Compiler-Design (CS304)

Phase-3 Report

1. Bharadwaja M Chittapragada (211CS216)
2. Rishi Diwaker (211CS243)
3. Rohit Kumar (211CS244)

Abstract:

This project delves into the realm of compiler design, with a specific focus on advancing the semantic analysis phase for the C programming language. At the heart of this endeavor lies the development of a LALR parser tailored to handle the intricacies of semantic analysis. The primary goal is to augment the compiler's ability to comprehend the semantics of C code, thereby contributing to the creation of more robust and efficient compilers.

Semantic analysis, the third phase in the compilation process, plays a critical role in ensuring that the source code not only adheres to syntactic rules but also possesses meaning and intent. The intricacies of the C language pose unique challenges, demanding a sophisticated parsing mechanism capable of handling complex type systems, intricate scope rules, and nuanced language constructs.

The LALR (Look-Ahead LR) parsing technique stands out as a promising solution for addressing the complexities of C's syntax and semantics. LALR parsers exhibit efficiency and power in handling a wide range of grammars, making them well-suited for languages like C with rich and intricate syntax. By implementing a LALR parser tailored to the semantic analysis phase, this project aims to elevate the compiler's ability to analyze C code with precision.

The project's scope encompasses various facets of semantic analysis, with an emphasis on the following key aspects:

1. **Type Checking and Inference:**
   The LALR parser will be designed to perform rigorous type checking, ensuring that operations are applied to compatible operands and detecting type-related errors early in the compilation process. Type inference mechanisms will also be explored to deduce types when explicit declarations are absent.

2. **Scope Resolution and Symbol Table Management:**
   C's scoping rules can be intricate, and the parser will be equipped to resolve variable and symbol scopes accurately. The project includes the implementation of an efficient symbol table management system, facilitating the tracking and resolution of symbols throughout the compilation process.

3. **Error Detection and Reporting:**
   A robust error detection and reporting mechanism will be integrated into the LALR parser. This involves identifying and reporting semantic errors such as undeclared variables, incompatible types, and other issues that may compromise the program's correctness and reliability.

4. **Handling Function Overloading and Polymorphism:**
   C, although not inherently object-oriented, exhibits features that can lead to scenarios resembling function overloading and polymorphism. The LALR parser will be designed to navigate such scenarios effectively, ensuring correct function resolution based on argument types and handling polymorphic behavior.

The project's significance lies in its potential to enhance the compilation of C programs by addressing semantic intricacies that are often challenging to capture using traditional parsing techniques. A successful implementation of the LALR parser for semantic analysis has broader implications for software development, impacting the reliability, and efficiency of compiled C code.

The anticipated outcome of this project is a sophisticated LALR parser integrated into the semantic analysis phase, contributing to the development of more resilient compilers for the C language. This research endeavor aligns with the broader goal of advancing compiler technology and lays the foundation for future optimizations and innovations in the realm of programming language processing.

Introduction:

Compiler design is a critical aspect of software development, playing a pivotal role in translating high-level programming languages into machine code that can be executed by a computer. The process of compilation is typically divided into several phases, with semantic analysis being a crucial stage in ensuring the correctness and meaningfulness of the source code.

Semantic analysis, also known as phase 3 in the compiler design process, goes beyond the syntax-focused phases of lexical and syntax analysis. While syntax analysis checks the structure and grammar of the source code, semantic analysis delves into the

meaning and intent behind the code. It ensures that the program is not only syntactically correct but also semantically meaningful, addressing issues related to type checking, scope resolution, and overall program behavior.

This project focuses on the semantic analysis phase, aiming to explore and implement techniques that enhance the compiler's ability to understand the semantics of the source code accurately. By doing so, the project seeks to contribute to the creation of more robust and reliable compilers, ultimately improving the quality of compiled programs.

Semantic Analysis in Detail:

Semantic analysis involves a series of tasks that aim to extract the meaning of the source code. Here are some key aspects of semantic analysis:

1. **Type Checking:** One of the primary responsibilities of semantic analysis is to ensure that the types used in the program are consistent and compatible. This involves verifying that operations are applied to operands of appropriate types, preventing type-related errors during program execution.

2. **Scope Resolution:** Semantic analysis identifies and resolves the scope of variables and symbols within the program. This includes determining the visibility and accessibility of variables, functions, and other entities, ensuring that they are used in a valid context.

3. **Declaration Checking:** The compiler checks whether variables and functions are declared before they are used, preventing issues related to undeclared or improperly declared entities.

4. **Control Flow Analysis:** Semantic analysis may involve analyzing the flow of control within the program to ensure that statements and structures are used in a logically sound manner. This helps in detecting issues like unreachable code or potential infinite loops.

5. **Function Overloading and Polymorphism:** In languages that support function overloading and polymorphism, semantic analysis ensures that function calls are resolved correctly based on the number and types of arguments.

6. **Error Handling:** Semantic analysis is responsible for detecting and reporting errors that cannot be identified during earlier phases. This includes a wide range of issues, such as incompatible types, undeclared variables, or violations of scoping rules.

By addressing these aspects, semantic analysis contributes significantly to the production of efficient and error-free compiled code, laying the groundwork for the subsequent phases of compilation.

Design of Program:

