

Semantic Analysis for the C Language



Date: 06-06-2020

**Submitted To:
Prof. P. Santhi Thilagam**

**Group Members:
Chandan Naik - 17CO212
Jelwin Rodrigues - 17CO218
Siddhartha M - 17CO246**

Abstract:

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 on 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. In the syntax analysis phase, we don't check if the input is semantically correct. After parser checks, if the code is structured correctly, semantic analysis phase checks if that syntax structure constructed in the source program derives any meaning or not. The output of the syntax analysis phase is a parse tree whereas that of the semantic phase is an annotated parse tree.

Semantic Analysis is the third phase of Compiler. Semantic Analysis makes sure that declarations and statements of the program are semantically correct. It is a collection of procedures which is called by parser as and when required by the grammar. Both syntax trees of the previous phase and symbol table are used to check the consistency of the given code. Type checking is an important part of the semantic analysis where the compiler makes sure that each operator has matching operands.

Semantic analysis is done by modifications in the parser code only.

The following tasks are performed in semantic analysis:

1. Label checking
2. Type checking
3. Array-bound checking

Contents:

	PAGE NUMBER
1. Introduction	
a. Semantic Analysis	4
b. Yacc Script	5
c. C-Program	6
2. Design of Programs	
a. Code	7
b. Explanation	31
3. Test Cases	
a. Without Errors	32
b. With Errors	36
4. Implementation	41
5. Results / Future work	43
6. References	43

List of Figures and Tables:

1. **Fig 1:** Output for test case of an Operator, Nested loops, Delimiters, Function, Assignments, Conditional Statement
2. **Fig 2:** Output for test case containing Loop Statements
3. **Fig 3:** Output for test case having Modifiers, Arithmetic Operations, Logical Operations.
4. **Fig4:** Output for test case having Function not declared
5. **Fig 5:** Output for test case having Function of type void.
6. **Fig 6:** Output for test case unmatched number of arguments.
7. **Fig 7:** Output for test case with Type mismatch.
8. **Fig 8:** Output for test case having Wrong array size.

Introduction:

Semantic Analysis:

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, $E \rightarrow E + T$

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.

After the lexical analysis stage, we get the stream of tokens from source C code which is given as input to the parser. Parser verifies that a string of token names can be generated by the grammar of the source language. We expect the parser to check the structure of the input program and report any syntax errors. The semantic analysis phase checks the semantics of the language.

The 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.

Semantic analysis typically involves the following tasks:

1. 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.)
2. Label Checking – Label references in a program must exist.
3. Array Bound Checking – When declaring an array, subscript should be defined properly.

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

1. Type mismatch
 - a. Return type mismatch.
 - b. Operations on mismatching variable types.
2. Undeclared variable
 - a. Check if the variable is undeclared globally.
 - b. Check if the variable is visible in the current scope.
3. Reserved identifier misuse.
 - a. Function name and variable name cannot be same.
 - b. Declaration of the keyword as a variable name.
4. Multiple declaration of variable in scope.

5. Accessing an out of scope variable.
6. Actual and formal parameter mismatch.

Yacc Script:

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine. The lexer can be used to make a simple parser. But it needs making extensive use of the user-defined states.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher-level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specifications.

Yacc is written in portable C. The class of specifications accepted is a very general one:

LALR(1) grammars with disambiguating rules.

The structure of our Yacc script is given below; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

```
Definition section
%%
Rules section
%%
C code section
```

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

In the rules section, each grammar rule defines a symbol in terms of:

1. Other symbols
2. Tokens (or terminal symbols) that come from the lexer.

Each rule can have an associated action, which is executed after all the component symbols of the rule have been parsed. Actions are basically C-program statements surrounded by curly braces.

The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs, it is more convenient to place this code in a separate file linked in at compile time.

C Program:

This section describes the input C program which is fed to the yacc script for parsing. The workflow is explained as under:

1. Compile the script using Yacc tool

```
$ yacc -d c_parser.y
```

2. Compile the flex script using Flex tool

```
$ flex c_lexer.l
```

3. After compiling the lex file, a lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generated after compiling the yacc script.
4. The three files, lex.yy.c, y.tab.c and y.tab.h are compiled together with the options -ll and -ly

```
$ gcc -o compiler lex.yy.c y.tab.h y.tab.c -ll -ly
```

5. The executable file is generated, which on running parses the C file given as a command-line input

```
$ ./compiler test.c
```

The script also has an option to take standard input instead of taking input from a file.

Design of Programs:

Code:

Updated Lexer Code:

```
%{
#include <stdio.h>
#include <string.h>
#include "y.tab.h"
#define ANSI_COLOR_RED "\x1b[31m"
#define ANSI_COLOR_GREEN "\x1b[32m"
#define ANSI_COLOR_YELLOW "\x1b[33m"
#define ANSI_COLOR_BLUE "\x1b[34m"
#define ANSI_COLOR_MAGENTA "\x1b[35m"
#define ANSI_COLOR_CYAN "\x1b[36m"
#define ANSI_COLOR_RESET "\x1b[0m"
struct symboltable
{
char name[ 100 ];
char class [100];
char type[ 100 ];
char value[ 100 ];
int nestval;
int lineno;
int length;
int params_count;
}ST[ 1001 ];
struct constanttable
{
char name[ 100 ];
char type[ 100 ];
int length;
}CT[ 1001 ];
int currnest = 0 ;
int params_count = 0 ;
```

```

extern int yylval;

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

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

int lookupCT ( char *str)
{
    int value = hash(str);
    if (CT[value].length == 0 )

```



```

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

else
{
int value = hash(str1);

```

```

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

```

```

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

ST[i].nestval = nest;
}
}
}
}
void insertSTparamscount ( char *s, int count)
{

