COMPILER DESIGN PROJECT REPORT - 3

SEMANTIC CHECKER FOR C PROGRAMMING LANGUAGE

Submitted By :

Arvind Ramachandran - 15CO111

Aswanth P P - 15CO112

DATE : 28/03/2018

TABLE OF CONTENTS

INTRODUCTION

A parser constructs parse trees in the syntax analysis phase. The plain parse-tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them. For example,

E → E + T

The above CFG production has no semantic rule associated with it, and it cannot help in making any sense of the production. Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination.

SEMANTIC ANALYSIS

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

CFG + semantic rules = Syntax Directed Definitions

For example:

int a = “value”;

should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis.

Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct, i.e , that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

Semantic analysis is not a separate module within a compiler. It is usually a collection of procedures called at appropriate times by the parser as the grammar requires it. Implementing the semantic actions is conceptually simpler in recursive descent parsing because they are simply added to the recursive procedures. Implementing the semantic actions in a table - action driven LL(1) parser requires the addition of a third type of variable to the productions and the necessary software routines to process it.

Part of semantic analysis is producing some sort of representation of the program, either object code or an intermediate representation of the program. One - pass compilers will generate object code without using an intermediate representation; code generation is part of the semantic actions performed during parsing. Other compilers will produce an intermediate representation during semantic analysis; most often it will be an abstract syntax tree or quadruples.

Semantic analysis typically involves:

- **Type checking** - Data types are used in a manner that is consistent with their definition (i. e., only with compatible data types, only with operations that are defined for them, etc.)
- **Label Checking** - Labels references in a program must exist.
- **Flow control checks** - Control structures must be used in their proper fashion (no GOTOs into a FORTRAN DO statement, no breaks outside a loop or switch statement, etc.)
- **Array-bound Checking** - Variables being used as an array index should be within the bounds of the array.
- **Scope Resolution** - We need to determine what identifiers are accessible at different points in the program.

Abstract syntax trees have one enormous advantage over other intermediate representations: they can be " decorated " , i.e., each node on the AST can have their attributes saved in the AST nodes, which can simplify the task of type checking as the parsing process continues.

ATTRIBUTE GRAMMAR

An attribute is a property whose value is assigned to a grammar symbol. Attribute computation functions (or semantic functions) are associated with the productions of a grammar and are used to compute the values of an attribute.

An attribute grammar is an extension to a context - free grammar that is used to describe features of a programming language that cannot be described in BNF or can only be described in BNF with great difficulty. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions. Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

E → E + T { E.value = E.value + T.value }

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories :

- **Synthesized attributes** - These attributes get values from the attribute values of their child nodes. Synthesized attributes never take values from their parent nodes or any sibling nodes. For example,

  S → ABC

  If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

- **Inherited attributes** - In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. For example,

  S → ABC

  A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

SYNTAX DIRECTED TRANSLATION

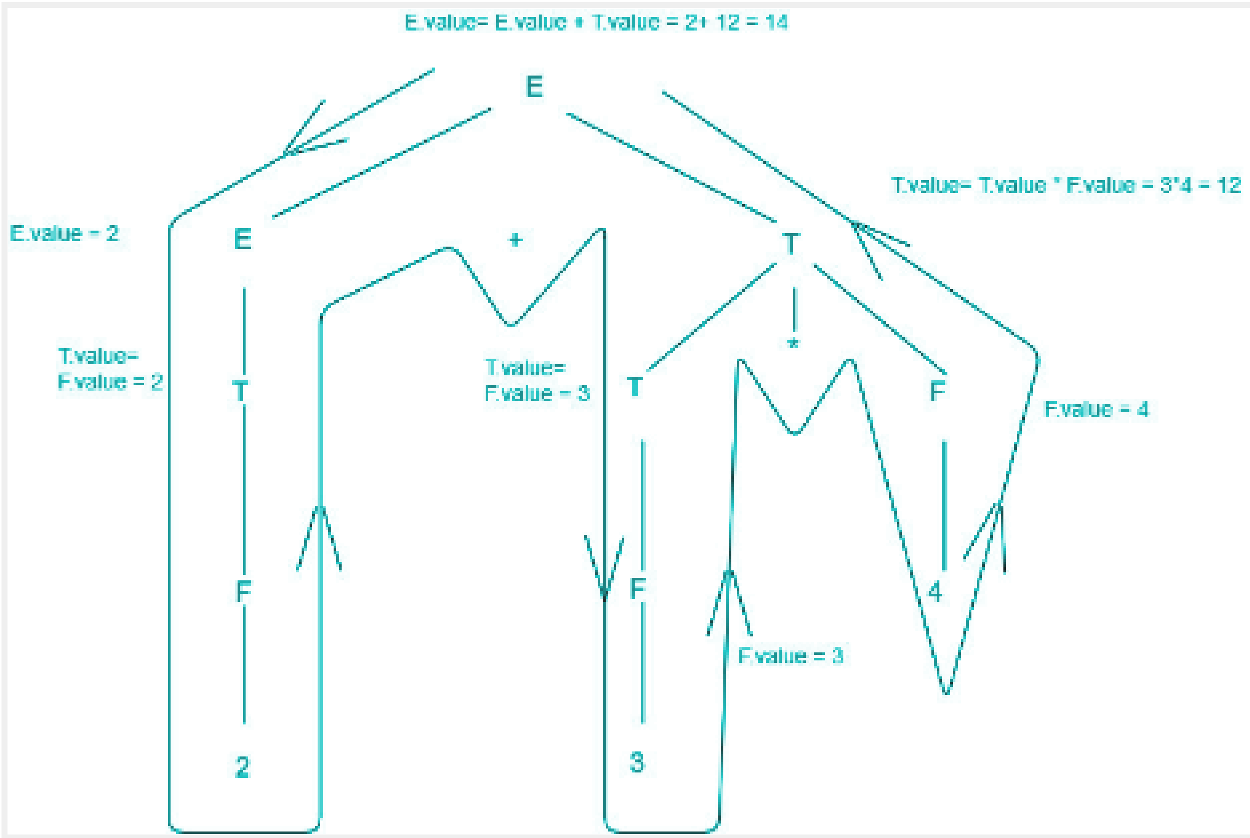
Parser uses a CFG(Context-free-Grammar) to validate the input string and produce output for next phase of the compiler. Output could be either a parse tree or abstract syntax tree. Now to interleave semantic analysis with syntax analysis phase of the compiler, we use Syntax Directed Translation.

