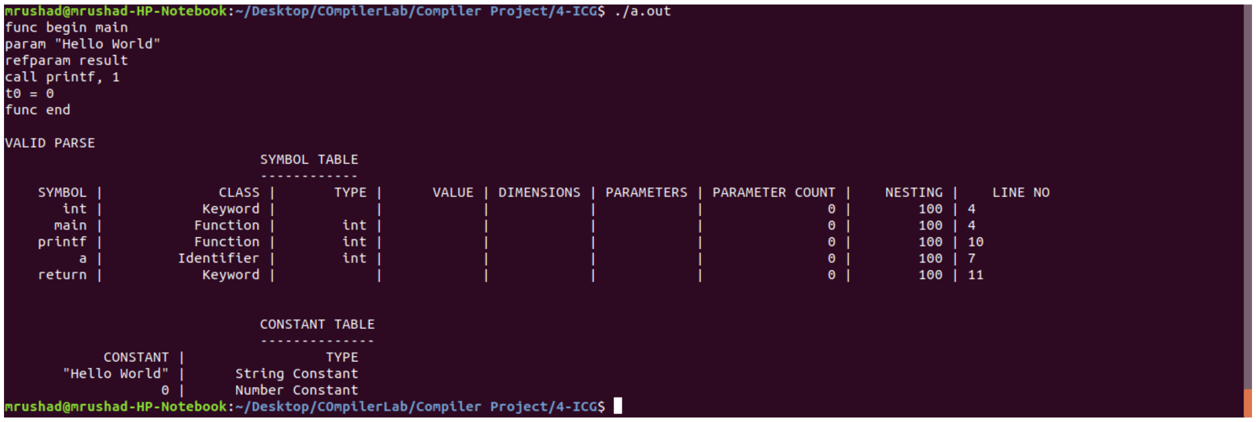
The yyparse() function was called to run the program on the given input file. After that, both the symbol table and the constant table were printed in order to show the result. Due to the ICG phase the 3 address code is also generated.

Test Cases (Error Free)

Testcase 1

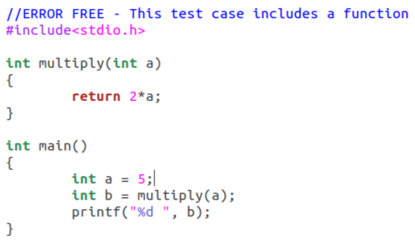


```
//ERROR FREE - This test includes a declaration and a print statement
#include<stdio.h>

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

        printf("Hello World");
        return 0;
}
```

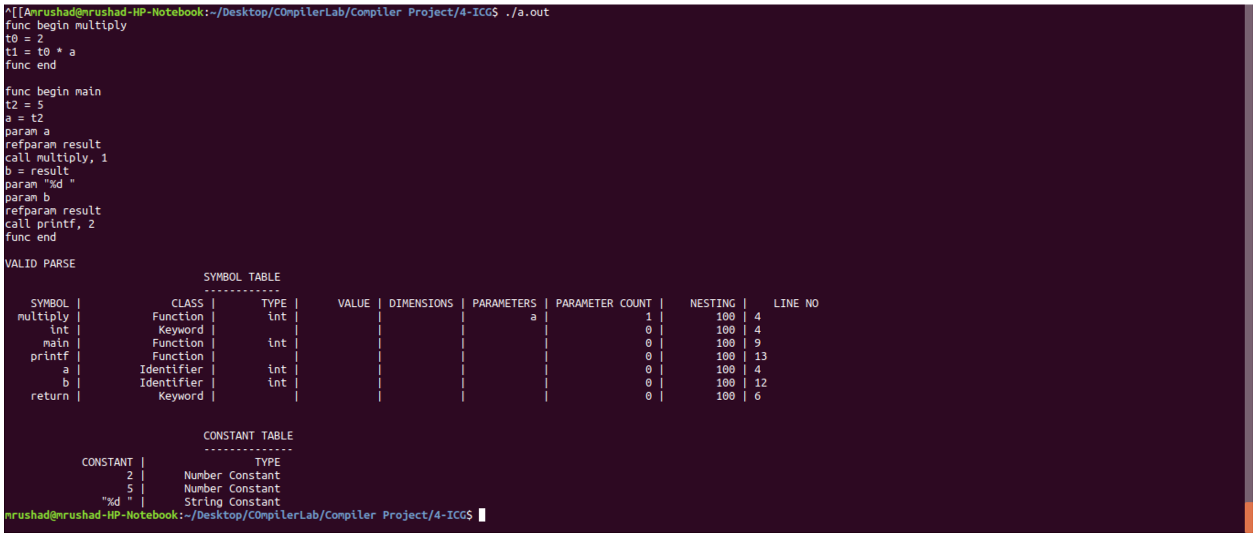


```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
param "Hello World"
refparam result
call printf, 1
t0 = 0
func end

VALID PARSE
                               SYMBOL TABLE
                               ------------

    SYMBOL |               CLASS |        TYPE |       VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |      NESTING |     LINE NO
       int |             Keyword |             |             |            |            |               0 |          100 | 4
      main |            Function |         int |             |            |            |               0 |          100 | 4
    printf |            Function |         int |             |            |            |               0 |          100 | 10
         a |          Identifier |         int |             |            |            |               0 |          100 | 7
    return |             Keyword |             |             |            |            |               0 |          100 | 11


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

            CONSTANT |                 TYPE
       "Hello World" |      String Constant
                   0 |      Number Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ 
```

Testcase 2

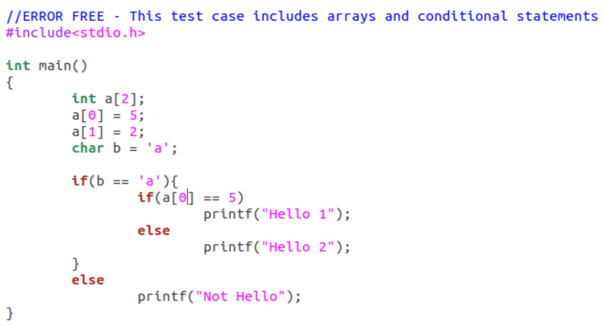


```
//ERROR FREE - This test case includes a function
#include<stdio.h>

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

int main()
{
        int a = 5;
        int b = multiply(a);
        printf("%d ", b);
}
```

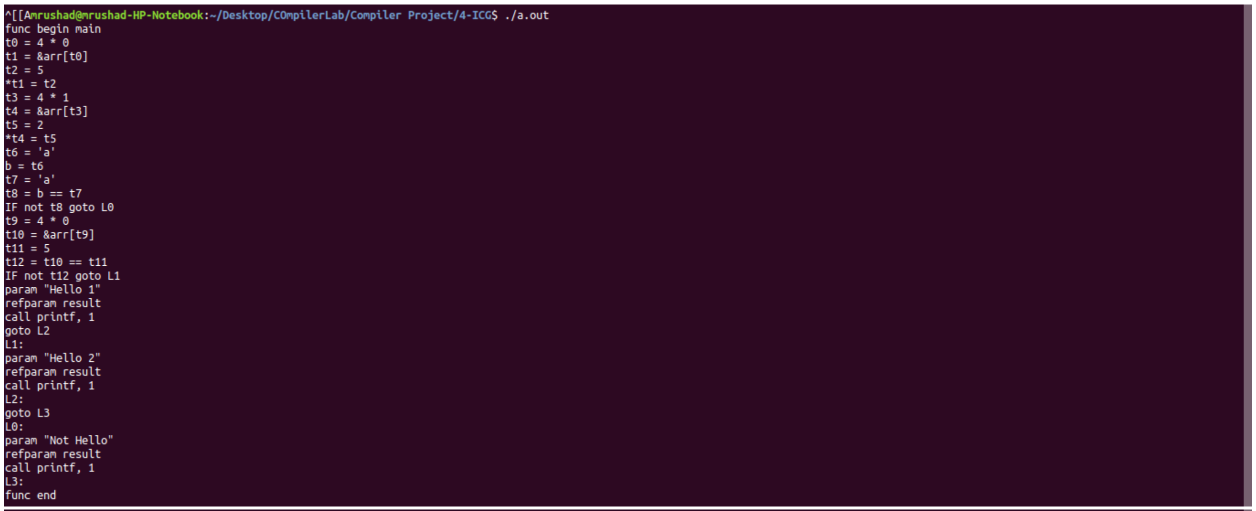


```
^[[Amrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin multiply
t0 = 2
t1 = t0 * a
func end

func begin main
t2 = 5
a = t2
param a
refparam result
call multiply, 1
b = result
param "%d "
param b
refparam result
call printf, 2
func end

VALID PARSE
                               SYMBOL TABLE
                               ------------

    SYMBOL |               CLASS |        TYPE |       VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |      NESTING |     LINE NO
  multiply |            Function |         int |             |            |          a |               1 |          100 | 4
       int |             Keyword |             |             |            |            |               0 |          100 | 4
      main |            Function |         int |             |            |            |               0 |          100 | 9
    printf |            Function |             |             |            |            |               0 |          100 | 13
         a |          Identifier |         int |             |            |            |               0 |          100 | 4
         b |          Identifier |         int |             |            |            |               0 |          100 | 12
    return |             Keyword |             |             |            |            |               0 |          100 | 6


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

            CONSTANT |                 TYPE
                   2 |      Number Constant
                   5 |      Number Constant
               "%d " |      String Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ 
```