```

```

for ( int i = 0 ; i < 1001 ; i++)
{
if ( strcmp (ST[i].name,s)== 0 )
{
ST[i].params_count = count;
}
}
}
int getSTparamscount ( char *s)
{
for ( int i = 0 ; i < 1001 ; i++)
{
if ( strcmp (ST[i].name,s)== 0 )
{
return ST[i].params_count;
}
}
return -2 ;
}
void insertSTF ( char *s)
{
for ( int i = 0 ; i < 1001 ; i++)
{
if ( strcmp (ST[i].name,s)== 0 )
{
strcpy (ST[i].class, "Function" );
return ;
}
}
}

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

```

```

strcpy (CT[value].type,str2);
CT[value].length = strlen (str1);
return ;
}
int pos = 0 ;
for ( int i = value + 1 ; i!=value ; i = (i+ 1 )% 1001 )
{
if (CT[i].length == 0 )
{
pos = i;
break ;
}
}
strcpy (CT[pos].name,str1);
strcpy (CT[pos].type,str2);
CT[pos].length = strlen (str1);
}
}
void deletedata ( int nesting)
{
for ( int i = 0 ; i < 1001 ; i++)
{
if (ST[i].nestval == nesting)
{
ST[i].nestval = 99999 ;
}
}

}
int checkscope ( char *s)
{
int flag = 0 ;
for ( int i = 0 ; i < 1000 ; i++)
{
if ( strcmp (ST[i].name,s)== 0 )
{
if (ST[i].nestval > currnest)
{
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].nestval == currnest)
{
return 1 ;
}
}
}
return 0 ;
}
int check_duplicate ( char * str)
{
for ( int i= 0 ; i< 1001 ; 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< 1001 ; 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 < 1001 ; i++ )
{
if ( strcmp (ST[i].name,s)== 0 )
{
return ST[i].type[ 0 ];
}
}
}
void printST ()
{
printf ( "%10s | %15s | %10s | %10s | %10s | %15s | %10s |\n" ,
"SYMBOL" , "CLASS" , "TYPE" , "VALUE" , "LINE NO" , "NESTING" ,
"PARAMS
COUNT" );
for ( int i= 0 ;i< 100 ;i++) {
printf ( "-" );
}
printf ( "\n" );
for ( int i = 0 ; i < 1001 ; i++)
{
if (ST[i].length == 0 )
{
continue ;
}
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].nestval, ST[i].params_count);
}
}
void printCT ()

```



```

{
printf ( "%10s | %15s\n" , "NAME" , "TYPE" );
for ( int i= 0 ;i< 81 ;i++) {
printf ( "-" );
}
printf ( "\n" );
for ( int i = 0 ; i < 1001 ; i++)
{
if (CT[i].length == 0 )
continue ;
printf ( "%10s | %15s\n" ,CT[i].name, CT[i].type);
}
}
char curid[ 20 ];
char curtype[ 20 ];
char curval[ 20 ];
}%
DE "define"
IN "include"
%%
\n {yylineno++;}
([#][ " " ]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?) )/[
"\n" |\/| "
" | "\t" ] { }
([#][ " " ]*({DE})[ " " ]*([A-Za-z]+)( " " )*[ 0-9 ]+)/[ "\n" |\/|
" " | "\t" ]
{ }
\/\/(.*)
{ }
\/\/*([^*]|[\r\n]|(\*+([^\r\n]|[\r\n])))\*+\/
{ }
[ \n\t] ;
";" { return ( ';' ); }
"," { return ( ',' ); }
("{") { return ( '{' ); }
("}") { return ( '}' ); }
("(" { return ( '(' ); }
")" { return ( ')' ); }
("[ " | "<:" ) { return ( '[' ); }
("]" | ">:" ) { return ( ']' ); }
":" { return ( ':' ); }

```

```
"." { return ( '.' ); }
"char" { strcpy (curtype,yytext); insertST(yytext,
"Keyword" ); return CHAR;}
"double" { strcpy (curtype,yytext); insertST(yytext, "Keyword" );
return
DOUBLE;}
"else" { insertST(yytext, "Keyword" ); return ELSE;}
"float" { strcpy (curtype,yytext); insertST(yytext, "Keyword" );
return FLOAT;}
"while" { insertST(yytext, "Keyword" ); return WHILE;}
"do" { insertST(yytext, "Keyword" ); return DO;}
"for" { insertST(yytext, "Keyword" ); return FOR;}
"if" { insertST(yytext, "Keyword" ); return IF;}
"int" { strcpy (curtype,yytext); insertST(yytext, "Keyword" );
return INT;}
"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;}
"unsigned" { insertST(yytext, "Keyword" ); return UNSIGNED;}
"void" { strcpy (curtype,yytext); insertST(yytext, "Keyword" );
return VOID;}
"break" { insertST(yytext, "Keyword" ); return BREAK;}
"++" { 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-Za-z\\'\\'"/[;|,|\\]|:|' {strcpy(curval,yytext);
insertCT(yytext," Character Constant "); return
character_constant;}
[a-zA-Z]([a-zA-Z]|([0-9]))*/\\[ {strcpy(curid,yytext);
insertST(yytext,
" Array Identifier "); return array_identifier;}
[1-9][0-9]*|0/[;|,|"
"\\)|<|>|=|\\!|\\||&|\\+|\\-|\\*|\\|/|\\%|~|\\||\\}|:|\\n|\\t|\\^]
{strcpy(curval,yytext); insertCT(yytext, " Number Constant ");
yylval =
atoi(yytext); return integer_constant;}
([0-9]*)\\.([0-9]+)/[;|,|"
"\\)|<|>|=|\\!|\\||&|\\+|\\-|\\*|\\|/|\\%|~|\\n|\\t|\\^]
{strcpy(curval,yytext); insertCT(yytext, " Floating Constant ");
return
float_constant;}
[A-Za-z_][A-Za-z_0-9]* {strcpy(curid,yytext); insertST(curid,"
Identifier ");

