Semantic Analyser for C- language

Course Project - 3 (July 2020 - Dec 2020)
CS305 Compiler Design Lab
National Institute of Technology, Karnataka

SUBMITTED TO:
Prof. P. Santhi Thilagam
CSE Dept, NITK

TEAM
Kesana Jahnavi (181CO127)
Shreeya Sanjay Sand (181CO150)
Shumbul Arifa (181CO152)
Keerti Chaudhary (181CO226)

Abstract

This report contains the details of the tasks finished as a part of Phase Three of Compiler Design Lab. We have developed a Semantic Analyser for C language which makes use of the parser of the previous phase. The objective of this assignment is to perform semantic analysis such as type and scope analysis and declaration processing, and integrate such analyses with the parser. 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 Bounds Checking

A compiler is a computer program that transforms source code written in a programming language (the source language) into another computer language (the target language), wi the latter often having a binary form known as object code. When executing, the compiler first parses a of the language statements syntactically one after the other and then, in one or more successive stages or passes, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Four Phases of the frontend compiler are Lexical phase, Syntax phase, Semantic phase and Intermediate code generation. Once the parse tree is generate the Semantic Analyser will check actual meaning of the statement parsed in parse tree. Semantic analysis is mainly a process in compiler construction after parsing to gather necessary semantic information fro the source code.

Semantic analyser checks whether syntax structure constructed in the source program derives any meaning or not. It is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. Semantic analyser is also called context sensitive analysis. This phase perform semantic checks such as type checking (checking for type errors), or definite assignment (variable to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis logically follows the parsing phase, and logically precedes the code generation phase. In this project we have done variable type checking, handling the scope of the variables, function parameters type checking, numbe of parameters matching in function call and array dimensionality check along with array index type checking. We have also handled the cases of undeclared variables and variable redeclaration. The semantic analyzer also checks if predefined functions (like printf, scanf, gets, getchar etc) are redefined in the program.

INDEX

List of Figures:

Overview

FEATURES

The following functionalities are being checked in Semantic Analysis:

- Symbol table insertion for different scopes
- Multiple functions
- Assignment expression
- Undeclared variable
- Redeclaration in same scope
- Out of scope
- Type mismatch
- Redeclaration of pre defined function
- Return type of function mismatch
- Same variable different scopes
- Different data types
- Usage of non-array variable with subscript
- Out of bounds subscript
- Usage of array identifier without subscript

RESULTS

- Semantic Errors in the source program along with appropriate error messages
- Symbol table is displayed with appropriate attributes.

TOOLS USED

- Flex
- Yacc

Introduction

SEMANTIC ANALYSIS

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation. An important part of the semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array. The language specification may permit some type conversions called coercions. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers

Semantics

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. These rules are set by the grammar of the language and evaluated in semantic analysis. The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking

- Array-bound checking

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information attributes) are appended to one or more of its non-terminals in order to provide context-sensitive nformation. Each attribute has a well-defined domain of values, such as integer, float, character, string nd expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help speci he syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree an pass values or information among the nodes of a tree.

YACC SCRIPT

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the tructures of his input, together with code to be invoked as each such structure is recognized. Yacc turn uch a specification into a subroutine that handles the input process; frequently, it is convenient and ppropriate to have most of the flow of control in the user's application handled by this subroutine.

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 onstructs such as names and numbers. The user-supplied routine may also handle idiomatic features uch as comment and continuation conventions, which typically defy easy grammatical specification. 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 hat 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 ny C code here, which will be copied verbatim into the generated source file. The rules section ssociates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code. The C code section contains C statements and unctions that are copied verbatim to the generated source file. These statements presumably contain code alled by the rules in the rules section. In large programs, it is more convenient to place this code in a eparate file linked in at compile time.