Parser.y

```
%{
	void yyerror(char* s);
	int yylex();

	#include "stdio.h"
	#include "stdlib.h"
	#include "ctype.h"
	#include "string.h"

	void ins();
	void insV();

	int flag=0;

	extern char curid[20];
	extern char curtype[20];
	extern char curval[20];

	extern int current_nesting;
	extern char line_dec_type;

	void deletedata (int );
	int checkscope(char*);
	int check_id_is_func(char *);
	void insertST(char*, char*);
	void insertSTnest(char*, int);
```

```
        void insertSTparamscount(char*, int);
        int getSTparamscount(char*);
        int check_duplicate(char*);
        int check_declaration(char*, char *);
        int check_params(char*);
        int duplicate(char *s);
        int checkarray(char*);
        char currfunctype[100];
        char currfunc[100];
        char currfunccall[100];
        void insertSTF(char*);
        char gettype(char*,int);
        char getfirst(char*);
        void set_line_dec_type(char);
        int check_line_dec_type(char);
        extern int params_count;
        int call_params_count;
        void append_dim(int);
%}

%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED
UNSIGNED
%token INTs FLOATs CHARs DOUBLEs
%token CONST STRUCT ENUM UNION
%token RETURN MAIN
%token VOID
%token WHILE FOR DO
%token BREAK
%token ENDIF
```

```
%token SWITCH CASE CONTINUE DEFAULT SPREAD
%token AUTO STATIC REGISTER EXTERN VOLATILE INLINE
%token PRINTF SCANF

%token identifier array_identifier func_identifier
%token integer_constant string_constant float_constant
character_constant

%nonassoc ELSE

%right leftshift_assignment_operator
rightshift_assignment_operator
%right XOR_assignment_operator OR_assignment_operator
%right AND_assignment_operator modulo_assignment_operator
%right multiplication_assignment_operator
division_assignment_operator
%right addition_assignment_operator
subtraction_assignment_operator
%right assignment_operator

%left OR_operator
%left AND_operator
%left pipe_operator
%left caret_operator
%left amp_operator
%left equality_operator inequality_operator
%left lessthan_assignment_operator lessthan_operator
greaterthan_assignment_operator greaterthan_operator
%left leftshift_operator rightshift_operator
%left add_operator subtract_operator
```

```
%left multiplication_operator division_operator modulo_operator

%right SIZEOF
%right tilde_operator exclamation_operator
%left increment_operator decrement_operator


%start program

%%
program
                : declaration_list;

declaration_list
                : declaration D

D
                : declaration_list
                | ;

declaration
                : variable_declaration
                | function_declaration
                | structure_definition
        | enum_declaration;

variable_declaration
                : type_specifier variable_declaration_list ';'
{set_line_dec_type('n');}
```

```
                | storage_classes type_specifier
variable_declaration_list ';' {set_line_dec_type('n');}
                | storage_classes CONST type_specifier
variable_declaration_list ';' {set_line_dec_type('n');}
                | CONST type_specifier variable_declaration_list ';'
{set_line_dec_type('n');}
                | structure_declaration ';' {set_line_dec_type('n');}
                ;

storage_classes
        : AUTO | STATIC | REGISTER | EXTERN | VOLATILE

variable_declaration_list
                : variable_declaration_list ','
variable_declaration_identifier | variable_declaration_identifier;

variable_declaration_identifier
                : identifier {if(duplicate(curid)){printf("\n");} ins();  }
vdi
                  | array_identifier {if(duplicate(curid)){printf("\n");}
ins();  } vdi;



vdi : identifier_array_type | assignment_operator
simple_expression {if (check_line_dec_type('i') && $2 ==
1){printf("\n");} else {printf("Types don't match\n");exit(0);} } ;

identifier_array_type
                : '[' initilization_params
```

```
                | ;

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

initilization
                : string_initilization
                | array_initialization
                | ;

type_specifier
                : INT {set_line_dec_type('i');}
                | CHAR {set_line_dec_type('i');}
                | FLOAT {set_line_dec_type('i');}
                | DOUBLE {set_line_dec_type('i');}
                | star_specifier {set_line_dec_type('i');}
                | LONG long_grammar
                | SHORT short_grammar
                | UNSIGNED unsigned_grammar
                | SIGNED signed_grammar
                | VOID;

star_specifier
                : INTs | CHARs | FLOATs | DOUBLEs |
multiple_stars;

multiple_stars : star_specifier multiplication_operator;
```

```
unsigned_grammar
                : INT | LONG long_grammar | SHORT
short_grammar | ;

signed_grammar
                : INT | LONG long_grammar | SHORT
short_grammar | ;

long_grammar
                : INT  | ;

short_grammar
                : INT | ;

structure_definition
                : struct_or_union identifier { ins(); } '{' V1  '}' ';';

V1 : variable_declaration V1 | ;

structure_declaration
                : struct_or_union identifier
variable_declaration_list;

struct_or_union
          : STRUCT
          | UNION ;

function_declaration
```

```
                : function_declaration_type
function_declaration_param_statement;

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

function_declaration_param_statement
                : params ')' statement;

params
                : parameters_list | ;

parameters_list
                : type_specifier { check_params(curtype); }
parameters_identifier_list { insertSTparamscount(currfunc,
params_count); };

parameters_identifier_list
                : param_identifier
parameters_identifier_list_breakup;

parameters_identifier_list_breakup
                : ',' parameters_list
                | ;

param_identifier
                : identifier { ins(); params_count++; }
param_identifier_breakup;
```

```
param_identifier_breakup
                    : '[' ']'
                    | ;

statement
                    : expression_statment | compound_statement
                    | conditional_statements | iterative_statements
                    | return_statement | break_statement |
continue_statement
                    | variable_declaration
          | switch_case
                    | printf_scanf_statements;

printf_scanf_statements
                    : printf_statement ';'
                    | scanf_statement ';'
                    ;

printf_statement
                    : PRINTF '(' printf_parameters ')' ;

scanf_statement
                    :  SCANF '(' scanf_parameters ')' ;

printf_parameters
                    : printf_parameters ',' expression
                    | string_constant;

scanf_parameters
```

```
                : scanf_parameters ',' identifier
                | scanf_parameters ',' amp_operator identifier
                | string_constant;

compound_statement
                : {current_nesting++;} '{'  statment_list  '}'
{printST(); deletedata(current_nesting);current_nesting--;}  ;

statment_list
                : statement statment_list
                | ;

expression_statment
                : expression ';'
                | ';' ;

conditional_statements
                : IF '(' simple_expression ')'
{if($3!=1){printf("Condition checking is not of type int\n");exit(0);}}
statement conditional_statements_breakup;

conditional_statements_breakup
                : ELSE statement
                | ;

iterative_statements
                : WHILE '(' expression ')'
{if($3!=1){printf("Condition checking is not of type int\n");exit(0);}}
statement
```

```
                | FOR '(' variable_declaration_list ';'
simple_expression ';' {if($5!=1){printf("Condition checking is not of
type int\n");exit(0);}} expression ')'
                | FOR '(' {current_nesting++;} type_specifier
variable_declaration_list ';' simple_expression ';'
{if($7!=1){printf("Condition checking is not of type int\n");exit(0);}}
expression ')'
                | DO statement WHILE '(' simple_expression
')'{if($5!=1){printf("Condition checking is not of type int\n");exit(0);}}
';';

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

yyerror("Function returns something but is declared void");
                                                }


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

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

                                        };
```

```
break_statement
                : BREAK ';' ;

continue_statement
                : CONTINUE ';' ;

string_initilization
                : assignment_operator string_constant {insV();} ;

array_initialization
                : assignment_operator '{' array_int_declarations '}';

array_int_declarations
                : integer_constant
array_int_declarations_breakup;

array_int_declarations_breakup
                : ',' array_int_declarations
                | ;

expression
                : mutable assignment_operator expression
{
                        if($1==1 && $3==1)

                        {
                                                        $$=1;
                                                        }
```

```
                                                    else
                                                    {$$=-1;
printf("Type mismatch\n"); exit(0);}
                                                }
                | mutable addition_assignment_operator
expression    {

                        if($1==1 && $3==1)
                                                    $$=1;
                                                    else
                                                    {$$=-1;
printf("Type mismatch\n"); exit(0);}
                                                }
                | mutable subtraction_assignment_operator
expression  {

                        if($1==1 && $3==1)
                                                    $$=1;
                                                    else
                                                    {$$=-1;
printf("Type mismatch\n"); exit(0);}
                                                }
                | mutable multiplication_assignment_operator
expression {

                        if($1==1 && $3==1)
                                                    $$=1;
                                                    else
                                                    {$$=-1;
printf("Type mismatch\n"); exit(0);}
```

```
                                                    }
                | mutable division_assignment_operator
expression          {

                    if($1==1 && $3==1)
                                                    $$=1;
                                                    else
                                                    {$$=-1;
printf("Type mismatch\n"); exit(0);}
                                                    }
                | mutable modulo_assignment_operator
expression          {

                    if($1==1 && $3==1)
                                                    $$=1;
                                                    else
                                                    {$$=-1;
printf("Type mismatch\n"); exit(0);}
                                                    }
                | mutable increment_operator
            {if($1 == 1) $$=1; else $$=-1;}
                | mutable decrement_operator
                {if($1 == 1) $$=1; else $$=-1;}
                | simple_expression {if($1 == 1) $$=1; else $$=-1;}
                ;


simple_expression
                : simple_expression OR_operator and_expression
{if($1 == 1 && $3==1) $$=1; else $$=-1;}
```

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

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


unary_relation_expression
                : exclamation_operator unary_relation_expression
{if($2==1) $$=1; else $$=-1;}
                | regular_expression {if($1 == 1) $$=1; else
$$=-1;} ;

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

relational_operators
                : greaterthan_assignment_operator |
lessthan_assignment_operator | greaterthan_operator
                | lessthan_operator | equality_operator |
inequality_operator ;

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

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

sum_operators
                : add_operator
                | subtract_operator ;

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

MULOP
                : multiplication_operator | division_operator |
modulo_operator ;

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

mutable
                : identifier {
                                        if(check_id_is_func(curid))
                                        {printf("Function name used as
Identifier\n"); exit(8);}
                                if(!checkscope(curid))

{printf("%s\n",curid);printf("Undeclared\n");exit(0);}
                                if(!checkarray(curid))
                                {printf("%s\n",curid);printf("Array ID has
no subscript\n");exit(0);}
```

```
                        if(gettype(curid,0)=='i' ||
gettype(curid,1)== 'c')
                        $$ = 1;
                        else
                        $$ = -1;
                        }
                | array_identifier
{if(!checkscope(curid)){printf("%s\n",curid);printf("Undeclared\n");e
xit(0);}} '[' expression ']'
                            {if(gettype(curid,0)=='i' ||
gettype(curid,1)== 'c')
                                $$ = 1;
                                else
                                $$ = -1;
                                }
                | amp_operator identifier {$$ = 1;};

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

call
                : identifier '('{
                        if(!check_declaration(curid, "Function"))
                        { printf("Need to declare function");
exit(0);}
                        insertSTF(curid);
                                strcpy(currfunccall,curid);
                        } arguments ')'
```

```
                                    { if(strcmp(currfunccall,"printf"))
                                        {

if(getSTparamscount(currfunccall)!=call_params_count)
                                                {
                                                    yyerror("function
arguments required does not match the passed arguments...");
                                                    exit(8);
                                                }
                                        }
                                    };

arguments
                : arguments_list | ;

arguments_list
                : expression { call_params_count++; } A ;

A
                : ',' expression { call_params_count++; } A
                | ;

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

enum_declaration
        : ENUM identifier '{' enum_list '}' ';'
```

```
        ;

enum_list
        : enumerator
        | enum_list ',' enumerator ;

enumerator
        : identifier
        | identifier assignment_operator integer_constant
        ;

switch_case
        : SWITCH  '(' identifier ')' '{' case_list '}' ;

case_list
        : case_entry BREAK ';'
        | case_list case_entry BREAK ';'
        ;

case_entry
        : CASE constant ':' statement
        | CASE constant SPREAD constant':' statement
        | DEFAULT ':' statement
        |
        ;

%%

extern FILE *yyin;
extern int yylineno;
```

```
extern char *yytext;
void insertSTtype(char *,char *);
void insertSTvalue(char *, char *);
void incertCT(char *, char *);
void printST();
void printCT();

int main(int argc , char **argv)
{
	yyin = fopen(argv[1], "r");
	yyparse();

	if(flag == 0)
	{
		printf( "PASSED: Semantic Phase\n");
		printf("%30s"  "PRINTING SYMBOL TABLE"  "\n\n", " ");
		printST();
		printf("%30s %s\n", " ", "------------------------------");
		printf("\n\n%30s"  "PRINTING CONSTANT TABLE"
"\n\n", " ");
		printCT();
	}
}

void yyerror(char *s)
{
	printf( "%d %s %s\n", yylineno, s, yytext);
	flag=1;
	printf( "FAILED: Semantic Phase Parsing failed\n" );
	exit(7);
```