```

```

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

```

Parser Code:

```

%{
void yyerror ( char * s);
int yylex ();
#include "stdio.h"
#include "stdlib.h"
#include "ctype.h"
#include "string.h"
void ins ();
void insV ();
int flag= 0 ;

```

```

#define ANSI_COLOR_RED "\x1b[31m"
#define ANSI_COLOR_GREEN "\x1b[32m"
#define ANSI_COLOR_CYAN "\x1b[36m"
#define ANSI_COLOR_RESET "\x1b[0m"
extern char curid[ 20 ];
extern char curtype[ 20 ];
extern char curval[ 20 ];
extern int currnest;
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 *);
extern int params_count;
int call_params_count;
%}
%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
%token RETURN MAIN
%token VOID
%token WHILE FOR DO
%token BREAK
%token ENDIF
%expect 1
%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
variable_declaration
: type_specifier variable_declaration_list ';'
variable_declaration_list
: variable_declaration_identifier V;
V
: ',' variable_declaration_list

```

```

| ;
variable_declaration_identifier
: identifier
{ if (duplicate(curid)){ printf ( "Duplicate\n" ); exit ( 0
);}insertSTnest(curid,cur
rnest); ins(); } vdi
| array_identifier
{ if (duplicate(curid)){ printf ( "Duplicate\n" ); exit ( 0
);}insertSTnest(curid,cur
rnest); ins(); } vdi;
vdi : identifier_array_type | assignment_operator expression ;
identifier_array_type
: '[' initialization_params
| ;
initialization_params
: integer_constant '[' initialization { if ($$ < 1 )
{ printf ( "Wrong array size\n" ); exit ( 0 );} }
| ']' string_initialization;
initialization
: string_initialization
| array_initialization
| ;
type_specifier
: INT | CHAR | FLOAT | DOUBLE
| LONG long_grammar
| SHORT short_grammar
| UNSIGNED unsigned_grammar
| SIGNED signed_grammar
| VOID ;
unsigned_grammar
: INT | LONG long_grammar | SHORT short_grammar | ;
signed_grammar
: INT | LONG long_grammar | SHORT short_grammar | ;
long_grammar
: INT | ;
short_grammar
: INT | ;
function_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();insertSTnest(curid, 1 ); params_count++; }
param_identifier_breakup;
param_identifier_breakup
: '[' ']'
| ;
statement
: expression_statment | compound_statement
| conditional_statements | iterative_statements
| return_statement | break_statement
| variable_declaration;
compound_statement
: {currnest++;} '{' statment_list '}'
{deletedata(currnest);currnest--;} ;
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 '(' simple_expression ')' { if ($ 3 != 1 ){ printf (
"Condition
checking is not of type int\n" ); exit ( 0 );}} statement
| FOR '(' expression ';' simple_expression ';'
{ if ($ 5 != 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 is void" );
}
if ((currfunctype[ 0 ]== 'i' || currfunctype[ 0 ]== 'c' ) && $ 2
!= 1 )
{
printf ( "Expression doesn't match return type of function\n" );
exit ( 0 );
}
};
break_statement
: BREAK ';' ;
string_initilization
: assignment_operator string_constant { insV(); };
array_initialization
: assignment_operator '{' array_int_declarations '}' ;
array_int_declarations
: integer_constant array_int_declarations_breakup;

```

```

array_int_declarations_breakup
: ',' array_int_declarations
| ;
expression
: mutable assignment_operator 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" ); exit ( 0 )
;}} '[' expression ']'
{ if (gettype(curid, 0 )== 'i' ||
gettype(curid, 1 )== 'c' )
$$ = 1 ;
else
$$ = -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 ( "Function not declared" ); exit ( 0 );}
insertSTF(curid);
strcpy (currfuncall,curid);
} arguments ')'
{ if ( strcmp (currfuncall, "printf" ))
{
if (getSTparamscount(currfuncall)!=call_params_count)

```

```

{
yyerror( "Number of
arguments in function call doesn't match number of parameters" );
//printf("Number of arguments in function call %s doesn't match
number of parameters\n",currfuncall);
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(); }
| character_constant{ insV(); $$= 1 ;};
%%
extern FILE *yyin;
extern int yylineno;
extern char *yytext;
void insertSTtype ( char *, char *);
void insertSTvalue ( char *, char *);
void incertCT ( char *, char *);
void printST ();
void printCT ();
int main ( int argc , char **argv)
{
yyin = fopen(argv[ 1 ], "r" );
yyparse();
if (flag == 0 )
{
printf (ANSI_COLOR_GREEN "Status: Parsing Complete - Valid"
ANSI_COLOR_RESET "\n" );
printf ( "%30s" ANSI_COLOR_CYAN "SYMBOL TABLE" ANSI_COLOR_RESET
"\n" ,
" " );
};

```

```

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

```

Explanation:

The lex code is detecting the tokens from the source code and returning the corresponding token to the parser. In phase 1 we were just printing the token and now we are returning the token so that parser uses it for further computation. We are using the symbol table and constant table of the previous phase only. We added functions like

`insertSTnest()`, `insertSTparamscount()`, `checkscope()`, `deletedata()`, `duplicate()` etc., in order to check the semantics. In the production rules of the grammar semantic actions are written and these are performed by the functions listed above.

Declaration Section

In this section we have included all the necessary header files, function declaration and flag that was needed in the code.