C PROGRAM

The workflow is explained as follows:

- Compile the script using Yacc tool.

```
$ yacc -d parser.y
```

- Compile the flex script using Flex tool.

```
$ lex scanner.l
```

- After compiling the lex file, lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generate after compiling the yacc script.
  $ gcc lex.yy.c y.tab.h y.tab.c -w
- The executable file is generated, which on running parses the C file given as a command line input.
  $ ./a.out tests/test_number.c

Code

Lexer Code

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

        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 && ST[i].nestval == currnest)
                 {
                        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;
                        }
                        char *scope_depth;
                        int j;
                        if(ST[i].nestval == 99999)
                        {
                                scope_depth = "*";
                        }
                        if(ST[i].nestval == 9999)
                        {
                                scope_depth = "**";
                        }
                        if(ST[i].nestval == 999)
                        {
                                scope_depth = "***";
                        }
                        if(ST[i].nestval == 99)
                        {
                                scope_depth = "****";
                        }
                        if(ST[i].nestval == 9)
                        {
                                scope_depth = "*****";
                        }
                        if(strcmp(ST[i].name, "main") == 0)
                        {
                                printf("%10s | %15s | %10s | %10s | %10d | %15s | %10d |\n",ST[i].name
ST[i].class, ST[i].type, ST[i].value, ST[i].lineno, scope_depth, 0);
                        }
                        else
                        {
                                printf("%10s | %15s | %10s | %10s | %10d | %15s | %10d |\n",ST[i].name
ST[i].class, ST[i].type, ST[i].value, ST[i].lineno, scope_depth, ST[i].params_count);
                        }
                }
```

```
    }


        void printCT()
        {
                printf("%10s | %15s\n","NAME", "TYPE");
                for(int i=0;i<50;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])))*\*+\/
                { }
[ \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-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");
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;
               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
```

```
%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_list ',' variable_declaration_identifier |
variable_declaration_identifier;

variable_declaration_identifier
                      : identifier
{if(duplicate(curid)){printf("Duplicate\n");exit(0);}insertSTnest(curid,currnest); ins();  } vdi
                        | array_identifier
{if(duplicate(curid)){printf("Duplicate\n");exit(0);}insertSTnest(curid,currnest); ins();  } vdi;



vdi : identifier_array_type | assignment_operator simple_expression  ;

identifier_array_type
                      : '[' initilization_params
                      | ;

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

initilization
                      : string_initilization
                      | 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 o
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 no
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 i
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("Typ
mismatch\n"); exit(0);}
                                                                             }
                      | mutable addition_assignment_operator expression     {

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

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

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

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

                               if($1==1 && $3==1)
                                                                                $$=1;
                                                                                else
                                                                                {$$=-1; printf("Typ
mismatch\n"); exit(0);}
                                                                             }
                      | mutable increment_operator                                              {if(
== 1) $$=1; else $$=-1;}
                      | mutable decrement_operator                                              {if(
== 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(currfunccall,curid);
                                   } arguments ')'
                                             { if(strcmp(currfunccall,"printf"))
                                                   {

       if(getSTparamscount(currfunccall)!=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", currfunccall);
                                                                  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("Status: Parsing Complete - Valid\n");
                printf("%50s", "SYMBOL TABLE\n");
                printf("%30s %s\n", " ", "+++++++++++++++++++++");
                printST();

                printf("\n\n%30s","CONSTANT TABLE\n");
                printf("%10s %s\n", " ", "++++++++++++++++++++++++++");
                printCT();
        }
}

void yyerror(char *s)
{
        printf("Line %d: %s %s\n", yylineno, s, yytext);
        flag=1;
        printf("Status: Parsing Failed - Invalid\n");
        exit(7);
}