```
}

void ins()
{
	insertSTtype(curid,curtype);
}

void insV()
{
	insertSTvalue(curid,curval);
}

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

Scanner.l

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

	struct symboltable
	{
		char name[100];
		char class[100];
		char type[100];
```

```
        char value[100];
        char dimension[100];
        int nesting_val;
        int lineno;
        int length;
        int params_count;
}ST[1007];

struct constanttable
{
        char name[100];
        char type[100];
        int length;
}CT[1007];

int current_nesting = 0;
int params_count = 0;
char line_dec_type = 'n';
extern int yylval;

void set_line_dec_type(char type){
        line_dec_type = type;
}
int check_line_dec_type(char type_to_check){
        return (line_dec_type == type_to_check);
}

char array_dim_str[10000] = "";

void append_dim(int* val_to_append){
```

```
        char temp[1000];
        sprintf(temp," %d",val_to_append);
        strcat(array_dim_str,temp);
}

int hash(char *str)
{
        int value = 0;
        for(int i = 0 ; i < strlen(str) ; i++)
        {
                value = 10*value + (str[i] - 'A');
                value = value % 1007;
                while(value < 0)
                        value = value + 1007;
        }
        return value;
}

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

                return value;
        }
```

```
	else
	{
		for(int i = value + 1 ; i!=value ; i = (i+1)%1007)
		{
			if(strcmp(ST[i].name,str)==0)
			{

				return i;
			}
		}
		return 0;
	}
}

int lookupCT(char *str)
{
	int value = hash(str);
	if(CT[value].length == 0)
		return 0;
	else if(strcmp(CT[value].name,str)==0)
		return 1;
	else
	{
		for(int i = value + 1 ; i!=value ; i = (i+1)%1007)
		{
			if(strcmp(CT[i].name,str)==0)
			{
				return 1;
			}
		}
```

```
                return 0;
        }
    }

    void insertSTline(char *str1, int line)
    {
        for(int i = 0 ; i < 1007 ; i++)
        {
            if(strcmp(ST[i].name,str1)==0)
            {
                ST[i].lineno = line;
            }
        }
    }


    void insertST(char *str1, char *str2)
    {
        if(lookupST(str1))
        {
            if(strcmp(ST[lookupST(str1)].class,"Identifier")==0
&& strcmp(str2,"Array Identifier")==0)
            {
                printf("Error use of array\n");
                exit(0);
            }
            return;
        }
        else
        {
```

```
                int value = hash(str1);
                if(ST[value].length == 0)
                {
                        strcpy(ST[value].name,str1);
                        strcpy(ST[value].class,str2);
                        ST[value].length = strlen(str1);
                        ST[value].nesting_val = current_nesting;
                        ST[value].params_count = -1;
                        insertSTline(str1,yylineno);
                        return;
                }

                int pos = 0;

                for (int i = value + 1 ; i!=value ; i = (i+1)%1007)
                {
                        if(ST[i].length == 0)
                        {
                                pos = i;
                                break;
                        }
                }

                strcpy(ST[pos].name,str1);
                strcpy(ST[pos].class,str2);
                ST[pos].length = strlen(str1);
                ST[pos].nesting_val = current_nesting;
                ST[pos].params_count = -1;
        }
}
```

```
	void insertSTtype(char *str1, char *str2)
	{
		for(int i = 0 ; i < 1007 ; i++)
		{
			if(strcmp(ST[i].name,str1)==0)
			{
				strcpy(ST[i].type,str2);
			}
		}
   }

	void insertSTvalue(char *str1, char *str2)
	{
		for(int i = 0 ; i < 1007 ; i++)
		{
			if(strcmp(ST[i].name,str1)==0)// &&
ST[i].nesting_val == current_nesting)
			{
				strcpy(ST[i].value,str2);
			}
		}
		printf("\n");
	}


	void insertSTnest(char *s, int nest)
	{
		if(lookupST(s) && ST[lookupST(s)].nesting_val != 9999)
		{
```

```
    int pos = 0;
    int value = hash(s);
            for (int i = value + 1 ; i!=value ; i = (i+1)%1007)
            {
                if(ST[i].length == 0)
                {
                    pos = i;
                    break;
                }
            }

            strcpy(ST[pos].name,s);
            strcpy(ST[pos].class,"Identifier");
            ST[pos].length = strlen(s);
            ST[pos].nesting_val = nest;
            ST[pos].params_count = -1;
            ST[pos].lineno = yylineno;
        }
        else
        {
            for(int i = 0 ; i < 1007 ; i++)
            {
                if(strcmp(ST[i].name,s)==0 )
                {
                    ST[i].nesting_val = nest;
                }
            }
        }
}
```

```
void insertSTparamscount(char *s, int count)
{
	for(int i = 0 ; i < 1007 ; i++)
	{
		if(strcmp(ST[i].name,s)==0 )
		{
			ST[i].params_count = count;
		}
	}
}

int getSTparamscount(char *s)
{
	for(int i = 0 ; i < 1007 ; i++)
	{
		if(strcmp(ST[i].name,s)==0 )
		{
			return ST[i].params_count;
		}
	}
	return -2;
}

void insertSTF(char *s)
{
	for(int i = 0 ; i < 1007 ; i++)
	{
		if(strcmp(ST[i].name,s)==0 )
		{
			strcpy(ST[i].class,"Function");
```

```
                return;
            }
        }

}

void insertCT(char *str1, char *str2)
{
        if(lookupCT(str1))
            return;
        else
        {
            int value = hash(str1);
            if(CT[value].length == 0)
            {
                strcpy(CT[value].name,str1);
                strcpy(CT[value].type,str2);
                CT[value].length = strlen(str1);
                return;
            }

            int pos = 0;

            for (int i = value + 1 ; i!=value ; i = (i+1)%1007)
            {
                if(CT[i].length == 0)
                {
                    pos = i;
                    break;
                }
```

```
            }

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

void deletedata (int nesting)
{
        for(int i = 0 ; i < 1007 ; i++)
        {
            if(ST[i].nesting_val == nesting)
            {
                ST[i].length = 0 ;
            }
        }


}

int checkscope(char *s)
{
        int flag = 0;
        for(int i = 0 ; i < 1000 ; i++)
        {
            if(strcmp(ST[i].name,s)==0)
            {
                if(ST[i].nesting_val > current_nesting)
                {
```

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

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

```
}

int checkarray(char *s)
{
	for(int i = 0 ; i < 1000 ; i++)
	{
		if(strcmp(ST[i].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].name,s)==0)
		{
			if(ST[i].nesting_val == current_nesting)
			{
			return 1;
			}
		}
	}
```

```
        return 0;
    }

    int check_duplicate(char* str)
    {
        for(int i=0; i<1007; i++)
        {
            if(strcmp(ST[i].name, str) == 0 &&
strcmp(ST[i].class, "Function") == 0)
            {
                printf("Function redeclaration not allowed\n");
                exit(0);
            }
        }
    }

    int check_declaration(char* str, char *check_type)
    {
        for(int i=0; i<1007; i++)
        {
            if(strcmp(ST[i].name, str) == 0 &&
strcmp(ST[i].class, "Function") == 0 ||
strcmp(ST[i].name,"printf")==0 )
            {
                return 1;
            }
        }
        return 0;
    }
```

```
int check_params(char* type_specifier)
{
	if(!strcmp(type_specifier, "void"))
	{
		printf("Parameters cannot be of type void\n");
		exit(0);
	}
	return 0;
}

char gettype(char *s, int flag)
{
		for(int i = 0 ; i < 1007 ; i++ )
		{
			if(strcmp(ST[i].name,s)==0)
			{
				return ST[i].type[0];
			}
		}

}



void process_array_dimensions(){
	// char* cur_array_str = strtok(array_dim_str,"\n");
	// while (cur_array_str != NULL){ // <array name> dim1
dim2
	//	char array_name[1000] = "";
	//	char value[100];
```

```
//      int index = 0;

//      while(index < strlen(cur_array_str)){
//              if(cur_array_str[index] == ' ') break;
//              strcat(array_name, cur_array_str[index]);
//              index++;
//      }

//      while(index < strlen(cur_array_str)){
//              strcat(value, cur_array_str[index]);
//              index++;
//      }

//      printf("Array name is %s\n",array_name);

//      insertSTvalue(array_name, value);

//      cur_array_str = strtok(NULL,"\n");
// }

// char input[] = "id 1 2 3 \n id 4 5 6 \n id 7 8 9 \n";

char *input = array_dim_str;
char id[100];
char numbers[100];
int offset = 0;

while (sscanf(input + offset, "%s %[^\n]s", id, numbers)
== 2) {
```

```
                offset += snprintf(NULL, 0, "%s %s", id, numbers)
+ 1;
                for(int i = 0 ; i < 1007 ; i++)
                {
                        if(strcmp(ST[i].name,id)==0)// &&
ST[i].nesting_val == current_nesting)
                        {
                                strcpy(ST[i].value,numbers);
                        }
                }
        }

    }

    void printST()
    {
        process_array_dimensions();


        printf("%10s | %15s | %10s | %10s | %10s | %15s |
%10s |\n","symbol name", "Class", "Type","Value", "Line No.",
"Nesting Count", "Count of Params");
        for(int i=0;i<100;i++) {
                printf("_");
        }
        printf("\n");
        for(int i = 0 ; i < 1007 ; i++)
        {
                if(ST[i].length == 0)
                {
```

```
            continue;
        }

        // int first_occurrance = 0;
        // int dup[1007];
        // for (int i = 0;i < 1007;i++){
        //      dup[i] = -1;
        // }