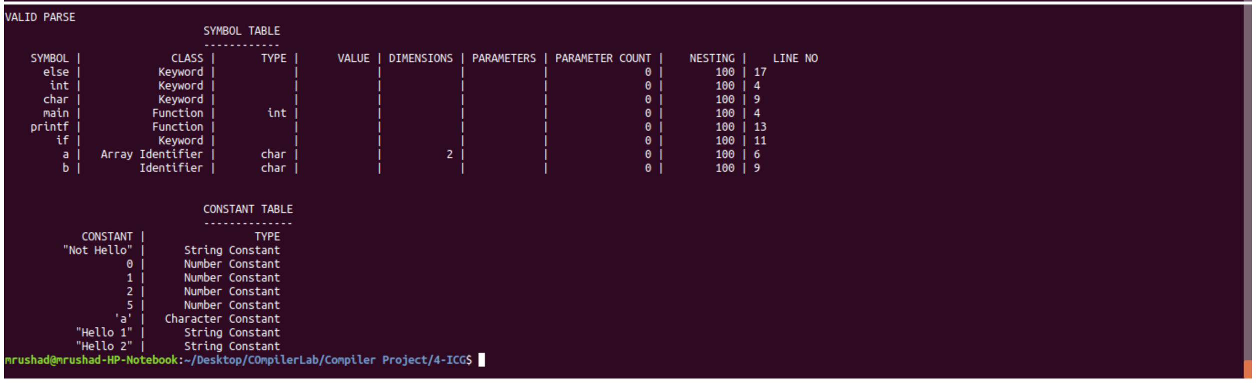
Testcase 3



```c
//ERROR FREE - This test case includes arrays and conditional statements
#include<stdio.h>

int main()
{
        int a[2];
        a[0] = 5;
        a[1] = 2;
        char b = 'a';

        if(b == 'a'){
                if(a[0] == 5)
                        printf("Hello 1");
                else
                        printf("Hello 2");
        }
        else
                printf("Not Hello");
}
```

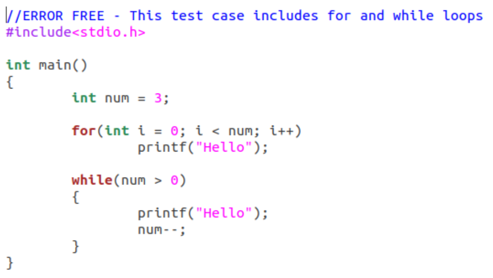


```
^[[Amrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
t0 = 4 * 0
t1 = &arr[t0]
t2 = 5
*t1 = t2
t3 = 4 * 1
t4 = &arr[t3]
t5 = 2
*t4 = t5
t6 = 'a'
b = t6
t7 = 'a'
t8 = b == t7
IF not t8 goto L0
t9 = 4 * 0
t10 = &arr[t9]
t11 = 5
t12 = t10 == t11
IF not t12 goto L1
param "Hello 1"
refparam result
call printf, 1
goto L2
L1:
param "Hello 2"
refparam result
call printf, 1
L2:
goto L3
L0:
param "Not Hello"
refparam result
call printf, 1
L3:
func end
```

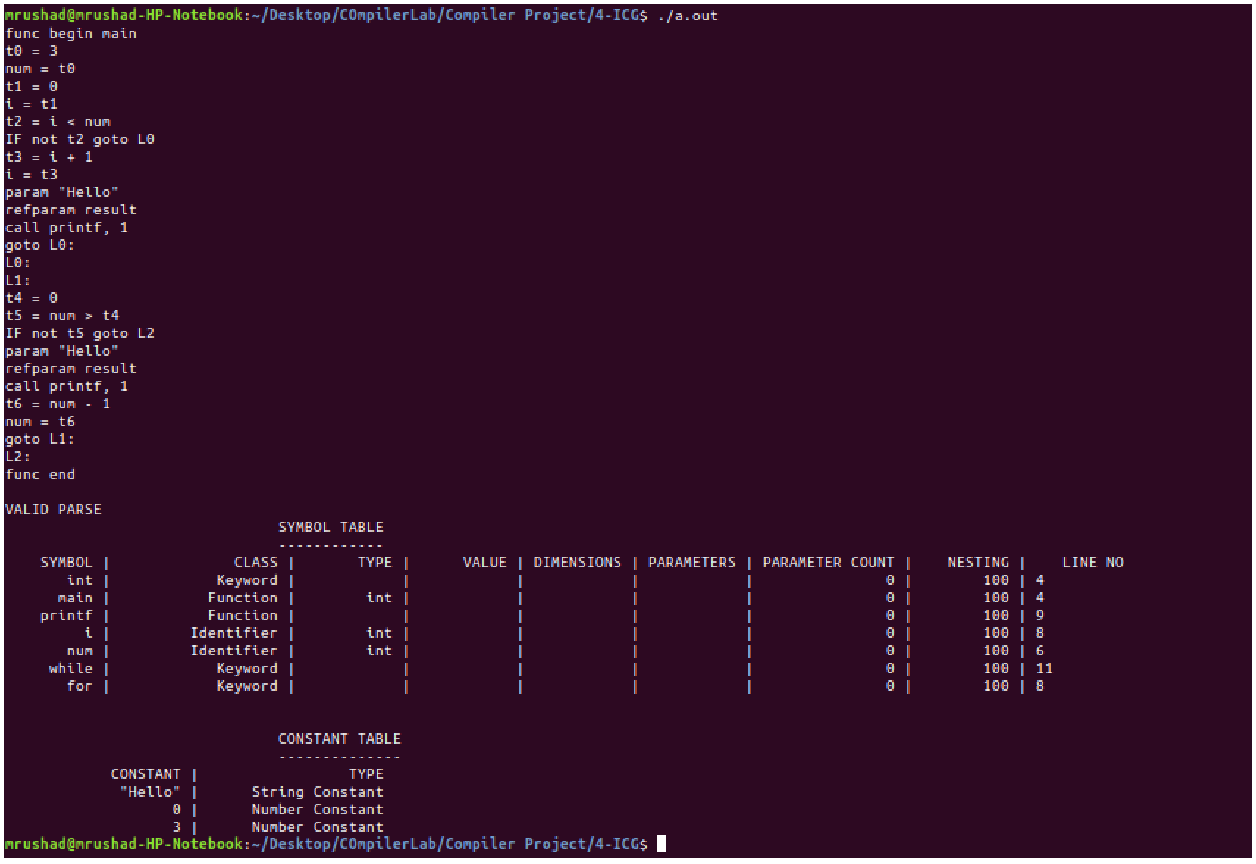


```
VALID PARSE
                              SYMBOL TABLE
                              ------------

    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |     NESTING |     LINE NO
      else |           Keyword |            |            |            |            |               0 |         100 | 17
       int |           Keyword |            |            |            |            |               0 |         100 | 4
      char |           Keyword |            |            |            |            |               0 |         100 | 9
      main |          Function |        int |            |            |            |               0 |         100 | 4
    printf |          Function |            |            |            |            |               0 |         100 | 13
        if |           Keyword |            |            |            |            |               0 |         100 | 11
         a |  Array Identifier |       char |            |          2 |            |               0 |         100 | 6
         b |        Identifier |       char |            |            |            |               0 |         100 | 9


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

        CONSTANT |                 TYPE
     "Not Hello" |      String Constant
               0 |      Number Constant
               1 |      Number Constant
               2 |      Number Constant
               5 |      Number Constant
             'a' |   Character Constant
       "Hello 1" |      String Constant
       "Hello 2" |      String Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ 
```

Testcase 4

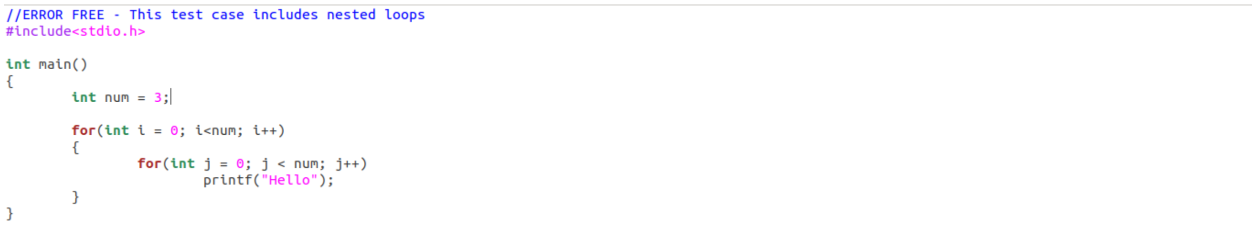


```
//ERROR FREE - This test case includes for and while loops
#include<stdio.h>

int main()
{
        int num = 3;

        for(int i = 0; i < num; i++)
                printf("Hello");

        while(num > 0)
        {
                printf("Hello");
                num--;
        }
}
```