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

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

int yywrap()
{
```

```
            return 1;
      }
```

Explanation

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

- Type mismatch
- Undeclared variable
- Reserved identifier misuse
- Multiple declaration of variable in a scope
- Accessing an out of scope variable
- Actual and formal parameter mismatch

Declaration Section
n this section we have included all the necessary header files, function declaration and lag 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 ccording to their associativity and precedence. This ensures the grammar we are giving to the parser i nambiguous as LALR (1) parser cannot work with ambiguous grammar.

Rules Section
n this section production rules for entire C language is written. The grammar productions do the synta nalysis 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 functior re called to do the necessary actions.

C-Program Section
n this section the parser links the extern functions, variables declared in the lexer, xternal files generated by the lexer etc. The main function takes the input source code ile and prints the final symbol table.

Test Cases

➢ Without errors

TEST CASE 1: output for testcase containing for loop and while loop

TEST CASE 2: output for testcase containing function

TEST CASE 3: output for testcase containing function

TEST CASE 4: output for testcase containing print statement

➢ With error:

TEST CASE 5: output displays error - function undeclared

TEST CASE 6: output displays error - void function

TEST CASE 7: output displays error - too many arguments

TEST CASE 8: output displays error - type mismatch

TEST CASE 9: output displays error - incorrect array size

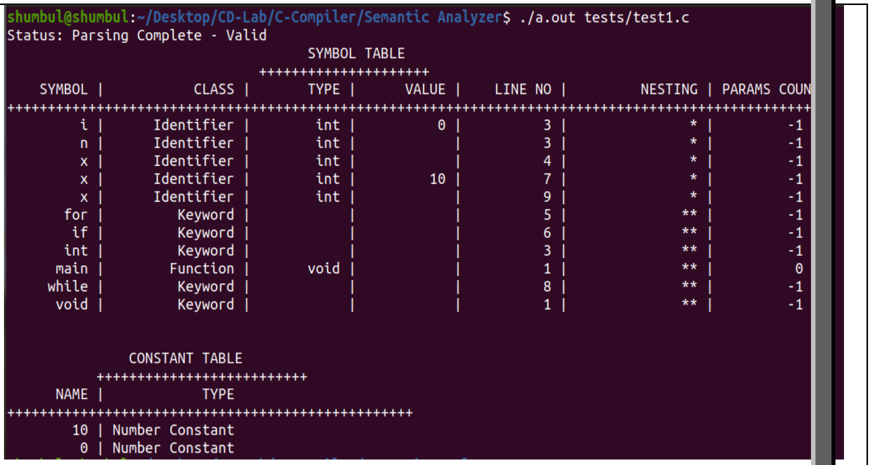
TEST CASE 10: output displays error - duplicate

TEST CASE 11: output displays error - array id has no subscript

TEST CASE 12: output displays error - condition checking is not int type

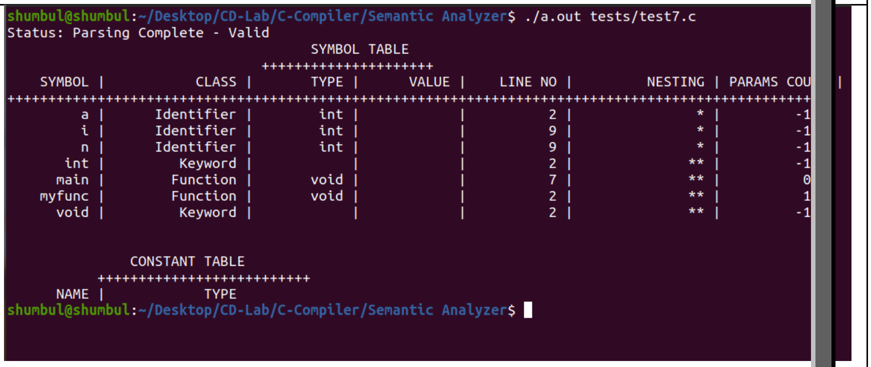
Code and Screenshots of Test Cases

```
#include<stdio.h>
void main ()
{
    int n,i;
    int x;
    for(i=0; i<n;i++) {
        if(i<10) {
            int x;
            while(x<10) {
                int x;
                x++;
            }
        }
    }
}
```



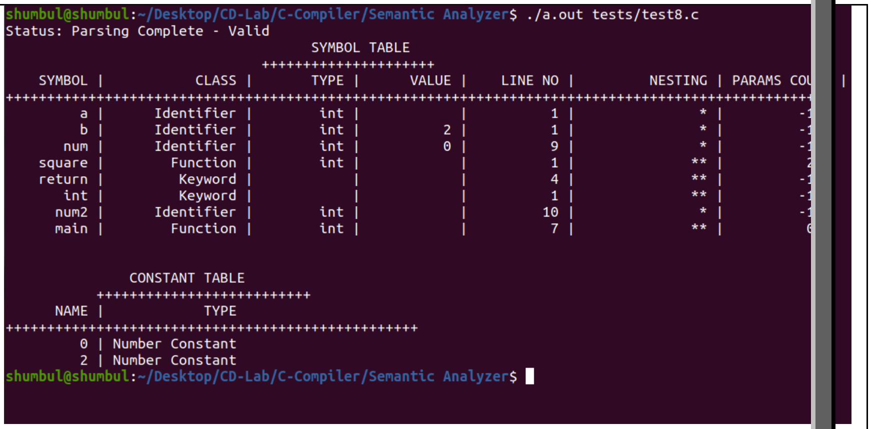
```
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyzer$ ./a.out tests/test1.c
Status: Parsing Complete - Valid
                                     SYMBOL TABLE
                              ++++++++++++++++++++
    SYMBOL |            CLASS |        TYPE |       VALUE |     LINE NO |         NESTING | PARAMS COUN
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
         i |       Identifier |         int |           0 |           3 |               * |         -1
         n |       Identifier |         int |             |           3 |               * |         -1
         x |       Identifier |         int |             |           4 |               * |         -1
         x |       Identifier |         int |          10 |           7 |               * |         -1
         x |       Identifier |         int |             |           9 |               * |         -1
       for |          Keyword |             |             |           5 |              ** |         -1
        if |          Keyword |             |             |           6 |              ** |         -1
       int |          Keyword |             |             |           3 |              ** |         -1
      main |         Function |        void |             |           1 |              ** |          0
     while |          Keyword |             |             |           8 |              ** |         -1
      void |          Keyword |             |             |           1 |              ** |         -1