Between declaration and rules section we have listed all the tokens which are returned by the lexer according to the precedence order. We also declared the operators here according to their associativity and precedence. This ensures the grammar we are giving to the parser is unambiguous as LALR(1) parser cannot work with ambiguous grammar.

Rules Section

In this section production rules for entire C language is written. The grammar productions does the syntax analysis of the source code. When a complete statement with proper syntax is matched by the parser. Along with rules semantic actions associated with the rules are also written and corresponding functions are called to do the necessary actions.

C-Program Section

In this section the parser links the extern functions, variables declared in the lexer, external files generated by the lexer etc. The main function takes the input source code file and prints the final symbol table.

Test Cases:

Valid test cases

Test 1: Operator, Nested loops, Delimiters, Function, Assignments, Conditional Statements

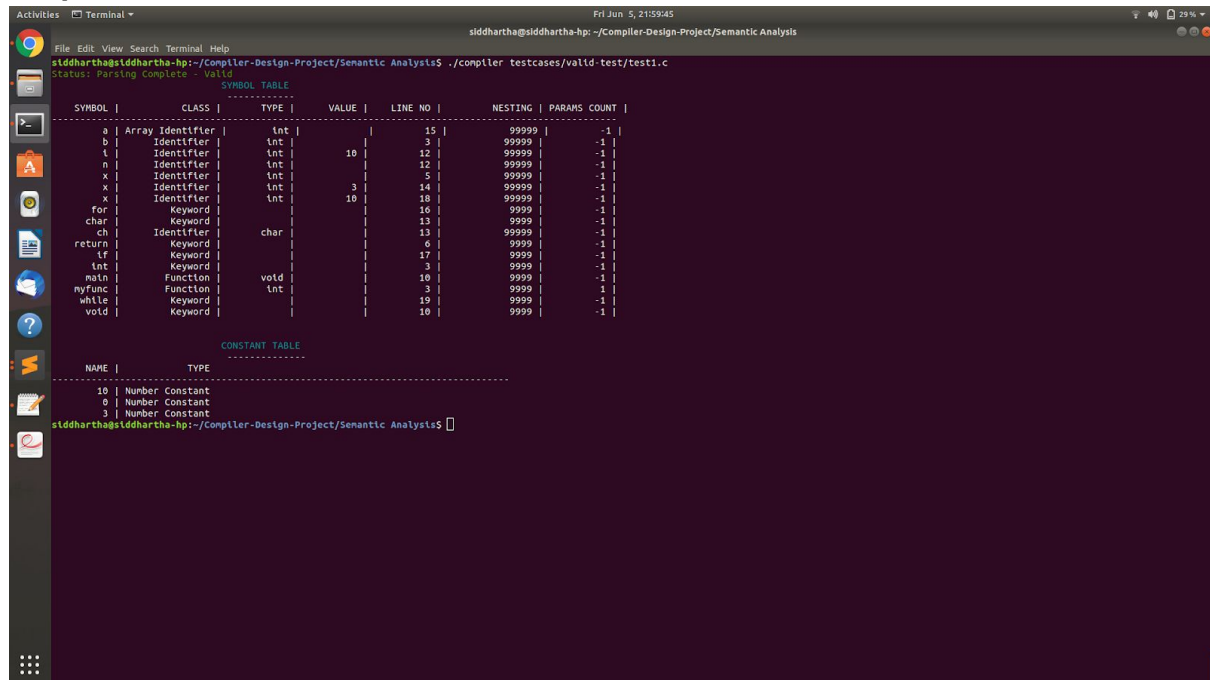
```
#include<stdio.h>

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

void main()
{
    int n,i;
    char ch;//Character Datatype
    int x;
    int a[10];
    for (i=0;i<10;i++){
        if(i<10){
            int x;
            while(x<10){
                x++;
            }
        }
    }
    x=3;
}
```

Status : Pass

Output :



```
Activities Terminal
siddhartha@siddhartha-hp: ~/Compiler-Design-Project/Semantic Analysis
siddhartha@siddhartha-hp:~/Compiler-Design-Project/Semantic Analysis$ ./compiler testcases/valid-test/test1.c
Status: Parsing complete - Valid

SYMBOL TABLE
-----
SYMBOL | CLASS | TYPE | VALUE | LINE NO | NESTING | PARAMS COUNT |
-----|-----|-----|-----|-----|-----|-----|
a | Array Identifier | int | | 15 | 99999 | -1 |
b | Identifier | int | | 3 | 99999 | -1 |
i | Identifier | int | 10 | 12 | 99999 | -1 |
n | Identifier | int | | 12 | 99999 | -1 |
x | Identifier | int | | 5 | 99999 | -1 |
x | Identifier | int | 3 | 14 | 99999 | -1 |
x | Identifier | int | 10 | 18 | 99999 | -1 |
for | Keyword | | | 16 | 9999 | -1 |
char | Keyword | | | 13 | 9999 | -1 |
ch | Identifier | char | | 13 | 99999 | -1 |
return | Keyword | | | 6 | 9999 | -1 |
if | Keyword | | | 17 | 9999 | -1 |
int | Keyword | | | 3 | 9999 | -1 |
main | Function | void | | 10 | 9999 | -1 |
myfunc | Function | int | | 3 | 9999 | 1 |
while | Keyword | | | 19 | 9999 | -1 |
void | Keyword | | | 10 | 9999 | -1 |

CONSTANT TABLE
-----
NAME | TYPE |
-----|-----|
10 | Number Constant |
0 | Number Constant |
3 | Number Constant |

siddhartha@siddhartha-hp:~/Compiler-Design-Project/Semantic Analysis$
```

Fig 1

Test 2 : Loop Statements

```
#include<stdio.h>

int main()
{
    int a = 5;
    while(a>0)
    {
        printf("Hello world");
        a--;
    }

    a=4;
    while(a>0)
    {
        printf("%d",a);
        a--;
        int b;
        b= 4;
        while(b>0)
        {
```

```

        printf("%d", a*b);
        b--;
    }
}

```

Status : Pass

Output:

Terminal window showing the output of a compiler. The status is 'Parsing Complete - Valid'. The terminal displays two tables: a SYMBOL TABLE and a CONSTANT TABLE.