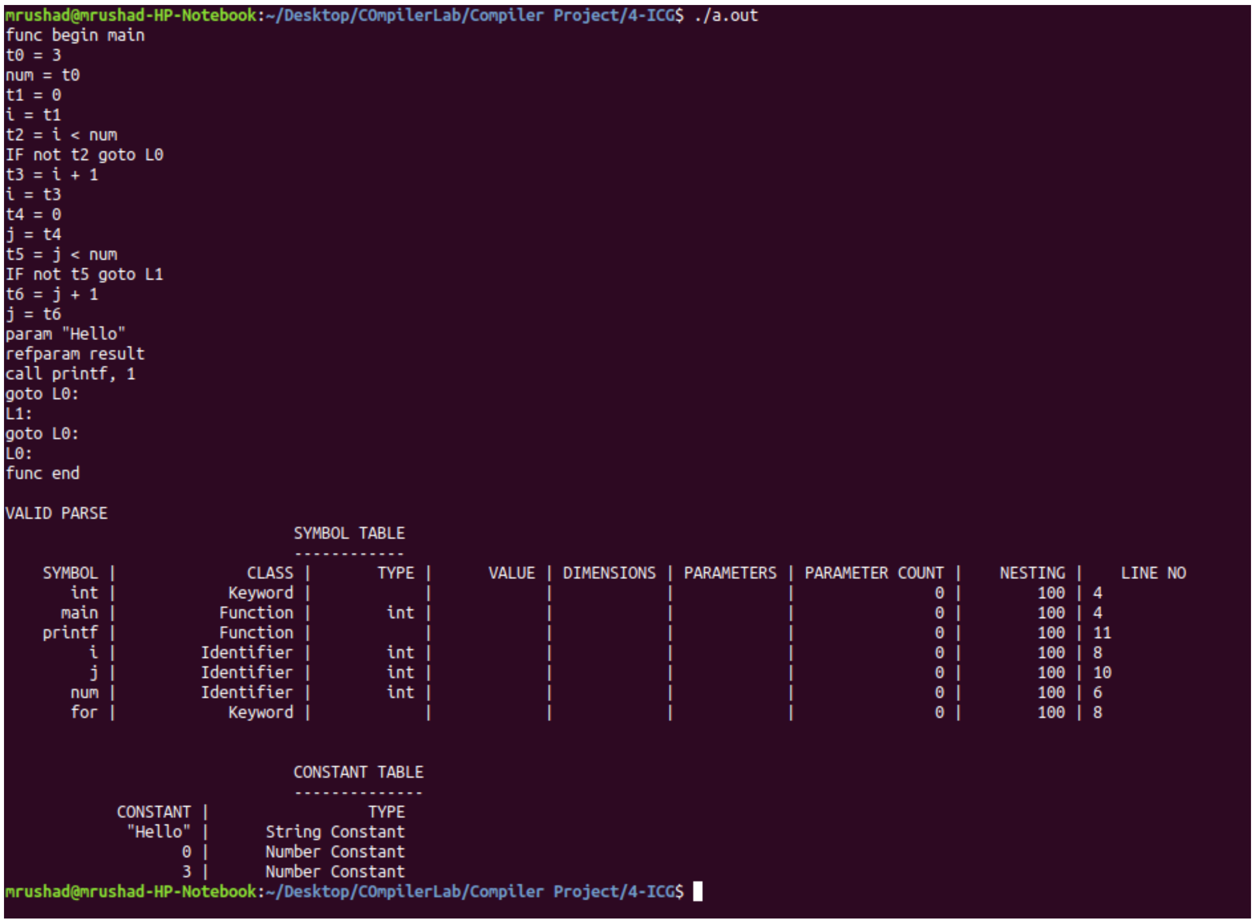


```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
t0 = 3
num = t0
t1 = 0
i = t1
t2 = i < num
IF not t2 goto L0
t3 = i + 1
i = t3
param "Hello"
refparam result
call printf, 1
goto L0:
L0:
L1:
t4 = 0
t5 = num > t4
IF not t5 goto L2
param "Hello"
refparam result
call printf, 1
t6 = num - 1
num = t6
goto L1:
L2:
func end

VALID PARSE
                               SYMBOL TABLE
                               ------------

    SYMBOL |               CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |             Keyword |            |            |            |            |               0 |        100 | 4
      main |            Function |        int |            |            |            |               0 |        100 | 4
    printf |            Function |            |            |            |            |               0 |        100 | 9
         i |          Identifier |        int |            |            |            |               0 |        100 | 8
       num |          Identifier |        int |            |            |            |               0 |        100 | 6
     while |             Keyword |            |            |            |            |               0 |        100 | 11
       for |             Keyword |            |            |            |            |               0 |        100 | 8


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

            CONSTANT |                TYPE
             "Hello" |      String Constant
                   0 |      Number Constant
                   3 |      Number Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$
```

Testcase 5

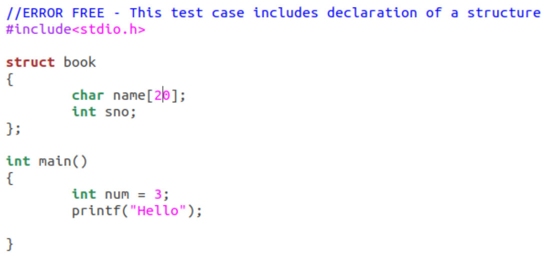


```c
//ERROR FREE - This test case includes nested loops
#include<stdio.h>

int main()
{
        int num = 3;

        for(int i = 0; i<num; i++)
        {
                for(int j = 0; j < num; j++)
                        printf("Hello");
        }
}
```

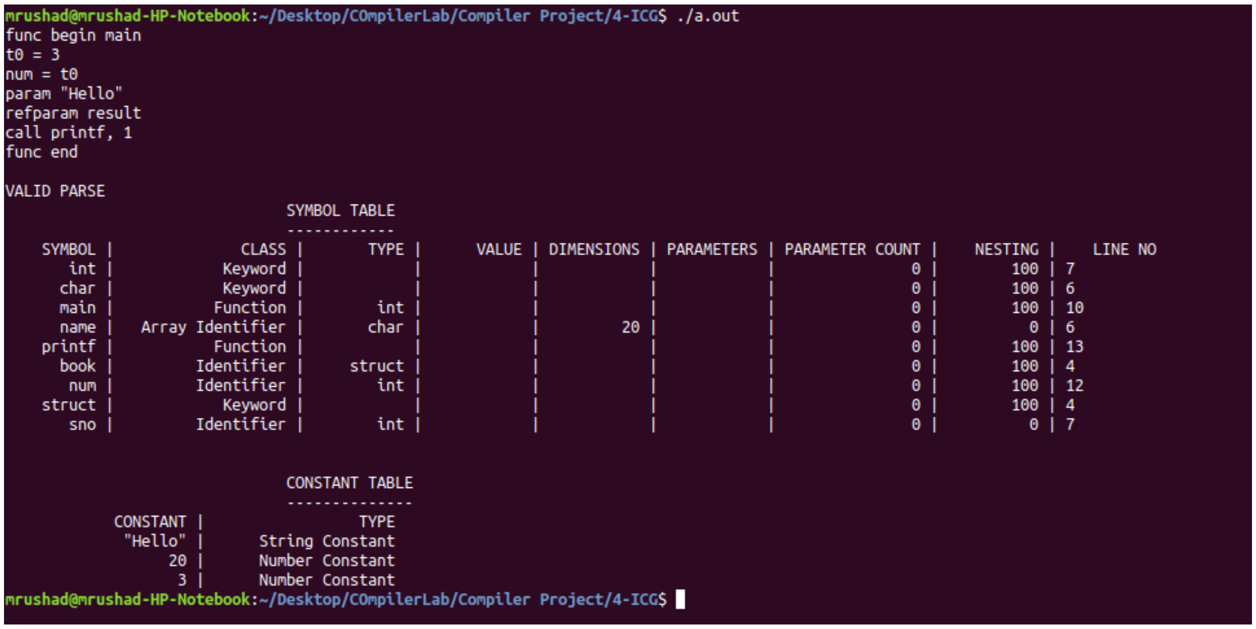


```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
t0 = 3
num = t0
t1 = 0
i = t1
t2 = i < num
IF not t2 goto L0
t3 = i + 1
i = t3
t4 = 0
j = t4
t5 = j < num
IF not t5 goto L1
t6 = j + 1
j = t6
param "Hello"
refparam result
call printf, 1
goto L0:
L1:
goto L0:
L0:
func end

VALID PARSE
                              SYMBOL TABLE
                              ------------

    SYMBOL |              CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |      NESTING |     LINE NO
       int |            Keyword |            |            |            |            |               0 |          100 | 4
      main |           Function |        int |            |            |            |               0 |          100 | 4
    printf |           Function |            |            |            |            |               0 |          100 | 11
         i |         Identifier |        int |            |            |            |               0 |          100 | 8
         j |         Identifier |        int |            |            |            |               0 |          100 | 10
       num |         Identifier |        int |            |            |            |               0 |          100 | 6
       for |            Keyword |            |            |            |            |               0 |          100 | 8


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

            CONSTANT |                TYPE
             "Hello" |     String Constant
                   0 |     Number Constant
                   3 |     Number Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$
```

Testcase 6

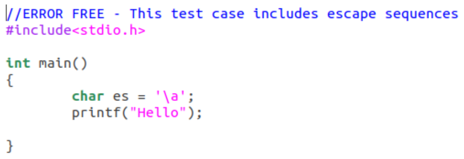


```
//ERROR FREE - This test case includes declaration of a structure
#include<stdio.h>

struct book
{
        char name[20];
        int sno;
};

int main()
{
        int num = 3;
        printf("Hello");

}
```

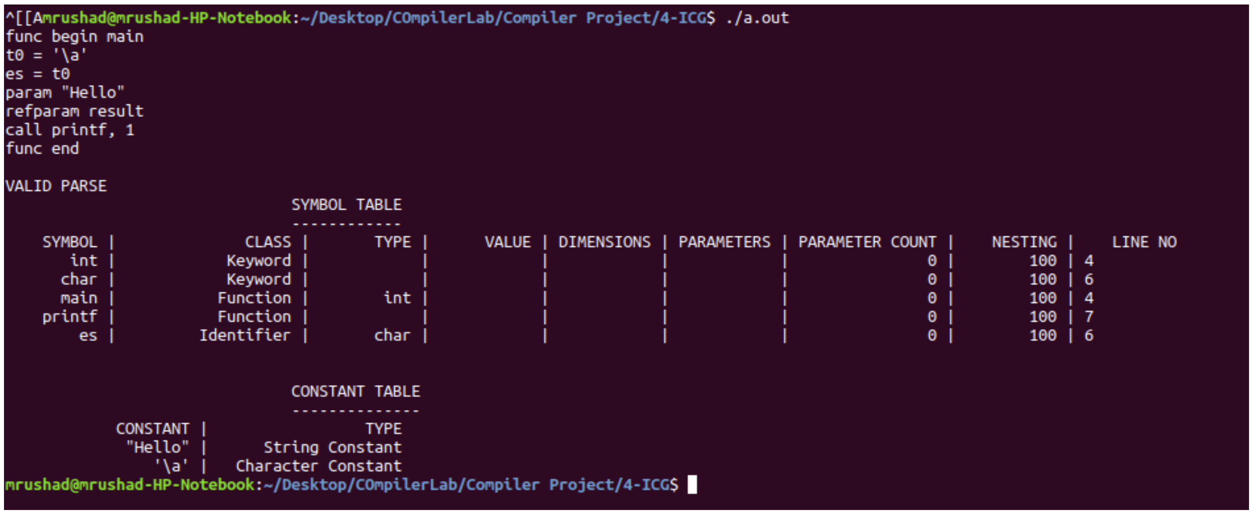


```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
t0 = 3
num = t0
param "Hello"
refparam result
call printf, 1
func end

VALID PARSE
                              SYMBOL TABLE
                              ------------

    SYMBOL |              CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |            Keyword |            |            |            |            |               0 |        100 | 7
      char |            Keyword |            |            |            |            |               0 |        100 | 6
      main |           Function |        int |            |            |            |               0 |        100 | 10
      name |   Array Identifier |       char |            |         20 |            |               0 |          0 | 6
    printf |           Function |            |            |            |            |               0 |        100 | 13
      book |         Identifier |     struct |            |            |            |               0 |        100 | 4
       num |         Identifier |        int |            |            |            |               0 |        100 | 12
    struct |            Keyword |            |            |            |            |               0 |        100 | 4
       sno |         Identifier |        int |            |            |            |               0 |          0 | 7


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

            CONSTANT |                    TYPE
             "Hello" |         String Constant
                  20 |         Number Constant
                   3 |         Number Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ 
```