               CONSTANT TABLE
           +++++++++++++++++++++++++
      NAME |            TYPE
++++++++++++++++++++++++++++++++++++++++++++++++++
        10 | Number Constant
         0 | Number Constant
```

```
#include<stdio.h>
void myfunc(int a)
{
    a++;
}

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



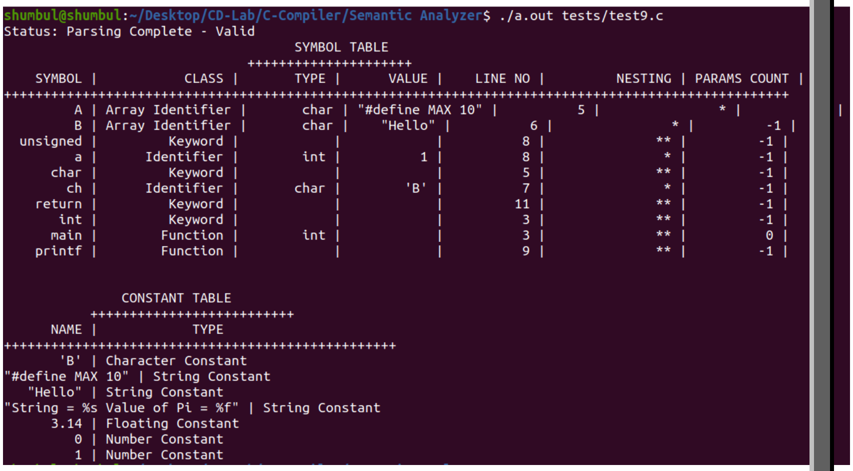
```
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyzer$ ./a.out tests/test7.c
Status: Parsing Complete - Valid
                                     SYMBOL TABLE
                              ++++++++++++++++++++
    SYMBOL |            CLASS |        TYPE |       VALUE |     LINE NO |         NESTING | PARAMS COU    |
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
         a |       Identifier |         int |             |           2 |               * |         -1
         i |       Identifier |         int |             |           9 |               * |         -1
         n |       Identifier |         int |             |           9 |               * |         -1
       int |          Keyword |             |             |           2 |              ** |         -1
      main |         Function |        void |             |           7 |              ** |          0
    myfunc |         Function |        void |             |           2 |              ** |          1
      void |          Keyword |             |             |           2 |              ** |         -1


               CONSTANT TABLE
           +++++++++++++++++++++++++
      NAME |            TYPE
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyzer$ 
```

```
#include<stdio.h>
int square (int a, int b)
{
    b = 2;
    return b;
}

int main ()
{
    int num=2;
    int num2;
    square(num,num);
     return 0;
}
```