        // for (int j = 0;j < 1007;j++){

        //      if(strcmp(ST[i].name,ST[j].name)==0) {

        //              // if(ST[i].nesting_val ==
ST[j].nesting_val) //Remove if this thing doesn't work
        //              {
        //                      first_occurrance = 1;
        //                      int idx = 0;
        //                      for (;idx<1007;idx++){
        //                              if (dup[idx] == -1){
        //                                      dup[idx] = ST[j].lineno;
        //                                      break;
        //                              }
        //                              // printf("hello");
        //                      }
        //              }
        //      }
        // }
        // char line_no_str[3000] = "";
```

```
                    // if (!first_occurrance){
                    //      continue;
                    // } else {

                    //      for (int idx = 0;idx < 1007;idx++){
                    //              if (dup[idx] == -1){
                    //                      break;
                    //              }
                    //              printf("hi");
                    //
sprintf(line_no_str,"%s%d->",line_no_str,dup[idx]);
                    //      }
                    // }

                    // printf("%10s | %15s | %10s | %10s | %10s |
%15d | %10d |\n",ST[i].name, ST[i].class, ST[i].type, ST[i].value,
line_no_str, ST[i].nesting_val, ST[i].params_count);
                    if(ST[i].params_count == -1){
                            printf("%10s | %15s | %10s | %10s | %10d |
%15d | %10d |\n",ST[i].name, ST[i].class, ST[i].type, ST[i].value,
ST[i].lineno, ST[i].nesting_val, 0);
                    } else {
                            printf("%10s | %15s | %10s | %10s | %10d |
%15d | %10d |\n",ST[i].name, ST[i].class, ST[i].type, ST[i].value,
ST[i].lineno, ST[i].nesting_val, ST[i].params_count);
                    }

                    /*printf("%10s | %15s | %10s | %10s |
",ST[i].name, ST[i].class, ST[i].type, ST[i].value);
            for (int j = 0;j < 100;j++){
```

```
            if (ST[i].lineno[j] == -1) break;
            printf("%d -> ",ST[i].lineno[j]);
        }
        printf("\n");*/
            }
                                        // printf("hello");
    }


        void printCT()
        {
            printf("%10s | %15s\n","constant name", "constant
type");
            for(int i=0;i<85;i++) {
                printf("_");
            }
            printf("\n");
            for(int i = 0 ; i < 1007 ; i++)
            {
                if(CT[i].length == 0)
                    continue;

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

            // printf("%s\n",array_dim_str);
        }
        char curid[20];
        char curtype[20];
        char curval[20];
```

```
%}

DE "define"
IN "include"

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

"const"                { insertST(yytext, "Keyword");  return CONST;}
```

```
"default"          { insertST(yytext, "Keyword");  return DEFAULT;}
"char"                    { strcpy(curtype,yytext); insertST(yytext,
"Keyword");return CHAR;}
"double"          { strcpy(curtype,yytext); insertST(yytext,
"Keyword"); return DOUBLE;}
"else"                    { insertST(yytext, "Keyword"); return ELSE;}
"float"                   { strcpy(curtype,yytext); insertST(yytext,
"Keyword"); return FLOAT;}
"while"                   { insertST(yytext, "Keyword"); return WHILE;}
"do"              { insertST(yytext, "Keyword"); return DO;}
"for"             { insertST(yytext, "Keyword"); return FOR;}
"if"              { insertST(yytext, "Keyword"); return IF;}
"int"             { strcpy(curtype,yytext); insertST(yytext,
"Keyword"); return INT;}
"long"                    { strcpy(curtype,yytext); insertST(yytext,
"Keyword"); return LONG;}
"return"          { insertST(yytext, "Keyword"); return RETURN;}
"short"                   { strcpy(curtype,yytext); insertST(yytext,
"Keyword"); return SHORT;}
"signed"          { strcpy(curtype,yytext); insertST(yytext,
"Keyword"); return SIGNED;}
"sizeof"          { insertST(yytext, "Keyword"); return SIZEOF;}
"struct"          { strcpy(curtype,yytext);   insertST(yytext,
"Keyword");  return STRUCT;}
"enum"               { strcpy(curtype,yytext); insertST(yytext,
"Keyword");  return ENUM;}
"union"              { strcpy(curtype,yytext); insertST(yytext,
"Keyword");  return UNION;}
"unsigned"           { insertST(yytext, "Keyword");   return
UNSIGNED;}
```

```
"void"                  { strcpy(curtype,yytext);   insertST(yytext,
"Keyword");  return VOID;}
"break"                 { insertST(yytext, "Keyword");  return
BREAK;}
"switch"          { insertST(yytext, "Keyword");  return SWITCH;}
"case"                  { insertST(yytext, "Keyword");  return CASE;}
"continue"        { insertST(yytext, "Keyword");  return CONTINUE;}

"int"\s*\*+         { strcpy(curtype,yytext); insertST(yytext,
"Keyword");return INTs;}
"float"\s*\*+       { strcpy(curtype,yytext); insertST(yytext,
"Keyword");return FLOATs;}
"char"\s*\*+        { strcpy(curtype,yytext); insertST(yytext,
"Keyword");return CHARs;}
"double"\s*\*+  { strcpy(curtype,yytext); insertST(yytext,
"Keyword");return DOUBLEs;}

"auto"          { insertST(yytext, "Keyword"); return AUTO; }
"static"        { insertST(yytext, "Keyword"); return STATIC; }
"register"      { insertST(yytext, "Keyword"); return REGISTER; }
"extern"        { insertST(yytext, "Keyword"); return EXTERN; }
"volatile"      { insertST(yytext, "Keyword"); return EXTERN; }
"inline"        { insertST(yytext, "Keyword"); return INLINE; }

"printf"          { insertST(yytext, "Keyword");  return PRINTF;}
"scanf"                 { insertST(yytext, "Keyword");  return
SCANF;}


"++"                { return increment_operator; }
```

```
"--"            { return decrement_operator; }
"<<"            { return leftshift_operator; }
">>"            { return rightshift_operator; }
"<="            { return lessthan_assignment_operator; }
"<"                 { return lessthan_operator; }
">="            { return greaterthan_assignment_operator; }
">"                 { return greaterthan_operator; }
"=="            { return equality_operator; }
"!="            { return inequality_operator; }
"&&"            { return AND_operator; }
"||"            { return OR_operator; }
"^"                 { return caret_operator; }
"*="            { return multiplication_assignment_operator; }
"/="            { return division_assignment_operator; }
"%="            { return modulo_assignment_operator; }
"+="            { return addition_assignment_operator; }
"-="            { return subtraction_assignment_operator; }
"<<="               { return leftshift_assignment_operator; }
">>="               { return rightshift_assignment_operator; }
"&="            { return AND_assignment_operator; }
"^="            { return XOR_assignment_operator; }
"|="            { return OR_assignment_operator; }
"&"                 { return amp_operator; }
"!"                 { return exclamation_operator; }
"~"                 { return tilde_operator; }
"-"                 { return subtract_operator; }
"+"                 { return add_operator; }
"*"                 { return multiplication_operator; }
"/"                 { return division_operator; }
"%"                 { return modulo_operator; }
```

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

\"[^\n]*\"/[;|,|\)]             {strcpy(curval,yytext);
insertCT(yytext,"String Constant"); return string_constant;}
\'[A-Z|a-z]\'/[;|,|\)|:]        {strcpy(curval,yytext);
insertCT(yytext,"Character Constant"); return
character_constant;}
[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[      {strcpy(curid,yytext);
insertST(yytext, "Array Identifier");
strcat(array_dim_str,"\n");strcat(array_dim_str,yytext); return
array_identifier;}
[1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^]
{strcpy(curval,yytext); insertCT(yytext, "Number Constant"); yylval
= atoi(yytext); return integer_constant;}
([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^]
{strcpy(curval,yytext); insertCT(yytext, "Floating Constant"); return
float_constant;}
[A-Za-z_][A-Za-z_0-9]* {strcpy(curid,yytext);
insertST(curid,"Identifier"); return identifier;}

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

```
                printf("ERR_UNMATCHED_COMMENT at line no.
%d\n",yylineno);
        }
        else if(yytext[0]=="")
        {
                printf("ERR_INCOMPLETE_STRING at line no.
%d\n",yylineno);
        }
        else
        {
                printf("ERROR at line no. %d\n",yylineno);
        }
        printf("%s\n", yytext);
        return 0;
}

%%
```

Explanation

1. **Type Checking:** Ensuring that the types of operands in expressions and statements are compatible with the language specifications. This involves verifying that operations are applied to appropriate data types, preventing potential runtime errors.

2. **Scope Resolution:** Determining the scope of variables, functions, and other symbols within the code. This involves

associating each identifier with its correct declaration, considering issues like nested scopes and variable shadowing.

3. **Symbol Table Management:** Building and maintaining a symbol table, which is a data structure that stores information about variables, functions, and other symbols in the program. The symbol table aids in resolving names and ensuring consistency in their usage.

4. **Error Detection and Reporting:** Identifying and reporting semantic errors that cannot be captured during lexical or syntax analysis. This includes issues like undeclared variables, incompatible types, and misuse of language constructs

5. **Intermediate Code Generation:** In some cases, semantic analysis involves generating an intermediate representation of the code that abstracts away from the specifics of the source or target architecture. This intermediate code serves as a bridge between the analysis and synthesis phases of the compiler.

Symbol Table structure

In the semantic phase, every function that is described has a symbol table associated with it. The symbol table for a function contains the following columns.

1. **Symbol name:** This field contains the name of the variables or the function itself, along with all keywords like char, float, int, while, for, etc.

2. **Class:** This field describes the type of the corresponding symbol name.

3. **Type:** This field declares the data type of each symbol name such as int, void, etc.

4. **Value:** For each variable, this field declares its value and the dimensions of the arrays being stored.

5. **Line Number:** This field contains the line number of occurrence of the corresponding symbol to identify where in the code it has been declared and used.

6. **Nesting Count:** This field contains the scope of the variable initially set to zero for global variables. For each nesting loop, the count is incremented by 1.

7. **Count of parameters:** This field is used to keep track of the number of parameters passed on as arguments during the function declaration, to check for argument count mismatches.

Test Cases:

Test 1:

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