Syntax Directed Translation are augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down the parse tree in form of attributes attached to the nodes. Syntax directed translation rules use 1) lexical values of nodes, 2) constants, 3) attributes associated to the non-terminals in their definitions.

For example,

E -> E+T { E.val = E.val + T.val } PR#1
E -> T { E.val = T.val } PR#2
T -> T*F { T.val = T.val * F.val } PR#3
T -> F { T.val = F.val } PR#4
F -> INTLIT { F.val = INTLIT.lexval } PR#5

Suppose we take the first SDT augmented to [E -> E+T] production rule. The translation rule in consideration has val as attribute for both the non-terminals – E & T. Right hand side of the translation rule corresponds to attribute values of right side nodes of the production rule and vice-versa.



Above diagram shows how semantic analysis could happen. The flow of information happens bottom-up and all the children attributes are computed before parents.

- **S-attributed SDT** - If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side). Attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
- **L-attributed SDT** - This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings. In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

IMPLEMENTATION

Scope

Implemented using stack. Variable called scope is used which is initially set to 0. Increment and push to stack when opening brace is found. On finding closing brace, pop out the last element from stack. This will assign unique value to each block of code. Stack as a whole contains the scope-id, which is unique. Incase of functions, the passed parameters are also part of the function block, therefore increment the scope variable when you encounter '(' .

Declaration Checking

To check whether an identifier is declared or not, check the symbol table corresponding to the value having scope-id equal to the current stack (which gives scope-id of the matched identifier). If not present, then undeclared.

Type Checking

When you encounter the type, use a global variable called flag which will indicate the type. For example, flag = 1 for int. When you encounter an assignment statement the current flag and the stored type value of the identifier has to match which is found out by consulting the symbol table.

Re-declaration Checking

When inserting a variable, check if the variable with same name exists with same scope-id. This is taken care by hashing the variable name along with the scope-id. Redeclaration is checked every time we insert identifier into the symbol table.

Array Dimension

When you encounter array declaration, pass the array dimension number into the symbol table. This makes sure that it's only positive integer integer.

Symbol Table

It contains fields:

1. Name : identifier name.
2. Token : token is constant or identifier.
3. Type : type of identifier whether int, float, void etc.

4. Scope : Scope could be global, function.
5. Scope-id : Unique value given to each block of code.
6. Function-id: Unique for each function declaration

All the semantic actions are written in **semantic.h** file, including maintaining symbol table .Necessary functions are called from **syntaxChecker.y** and **lexicalAnalyzer.l** for performing semantic actions and to report semantic error if any. Source code for all the files are included below.

SOURCE CODE

Lexical Analyzer

This is the lex program that contains regular expressions and returns whatever tokens are required.

```
%{
#include <stdio.h>
#include "y.tab.h"
int lineCount=1;
int nestedCommentCount=0;
int commentFlag=0;
char *tablePtr;
void addToken(char*);
%}

digit                   [0-9]
letter                  [a-zA-Z_]
hex                     [a-fA-F0-9]
E                       [Ee][+-]?{digit}+
FS                      (f|F|l|letter)
IS                      (u|U|l|letter)*

singlelineComment (\/\/.*)
multilineCommentStart (\/\*)
multilineCommentEnd (\*\/)
%x DETECT_COMMENT

%%


{singlelineComment}                     { lineCount++; }

{multilineCommentStart}                 { BEGIN(DETECT_COMMENT);
                                          nestedCommentCount++;
                                        }

<DETECT_COMMENT>{multilineCommentStart} { nestedCommentCount++;
                                          if(nestedCommentCount>1)
                                                commentFlag = 1;
                                        }

<DETECT_COMMENT>{multilineCommentEnd}   { BEGIN(INITIAL); lineCount++;
                                                 if(nestedCommentCount>0)
                                                        nestedCommentCount--;
                                                if(nestedCommentCount==0)
                                                        BEGIN(INITIAL);
                                        }

<DETECT_COMMENT>\n              {lineCount++;}
<DETECT_COMMENT>.               {}
```

FIG. 1

```



"auto"                  {  return(AUTO); }
"break"                 {  return(BREAK); }
"case"                  {  return(CASE); }
"char"                  {  return(CHAR); }
"const"                 {  return(CONST); }
"continue"              {  return(CONTINUE); }
"default"               {  return(DEFAULT); }
"do"                    {  return(DO); }
"double"                {  return(DOUBLE); }
"else"                  {  return(ELSE); }
"enum"                  {  return(ENUM); }
"extern"                {  return(EXTERN); }
"float"                 {  return(FLOAT); }
"for"                   {  return(FOR); }
"goto"                  {  return(GOTO); }
"if"                    {  return(IF); }
"int"                   {  return(INT); }
"long"                  {  return(LONG); }
"register"              {  return(REGISTER); }
"return"                {  return(RETURN); }
"short"                 {  return(SHORT); }
"signed"                {  return(SIGNED); }
"sizeof"                {  return(SIZEOF); }
"static"                {  return(STATIC); }
"struct"                {  return(STRUCT); }
"switch"                {  return(SWITCH); }
"typedef"               {  return(TYPEDEF); }
"union"                 {  return(UNION); }
"unsigned"              {  return(UNSIGNED); }
"void"                  {  return(VOID); }
"volatile"              {  return(VOLATILE); }
"while"                 {  return(WHILE); }

{letter}({letter}|{digit})*           {  addToken(yytext); return(IDENTIFIER); }
{letter}?\"(\\.|[^\\"])*\"            {  addToken(yytext); return(STRING_LITERAL); }

0[xX]{hex}+{IS}?                      {  addToken(yytext); return(CONSTANT); }
0{digit}+{IS}?                        {  addToken(yytext); return(CONSTANT); }
{hex}+{IS}?                           {  addToken(yytext); return(CONSTANT); }
{letter}?'(\\.|[^\\'])+'              {  addToken(yytext); return(CONSTANT); }
{digit}+{E}{FS}?                      {  addToken(yytext); return(CONSTANT); }
```