Testcase 7

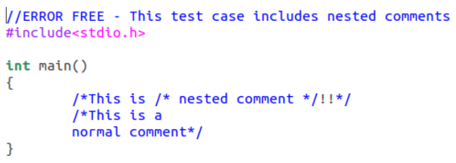


```
//ERROR FREE - This test case includes escape sequences
#include<stdio.h>

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

}
```

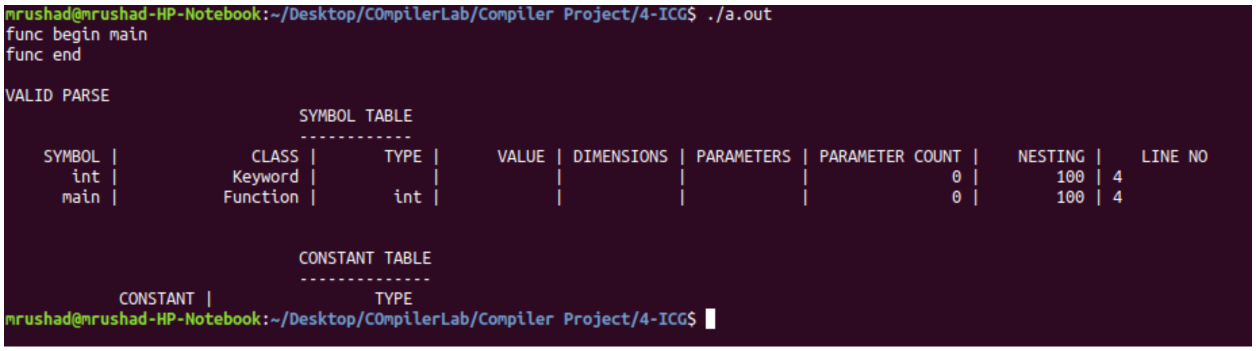


```
^[[Amrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
t0 = '\a'
es = t0
param "Hello"
refparam result
call printf, 1
func end

VALID PARSE
                              SYMBOL TABLE
                              ------------

    SYMBOL |              CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |            Keyword |            |            |            |            |               0 |        100 | 4
      char |            Keyword |            |            |            |            |               0 |        100 | 6
      main |           Function |        int |            |            |            |               0 |        100 | 4
    printf |           Function |            |            |            |            |               0 |        100 | 7
        es |         Identifier |       char |            |            |            |               0 |        100 | 6


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

            CONSTANT |                    TYPE
             "Hello" |         String Constant
                '\a' |      Character Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ 
```

Testcase 8

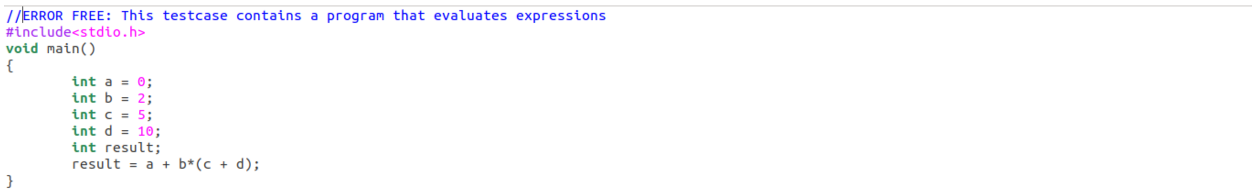


```
//ERROR FREE - This test case includes nested comments
#include<stdio.h>

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

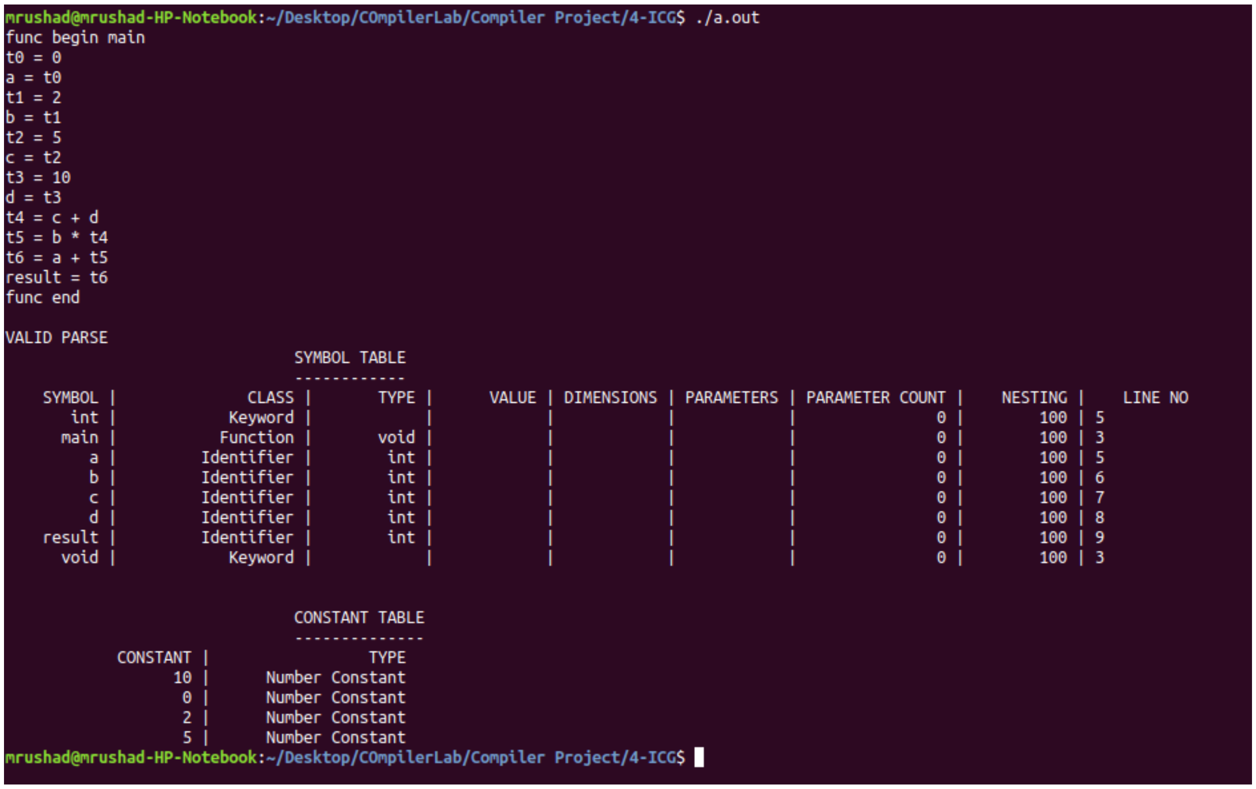


```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
func end

VALID PARSE
                                SYMBOL TABLE
                                ------------
    SYMBOL |               CLASS |        TYPE |       VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |     NESTING |     LINE NO
       int |             Keyword |             |             |            |            |               0 |         100 | 4
      main |            Function |         int |             |            |            |               0 |         100 | 4


                                CONSTANT TABLE
                                --------------
            CONSTANT |                   TYPE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$
```

Testcase 9

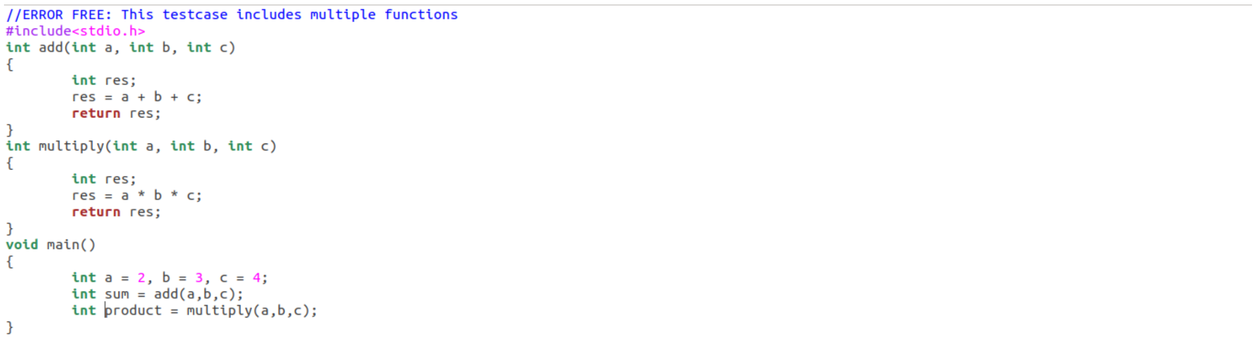