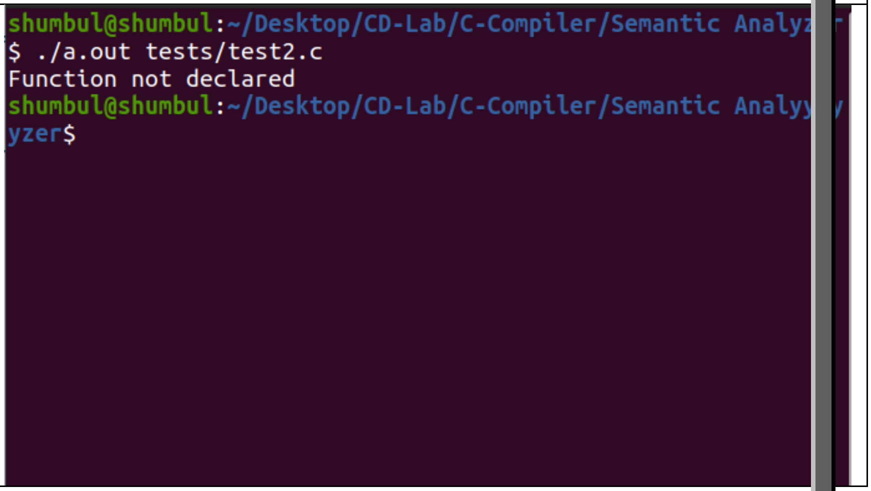
```
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyzer$ ./a.out tests/test8.c
Status: Parsing Complete - Valid
                                        SYMBOL TABLE
                              ++++++++++++++++++++++
    SYMBOL |            CLASS |         TYPE |       VALUE |     LINE NO |          NESTING | PARAMS COU     |
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
         a |       Identifier |          int |             |           1 |                * |          -1
         b |       Identifier |          int |           2 |           1 |                * |          -1
       num |       Identifier |          int |           0 |           9 |                * |          -1
    square |         Function |          int |             |           1 |               ** |           2
    return |          Keyword |              |             |           4 |               ** |          -1
       int |          Keyword |              |             |           1 |               ** |          -1
      num2 |       Identifier |          int |             |          10 |                * |          -1
      main |         Function |          int |             |           7 |               ** |           0


                  CONSTANT TABLE
              ++++++++++++++++++++++++++
      NAME |             TYPE
+++++++++++++++++++++++++++++++++++++++++++++++++++++
         0 | Number Constant
         2 | Number Constant
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyzer$
```

```
#include<stdio.h>
#define NUM 5

int main ()
{
char A [] = "#define MAX 10";
char B [] = "Hello";
char ch  = 'B';
unsigned int a = 1;
printf("String = %s Value of Pi =
%f", 3.14);

     return 0;
}
```



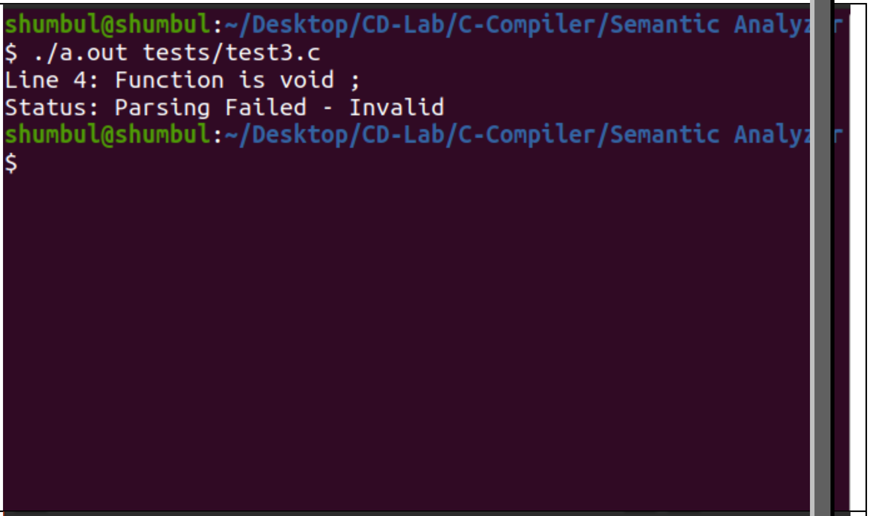
```
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyzer$ ./a.out tests/test9.c
Status: Parsing Complete - Valid
                                      SYMBOL TABLE
                              ++++++++++++++++++++++
    SYMBOL |            CLASS |         TYPE |       VALUE |    LINE NO |          NESTING | PARAMS COUNT |
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
         A | Array Identifier |         char | "#define MAX 10" |          5 |                * |              |
         B | Array Identifier |         char |     "Hello" |          6 |                * |           -1 |
  unsigned |          Keyword |              |             |          8 |               ** |           -1 |
         a |       Identifier |          int |           1 |          8 |                * |           -1 |
      char |          Keyword |              |             |          5 |               ** |           -1 |
        ch |       Identifier |         char |         'B' |          7 |                * |           -1 |
    return |          Keyword |              |             |         11 |               ** |           -1 |
       int |          Keyword |              |             |          3 |               ** |           -1 |
      main |         Function |          int |             |          3 |               ** |            0 |
    printf |         Function |              |             |          9 |               ** |           -1 |


                  CONSTANT TABLE
              ++++++++++++++++++++++++++
      NAME |             TYPE
+++++++++++++++++++++++++++++++++++++++++++++++++++++
       'B' | Character Constant
"#define MAX 10" | String Constant
   "Hello" | String Constant
"String = %s Value of Pi = %f" | String Constant
      3.14 | Floating Constant
         0 | Number Constant
         1 | Number Constant
```

```
#include<stdio.h>
void main ()
{
   int i,n;
   myfunc(i);
}
```