SYMBOL	CLASS	TYPE	VALUE	LINE NO	NESTING	PARAMS COUNT
a	Identifier	int	0	5	99999	-1
b	Identifier	int	0	17	99999	-1
int	Keyword			3	9999	-1
main	Function	int		3	9999	-1
printf	Function			8	9999	-1
while	Keyword			6	9999	-1

NAME	TYPE
"Hello world"	String Constant
"sd"	String Constant
0	Number Constant
4	Number Constant
5	Number Constant

Fig 2

Test Case 3: Modifiers, Arithmetic Operations, Logical Operations

```

#include<stdio.h>
void main(){
    int a,b,c,d,e,f,g,h;
    c=a+b;
    d=a*b;
    e=a/b;
    f=a%b;
    g=a&&b;
    h=a||b;
    h=a*(a+b);
    h=a*a+b*b;
}

```

Status : Pass

Output:

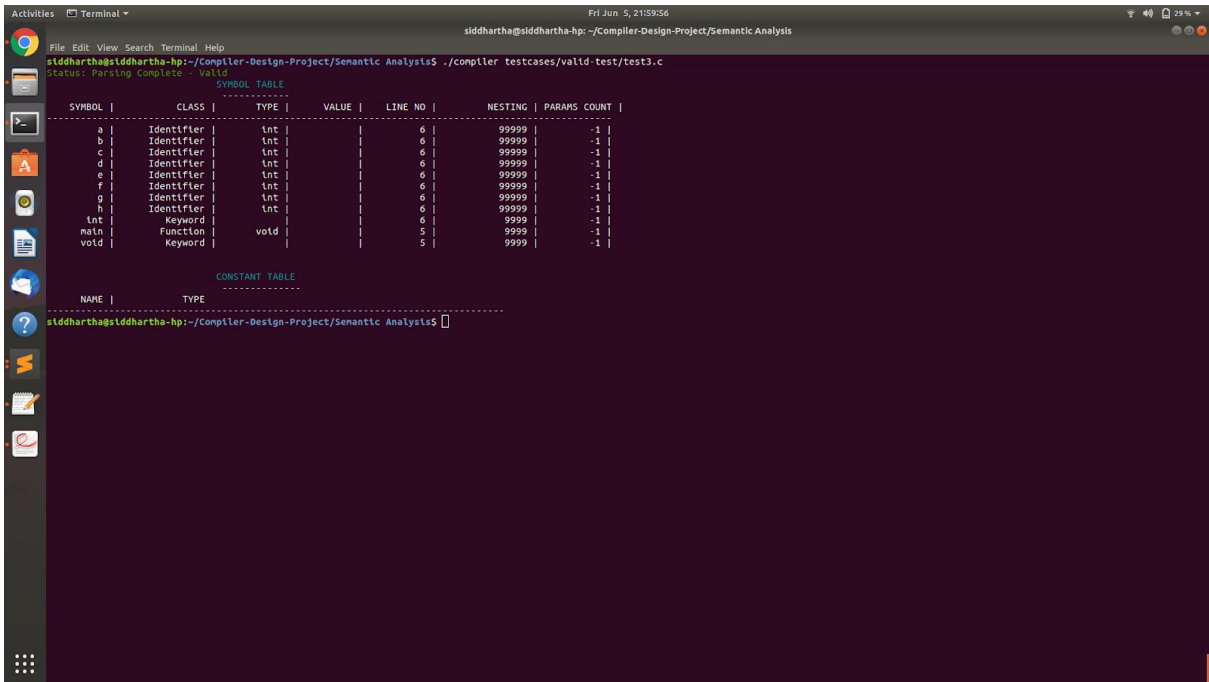


Fig 3

Invalid Test Cases

Test 1: Function not declared

```
#include<stdio.h>

void main()
{
    int i,n;

    myfunc(i);

}
```

Status : Pass

Output :