FIG. 2

```
{digit}*"."{digit}+({E})?{FS}?        {  addToken(yytext); return(CONSTANT); }
{digit}+"."{digit}*({E})?{FS}?        {  addToken(yytext); return(CONSTANT); }

"..."                   {  return(ELLIPSIS); }
">>="                   {  return(RIGHT_ASSIGN); }
"<<="                   {  return(LEFT_ASSIGN); }
"+="                    {  return(ADD_ASSIGN); }
"-="                    {  return(SUB_ASSIGN); }
"*="                    {  return(MUL_ASSIGN); }
"/="                    {  return(DIV_ASSIGN); }
"%="                    {  return(MOD_ASSIGN); }
"&="                    {  return(AND_ASSIGN); }
"^="                    {  return(XOR_ASSIGN); }
"|="                    {  return(OR_ASSIGN); }
">>"                    {  return(RIGHT_OP); }
"<<"                    {  return(LEFT_OP); }
"++"                    {  return(INC_OP); }
"--"                    {  return(DEC_OP); }
"->"                    {  return(PTR_OP); }
"&&"                    {  return(AND_OP); }
"||"                    {  return(OR_OP); }
"<="                    {  return(LE_OP); }
">="                    {  return(GE_OP); }
"=="                    {  return(EQ_OP); }
"!="                    {  return(NE_OP); }
";"                     {  return(';'); }
("{"|"<%")              {  return('{'); }
("}"|"%>")              {  return('}'); }
","                     {  return(','); }
":"                     {  return(':'); }
"="                     {  return('='); }
"("                     {  return('('); }
")"                     {  return(')'); }
("["|"<:")              {  return('['); }
("]"|":>")              {  return(']'); }
"."                     {  return('.'); }
"&"                     {  return('&'); }
"!"                     {  return('!'); }
"~"                     {  return('~'); }
"-"                     {  return('-'); }
"+"                     {  return('+'); }
"*"                     {  return('*'); }
"/"                     {  return('/'); }
"%"                     {  return('%'); }
"<"                     {  return('<'); }
```

FIG. 3

```
">"                     {  return('>'); }
"^"                     {  return('^'); }
"|"                     {  return('|'); }
"?"                     {  return('?'); }


"#include"(.)*"\n"      { lineCount++; }
"#define"(.)*"\n"       { lineCount++; }

[ ]                     {}
[\t\v\f]                {}
[\n]                    {lineCount++;}
.                       { }

%%
yywrap()
{
        return(1);
}

void addToken(char *yytext)
{
  int len = strlen(yytext);
  tablePtr = (char*)malloc((len+1)*sizeof(char));
  strcpy(tablePtr, yytext);
}
```

FIG. 4

Syntax Analyzer

This is the main parser code, a yacc file which contains the declarations, rules and programs and defines the actions which should be taken for various cases.

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

extern FILE *yyin;
extern int lineCount;
extern char *tablePtr;
extern char *tablePtr;
extern int nestedCommentCount;
extern int commentFlag;
extern int arrayIndexErr;

char *sourceCode=NULL;
int errorFlag=0;
void makeList(char *,char,int);
%}

%token  AUTO BREAK  CASE CHAR  CONST  CONTINUE  DEFAULT  DO DOUBLE  ELSE ENUM
%token EXTERN FLOAT  FOR GOTO  IF INT LONG REGISTER  RETURN SHORT SIGNED

%token SIZEOF STATIC STRUCT SWITCH TYPEDEF UNION UNSIGNED VOID VOLATILE WHILE

%token IDENTIFIER CONSTANT FLCONSTANT STRING_LITERAL

%token ELLIPSIS

%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
%start translation_unit
%%

primary_expression
        : IDENTIFIER            { $$=checkScope(tablePtr,lineCount); }
        | FLCONSTANT            {tempCheckType=4;}
        | CONSTANT              { addConstant(tablePtr, lineCount);}
        | STRING_LITERAL        { makeList(tablePtr, 's', lineCount);}
        | '(' expression ')'    { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); $$=$2; }
```

FIG. 1