```
//ERROR FREE: This testcase contains a program that evaluates expressions
#include<stdio.h>
void main()
{
        int a = 0;
        int b = 2;
        int c = 5;
        int d = 10;
        int result;
        result = a + b*(c + d);
}
```

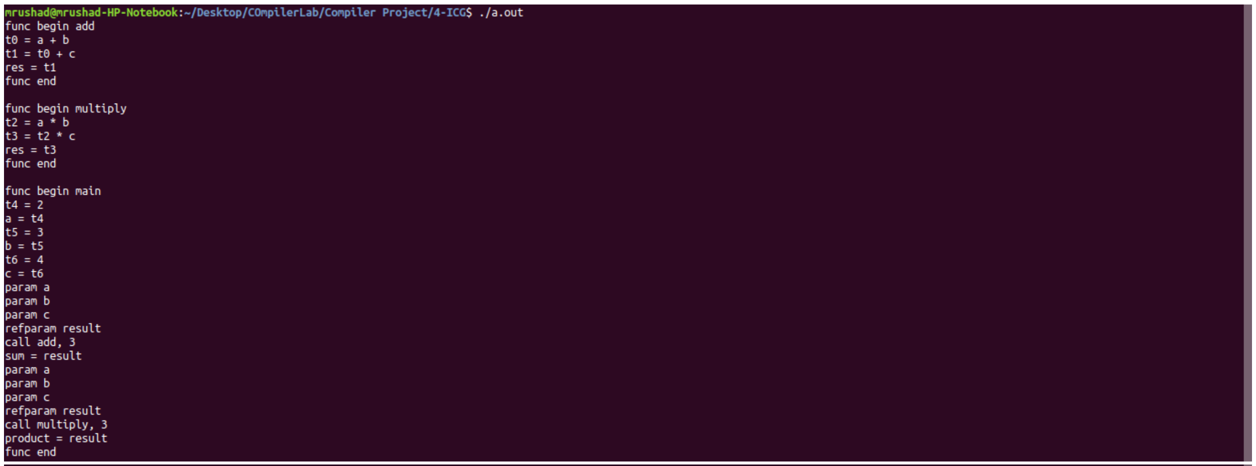


```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
t0 = 0
a = t0
t1 = 2
b = t1
t2 = 5
c = t2
t3 = 10
d = t3
t4 = c + d
t5 = b * t4
t6 = a + t5
result = t6
func end

VALID PARSE
                                SYMBOL TABLE
                                ------------
    SYMBOL |               CLASS |        TYPE |       VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |     NESTING |     LINE NO
       int |             Keyword |             |             |            |            |               0 |         100 | 5
      main |            Function |        void |             |            |            |               0 |         100 | 3
         a |          Identifier |         int |             |            |            |               0 |         100 | 5
         b |          Identifier |         int |             |            |            |               0 |         100 | 6
         c |          Identifier |         int |             |            |            |               0 |         100 | 7
         d |          Identifier |         int |             |            |            |               0 |         100 | 8
    result |          Identifier |         int |             |            |            |               0 |         100 | 9
      void |             Keyword |             |             |            |            |               0 |         100 | 3


                                CONSTANT TABLE
                                --------------
            CONSTANT |                   TYPE
                  10 |       Number Constant
                   0 |       Number Constant
                   2 |       Number Constant
                   5 |       Number Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$
```

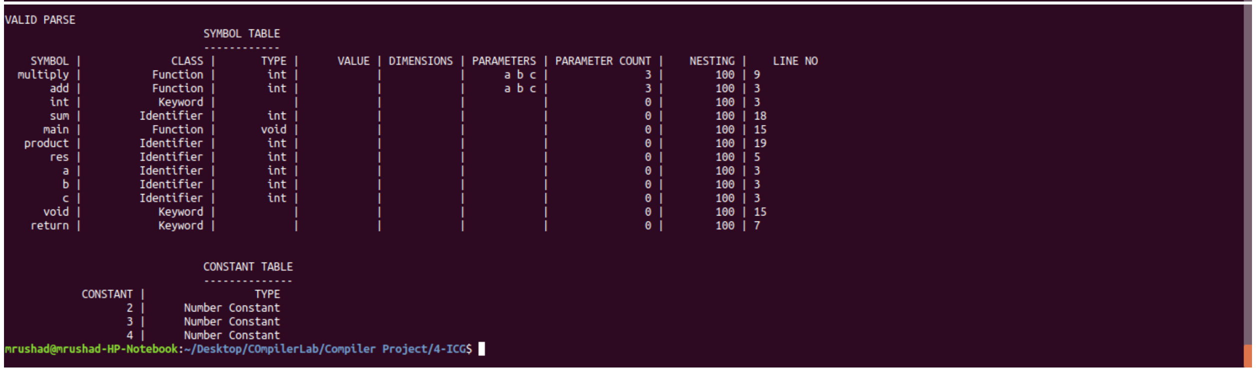
Testcase 10



```
//ERROR FREE: This testcase includes multiple functions
#include<stdio.h>
int add(int a, int b, int c)
{
        int res;
        res = a + b + c;
        return res;
}
int multiply(int a, int b, int c)
{
        int res;
        res = a * b * c;
        return res;
}
void main()
{
        int a = 2, b = 3, c = 4;
        int sum = add(a,b,c);
        int product = multiply(a,b,c);
}
```

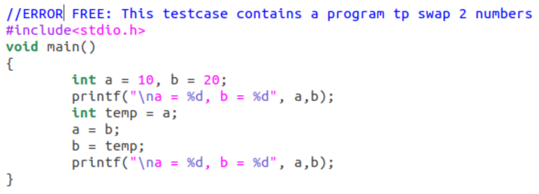


```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin add
t0 = a + b
t1 = t0 + c
res = t1
func end

func begin multiply
t2 = a * b
t3 = t2 * c
res = t3
func end

func begin main
t4 = 2
a = t4
t5 = 3
b = t5
t6 = 4
c = t6
param a
param b
param c
refparam result
call add, 3
sum = result
param a
param b
param c
refparam result
call multiply, 3
product = result
func end
```

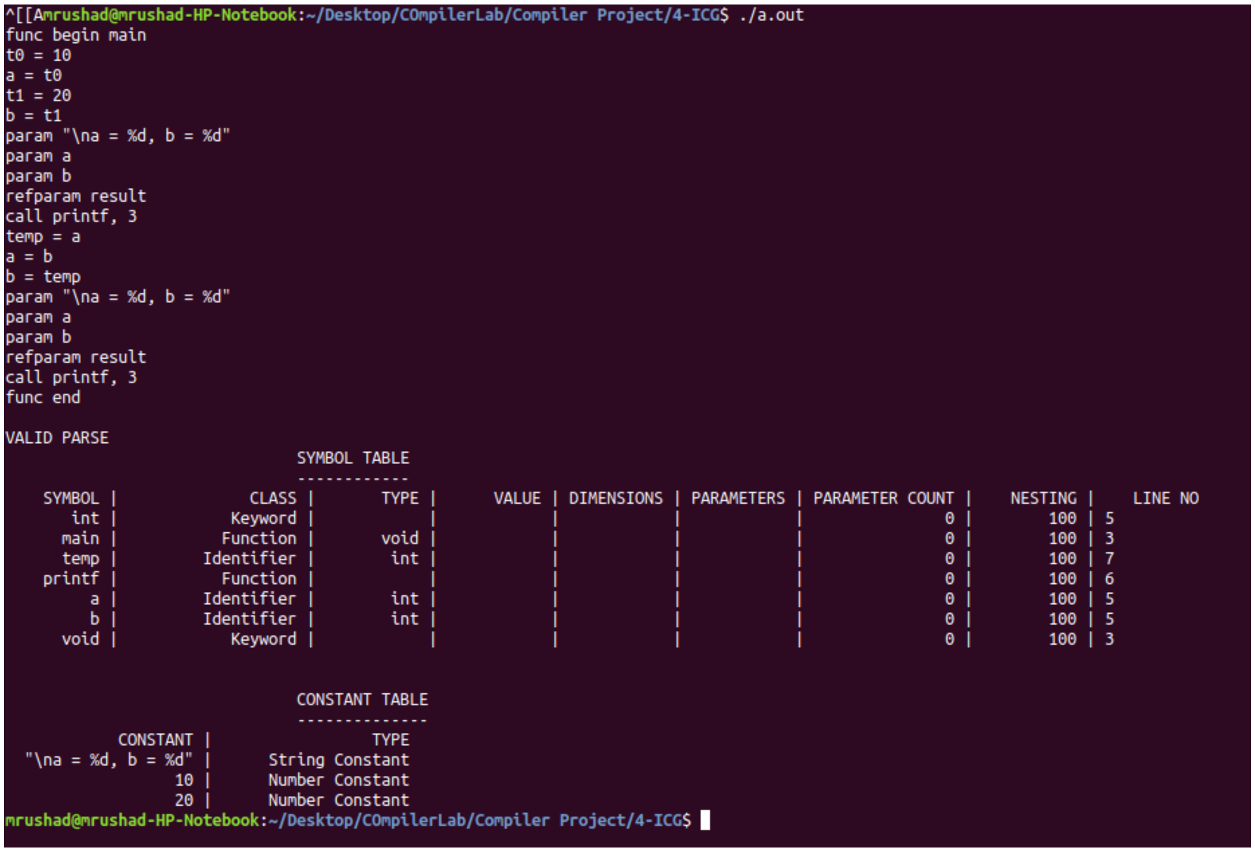


```
VALID PARSE
                               SYMBOL TABLE
                               ------------
    SYMBOL |           CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |     NESTING |    LINE NO
  multiply |        Function |        int |            |            |      a b c |               3 |         100 | 9
       add |        Function |        int |            |            |      a b c |               3 |         100 | 3
       int |         Keyword |            |            |            |            |               0 |         100 | 3
       sum |      Identifier |        int |            |            |            |               0 |         100 | 18
      main |        Function |       void |            |            |            |               0 |         100 | 15
   product |      Identifier |        int |            |            |            |               0 |         100 | 19
       res |      Identifier |        int |            |            |            |               0 |         100 | 5
         a |      Identifier |        int |            |            |            |               0 |         100 | 3
         b |      Identifier |        int |            |            |            |               0 |         100 | 3
         c |      Identifier |        int |            |            |            |               0 |         100 | 3
      void |         Keyword |            |            |            |            |               0 |         100 | 15
    return |         Keyword |            |            |            |            |               0 |         100 | 7