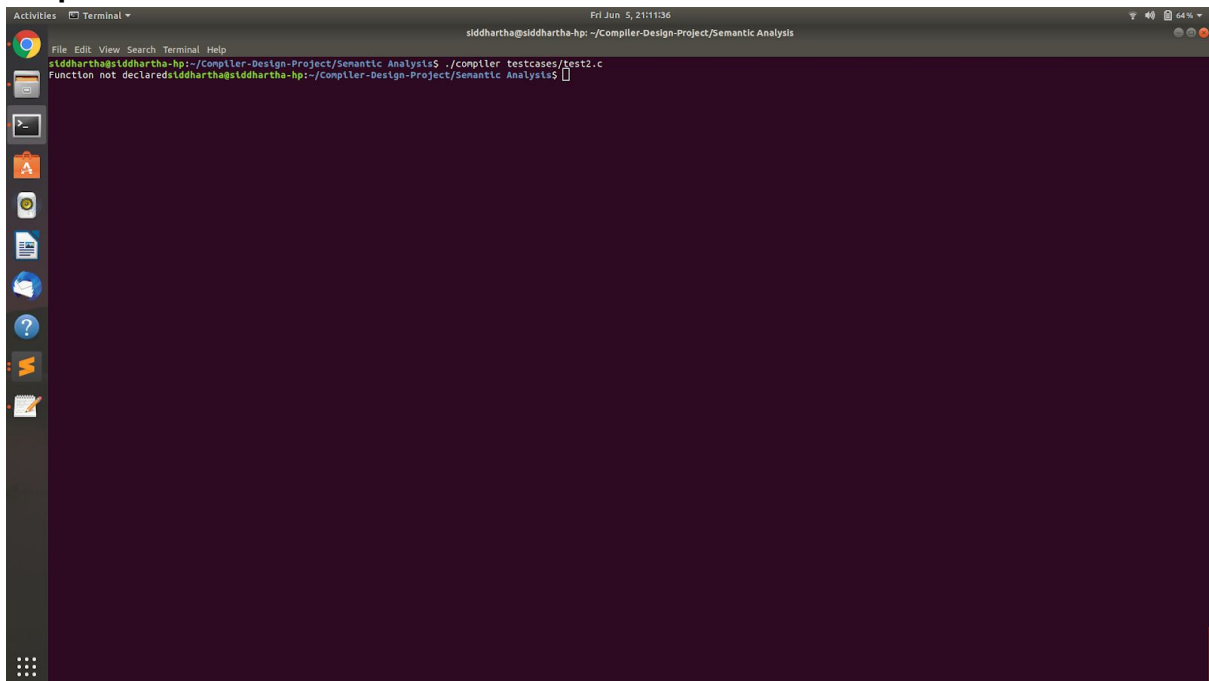


Fig 4

Test 2: Function of type void

```
#include<stdio.h>

void myfunc(int a)
{
```

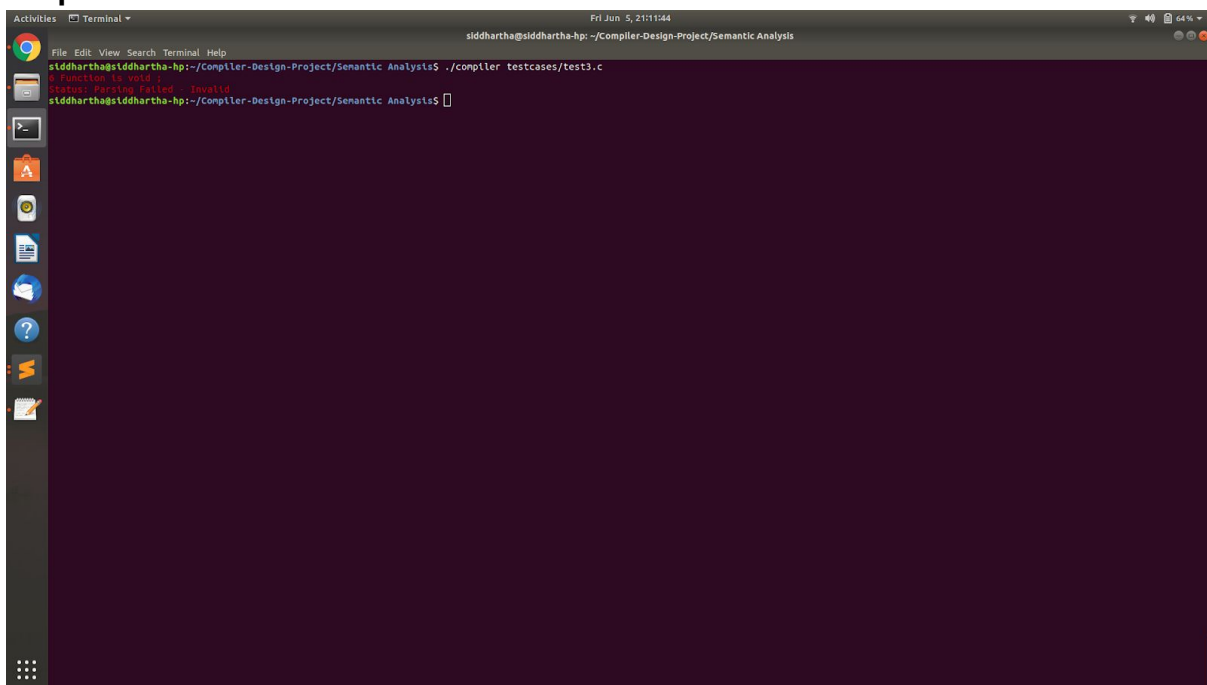
```
    return a;
}

void main()
{
    int i,n;

    myfunc(i);
}
```

Status : Pass

Output :

A screenshot of a Linux terminal window. The title bar shows 'Activities', 'Terminal', and the date 'Fri Jun 5, 21:11:44'. The terminal text shows the user 'siddhartha@siddhartha-hp' in the directory '~/Compiler-Design-Project/Semantic Analysis'. The command './compiler testcases/test3.c' has been executed, resulting in a 'Function to void' error and a 'Status: Parsing Failed - Invalid' message. The terminal has a dark purple background and a sidebar with application icons on the left.

```
Activities Terminal Fri Jun 5, 21:11:44 siddhartha@siddhartha-hp: ~/Compiler-Design-Project/Semantic Analysis
siddhartha@siddhartha-hp:~/Compiler-Design-Project/Semantic Analysis$ ./compiler testcases/test3.c
Function to void
Status: Parsing Failed - Invalid
siddhartha@siddhartha-hp:~/Compiler-Design-Project/Semantic Analysis$
```

Fig 5

Test 3: Unmatched number of arguments

```
#include<stdio.h>

int myfunc(int a)
{
    return a;
}
```

```
void main()
{
    int i,n;

    myfunc(i,n);

}
```

Status : Pass

Output :