```
        ;
postfix_expression
        : primary_expression   {$$=$1;}
        | postfix_expression '[' expression ']'                 { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | postfix_expression '(' ')'                            { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | postfix_expression '(' argument_expression_list ')'   { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | postfix_expression '.' IDENTIFIER                     { makeList(tablePtr, 'v', lineCount);}
        | postfix_expression PTR_OP IDENTIFIER                  { makeList(tablePtr, 'v', lineCount);}
        | postfix_expression INC_OP                             { makeList(tablePtr, 'o', lineCount);}
        | postfix_expression DEC_OP                             { makeList(tablePtr, 'o', lineCount);}
        ;
argument_expression_list
        : assignment_expression {$$=$1;}
        | argument_expression_list ',' assignment_expression { makeList(",",'p', lineCount); }
        ;
unary_expression
        : postfix_expression {$$=$1;}
        | INC_OP unary_expression       { makeList("++",'o', lineCount); }
        | DEC_OP unary_expression       { makeList("--",'o', lineCount); }
        | unary_operator cast_expression { makeList("sizeof",'o', lineCount); }
        | SIZEOF unary_expression       { makeList("sizeof",'o', lineCount); }
        | SIZEOF '(' type_name ')'      { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        ;
unary_operator
        : '&' { makeList("&",'o', lineCount); }
        | '*' { makeList("*",'o', lineCount); }
        | '+' { makeList("+",'o', lineCount); }
        | '-' { makeList("-",'o', lineCount); }
        | '~' { makeList("~",'o', lineCount); }
        | '!' { makeList("!",'o', lineCount); }
        ;
cast_expression
        : unary_expression   {$$=$1;}
        | '(' type_name ')' cast_expression { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        ;
multiplicative_expression
        : cast_expression {$$=$1;}
        | multiplicative_expression '*' cast_expression { makeList("*",'o', lineCount); }
        | multiplicative_expression '/' cast_expression { makeList("/",'o', lineCount); }
        | multiplicative_expression '%' cast_expression { makeList("%",'o', lineCount); }
        ;
additive_expression
        : multiplicative_expression {$$=$1;}
        | additive_expression '+' multiplicative_expression { makeList("+",'o', lineCount); }
        | additive_expression '-' multiplicative_expression { makeList("-",'o', lineCount); }
```

FIG. 2

```
          ;
shift_expression
          : additive_expression {$$=$1;}
          | shift_expression LEFT_OP additive_expression  { makeList("<<",'o', lineCount); }
          | shift_expression RIGHT_OP additive_expression { makeList(">>",'o', lineCount); }
          ;
relational_expression
          : shift_expression {$$=$1;}
          | relational_expression '<' shift_expression
          | relational_expression '>' shift_expression
          | relational_expression LE_OP shift_expression { makeList("<=",'o', lineCount); }
          | relational_expression GE_OP shift_expression { makeList(">=",'o', lineCount); }
          ;
equality_expression
          : relational_expression {$$=$1;}
          | equality_expression EQ_OP relational_expression { makeList("==",'o', lineCount); }
          | equality_expression NE_OP relational_expression { makeList("!=",'o', lineCount); }
          ;
and_expression
          : equality_expression {$$=$1;}
          | and_expression '&' equality_expression        { makeList("&", 'o', lineCount);}
          ;
exclusive_or_expression
          : and_expression {$$=$1;}
          | exclusive_or_expression '^' and_expression    { makeList("^", 'o', lineCount); }
          ;
inclusive_or_expression
          : exclusive_or_expression {$$=$1;}
          | inclusive_or_expression '|' exclusive_or_expression { makeList("|", 'o', lineCount); }
          ;
logical_and_expression
          : inclusive_or_expression {$$=$1;}
          | logical_and_expression AND_OP inclusive_or_expression { makeList("&&", 'o', lineCount); }
          ;
logical_or_expression
          : logical_and_expression {$$=$1;}
          | logical_or_expression OR_OP logical_and_expression { makeList("||", 'o', lineCount); }
          ;
conditional_expression
          : logical_or_expression {$$=$1;}
          | logical_or_expression '?' expression ':' conditional_expression { makeList("?:",'o', lineCount); }
          ;
assignment_expression
          : conditional_expression {$$=$1;}
          | unary_expression assignment_operator assignment_expression {$$=$3; checkType($1,$3,lineCount);}
```

FIG. 3

```
          ;
assignment_operator
          : '='           { makeList("=",'o', lineCount); }
          | MUL_ASSIGN    { makeList("*=",'o', lineCount); }
          | DIV_ASSIGN    { makeList("/=",'o', lineCount); }
          | MOD_ASSIGN    { makeList("%=",'o', lineCount); }
          | ADD_ASSIGN    { makeList("+=",'o', lineCount); }
          | SUB_ASSIGN    { makeList("-=",'o', lineCount); }
          | LEFT_ASSIGN   { makeList("<<=",'o', lineCount); }
          | RIGHT_ASSIGN  { makeList(">==",'o', lineCount); }
          | AND_ASSIGN    { makeList("&=",'o', lineCount); }
          | XOR_ASSIGN    { makeList("^=",'o', lineCount); }
          | OR_ASSIGN     { makeList("|=",'o', lineCount); }
          ;
expression
          : assignment_expression {$$=$1;}
          | expression ',' assignment_expression { makeList(",", 'p', lineCount); }
          ;
constant_expression
          : conditional_expression
          ;
declaration
          : declaration_specifiers ';'                       { makeList(";", 'p', lineCount);typeBuffer=' '; }
          | declaration_specifiers init_declarator_list ';' { makeList(";", 'p', lineCount); typeBuffer=' ';}
          ;
declaration_specifiers
          : storage_class_specifier
          | storage_class_specifier declaration_specifiers
          | type_specifier
          | type_specifier declaration_specifiers
          | type_qualifier
          | type_qualifier declaration_specifiers
          ;
init_declarator_list
          : init_declarator
          | init_declarator_list ',' init_declarator { makeList(",", 'p', lineCount); }
          ;
init_declarator
          : declarator
          | declarator '=' initializer { makeList("=", 'o', lineCount); }
          ;
storage_class_specifier
          : TYPEDEF       { makeList("typedef", 'k', lineCount);}
          | EXTERN        { makeList("extern", 'k', lineCount);}
          | STATIC        { makeList("static", 'k', lineCount);}
```

FIG. 4