                               CONSTANT TABLE
                               --------------
            CONSTANT |                 TYPE
                   2 |      Number Constant
                   3 |      Number Constant
                   4 |      Number Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ 
```

Testcase 11

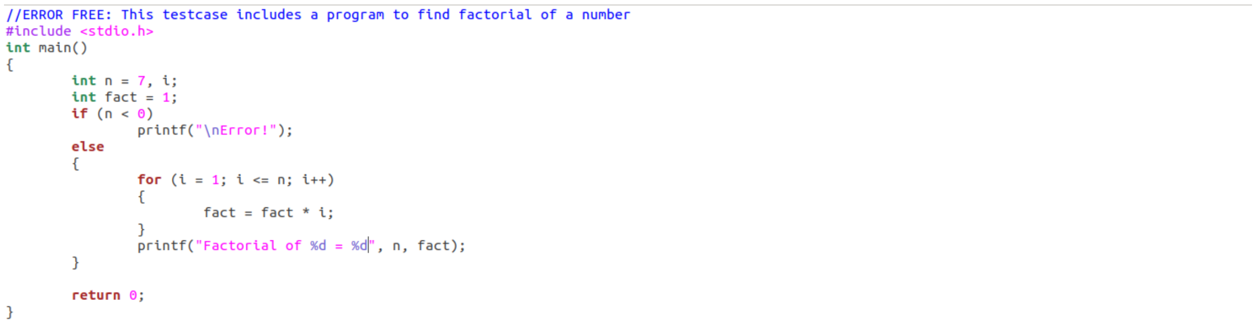


```
//ERROR FREE: This testcase contains a program tp swap 2 numbers
#include<stdio.h>
void main()
{
        int a = 10, b = 20;
        printf("\na = %d, b = %d", a,b);
        int temp = a;
        a = b;
        b = temp;
        printf("\na = %d, b = %d", a,b);
}
```

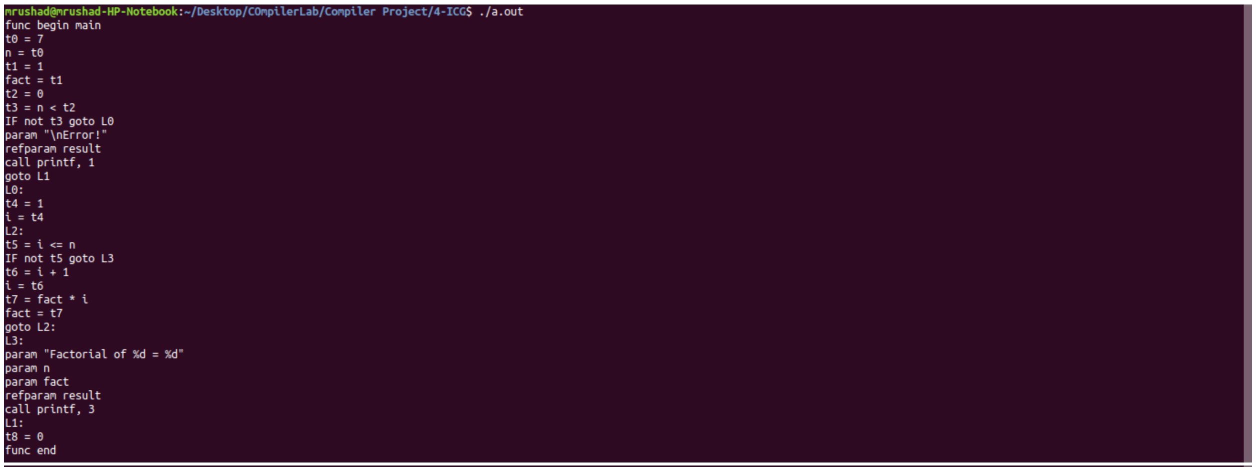


```
^[[Amrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
t0 = 10
a = t0
t1 = 20
b = t1
param "\na = %d, b = %d"
param a
param b
refparam result
call printf, 3
temp = a
a = b
b = temp
param "\na = %d, b = %d"
param a
param b
refparam result
call printf, 3
func end

VALID PARSE
                               SYMBOL TABLE
                               ------------

    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |     NESTING |     LINE NO
       int |           Keyword |            |            |            |            |               0 |         100 | 5
      main |          Function |       void |            |            |            |               0 |         100 | 3
      temp |        Identifier |        int |            |            |            |               0 |         100 | 7
    printf |          Function |            |            |            |            |               0 |         100 | 6
         a |        Identifier |        int |            |            |            |               0 |         100 | 5
         b |        Identifier |        int |            |            |            |               0 |         100 | 5
      void |           Keyword |            |            |            |            |               0 |         100 | 3


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

           CONSTANT |            TYPE
  "\na = %d, b = %d" |     String Constant
                 10 |     Number Constant
                 20 |     Number Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$
```

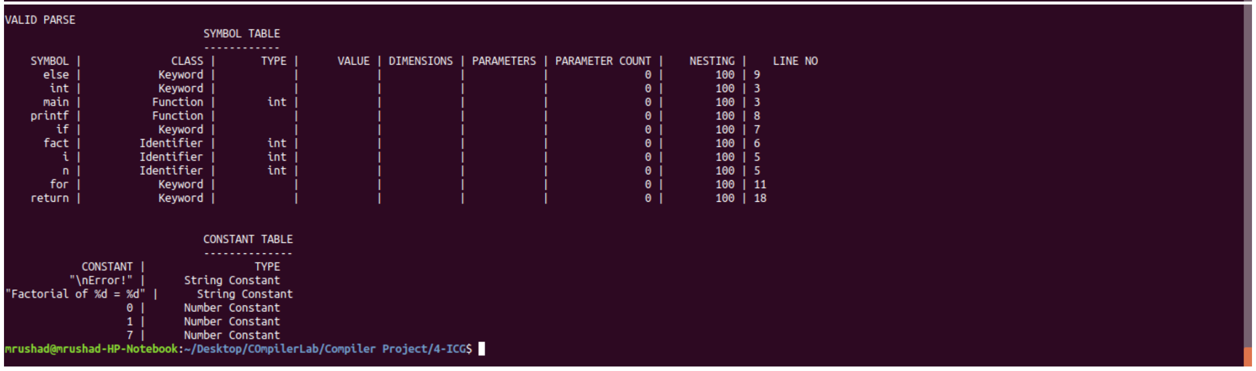
Testcase 12



```c
//ERROR FREE: This testcase includes a program to find factorial of a number
#include <stdio.h>
int main()
{
        int n = 7, i;
        int fact = 1;
        if (n < 0)
                printf("\nError!");
        else
        {
                for (i = 1; i <= n; i++)
                {
                        fact = fact * i;
                }
                printf("Factorial of %d = %d", n, fact);
        }

        return 0;
}
```



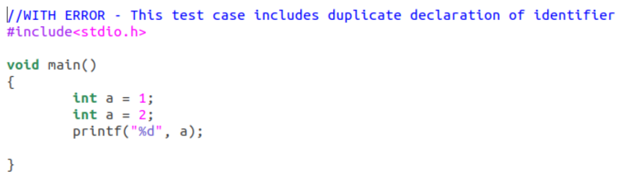
```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
t0 = 7
n = t0
t1 = 1
fact = t1
t2 = 0
t3 = n < t2
IF not t3 goto L0
param "\nError!"
refparam result
call printf, 1
goto L1
L0:
t4 = 1
i = t4
L2:
t5 = i <= n
IF not t5 goto L3
t6 = i + 1
i = t6
t7 = fact * i
fact = t7
goto L2:
L3:
param "Factorial of %d = %d"
param n
param fact
refparam result
call printf, 3
L1:
t8 = 0
func end
```



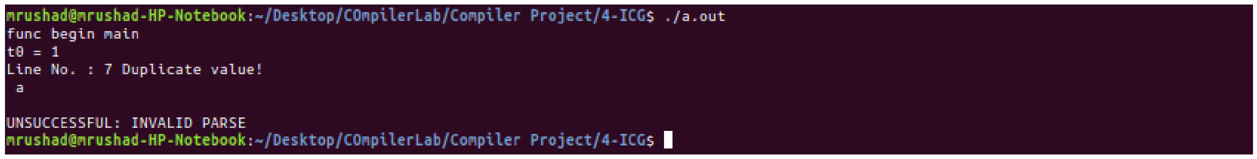
```
VALID PARSE
                              SYMBOL TABLE
                              ------------
    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |     NESTING |     LINE NO
      else |           Keyword |            |            |            |            |               0 |         100 | 9
       int |           Keyword |            |            |            |            |               0 |         100 | 3
      main |          Function |        int |            |            |            |               0 |         100 | 3
    printf |          Function |            |            |            |            |               0 |         100 | 8
        if |           Keyword |            |            |            |            |               0 |         100 | 7
      fact |        Identifier |        int |            |            |            |               0 |         100 | 6
         i |        Identifier |        int |            |            |            |               0 |         100 | 5
         n |        Identifier |        int |            |            |            |               0 |         100 | 5
       for |           Keyword |            |            |            |            |               0 |         100 | 11
    return |           Keyword |            |            |            |            |               0 |         100 | 18


                              CONSTANT TABLE
                              --------------
          CONSTANT |                  TYPE
        "\nError!" |       String Constant
"Factorial of %d = %d" |       String Constant
                 0 |       Number Constant
                 1 |       Number Constant
                 7 |       Number Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$
```

Test Cases (With Error)

Testcase 13

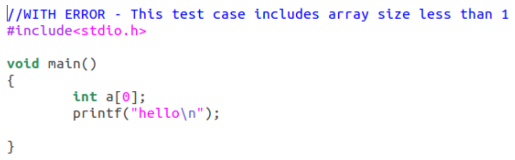