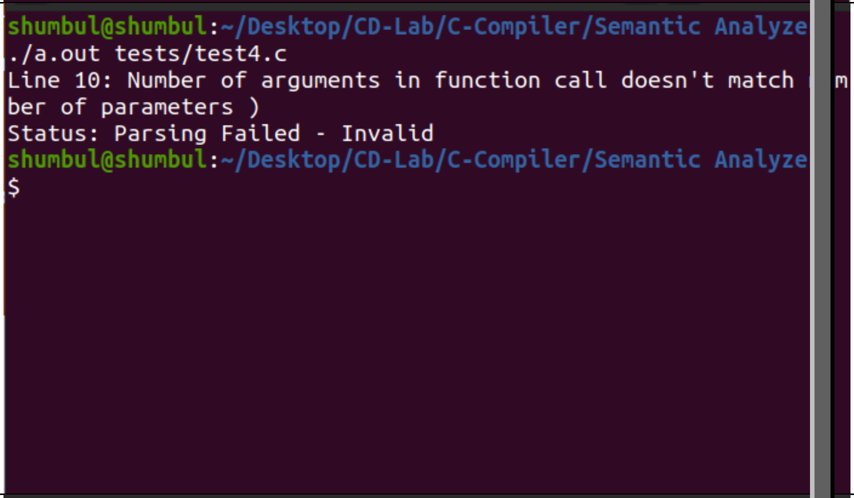
```
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyz
$ ./a.out tests/test2.c
Function not declared
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analy
yzer$
```

```
#include<stdio.h>

void myfunc(int a)
{
    return a;
}

void main ()
{
    int i,n;

    myfunc(i);
}
```



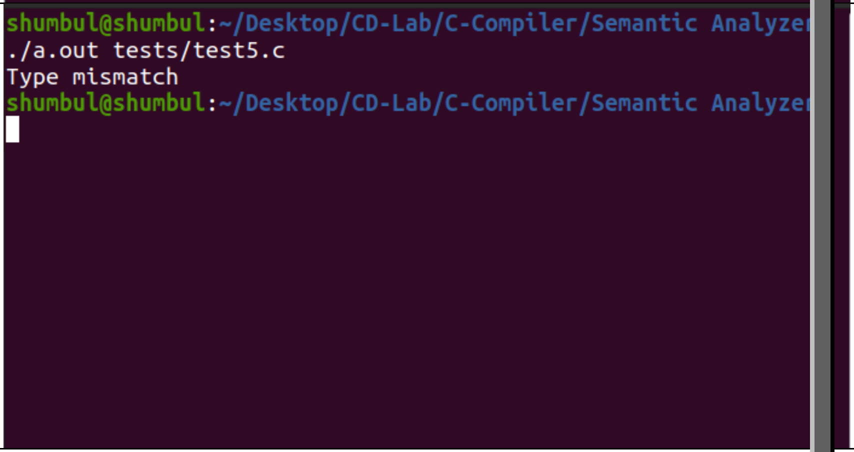
```
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyz r
$ ./a.out tests/test3.c
Line 4: Function is void ;
Status: Parsing Failed - Invalid
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyz r
$
```

```
#include<stdio.h>

int myfunc(int a)
{
    return a;
}

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

}
```