```
        | AUTO          { makeList("auto", 'k', lineCount);}
        | REGISTER      { makeList("register", 'k', lineCount);}
        ;
type_specifier
        : VOID          { makeList("void", 'k', lineCount);typeBuffer='v';}
        | CHAR          { makeList("char", 'k', lineCount); typeBuffer='c';}
        | SHORT         { makeList("short", 'k', lineCount);}
        | INT           { makeList("int", 'k', lineCount); typeBuffer='i';}
        | LONG          { makeList("long", 'k', lineCount);}
        | FLOAT         { makeList("float", 'k', lineCount);    typeBuffer='f';}
        | DOUBLE        { makeList("double", 'k', lineCount);}
        | SIGNED        { makeList("signed", 'k', lineCount);}
        | UNSIGNED      { makeList("unsigned", 'k', lineCount);}
        | struct_or_union_specifier
        | enum_specifier
        | TYPE_NAME
        ;
struct_or_union_specifier
        : struct_or_union IDENTIFIER '{' struct_declaration_list '}'
        | struct_or_union '{' struct_declaration_list '}'
        | struct_or_union IDENTIFIER
        ;
struct_or_union
        : STRUCT        { makeList("struct", 'k', lineCount);}
        | UNION         { makeList("union", 'k', lineCount);}
        ;
struct_declaration_list
        : struct_declaration
        | struct_declaration_list struct_declaration
        ;
struct_declaration
        : specifier_qualifier_list struct_declarator_list ';' { makeList(";", 'p', lineCount); }
        ;
specifier_qualifier_list
        : type_specifier specifier_qualifier_list
        | type_specifier
        | type_qualifier specifier_qualifier_list
        | type_qualifier
        ;
struct_declarator_list
        : struct_declarator
        | struct_declarator_list ',' struct_declarator { makeList(",", 'p', lineCount); }
        ;
struct_declarator
        : declarator
```

FIG. 5

```
        | ':' constant_expression               { makeList(":", 'p', lineCount); }
        | declarator ':' constant_expression    { makeList(":", 'p', lineCount); }
        ;
enum_specifier
        : ENUM '{' enumerator_list '}'                  { makeList("enum", 'k', lineCount);}
        | ENUM IDENTIFIER '{' enumerator_list '}'       { makeList("enum", 'k', lineCount); makeList(tablePtr, 'v', lineCount); }
        | ENUM IDENTIFIER                               { makeList("enum", 'k', lineCount); makeList(tablePtr, 'v', lineCount); }
        ;
enumerator_list
        : enumerator
        | enumerator_list ',' enumerator { makeList(",", 'p', lineCount); }
        ;
enumerator
        : IDENTIFIER                            { makeList(tablePtr, 'v', lineCount); }
        | IDENTIFIER '=' constant_expression    { makeList("=", 'o', lineCount); makeList("tablePtr", 'v', lineCount); }
        ;
type_qualifier
        : CONST         { makeList("const", 'k', lineCount); }
        | VOLATILE      { makeList("volatile", 'k', lineCount); }
        ;
declarator
        : pointer direct_declarator
        | direct_declarator
        ;
direct_declarator
        : IDENTIFIER                                    {  checkDeclaration(tablePtr,lineCount,scopeCount);}
        | '(' declarator ')'                            {  makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | direct_declarator '[' constant_expression ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | direct_declarator '[' ']'                     { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | direct_declarator '(' parameter_type_list ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | direct_declarator '(' identifier_list ')'     { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | direct_declarator '(' ')'                     { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        ;
pointer
        : '*'                                   { makeList("*", 'o', lineCount); }
        | '*' type_qualifier_list               { makeList("*", 'o', lineCount); }
        | '*' pointer                           { makeList("*", 'o', lineCount); }
        | '*' type_qualifier_list pointer       { makeList("*", 'o', lineCount); }
        ;
type_qualifier_list
        : type_qualifier
        | type_qualifier_list type_qualifier
        ;
parameter_type_list
        : parameter_list
```

FIG. 6

```
        | parameter_list ',' ELLIPSIS          { makeList(",", 'p', lineCount); makeList("::", 'o', lineCount); }
        ;
parameter_list
        : parameter_declaration
        | parameter_list ',' parameter_declaration { makeList(",", 'p', lineCount); }
        ;
parameter_declaration
        : declaration_specifiers declarator
        | declaration_specifiers abstract_declarator
        | declaration_specifiers
        ;
identifier_list
        : IDENTIFIER                           { checkDeclaration(tablePtr,lineCount,scopeCount);}
        | identifier_list ',' IDENTIFIER       { checkDeclaration(tablePtr,lineCount,scopeCount);makeList(",", 'p', lineCount); }
        ;
type_name
        : specifier_qualifier_list
        | specifier_qualifier_list abstract_declarator
        ;
abstract_declarator
        : pointer
        | direct_abstract_declarator
        | pointer direct_abstract_declarator
        ;
direct_abstract_declarator
        : '(' abstract_declarator ')'                                  { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | '[' ']'                                                      { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | '[' constant_expression ']'                                  { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | direct_abstract_declarator '[' ']'                           { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | direct_abstract_declarator '[' constant_expression ']'       { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | '(' ')'                                                      { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | '(' parameter_type_list ')'                                  { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | direct_abstract_declarator '(' ')'                           { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | direct_abstract_declarator '(' parameter_type_list ')'       { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        ;
initializer
        : assignment_expression {$$=$1;}
        | '{' initializer_list '}'
        | '{' initializer_list ',' '}'
        ;
initializer_list
        : initializer
        | initializer_list ',' initializer { makeList(",", 'p', lineCount); }
        ;
statement
```

FIG. 7