```
int myfunc(int b)
{
    int x;
    return x;
}

void main()
{
    int n;
    char ch;
    int x = 9;
    int a[10][8];
    float basdf[5][13];
    for (int i=0;i<10;i++){
        if(i<10){
            int y;
            while(x<10){
                x++;
            }
        }
    }
   x=3;

}
```

```
rishi@rishi-HP-Pavilion-Gaming-Laptop-15-dk2xxx:~/phase3$ ./a.out tests/test1.c

symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          b |     Identifier |        int |            |          3 |               0 |               0 |
          x |     Identifier |        int |            |          5 |               1 |               0 |
        int |        Keyword |            |            |          3 |               0 |               0 |
     return |        Keyword |            |            |          6 |               1 |               0 |
     myfunc |       Function |        int |            |          3 |               0 |               1 |
```

```
symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
       char |        Keyword |            |            |         12 |               1 |               0 |
          a | Array Identifier |        int |       10 8 |         14 |               1 |               0 |
          b |     Identifier |        int |            |          3 |               0 |               0 |
          i |     Identifier |        int |         10 |         16 |               2 |               0 |
          n |     Identifier |        int |            |         11 |               1 |               0 |
          x |     Identifier |        int |         10 |         13 |               1 |               0 |
          y |     Identifier |        int |            |         18 |               4 |               0 |
      float |        Keyword |            |            |         15 |               1 |               0 |
        for |        Keyword |            |            |         16 |               1 |               0 |
       main |       Function |       void |            |          9 |               0 |               0 |
         ch |     Identifier |       char |            |         12 |               1 |               0 |
      while |        Keyword |            |            |         19 |               4 |               0 |
         if |        Keyword |            |            |         17 |               3 |               0 |
        int |        Keyword |            |            |          3 |               0 |               0 |
       void |        Keyword |            |            |          9 |               0 |               0 |
      basdf | Array Identifier |      float |      5 13 |         15 |               1 |               0 |
     myfunc |       Function |        int |            |          3 |               0 |               1 |
```

```
                          PRINTING CONSTANT TABLE

 constant name |   constant type
 ____________________________________________________________________________________________
          10 | Number Constant
          13 | Number Constant
           0 | Number Constant
           3 | Number Constant
           5 | Number Constant
           8 | Number Constant
           9 | Number Constant
```

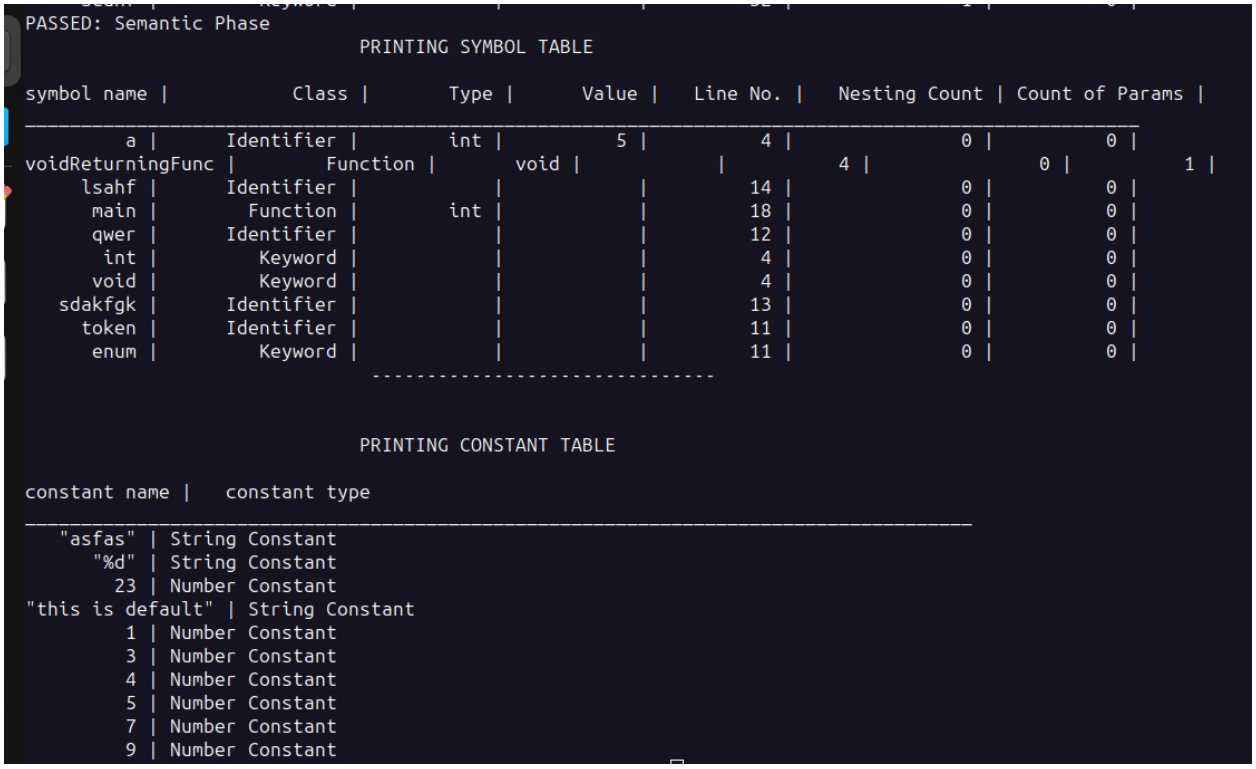
Test2

```
//error test for function return mismatch
#include<stdio.h>

void voidReturningFunc(int a)
{
```

```
    int abc=1;
    a = a + 5;

}

enum token {
    qwer=9,
    sdakfgk=23,
    lsahf
};


int main()
{
    int i,n;

    voidReturningFunc(i);

    switch (i) {
        case 1:
            printf("asfas");
            break;
        case 3:
            printf("asfas");
            break;
        case 4 ... 7:
            scanf("%d", n);
            break;
        default:
            printf("this is default");
```

```
        break;
    }


    printf("%d",i);
}
```

```
rishi@rishi-HP-Pavilion-Gaming-Laptop-15-dk2xxx:~/phase3$ ./a.out tests/test2.c



symbol name |           Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          a |      Identifier |        int |          5 |          4 |               0 |               0 |
voidReturningFunc |        Function |       void |            |          4 |               0 |               1 |
        int |         Keyword |            |            |          4 |               0 |               0 |
        abc |      Identifier |        int |          1 |          6 |               1 |               0 |
       void |         Keyword |            |            |          4 |               0 |               0 |




symbol name |           Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          a |      Identifier |        int |          5 |          4 |               0 |               0 |
          i |      Identifier |        int |          7 |         20 |               1 |               0 |
          n |      Identifier |        int |            |         20 |               1 |               0 |
voidReturningFunc |        Function |       void |            |          4 |               0 |               1 |
      lsahf |      Identifier |            |            |         14 |               0 |               0 |
      break |         Keyword |            |            |         27 |               1 |               0 |
       main |        Function |        int |            |         18 |               0 |               0 |
       qwer |      Identifier |            |            |         12 |               0 |               0 |
        int |         Keyword |            |            |          4 |               0 |               0 |
       case |         Keyword |            |            |         25 |               1 |               0 |
    default |         Keyword |            |            |         34 |               1 |               0 |
       void |         Keyword |            |            |          4 |               0 |               0 |
     switch |         Keyword |            |            |         24 |               1 |               0 |
     printf |         Keyword |            |            |         26 |               1 |               0 |
    sdakfgk |      Identifier |            |            |         13 |               0 |               0 |
      token |      Identifier |            |            |         11 |               0 |               0 |
       enum |         Keyword |            |            |         11 |               0 |               0 |
      scanf |         Keyword |            |            |         32 |               1 |               0 |
PASSED: Semantic Phase
                            PRINTING SYMBOL TABLE
```

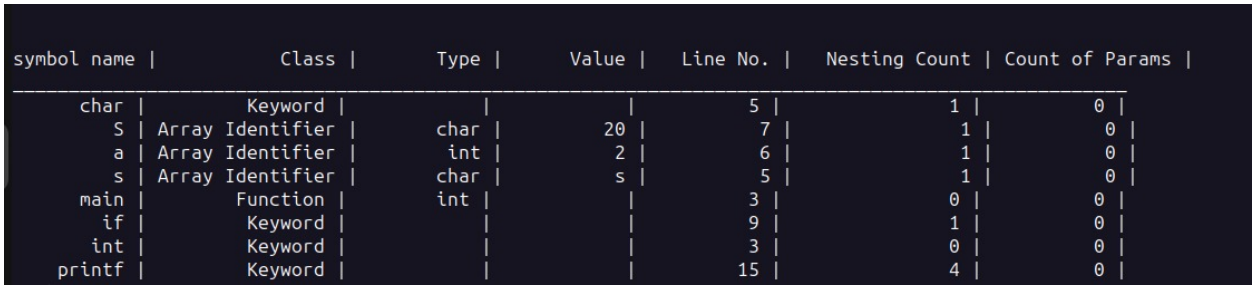