```
//WITH ERROR - This test case includes duplicate declaration of identifier
#include<stdio.h>

void main()
{
        int a = 1;
        int a = 2;
        printf("%d", a);

}
```

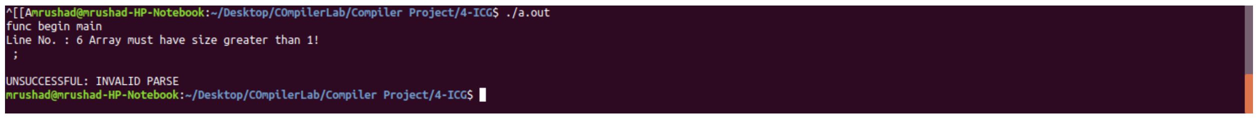


```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
t0 = 1
Line No. : 7 Duplicate value!
 a

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$
```

Testcase 14

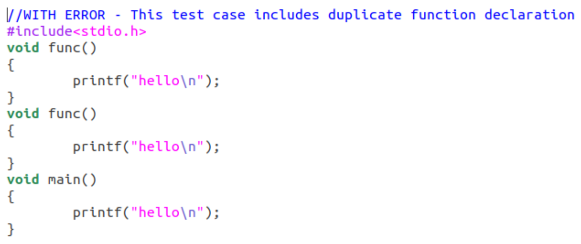


```
//WITH ERROR - This test case includes array size less than 1
#include<stdio.h>

void main()
{
        int a[0];
        printf("hello\n");

}
```

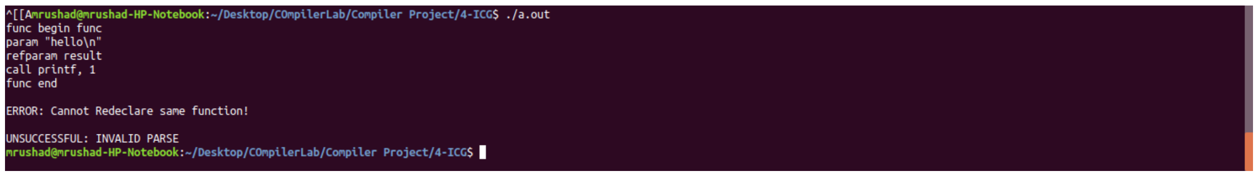


```
^[[Amrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
Line No. : 6 Array must have size greater than 1!
 ;

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$
```

Testcase 15



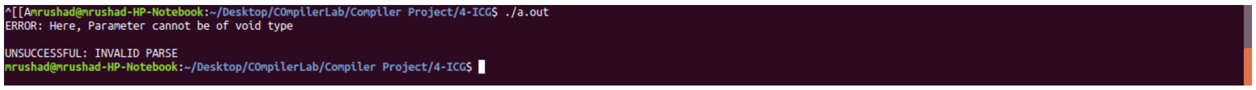
```
//WITH ERROR - This test case includes duplicate function declaration
#include<stdio.h>
void func()
{
        printf("hello\n");
}
void func()
{
        printf("hello\n");
}
void main()
{
        printf("hello\n");
}
```



```
^[[Amrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin func
param "hello\n"
refparam result
call printf, 1
func end

ERROR: Cannot Redeclare same function!

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$
```

Testcase 16

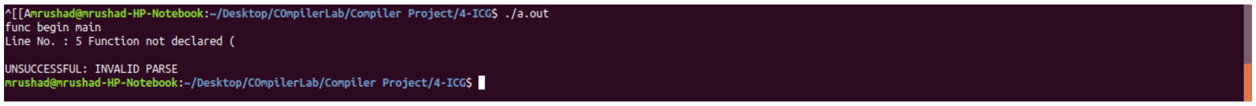
```c
//WITH ERROR - This test case includes void parameter for function
#include<stdio.h>
void func(void x)
{
        printf("hello\n");
}
void main()
{
        printf("hello\n");
}
```



```
^[[Amrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
ERROR: Here, Parameter cannot be of void type

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ 
```

Testcase 17

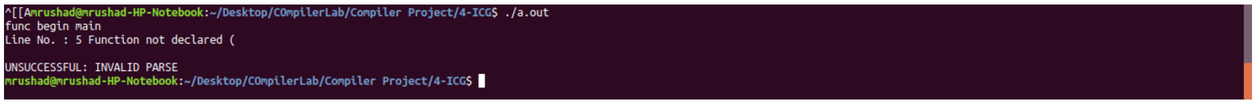
```c
//WITH ERROR - This test case includes a function call to undeclared function
#include<stdio.h>
void main()
{
        func();
}
```



```
^[[Amrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
Line No. : 5 Function not declared (

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ 
```

Testcase 18

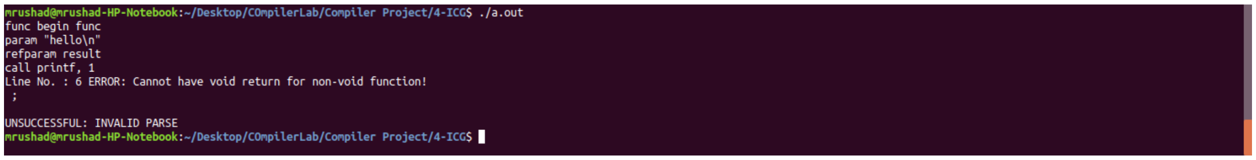
```c
//WITH ERROR - This test case includes a function call to function declared after main
#include<stdio.h>
void main()
{
        func();
}
void func()
{
        printf("hello\n");
}
```



```
^[[Amrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
Line No. : 5 Function not declared (

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ 
```

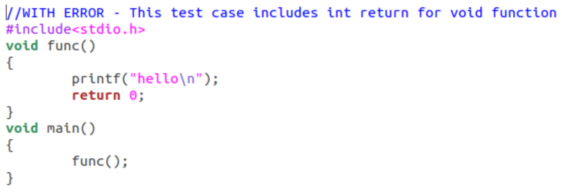
Testcase 19

```c
//WITH ERROR - This test case includes void return for int function
#include<stdio.h>
int func()
{
        printf("hello\n");
        return;
}
void main()
{
        func();
}
```

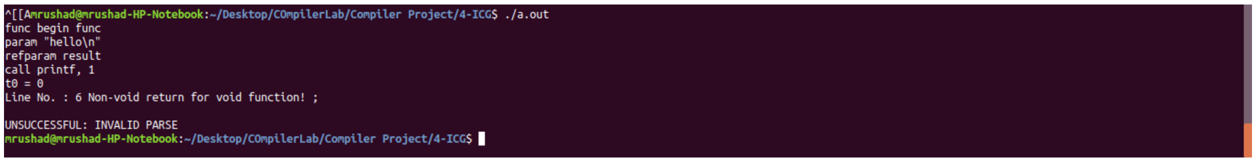


```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin func
param "hello\n"
refparam result
call printf, 1
Line No. : 6 ERROR: Cannot have void return for non-void function!
 ;

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ 
```

Testcase 20

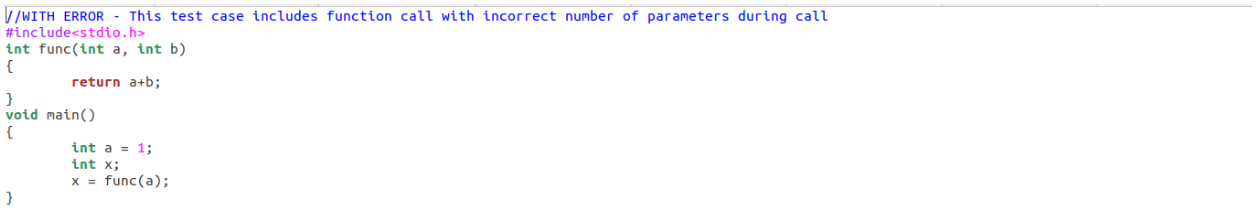


```
//WITH ERROR - This test case includes int return for void function
#include<stdio.h>
void func()
{
	printf("hello\n");
	return 0;
}
void main()
{
	func();
}
```

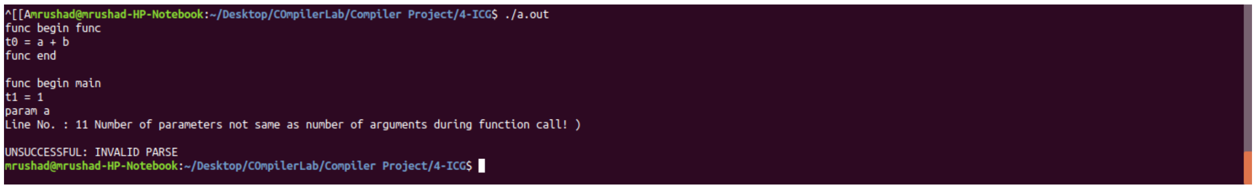


```
^[[Amrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin func
param "hello\n"
refparam result
call printf, 1
t0 = 0
Line No. : 6 Non-void return for void function! ;

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ 
```

Testcase 21

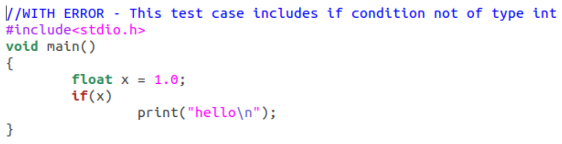


```
//WITH ERROR - This test case includes function call with incorrect number of parameters during call
#include<stdio.h>
int func(int a, int b)
{
	return a+b;
}
void main()
{
	int a = 1;
	int x;
	x = func(a);
}
```

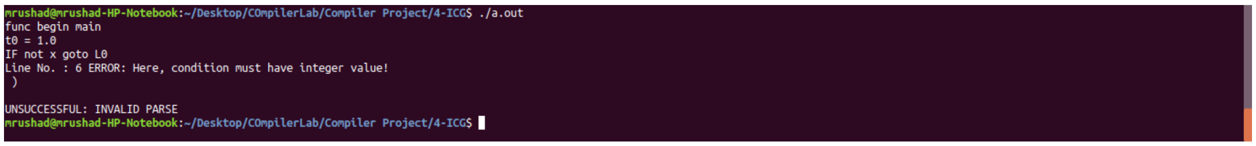