```
        : labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        ;
labeled_statement
        : IDENTIFIER ':' statement                       { makeList(tablePtr, 'v', lineCount);  }
        | CASE constant_expression ':'  statement        { makeList(":", 'p', lineCount); makeList("case", 'k', lineCount);}
        | DEFAULT ':' statement                          { makeList(":", 'p', lineCount); makeList("default", 'k', lineCount); }
        ;
compound_statement
        : '{' '}'
        | '{' statement_list '}'
        | '{' declaration_list '}'
        | '{' declaration_list statement_list '}'
        ;
declaration_list
        : declaration
        | declaration_list declaration
        ;
statement_list
        : statement
        | statement_list statement
        ;
expression_statement
        : ';'                          { makeList(";", 'p', lineCount); }
        | expression ';'               { makeList(";", 'p', lineCount); }
        ;
selection_statement
        : IF '(' expression ')' statement %prec LOWER_THAN_ELSE
                                  { makeList("if", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount);}
        | IF '(' expression ')' statement ELSE statement
                                  { makeList("if", 'k', lineCount);  makeList("else", 'k', lineCount); makeList("(", 'p', lineCount);
                                  }
        | SWITCH '(' expression ')' statement
                                  { makeList("switch", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        ;
iteration_statement
        : WHILE '(' expression ')' statement
                          { makeList("while", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | DO statement WHILE '(' expression ')' ';'
                          { makeList("do", 'k', lineCount); makeList("while", 'k', lineCount); makeList("(", 'p', lineCount);
                          }
```

FIG. 8

```
        | FOR '(' expression_statement expression_statement ')' statement
                        { makeList("for", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | FOR '(' expression_statement expression_statement expression ')' statement
                        { makeList("for", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        ;
jump_statement
        : GOTO IDENTIFIER ';'   { makeList("goto", 'k', lineCount); makeList(";", 'p', lineCount); makeList(tablePtr, 'v', lineCount);}
        | CONTINUE ';'          { makeList("continue", 'k', lineCount); makeList(";", 'p', lineCount); }
        | BREAK ';'             { makeList("break", 'k', lineCount); makeList(";", 'p', lineCount);}
        | RETURN ';'            { makeList("return", 'k', lineCount); makeList(";", 'p', lineCount);}
        | RETURN expression ';' { makeList("return", 'k', lineCount); makeList(";", 'p', lineCount);}
        ;
translation_unit
        : external_declaration
        | translation_unit external_declaration
        ;
external_declaration
        : function_definition
        | declaration
        ;
function_definition
        : declaration_specifiers declarator declaration_list compound_statement
        | declaration_specifiers declarator compound_statement
        | declarator declaration_list compound_statement
        | declarator compound_statement
        ;
%%

void yyerror()
{
        errorFlag=1;
        fflush(stdout);
        printf("\n%s : %d :Syntax error \n",sourceCode,lineCount);
}

void main(int argc,char **argv){
        if(argc<=1){

                printf("Invalid ,Expected Format : ./a.out <\"sourceCode\"> \n");
                return 0;
        }

        yyin=fopen(argv[1],"r");

        sourceCode=(char *)malloc(strlen(argv[1])*sizeof(char));
```

FIG. 9

```
        sourceCode=argv[1];
        yyparse();

        if(nestedCommentCount!=0){
                errorFlag=1;
                printf("%s : %d : Comment Does Not End\n",sourceCode,lineCount);

        }

        if(commentFlag==1){
                errorFlag=1;
                printf("%s : %d : Nested Comment\n",sourceCode,lineCount);
        }

        if(!errorFlag  && !semanticErr  && arrayIndexErr!=1){

                printf("\n\n\t\t%s Parsing Completed\n\n",sourceCode);

                FILE *writeParsed=fopen("parsedTable.txt","w");
                fprintf(writeParsed,"\n\t\t\t\tParsed Table\n\n\t\tToken\t\t\t\t\t\tType\t\t\t\t\t\t\tLine Number\n");
                for(tokenList *ptr=parsedPtr;ptr!=NULL;ptr=ptr->next){
                        fprintf(writeParsed,"\n%20s%30.30s%60s",ptr->token,ptr->type,ptr->line);
                }

                FILE *writeSymbol=fopen("symbolTable.txt","w");
                fprintf(writeSymbol,"\n\t\t\t\tSymbolTable\n\n\t\tToken\t\t\t\tType\t\tLine Number\t\t\t\tScope\t\tFunction Number\n");
                for(tokenList *ptr=symbolPtr;ptr!=NULL;ptr=ptr->next){
                        fprintf(writeSymbol,"\n%20s%30.30s%30s%30s\t%d\t%d",ptr->token,ptr->type,ptr->line,ptr->scope,ptr->scopeValue,ptr->funcCount
                }

                FILE *writeConstant=fopen("constantTable.txt","w");
                fprintf(writeConstant,"\n   \t\t\t\t\t\t\t\tConstant Table \n\n \t\t\t\t\t\tValue\t\t\t\t\t\t\tLine Number\n");
                for(tokenList *ptr=constantPtr;ptr!=NULL;ptr=ptr->next)
                fprintf(writeConstant,"\n%50s%60s",ptr->token,ptr->line);

                fclose(writeSymbol);
                fclose(writeConstant);
                fclose(writeParsed);
        }
printf("\n\n");
}
```

FIG. 10

Semantic Analyzer

This is the semantic phase code , a c header file which contains the function that is called by different production rule in parser phase and taken care of symbol table entry.