```
PASSED: Semantic Phase
                            PRINTING SYMBOL TABLE

symbol name |           Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          a |      Identifier |        int |          5 |          4 |               0 |               0 |
voidReturningFunc |        Function |       void |            |          4 |               0 |               1 |
      lsahf |      Identifier |            |            |         14 |               0 |               0 |
       main |        Function |        int |            |         18 |               0 |               0 |
       qwer |      Identifier |            |            |         12 |               0 |               0 |
        int |         Keyword |            |            |          4 |               0 |               0 |
       void |         Keyword |            |            |          4 |               0 |               0 |
    sdakfgk |      Identifier |            |            |         13 |               0 |               0 |
      token |      Identifier |            |            |         11 |               0 |               0 |
       enum |         Keyword |            |            |         11 |               0 |               0 |
                              ------------------------------


                            PRINTING CONSTANT TABLE

constant name |   constant type
_____________________________________________________________________________
    "asfas" | String Constant
       "%d" | String Constant
         23 | Number Constant
"this is default" | String Constant
          1 | Number Constant
          3 | Number Constant
          4 | Number Constant
          5 | Number Constant
          7 | Number Constant
          9 | Number Constant
```

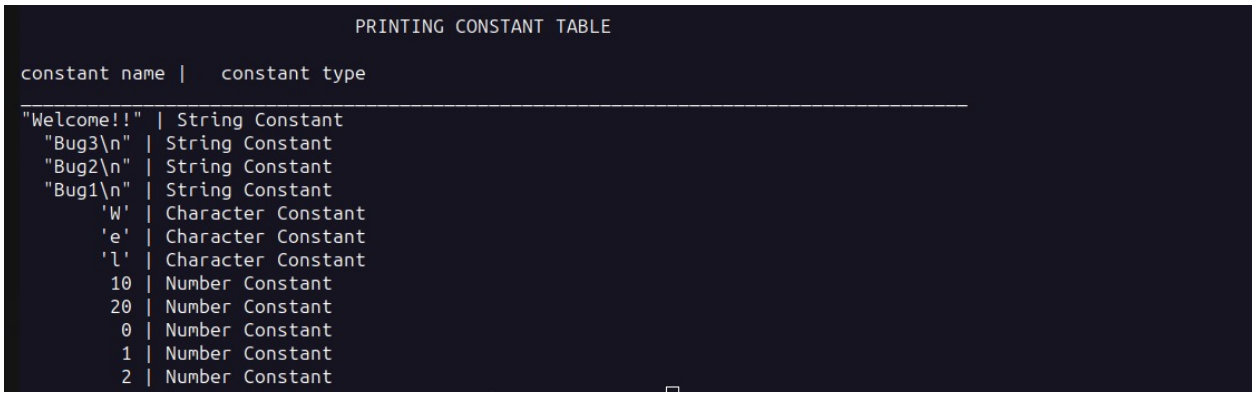
Test3

```
#include <stdio.h>

int main()
{
    char s[10] = "Welcome!!";
    int a[2] = {1, 2};
    char S[20];

    if (s[0] == 'W')
    {
        if (s[1] == 'e')
        {
            if (s[2] == 'l')
            {
```

```
            printf("Welcome!!");
        }

        else
            printf("Bug1\n");
    }
    else
        printf("Bug2\n");
}

else
    printf("Bug3\n");
}
```



symbol name	Class	Type	Value	Line No.	Nesting Count	Count of Params
char	Keyword			5	1	0
S	Array Identifier	char	20	7	1	0
a	Array Identifier	int	2	6	1	0
s	Array Identifier	char	s	5	1	0
main	Function	int		3	0	0
if	Keyword			9	1	0
int	Keyword			3	0	0
printf	Keyword			15	4	0



PRINTING CONSTANT TABLE

constant name	constant type
"Welcome!!"	String Constant
"Bug3\n"	String Constant
"Bug2\n"	String Constant
"Bug1\n"	String Constant
'W'	Character Constant
'e'	Character Constant
'l'	Character Constant
10	Number Constant
20	Number Constant
0	Number Constant
1	Number Constant
2	Number Constant

Test 4

```
#include <stdio.h>

int square(int a)
```

```
{
    return (a * a);
}

struct abc
{
    int a;
    char b;
};

int main()
{
    struct abc A;
    const int num = 2;
    int x;
    printf("Square of %d is %d", num, square(num));
    scanf("%d", &x);

    return 0;
}
```

```
rishi@rishi-HP-Pavilion-Gaming-Laptop-15-dk2xxx:~/phase3$ ./a.out tests/test4.c
symbol name |           Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          a |      Identifier |        int |            |          3 |               0 |               0 |
     square |        Function |        int |            |          3 |               0 |               1 |
        int |         Keyword |            |            |          3 |               0 |               0 |
     return |         Keyword |            |            |          5 |               1 |               0 |
```

```
symbol name |           Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          A |      Identifier |     struct |            |         16 |               1 |               0 |
       char |         Keyword |            |            |         11 |               0 |               0 |
        num |      Identifier |        int |          2 |         17 |               1 |               0 |
          a |      Identifier |        int |            |          3 |               0 |               0 |
          b |      Identifier |       char |            |         11 |               0 |               0 |
          x |      Identifier |        int |          0 |         18 |               1 |               0 |
     square |        Function |        int |            |          3 |               0 |               1 |
       main |        Function |        int |            |         14 |               0 |               0 |
      const |         Keyword |            |            |         17 |               1 |               0 |
        int |         Keyword |            |            |          3 |               0 |               0 |
        abc |      Identifier |     struct |            |          8 |               0 |               0 |
     printf |         Keyword |            |            |         19 |               1 |               0 |
     return |         Keyword |            |            |         22 |               1 |               0 |
     struct |         Keyword |            |            |          8 |               0 |               0 |
      scanf |         Keyword |            |            |         20 |               1 |               0 |
PASSED: Semantic Phase
                            PRINTING SYMBOL TABLE

symbol name |           Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
       char |         Keyword |            |            |         11 |               0 |               0 |
          a |      Identifier |        int |            |          3 |               0 |               0 |
          b |      Identifier |       char |            |         11 |               0 |               0 |
     square |        Function |        int |            |          3 |               0 |               1 |
       main |        Function |        int |            |         14 |               0 |               0 |
        int |         Keyword |            |            |          3 |               0 |               0 |
        abc |      Identifier |     struct |            |          8 |               0 |               0 |
     struct |         Keyword |            |            |          8 |               0 |               0 |
                              -------------------------------
```

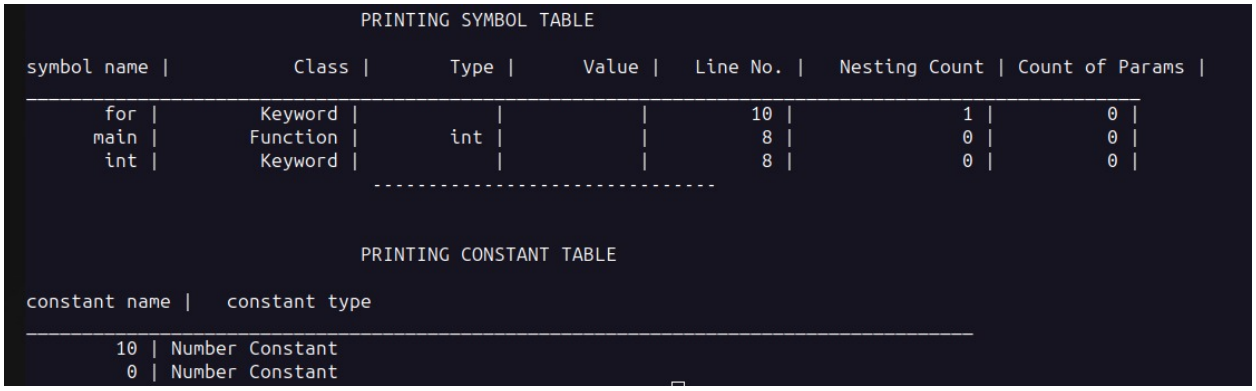
```
                            PRINTING CONSTANT TABLE

constant name |   constant type
_____________________________________________________________________________
"Square of %d is %d" | String Constant
        "%d" | String Constant
           0 | Number Constant
           2 | Number Constant
```

Test5

//for loop
//continue
//while loop
//do while loop