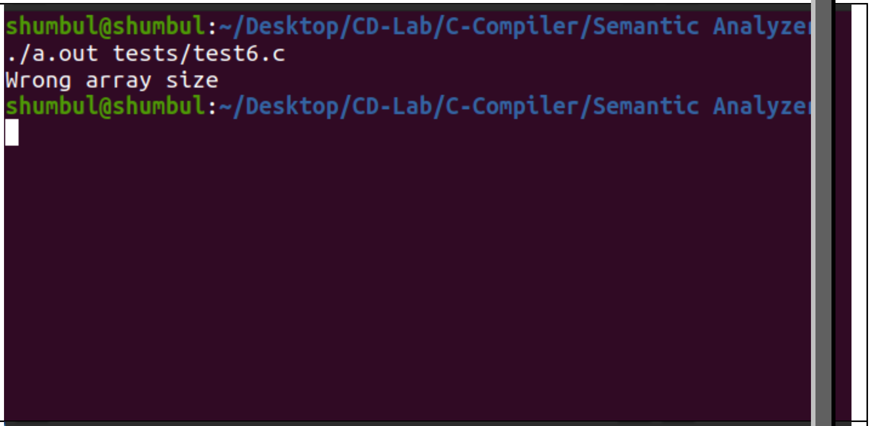


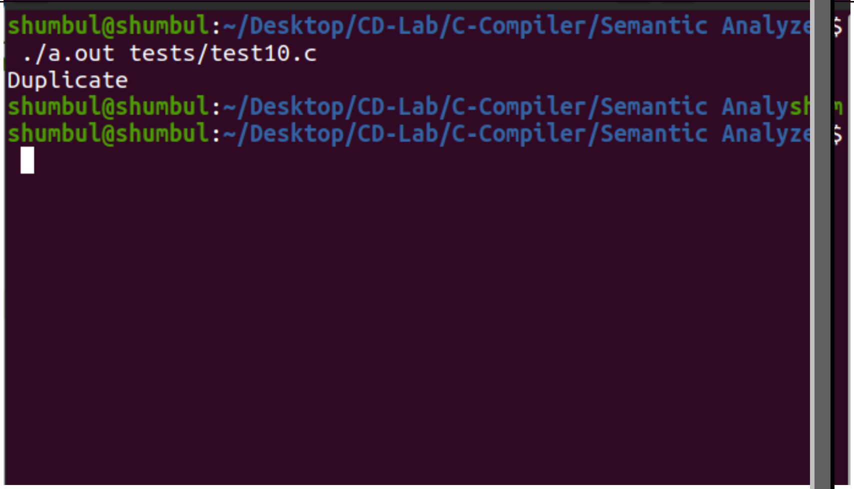
```
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyze
./a.out tests/test4.c
Line 10: Number of arguments in function call doesn't match ( m
ber of parameters )
Status: Parsing Failed - Invalid
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyze
$
```

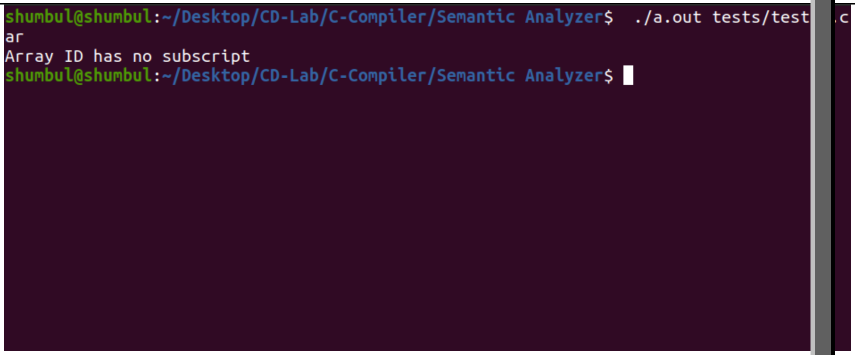
```
#include <stdio.h>
int main ()
{
    int p = 8.6;
    int q=23;
    float f=6.7;
    q=f;
    //type mismatch (int = float)
    return 0;
}
```



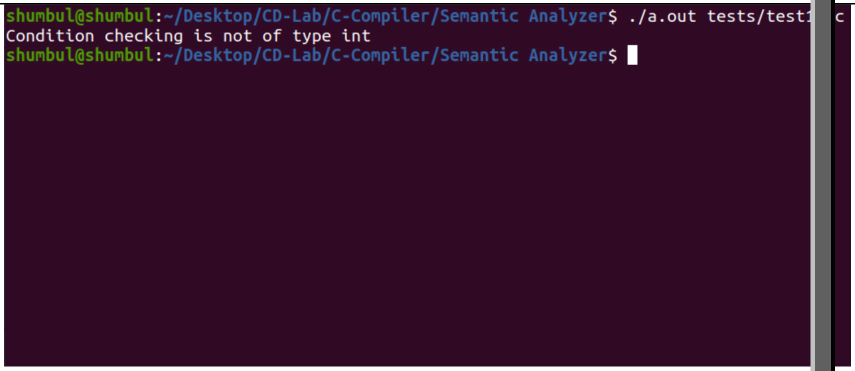
```
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyzer
./a.out tests/test5.c
Type mismatch
shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyzer
█
```