```c
struct tokenList
{
        char *token,type[20],line[100];
        char *scope[20];
        int scopeValue;
        int funcCount;
        struct tokenList *next;
};
typedef struct tokenList tokenList;
struct funcNode{
        char funcName[30];
        int line;
        char funcReturn[20];
        struct  funcNode *next;
};
typedef struct funcNode funcNode;


tokenList *symbolPtr = NULL;
tokenList *constantPtr = NULL;
tokenList *parsedPtr =NULL;
extern int functionCount;
extern int scopeCount;
char typeBuffer=' ';
char *sourceCode;
int tempCheckType=3;

int semanticErr=0,lineSemanticCount;
int checkScope(char *tempToken,int lineCount)
{       tokenList *temp=NULL;
        char type[20];
        int flag=0,tempFlag=0;
        for(tokenList *p=symbolPtr;p!=NULL;p=p->next){
                if(strcmp(tempToken,"printf")==0 ||  strcmp(tempToken,"scanf")==0){
                        tempFlag=1;
                 }
                else{
                        if(strcmp(tempToken,p->token)==0){
                                strcpy(type,p->type);
                                flag=1;
                                break;
                        }
                }

        }
```

FIG. 1

```c
	if (flag == 0 && tempFlag ==0 )
	{
		printf("\n%s : %d :Undeclared variable  \n",sourceCode,lineCount-1);
		semanticErr=1;
	}
	else
	{
		addSymbol(tempToken,lineCount);
		if(strcmp(type,"VOID")==0)
			return(1);
		if(strcmp(type,"CHAR")==0)
			return(2);
		if(strcmp(type,"INT")==0)
			return(3);
		if(strcmp(type,"FLOAT")==0)
			return(4);
	}

}

void checkType(int value1,int value2,int lineCount)
{	lineSemanticCount=lineCount;
	if(value2 == 0)
		value2 = tempCheckType;
	if(value1!=value2)
	{
		printf("\n%s : %d :Type Mismatch error \n",sourceCode,lineSemanticCount-1);
		semanticErr=1;
	}
	tempCheckType=3;
}
void checkDeclaration(char *tokenName,int tokenLine,int scopeVal){
	char type[20];
	char line[39],lineBuffer[19];
	snprintf(lineBuffer, 19, "%d", tokenLine);
	strcpy(line," ");
	strcat(line,lineBuffer);
	switch(typeBuffer){
		case 'i': strcpy(type,"INT"); break;
		case 'f': strcpy(type,"FLOAT");break;
		case 'v': strcpy(type,"VOID");break;
		case 'c': strcpy(type,"CHAR");break;

	}
	for(tokenList *p=symbolPtr;p!=NULL;p=p->next){
```

FIG. 2

```c
		if(strcmp(p->token,tokenName)==0 && p->scopeValue == scopeCount && p->funcCount == functionCount){
			semanticErr=1;
			if(strcmp(p->type,type)==0){
				printf("\n%s : %d :Multiple Declaration \n",sourceCode,tokenLine);
				return;
			}
			else{
				printf("\n%s : %d :Multiple Declration with Different Type \n",sourceCode,tokenLine);
				return;
			}
		}
	}
	addSymbol(tokenName,tokenLine,scopeCount);

}
void checkArray(int val,int lineCount){
	if(val<0){
		semanticErr=1;
		printf("\n%s : %d :Array Index error\n",sourceCode,lineCount-1);
	}
}
void addSymbol(char *tokenName,int tokenLine,int scopeVal){
	char line[39],lineBuffer[19];
	snprintf(lineBuffer, 19, "%d", tokenLine);
	strcpy(line," ");
	strcat(line,lineBuffer);
	char type[20];
	for(tokenList *p=symbolPtr;p!=NULL;p=p->next)
			if(strcmp(p->token,tokenName)==0 && p->scopeValue == scopeCount && p->funcCount ==functionCount ){
				strcat(p->line,line);
				return;
			}
	tokenList *temp=(tokenList *)malloc(sizeof(tokenList));
	temp->token=(char *)malloc(strlen(tokenName)+1);
	strcpy(temp->token,tokenName);
	switch(typeBuffer){
		case 'i': strcpy(temp->type,"INT"); break;
		case 'f': strcpy(temp->type,"FLOAT");break;
		case 'v': strcpy(temp->type,"VOID");break;
		case 'c': strcpy(temp->type,"CHAR");break;
	}
	temp->funcCount=functionCount;
	if(scopeCount==0){
		strcpy(temp->scope,"GLOBAL");
		temp->scopeValue=scopeCount;
```

FIG. 3

```
		temp->scopeValue=scopeCount;

	}
	else{
		strcpy(temp->scope,"NESTING");
		temp->scopeValue=scopeCount;
	}
	strcpy(temp->line,line);
	temp->next=NULL;
	tokenList *p=symbolPtr;
	if(p==NULL){
		symbolPtr=temp;
	}
	else{
		while(p->next!=NULL){
			p=p->next;
		}
		p->next=temp;
	}


}
void addConstant(char *tokenName,int tokenLine){
	char line[39],lineBuffer[19];
	snprintf(lineBuffer, 19, "%d", tokenLine);
	strcpy(line," ");
	strcat(line,lineBuffer);
	for(tokenList *p=constantPtr;p!=NULL;p=p->next)
			if(strcmp(p->token,tokenName)==0){
				strcat(p->line,line);
				return;
			}
		tokenList *temp=(tokenList *)malloc(sizeof(tokenList));
		temp->token=(char *)malloc(strlen(tokenName)+1);
		strcpy(temp->token,tokenName);
		strcpy(temp->line,line);
		temp->next=NULL;

		tokenList *p=constantPtr;
		if(p==NULL){
			constantPtr=temp;
		}
		else{
			while(p->next!=NULL){
				p=p->next;
			}
```

FIG. 4

```
			p->next=temp;
		}


}
void makeList(char *tokenName,char tokenType, int tokenLine)
{
	char line[39],lineBuffer[19];

	snprintf(lineBuffer, 19, "%d", tokenLine);
	strcpy(line," ");
	strcat(line,lineBuffer);
	char type[20];
	switch(tokenType)
	{
			case 'c':
					strcpy(type,"Constant");
					break;
			case 'v':
					strcpy(type,"Identifier");
					break;
			case 'p':
					strcpy(type,"Punctuator");
					break;
			case 'o':
					strcpy(type,"Operator");
					break;
			case 'k':
					strcpy(type,"Keyword");
					break;
			case 's':
					strcpy(type,"String Literal");
					break;
			case 'd':
					strcpy(type,"Preprocessor Statement");
					break;
	}


	for(tokenList *p=parsedPtr;p!=NULL;p=p->next)
			if(strcmp(p->token,tokenName)==0){
				strcat(p->line,line);
				return;
			}
		tokenList *temp=(tokenList *)malloc(sizeof(tokenList));
```

FIG. 5