```c
#include<stdio.h>

int main()
{
    for (int a = 0; a < 10; a++){
        continue;
    }
    int x = 0;
    while(x>0) {
        x--;
    }

    do {
        x++;
    }while(x<10);
}
```

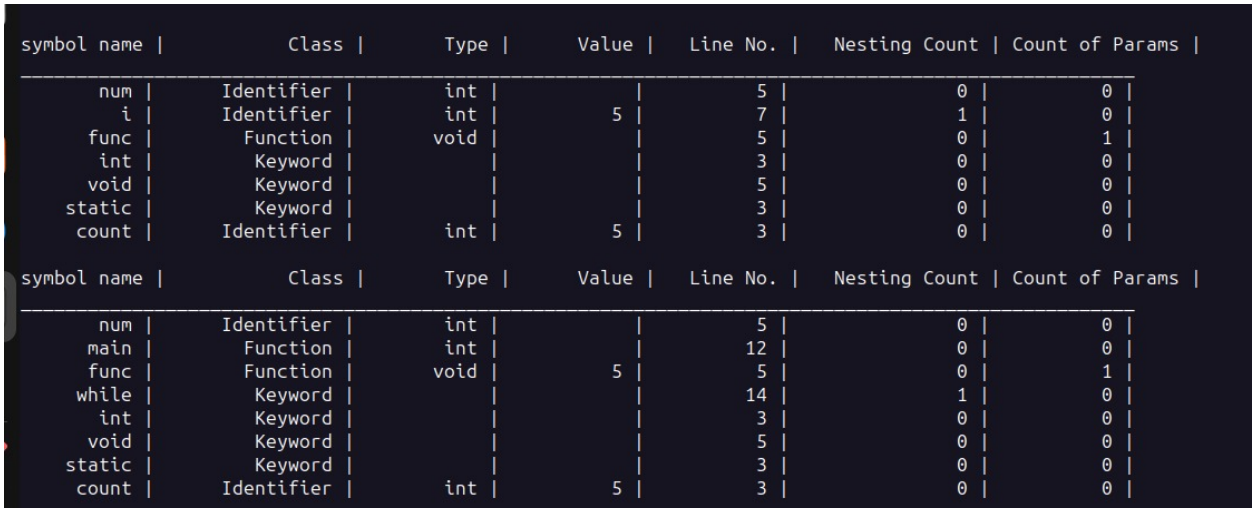
```
rishi@rishi-HP-Pavilion-Gaming-Laptop-15-dk2xxx:~/phase3$ ./a.out tests/test5.c


symbol name |          Class |      Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          a |     Identifier |       int |         10 |         10 |               2 |               0 |
        for |        Keyword |           |            |         10 |               1 |               0 |
       main |       Function |       int |            |          8 |               0 |               0 |
        int |        Keyword |           |            |          8 |               0 |               0 |
   continue |        Keyword |           |            |         11 |               3 |               0 |



symbol name |          Class |      Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          a |     Identifier |       int |         10 |         10 |               2 |               0 |
          x |     Identifier |       int |          0 |         13 |               2 |               0 |
        for |        Keyword |           |            |         10 |               1 |               0 |
       main |       Function |       int |            |          8 |               0 |               0 |
      while |        Keyword |           |            |         14 |               2 |               0 |
        int |        Keyword |           |            |          8 |               0 |               0 |
```



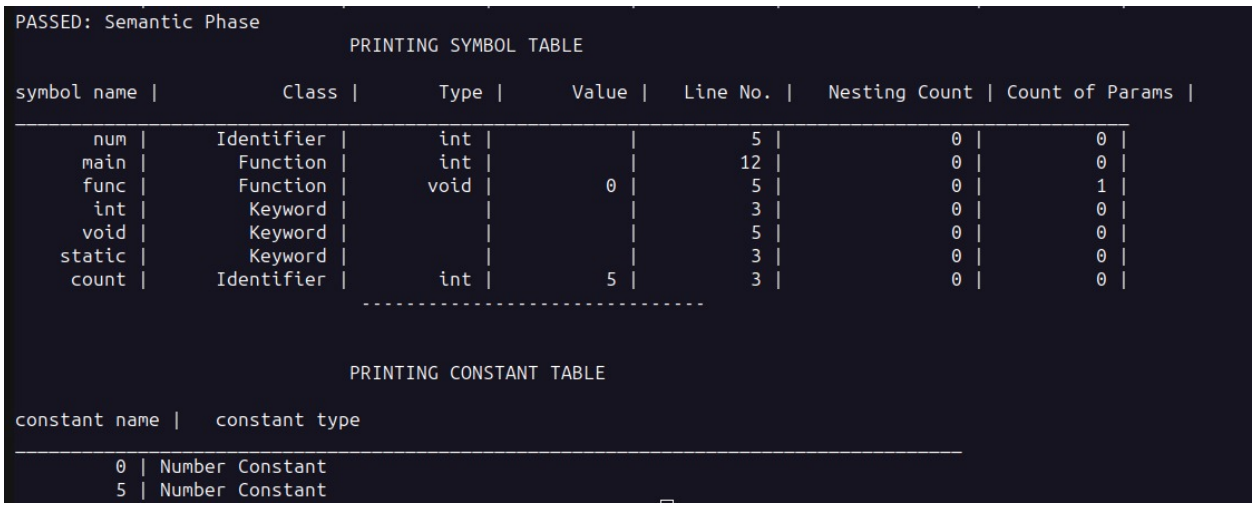
```
                            PRINTING SYMBOL TABLE

symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
        for |        Keyword |            |            |         10 |               1 |               0 |
       main |       Function |        int |            |          8 |               0 |               0 |
        int |        Keyword |            |            |          8 |               0 |               0 |
                              ------------------------------


                            PRINTING CONSTANT TABLE

constant name |   constant type
_____________________________________________________________________________
         10 | Number Constant
          0 | Number Constant
```

Test7

```
#include <stdio.h>

static int count = 5; /* global variable */

void func(int num)
{
    static int i = 5;
    i++;
    num++;
}

int main()
{
    while (count--)
    {
        func(5);
    }

    return 0;
}
```



```
symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
        num |     Identifier |        int |            |          5 |               0 |               0 |
          i |     Identifier |        int |          5 |          7 |               1 |               0 |
       func |       Function |       void |            |          5 |               0 |               1 |
        int |        Keyword |            |            |          3 |               0 |               0 |
       void |        Keyword |            |            |          5 |               0 |               0 |
     static |        Keyword |            |            |          3 |               0 |               0 |
      count |     Identifier |        int |          5 |          3 |               0 |               0 |

symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
        num |     Identifier |        int |            |          5 |               0 |               0 |
       main |       Function |        int |            |         12 |               0 |               0 |
       func |       Function |       void |          5 |          5 |               0 |               1 |
      while |        Keyword |            |            |         14 |               1 |               0 |
        int |        Keyword |            |            |          3 |               0 |               0 |
       void |        Keyword |            |            |          5 |               0 |               0 |
     static |        Keyword |            |            |          3 |               0 |               0 |
      count |     Identifier |        int |          5 |          3 |               0 |               0 |
```



```
PASSED: Semantic Phase
                        PRINTING SYMBOL TABLE

symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
        num |     Identifier |        int |            |          5 |               0 |               0 |
       main |       Function |        int |            |         12 |               0 |               0 |
       func |       Function |       void |          0 |          5 |               0 |               1 |
        int |        Keyword |            |            |          3 |               0 |               0 |
       void |        Keyword |            |            |          5 |               0 |               0 |
     static |        Keyword |            |            |          3 |               0 |               0 |
      count |     Identifier |        int |          5 |          3 |               0 |               0 |
                          ------------------------------


                        PRINTING CONSTANT TABLE

constant name |   constant type
_____________________________________________________________________________
            0 | Number Constant
            5 | Number Constant
```

Test8

```
#include <stdio.h>

int main()
{
// switch variable
        int var = 5;


// switch statement
```

```
	switch (var) {
		case 1:
			printf("Case 1 is Matched.");
			break;

		case 2:
			printf("Case 2 is Matched.");
			break;

		case 3:
			printf("Case 3 is Matched.");
			break;

		default:
			printf(".");
			break;
	}

	return 0;
}
```

```
symbol name |           Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
     break |         Keyword |            |            |         13 |               1 |               0 |
      main |        Function |        int |            |          3 |               0 |               0 |
       int |         Keyword |            |            |          3 |               0 |               0 |
      case |         Keyword |            |            |         11 |               1 |               0 |
   default |         Keyword |            |            |         23 |               1 |               0 |
       var |      Identifier |        int |          0 |          6 |               1 |               0 |
    switch |         Keyword |            |            |         10 |               1 |               0 |
    printf |         Keyword |            |            |         12 |               1 |               0 |
    return |         Keyword |            |            |         28 |               1 |               0 |
PASSED: Semantic Phase
                            PRINTING SYMBOL TABLE

symbol name |           Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
      main |        Function |        int |            |          3 |               0 |               0 |
       int |         Keyword |            |            |          3 |               0 |               0 |
                             ------------------------------


                            PRINTING CONSTANT TABLE

constant name |   constant type
_____________________________________________________________________________
"Case 3 is Matched." | String Constant
"Case 2 is Matched." | String Constant
       "." | String Constant
"Case 1 is Matched." | String Constant
         0 | Number Constant
         1 | Number Constant
         2 | Number Constant
         3 | Number Constant
         5 | Number Constant
```

Test9