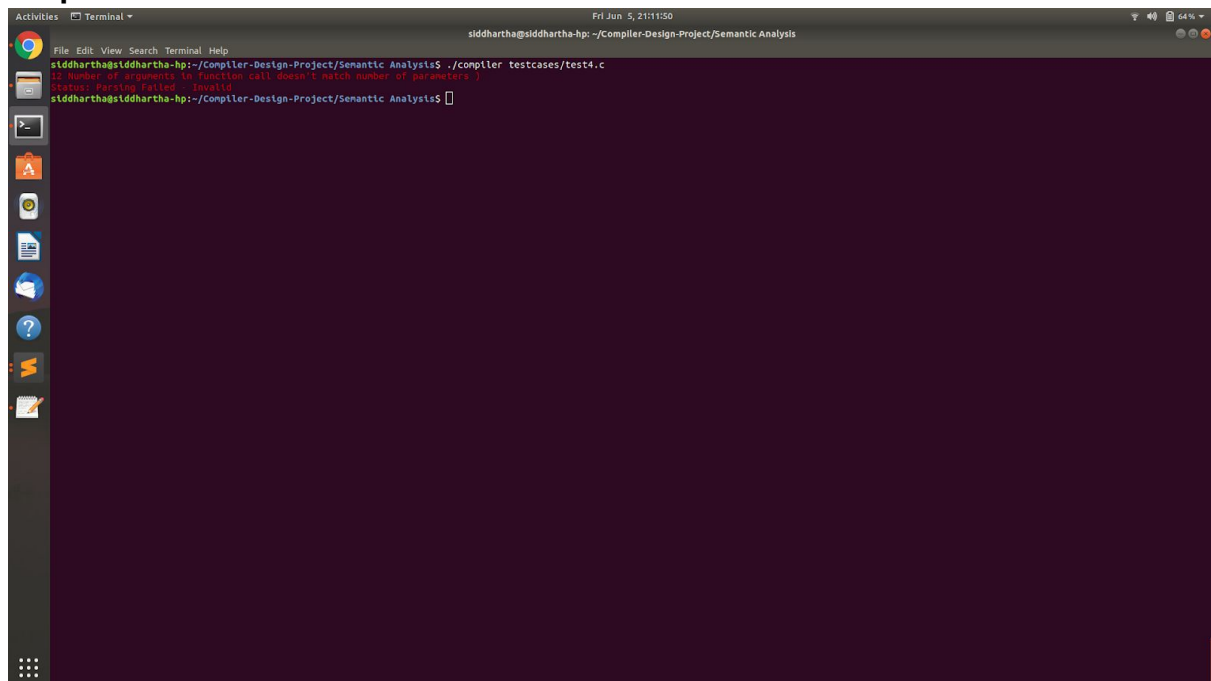


Fig 6

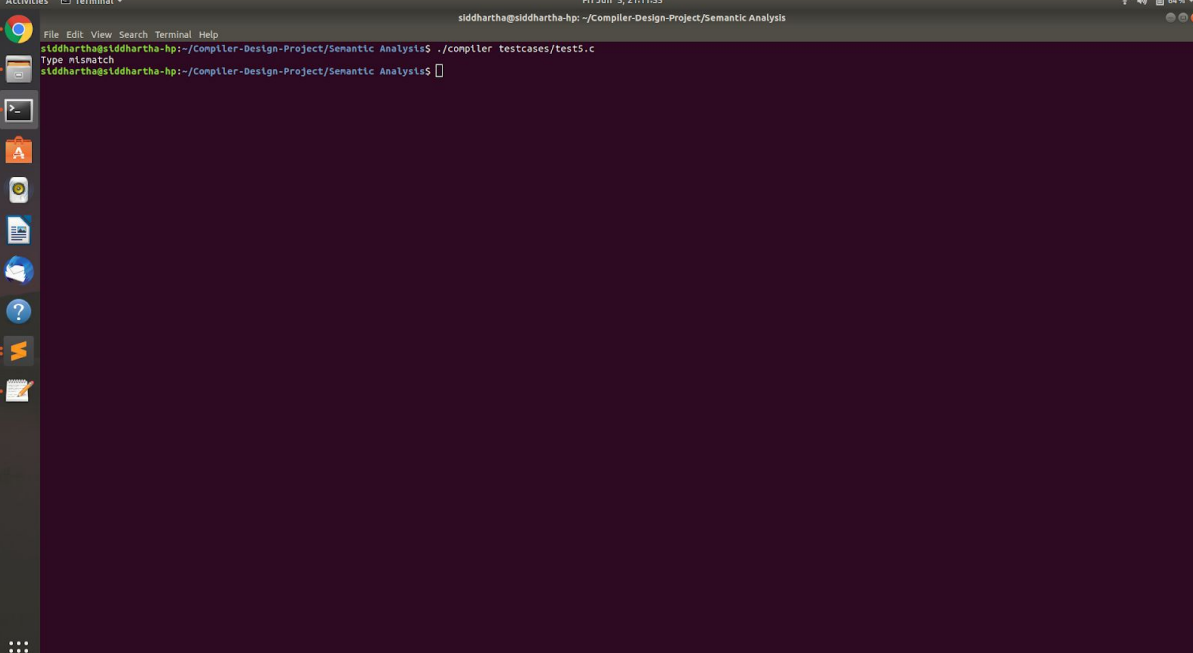
Test 4: Type mismatch

```
#include<stdio.h>

void main()
{
    int i=3,n=6;
    float a=0.0;
    a = i+n;
}
```

Status : Pass

Output :