```c
				case 'k':
						strcpy(type,"Keyword");
						break;
				case 's':
						strcpy(type,"String Literal");
						break;
				case 'd':
						strcpy(type,"Preprocessor Statement");
						break;
	}


	for(tokenList *p=parsedPtr;p!=NULL;p=p->next)
				if(strcmp(p->token,tokenName)==0){
					strcat(p->line,line);
					return;
				}
			tokenList *temp=(tokenList *)malloc(sizeof(tokenList));
			temp->token=(char *)malloc(strlen(tokenName)+1);
			strcpy(temp->token,tokenName);
			strcpy(temp->type,type);
			strcpy(temp->line,line);
			temp->next=NULL;

			tokenList *p=parsedPtr;
			if(p==NULL){
				parsedPtr=temp;
			}
			else{
				while(p->next!=NULL){
					p=p->next;
				}
				p->next=temp;
			}
}
```

FIG. 6

EXECUTION OF THE CODE

The following is a shell script to automate the compilation and execution of the mini compiler consist of lexical ,syntax and semantic phases.There is sample input source code given .execution of the same is done and output is shown as sample output and respective symbol table is generated

Shell Script

```sh
#!/bin/sh
lex lexicalAnalyzer.l
yacc -d syntaxChecker.y
gcc lex.yy.c y.tab.c -w -g
./a.out input.c
rm y.tab.c y.tab.h lex.yy.c
```

Sample Input

```c
#include<stdio.h>
void func(){
        int a;
}
void main()
{

        int a=10,b;
        char c;
        int f[100][10];
        a=b;
        a=b+12;
        printf("\nHello World");
        scanf();
        {
        int y;
        int a;
        }


}

```

Sample Output

```
Terminal File Edit View Search Terminal Help                    (100%)  Sat Mar 24 3:45 PM
aswanth@hp-notebook:~/Projects/CD/Compiler-Design/Semantic Analyzer/src$ ./compile.sh

            input.c Parsing Completed
```

Symbol Table

SymbolTable

Token	Type	Line Number	Scope	Function Number
func	VOID	2	GLOBAL 0	1
a	INT	3	NESTING 1	1
main	VOID	5	GLOBAL 0	2
a	INT	8 11 12	NESTING 1	2
b	INT	8 11 12	NESTING 1	2
c	CHAR	9	NESTING 1	2
f	INT	10	NESTING 1	2
printf		13	NESTING 1	2
scanf		14	NESTING 1	2
y	INT	16	NESTING 2	2
a	INT	17	NESTING 2	2

TEST CASES

Different test cases are added to check implementation of semantic phase.The validation of the same is done below in a table format. Source code for the 6 test cases are added as screenshot below.

Test case Source code

Case1.c

```c
//Testcase to check multiple declaration

#include <stdio.h>
int main()
{
        int a = 5, b = 3;
        float x;
        char b;

        int a = b + 10;
}
```

Case2.c

```c
// Testcase to check undeclared variables
#include<stdio.h>
int main()
{
        int a=10,b=50,c=10;
        d = a+b;
        printf("The sum of %d + %d = %d",a,b,d);
}
```

Case3.c

```c
// Testcase to check type mismatch
#include<stdio.h>
int main()
{
	int l=10,a=0,i;
	float l=5.0;
	for(i=0;i<l;i++)
	{
		printf("\nHello World");
	}


	l = l + 3.14;
	printf("%d",l);
}
```

Case4.c

```c
//Testcase to check scope of variables
#include<stdio.h>
int func(){
	int a;
}
int main()
{
	int a;
	{
		int a;
	}
	func();
}
```

Case5.c

```c
// Testcase to check scope of a variable

#include <stdio.h>
void main()
{
    int a = 10, b = 5;

    {
        printf("Enter a variable");
        int b;
        scanf("%d",&b);
    }
}
```

Case6.c

```c
// Testcase to check array dimensions

#include <stdio.h>
int main()
{
    int i, arr1[100], arr2[100];
    for(i=0;i<100;i++)
    {
        scanf("%d",&arr1[i]);
        arr2[i] = arr1[i] * 2;
    }

    for(i=0;i<100;i++)
    {
        printf("%d",arr2[i]);
    }
}
```

Test Case Evaluation

File Name	Purpose	Expected Output	Explanation	Status
case1.c	Check Multiple Declaration	case1.c : 9 :Multiple declaration with different Type case1.c : 11 :Multiple Declaration	Multiple declaration of variable a and b	**Passed**
case2.c	Check Undeclared variable	case2.c : 6 :Undeclared variable case2.c : 6 :Type Mismatch error case2.c : 7 :Undeclared variable	Variable d is undeclared	**Passed**
case3.c	Check Type mismatch variable	case3.c : 7 :Multiple Declaration with Different Type case3.c : 7 :Type Mismatch error	Variable l is of type *int* but assignment expression of type float	**Passed**

case4.c	Check scope of variable	case4.c Parsing Completed	Though multiple declaration of a but all under different scope	**Passed**
case5.c	Check scope of variable with functions	case5.c Parsing Completed	Multiple declaration under different compound statement	**Passed**
case6.c	Check array dimension	case6.c : 7 : Invalid array Index	*arr2* has array dimension negative	**Passed**

CONCLUSION

After the third phase of this project, we have successfully implemented and added a Semantic Analyzer to the C Compiler. Semantic analysis checks whether the parse tree constructed follows the rules of language. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not, etc. The semantic analyzer produces an annotated syntax tree as an output.