```c
//modifiers
//arithmetic operation
//logical operations

#include<stdio.h>
int lauda = 1;
int func(int var) {
    return 1;
}

int lassan() {
    int p = 9;
    return p;
}

int main()
{
    long int a, b;
    unsigned long int x;
    signed short int y;
    signed short z;
    int w;
    a = 23;
    a = 20;
    b = 15;
    int c = a + b;
    printf("%d",c);
    c = a - b;
    printf("%d",c);
    c = a * b;
    printf("%d",c);
    c = a/b;
    printf("%d",c);
    c = a%b;
    printf("%d",c);

    c = (a>=b);
    printf("%d",c);
    c = (a<=b);
    printf("%d",c);
    c = (a==b);
    printf("%d",c);
    c = (a!=b);
    printf("%d",c);
```

```
symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          a |     Identifier |        int |         20 |         18 |               1 |               0 |
          b |     Identifier |        int |         15 |         18 |               1 |               0 |
          c |     Identifier |        int |            |         26 |               1 |               0 |
          w |     Identifier |        int |            |         22 |               1 |               0 |
          x |     Identifier |        int |            |         19 |               1 |               0 |
          y |     Identifier |        int |            |         20 |               1 |               0 |
          z |     Identifier |      short |            |         21 |               1 |               0 |
     signed |        Keyword |            |            |         20 |               1 |               0 |
       main |       Function |        int |            |         16 |               0 |               0 |
      lauda |     Identifier |        int |          1 |          6 |               0 |               0 |
      short |        Keyword |            |            |         20 |               1 |               0 |
       func |       Function |        int |            |          7 |               0 |               1 |
        int |        Keyword |            |            |          6 |               0 |               0 |
   unsigned |        Keyword |            |            |         19 |               1 |               0 |
        var |     Identifier |        int |          1 |          7 |               0 |               0 |
     printf |        Keyword |            |            |         27 |               1 |               0 |
       long |        Keyword |            |            |         18 |               1 |               0 |
     lassan |       Function |        int |            |         11 |               0 |               0 |
PASSED: Semantic Phase
                            PRINTING SYMBOL TABLE

symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
       main |       Function |        int |            |         16 |               0 |               0 |
      lauda |     Identifier |        int |          1 |          6 |               0 |               0 |
       func |       Function |        int |            |          7 |               0 |               1 |
        int |        Keyword |            |            |          6 |               0 |               0 |
        var |     Identifier |        int |          1 |          7 |               0 |               0 |
     lassan |       Function |        int |            |         11 |               0 |               0 |
                              ------------------------------

                            PRINTING CONSTANT TABLE

constant name |   constant type
_____________________________________________________________________________
        "%d" | String Constant
          15 | Number Constant
          20 | Number Constant
          23 | Number Constant
           1 | Number Constant
           9 | Number Constant
```

Test 10

```
        0 | Number Constant
211CS216@ubuntu18:~/phase3/Semantic Analysis$ cat tests/test10.c
#include <stdio.h>
// type match, scope, array
int a;

int main(){

    a = 'p';
    int num = 7.8;

    float f = 5.6;
    float t = 'c';

    double n = 5.6;
    double m = 'c';

    char q = 'x';
    char u = 65;

    return 0;
```

```
symbol name |           Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
      char |         Keyword |            |            |         16 |               1 |               0 |
       num |      Identifier |        int |        7.8 |          8 |               1 |               0 |
         a |      Identifier |        int |        'p' |          3 |               0 |               0 |
         f |      Identifier |      float |        5.6 |         10 |               1 |               0 |
         m |      Identifier |     double |        'c' |         14 |               1 |               0 |
         n |      Identifier |     double |        5.6 |         13 |               1 |               0 |
         q |      Identifier |       char |        'x' |         16 |               1 |               0 |
         t |      Identifier |      float |        'c' |         11 |               1 |               0 |
         u |      Identifier |       char |          0 |         17 |               1 |               0 |
     float |         Keyword |            |            |         10 |               1 |               0 |
      main |        Function |        int |            |          5 |               0 |               0 |
       int |         Keyword |            |            |          3 |               0 |               0 |
    return |         Keyword |            |            |         19 |               1 |               0 |
    double |         Keyword |            |            |         13 |               1 |               0 |
PASSED: Semantic Phase
                            PRINTING SYMBOL TABLE
symbol name |           Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
         a |      Identifier |        int |        'p' |          3 |               0 |               0 |
      main |        Function |        int |            |          5 |               0 |               0 |
       int |         Keyword |            |            |          3 |               0 |               0 |
                              -------------------------------


                            PRINTING CONSTANT TABLE

constant name |   constant type
_____________________________________________________________________________
       5.6 | Floating Constant
       'c' | Character Constant
       7.8 | Floating Constant
       'p' | Character Constant
        65 | Number Constant
       'x' | Character Constant
         0 | Number Constant
```

Test 11

```
}211CS216@ubuntu18:~/phase3/Semantic Analysis$ cat tests/test11.c
//for loop
//continue
//while loop
//do while loop

#include<stdio.h>

int main()
{
    for (int a = 0; a < 10; a++){
        continue;
    }

    int x = 0;
    int* y = &x;
    const int** z = &y;
    while(x>0) {
        x--;
    }

    do {
        x++;
    }while(x<10);
```

```
}211CS216@ubuntu18:~/phase3/Semantic Analysis$ ./a.out tests/test11.c



symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          a |     Identifier |        int |         10 |         10 |               2 |               0 |
        for |        Keyword |            |            |         10 |               1 |               0 |
       main |       Function |        int |            |          8 |               0 |               0 |
        int |        Keyword |            |            |          8 |               0 |               0 |
   continue |        Keyword |            |            |         11 |               3 |               0 |
```

```
symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          a |     Identifier |        int |         10 |         10 |               2 |               0 |
          x |     Identifier |        int |         10 |         14 |               2 |               0 |
          y |     Identifier |       int* |            |         15 |               2 |               0 |
          z |     Identifier |      int** |            |         16 |               2 |               0 |
        for |        Keyword |            |            |         10 |               1 |               0 |
       main |       Function |        int |            |          8 |               0 |               0 |
      const |        Keyword |            |            |         16 |               2 |               0 |
         do |        Keyword |            |            |         21 |               2 |               0 |
      while |        Keyword |            |            |         17 |               2 |               0 |
        int |        Keyword |            |            |          8 |               0 |               0 |
       int* |        Keyword |            |            |         15 |               2 |               0 |
      int** |        Keyword |            |            |         16 |               2 |               0 |
PASSED: Semantic Phase
                          PRINTING SYMBOL TABLE
symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
        for |        Keyword |            |            |         10 |               1 |               0 |
       main |       Function |        int |            |          8 |               0 |               0 |
        int |        Keyword |            |            |          8 |               0 |               0 |
                              --------------------------------


                          PRINTING CONSTANT TABLE

constant name |   constant type
_____________________________________________________________________________________________
         10 | Number Constant
          0 | Number Constant
211CS216@ubuntu18:~/phase3/Semantic Analysis$
```

Test 12

```
211CS216@ubuntu18:~/phase3/Semantic Analysis$ cat tests/test12.c
#include<stdio.h>

char myfunc(int p, int q)
{
    int x;
    return x;
}

void main()
{
    // int a, a;
    int a = 9;
    int a = 7;
    // int b[-1];
    // int c[0];
    int x, y, z;
    z = myfunc(x, y);
```

```
211CS216@ubuntu18:~/phase3/Semantic Analysis$ ./a.out tests/test12.c
symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
      char |        Keyword |            |            |          3 |               0 |               0 |
         p |     Identifier |        int |            |          3 |               0 |               0 |
         q |     Identifier |        int |            |          3 |               0 |               0 |
         x |     Identifier |        int |            |          5 |               1 |               0 |
       int |        Keyword |            |            |          3 |               0 |               0 |
    return |        Keyword |            |            |          6 |               1 |               0 |
    myfunc |       Function |       char |            |          3 |               0 |               2 |
```

Limitations

- Few shift-reduce conflicts.
- Requires a main function to operate.
- Does not keep track of format specifiers in printf and scanf.
- Doesn’t handle niche and uncommon keywords.

2. Results of the Code:

- **Semantic Analysis:** The primary result of the code is the semantic analysis of the C language source code. It checks for various semantic errors such as undeclared variables,

duplicate declarations, type mismatches, and other violations of the language's semantics.

- **Symbol Table and Constant Table:** The code generates and populates a symbol table to keep track of variables and functions along with their types. It also manages a constant table for storing constant values.

- **Error Reporting:** In case of semantic errors, the code provides detailed error messages, including line numbers and context, to assist in debugging.

3. Future Work:

- **Extended Language Support:** The code can be extended to support more features of the C language or other programming languages. This might include handling additional language constructs, preprocessor directives, or specific libraries.

- **Optimizations:** The current code focuses on correctness and basic semantic analysis. Future work could involve incorporating optimizations for performance improvements or generating more informative error messages.

- **Error Recovery:** Enhancing error recovery mechanisms would make the parser more robust, allowing it to continue parsing after encountering errors and potentially identifying more issues in a single run.

- **Security Checks:** Additional checks related to security, such as buffer overflows or other vulnerabilities, could be integrated into the semantic analysis phase.

- **Integration with Code Generation:** Depending on the project's goals, integrating the semantic analyzer with a code generator could be considered to produce executable code.

Addressing these aspects would contribute to the completeness, efficiency, and usability of the semantic analysis process for C language source code.