The screenshot shows a terminal window with a dark purple background. The title bar at the top reads "Activities Terminal" and "Fri Jun 5, 21:11:55". The terminal content shows the user "siddhartha@siddhartha-hp" in the directory "~/Compiler-Design-Project/Semantic Analysis". The command ". /compiler testcases/test5.c" has been executed, resulting in the output "Type mismatch". The terminal window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". A sidebar on the left contains various application icons.

```
Activities Terminal
siddhartha@siddhartha-hp: ~/Compiler-Design-Project/Semantic Analysis
siddhartha@siddhartha-hp:~/Compiler-Design-Project/Semantic Analysis$ ./compiler testcases/test5.c
Type mismatch
siddhartha@siddhartha-hp:~/Compiler-Design-Project/Semantic Analysis$
```

Fig 7

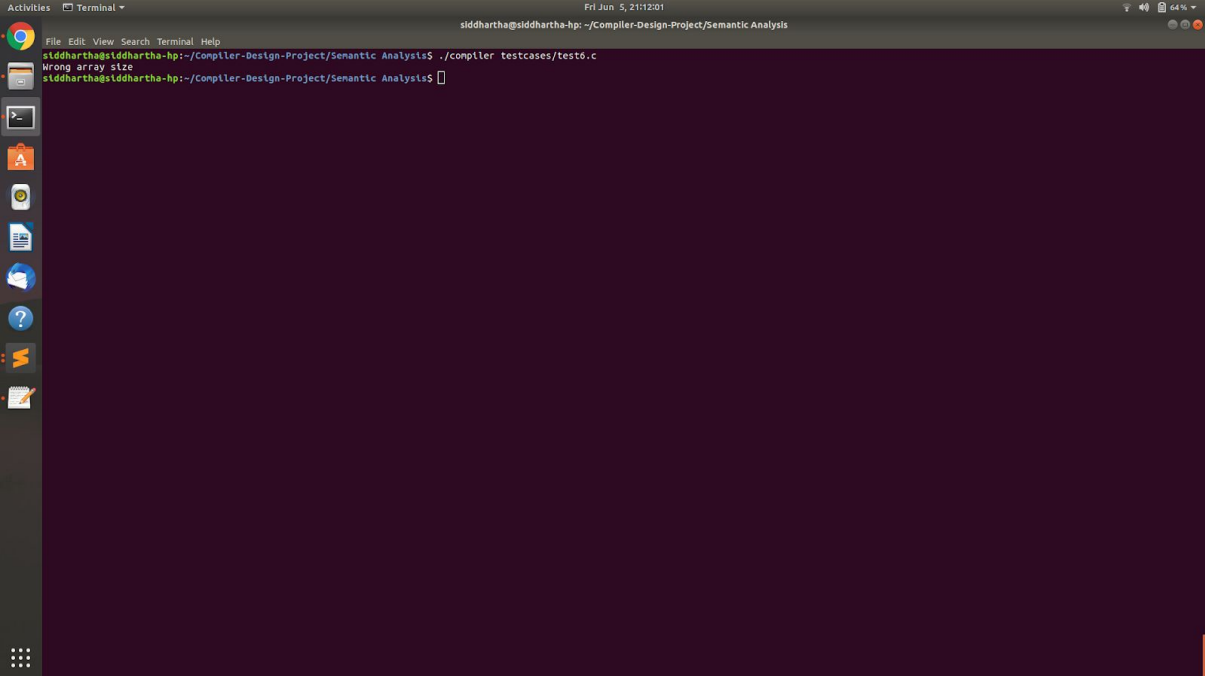
Test 5: Wrong array size

```
#include<stdio.h>

void main()
{
    int a[0];
}
```

Status : Pass

Output :



A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help) and a status bar (Fri Jun 5, 21:12:01, siddhartha@siddhartha-hp: ~/Compiler-Design-Project/Semantic Analysis, 64%). The terminal shows the following commands and output:

```
siddhartha@siddhartha-hp:~/Compiler-Design-Project/Semantic Analysis$ ./compiler testcases/test6.c
Wrong array size
siddhartha@siddhartha-hp:~/Compiler-Design-Project/Semantic Analysis$
```

Fig 8

Implementation:

The lexer code submitted in the previous phase took care of most of the features of C using regular expressions. Some special corner cases were taken care of using custom regex. These were:

- A. The Regex for Identifiers
- B. Multiline comments should be supported
- C. Literals
- D. Error Handling for Incomplete String
- E. Error Handling for Nested Comments

Scope

Implemented using stack. Variable called scope is used which is initially set to 0. Increment and push to the stack when the opening brace is found. On finding the closing brace, pop out the last element from the stack. This will assign unique value to each block of code. Stack as a whole contains the scope-id, which is unique. In case 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. Redclaration 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 the 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

The following functions were written in order to check semantics:

1. `insertSTnest()` - This function was used to insert the nesting value of an identifier to the symbol table.
2. `insertSTparamscount()` - Inserts the count of number of parameters for a function
3. `getSTparamscount()` - Get the number of parameters in a function
4. `deletedata()` - This function deletes the data when its scope is over.
5. `checkscope()` - It checks whether the identifier is declared in the current scope or not.
6. `check_id_is_func()` - Check if the identifier is declared as a function or not.
7. `checkarray()` - It checks whether the identifier is of array data type or not. If yes it returns true else false.
8. `duplicate()` - It checks if the identifier was already declared or not.
9. `check_duplicate()` - It checks if the function is re-declared or not.
10. `check_declaration()` - It checks if the function is declared or not,
11. `check_params()` - it checks whether the parameters used in function definition are not of type void.
12. `char gettype()` - it returns the first char of the data type of identifier

Future Work:

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.

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

References:

1. <http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf>
2. Compilers: Principles, Techniques, and Tools: Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
3. <https://www.gatevidyalay.com/compiler-design/>