#include<stdio.h>
 	
void main ()
{
    int a [0];
}
	
	shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyze
	./a.out tests/test6.c
	Wrong array size
	shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyze

//conflicting types of a variable
 	
#include<stdio.h>
 	
int main () {
 	
   int x;
 	
   char x;
 	
   float y;
 	
   char z;
 	
     float [10] y;
 	
     x = 2.5;
 	
   jjj = 25;
 	
   b = y;
 	
}
	
	shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyze $
	 ./a.out tests/test10.c
	Duplicate
	shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analysh n
	shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyze $

#include<stdio.h>
void main ()
{
    int n,i;
    int x;
    int ar[10];
 	
    for(i=0; i<n;i++) {
        ar[i]=0;
        if(i<10) {
            int x;
            while(ar==0) {
                int x;
                x++;
	
	shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyzer$ ./a.out tests/test .c
	ar
	Array ID has no subscript
	shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyzer$

}
      }
   }
}	

#include<stdio.h>
void main ()
{
   int n,i;
   int x;
   float f=1.2345;
 	
   for(i=0; i<n;i++) {
      if(i<10) {
         int x;
         while(f==0) {
            int x;
            x++;
         }
      }
   }
}
	
	shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyzer$ ./a.out tests/test1 c
	Condition checking is not of type int
	shumbul@shumbul:~/Desktop/CD-Lab/C-Compiler/Semantic Analyzer$

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:

- The Regex for Identifiers
- Multiline comments should be supported
- Literals
- Error Handling for Incomplete String
- Error Handling for Nested Comments

The parser code requires exhaustive token recognition and because of this reason, we utilized the lexer code given under the C specifications with the parser. The parser implements C grammar using a number of production rules. The parser takes tokens from the lexer output, one at a time and applies the corresponding production rules to append to the symbol table with type, value and line of declaration. If the parsing is not successful, the parser outputs the line number with the corresponding error. Along with this, semantic actions were also added to each production rule to check if the structure created has some meaning or not.

The semantic analyzer is built by adding subroutines and C functions for the grammar rules defined in the syntax analyzer phase of the compiler. The symbol table of syntax phase is updated in the semantic phase. The symbol table contains the following columns:

- Serial No - represents the number of entries in the symbol table

- Identifier - specifies the name of the variables which are identifiers
- Scope - specifies the scope of the variables
- Value - represents the mathematical value of a variable if initialized/defined
- Type - specifies the data type of the variable
- Dimension - specifies the dimension if the identifier is an array
- Parameter type - specifies the data types of function arguments/parameters
- Parameter list - specifies the names of function parameters

Conclusion

The lexical analyzer, syntax analyzer and the semantic analyzer for a subset of C anguage, which include selection statements, compound statements, iteration tatements (for, while and do-while) and user defined functions is generated. It is important to define nambiguous grammar in the syntax analysis phase. The emantic analyzer performs type checking, reports various errors such as ndeclared variable, type mismatch, errors in function call (number and datatypes of parameters nismatch) and errors in array indexing.

Results

The lex file (parser.l) and yacc (parser.y) are compiled using following commands:

```
yacc -d parser.y
flex scanner.l
gcc y.tab.c lex.yy.c -w
./a.out tests/test_number.c
```

Tokens recognized by the lexer are successfully parsed in the parser. The output displays the set of dentifiers and constants present in the program with their types. The parser generates error messages i ase of any syntactical or semantic errors in the test program.

Future Work

We have implemented the parser and semantic analyzer for only a subset of C anguage. The future work may include defining the grammar and specifying the emantics for switch statements, predefined functions (like string functions, fileread and write function), ump statements and enumerations. The yacc script presented in this report takes care of all the rules of C anguage but is not fully exhaustive in nature. Our future work would include making the script even nore robust to handle all aspects of C language and making it more efficient. We would also work on ntermediate Code Generator.

References

- Compilers Principles, Techniques and Tool by Alfred V.Aho, Monica S. Lam, Ravi Sethi, Jeffry D. Ullman
- http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf - Lex and Yacc Tutoria by Tom Nieman
- http://dinosaur.compilertools.net/yacc/
- https://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/tutorial-large.pdf

- http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf
- https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm - Compiler Design Semantic Analysis by TutorialsPoint

****