```
^[[Amrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin func
t0 = a + b
func end

func begin main
t1 = 1
param a
Line No. : 11 Number of parameters not same as number of arguments during function call! )

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ 
```

Testcase 22

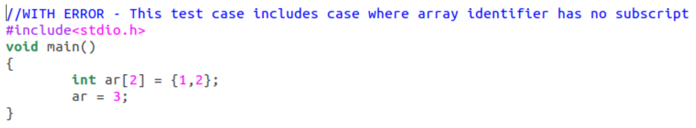


```
//WITH ERROR - This test case includes if condition not of type int
#include<stdio.h>
void main()
{
	float x = 1.0;
	if(x)
		print("hello\n");
}
```

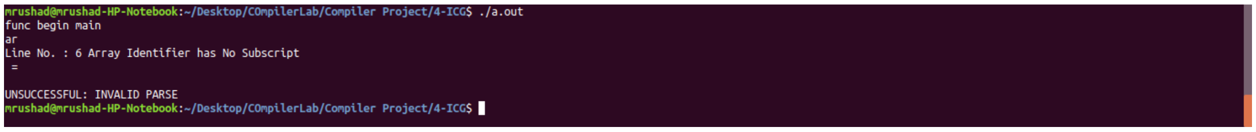


```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
t0 = 1.0
IF not x goto L0
Line No. : 6 ERROR: Here, condition must have integer value!
 )

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ 
```

Testcase 23

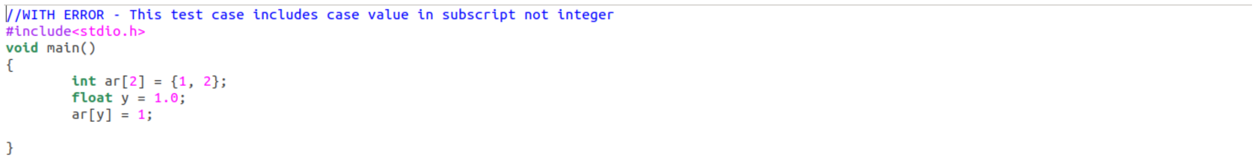


```
//WITH ERROR - This test case includes case where array identifier has no subscript
#include<stdio.h>
void main()
{
        int ar[2] = {1,2};
        ar = 3;
}
```

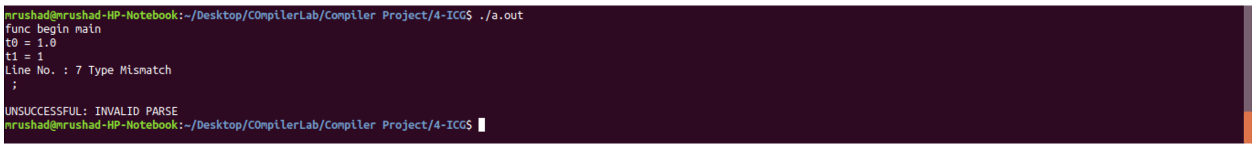


```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
ar
Line No. : 6 Array Identifier has No Subscript
 =

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$
```

Testcase 24

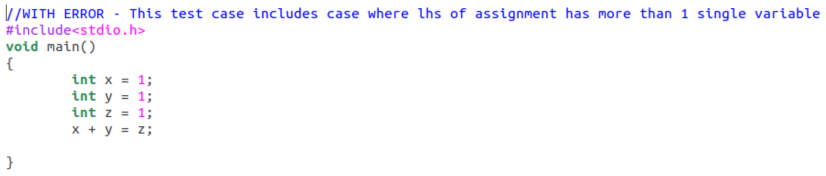


```
//WITH ERROR - This test case includes case value in subscript not integer
#include<stdio.h>
void main()
{
        int ar[2] = {1, 2};
        float y = 1.0;
        ar[y] = 1;

}
```

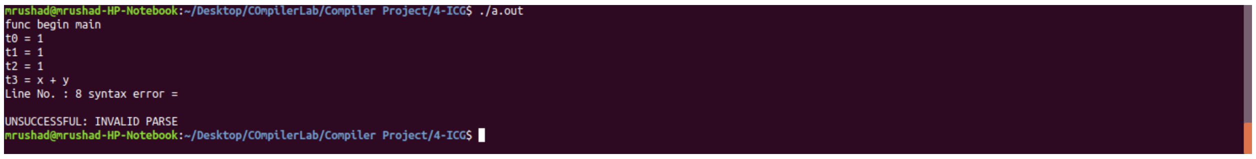


```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
t0 = 1.0
t1 = 1
Line No. : 7 Type Mismatch
 ;

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$
```

Testcase 25

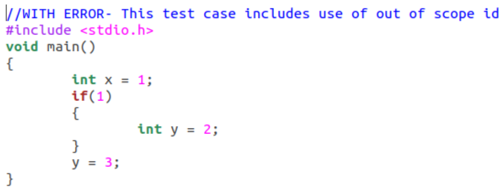


```
//WITH ERROR - This test case includes case where lhs of assignment has more than 1 single variable
#include<stdio.h>
void main()
{
        int x = 1;
        int y = 1;
        int z = 1;
        x + y = z;

}
```

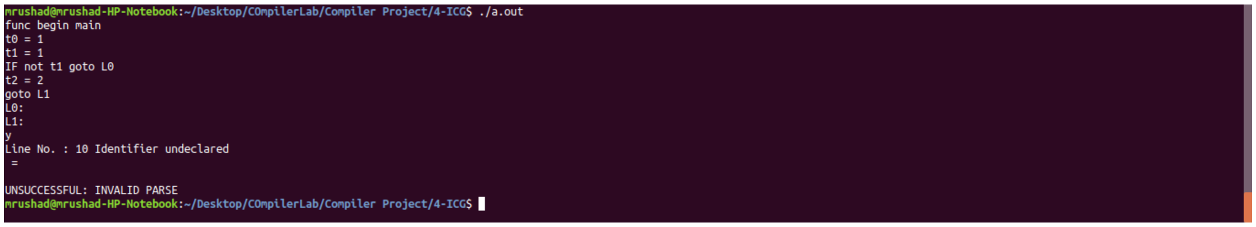


```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
t0 = 1
t1 = 1
t2 = 1
t3 = x + y
Line No. : 8 syntax error =

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$
```

Testcase 26



```
//WITH ERROR- This test case includes use of out of scope id
#include <stdio.h>
void main()
{
        int x = 1;
        if(1)
        {
                int y = 2;
        }
        y = 3;
}
```



```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$ ./a.out
func begin main
t0 = 1
t1 = 1
IF not t1 goto L0
t2 = 2
goto L1
L0:
L1:
y
Line No. : 10 Identifier undeclared
 =

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/4-ICG$
```

Implementation

The regular expressions used to identify the different tokens of the C language are fairly straightforward. Similar lexer code to the one submitted in the previous phase was used. A few features require a significant amount of thought are mentioned below:
• The Regex for Identifiers
• Support for Multi-line and Nested Comments and Error Handling for Unmatched Comments
• Literals
• Error Handling for Incomplete String
The parser uses a number of grammar production rules to implement the C programming language grammar. The parser also takes the help from the lexer for its functioning. The lexer outputs tokens one at a time and these tokens are used by the parser. The parser then applies the corresponding production rules on the token to insert the type, value, array dimensions, function parameters etc. into the symbol table. Along with this, semantic actions were also added to each Production rule to check if the structure created has some meaning or not.
Then we added the function to generate the 3 address code with production so that we can generate the desired intermediate code. In order to generate 3 address code we make use of stack. When we see an operator, operand or constant we push it into the stack. Whenever the reduction function is generated the 3 address code by creating a new temp variable and by making use of elements in the stack, soon after the elements are popped from the stack and we push the temp variable into the stack so that it gets used in later computation. Similarly functions like labels were used to assign appropriate labels while using conditional statements or iterative statements. All the functions used are described below:

1. generate_code():This function is called whenever a reduction of an expression takes place. It creates the temporary variable and displays the desired 3 address code i.e x = y op z.
2. generate_constant_code(): This function is especially written for reductions of expression involving constants since its 3 address code is x op z.
3. check_unary(): This function verifies whether the operater is binary eg ‘++’. If yes return true else false
4. generate_unary_code():This function is purposed to generate 3 address code for unary operations. It makes use of check_unary() function above.
5. control_start_label(): It is used while evaluating conditions of loops or if statement. If the condition is not satisfied then it states where to jump to i.e. on which label the control should go.
6. if_end_label(): It is used when the statement related to the if statement is over. It indicates where the control flow should go once that block is over i.e. it jumps the else statement block.
7. else_label(): it is used after the whole if else construct is over. It gives label that tells where to jump after the if block is executed.
8. loop_label(): it is used to give labels to starting of loops.
9. loop_end_label():it is used after the statement block of the loop. It indicates the label to jump to and also generates the label where the control should go once the loop is terminated.

10. start_function(): it indicates the start of a function
11. end_function(): it indicates the ending of a function.
12. generate_arguments_code(): it indicates the ending of a function.
13. generate_call_code() : It calls the function i.e. displays the appropriate function call according to 3 address code.

Future work

The yacc script presented in this report takes care of all the rules of C language, but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle all aspects of C language and making it more efficient.

Results

The parser was able to be successfully parse the tokens recognized by the flex script. The program gives us the list of the symbols and constants. The program also detects syntactical errors and semantics errors if found. Thus the semantic stage is an essential part of the compiler and is needed for the simplifications of the compiler. It makes the compiler more efficient and robust

References

• Compiler Principles, Techniques and Tool by Alfred V. Aho, Monica S. Lam,
Ravi Sethi, Jefferey D. Ullman

• https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_an
alysis