Semantic Analyzer for the C Language

National Institute of Technology Karnataka Surathkal

Date: 21-10-2020

Submitted To: Mr. Ankit Kurup

Group Members:

Paranjaya Saxena	**181CO138**
Ameya Deshpande	**181CO205**
Manas Trivedi	**181CO231**
Shuddhatm Choudhary	**181CO251**

Abstract:

Programming languages are notations for describing computations to people and machines. All software running on any device was written in some programming language. A program written in a programming language cannot be run directly on a device, it must first be translated into a form which can be executed by a computer. The system which does this translation is called a compiler. It converts programs written in high-level languages to a low-level language which can be directly executed.

A compiler works in a series of phases to produce machine code:
1. Lexical Analysis, which produces a stream of tokens.
2. Syntax Analysis, or 'Parsing', which produces a syntax tree.
3. Semantic Analysis, which checks the source program for semantic consistency.
4. Intermediate Code Generation, which generates a low-level intermediate representation of the source program.
5. Code Optimization, which attempts to improve the intermediate code.
6. Code Generation, which generates the target machine code.

The project 'Semantic Analyzer for the C Language' is an attempt to understand and implement the third phase of a C compiler, i.e. Semantic Analysis.

Contents

1 Introduction

1.1 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 of conversions called coercions. For example, a binary arithmetic operator may be applied to either a pair of integers or a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Semantics

The semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics helps 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

An 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 to provide context-sensitive information. Each attribute has a well-defined domain of values, such as integer, float, character, string, and expressions.

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

1.2 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 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 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 import 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. The rules section associates 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 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.

1.3 C Program

The workflow is explained as under:

- Compile the script using the Yacc tool
  ```
  $ yacc -d parser.y
  ```
- Compile the lex script using the lex tool
  ```
  $ lex scanner.l
  ```
- After compiling the lex file, the lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generated after compiling the yacc script.
- The compilation is done with the following command:
  ```
  $ gcc lex.yy.c y.tab.c
  ```

2 Design of Programs

2.1 Code

parser.y

```
%{
	void yyerror(char* s);
	int yylex();
	#include "stdio.h"
	#include "stdlib.h"
	#include "ctype.h"
	#include "string.h"
	void ins();
	void insV();
	int flag=0;
	extern char curid[20];
	extern char curtype[20];
	extern char curval[20];
	extern int currnest;
	void deleteData (int );
	int identifierInScope(char*);
	int isIdentifierAFunc(char *);
	void insertST(char*, char*);
	void insertIdentifierNestVal(char*, int);
	void insertFuncArgsCount(char*, int);
	int getFuncArgsCount(char*);
	int isFuncRedeclared(char*);
```

```
      int isFuncDeclared(char*, char *);
      int areFuncArgsNotVoid(char*);
      int isIdentifierAlreadyDeclared(char *s);
      int isIdentifierAnArray(char*);
      char currfunctype[100];
      char currfunc[100];
      char currfunccall[100];
      void insertSTF(char*);
      char getFirstCharOfIDDatatype(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_list ','
variable_declaration_identifier | variable_declaration_identifier;
```

```
variable_declaration_identifier
                : identifier
{if(isIdentifierAlreadyDeclared(curid)){printf("Identifier is already
declared!\n");exit(0);}insertIdentifierNestVal(curid,currnest);
ins();  } vdi
                  | array_identifier
{if(isIdentifierAlreadyDeclared(curid)){printf("Identifier is already
declared!\n");exit(0);}insertIdentifierNestVal(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);
isFuncRedeclared(curid); insertSTF(curid); ins(); };

function_declaration_param_statement
                : params ')' statement;

params
                : parameters_list | ;

parameters_list
                : type_specifier { areFuncArgsNotVoid(curtype); }
parameters_identifier_list { insertFuncArgsCount(currfunc,
params_count); };

parameters_identifier_list
                : param_identifier
parameters_identifier_list_breakup;

parameters_identifier_list_breakup
                : ',' parameters_list
                | ;

param_identifier
```

```
                : identifier {
ins();insertIdentifierNestVal(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(isIdentifierAFunc(curid))
                                    {printf("Function name used as
Identifier\n"); exit(8);}
                                if(!identifierInScope(curid))

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

if(getFirstCharOfIDDatatype(curid,0)=='i' ||
getFirstCharOfIDDatatype(curid,1)== 'c')
                                $$ = 1;
                                else
```

```
                                $$ = -1;
                                }
                | array_identifier
{if(!identifierInScope(curid)){printf("%s\n",curid);printf("Undeclare
d\n");exit(0);}} '[' expression ']'

{if(getFirstCharOfIDDatatype(curid,0)=='i' ||
getFirstCharOfIDDatatype(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(!isFuncDeclared(curid, "Function"))
                                { printf("Function not declared");
exit(0);}
                                insertSTF(curid);
                                    strcpy(currfunccall,curid);
                                } arguments ')'
                                    { if(strcmp(currfunccall,"printf"))
                                        {

if(getFuncArgsCount(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("%30sSYMBOL TABLE\n", " ");
            printf("%30s %s\n", " ", "------------");
```

```
            printST();

            printf("\n\n%30sCONSTANT TABLE\n", " ");
            printf("%30s %s\n", " ", "--------------");
            printCT();
      }
}

void yyerror(char *s)
{
      printf("%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;
}
```

scanner.l

```
%{
#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 insertIdentifierNestVal(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 insertFuncArgsCount(char *s, int count) {
    for(int i = 0 ; i < 1001 ; i++)
        if(strcmp(ST[i].name,s)==0 ) ST[i].params_count = count;
}

int getFuncArgsCount(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 identifierInScope(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 isIdentifierAFunc(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 isIdentifierAnArray(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 isIdentifierAlreadyDeclared(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 isFuncRedeclared(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 isFuncDeclared(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 areFuncArgsNotVoid(char* type_specifier)
    {
        if(!strcmp(type_specifier, "void"))
        {
            printf("Parameters cannot be of type void\n");
            exit(0);
        }
```

```
        return 0;
    }

    char getFirstCharOfIDDatatype(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 | %10s
|\n","Symbol", "Class", "Type","Value", "Line no.", "Arg count");
        for(int i=0;i<100;i++) {
            printf("-");
        }
        printf("\n");
        for(int i = 0 ; i < 1001 ; i++)
        {
            if(ST[i].length == 0)
            {
                continue;
            }
            if (ST[i].params_count > 0) {
                printf("%10s | %15s | %10s | %10s | %10d | %10d
|\n",ST[i].name, ST[i].class, ST[i].type, ST[i].value, ST[i].lineno,
ST[i].params_count);
            } else {
                printf("%10s | %15s | %10s | %10s | %10d | \t\t
|\n",ST[i].name, ST[i].class, ST[i].type, ST[i].value, ST[i].lineno);
            }
        }
    }
```

```
    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])))*\*+\/
                            { }
[ \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;
}

%%
```

2.2 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 the parser uses it for further computation. We are using the symbol table and a constant table obtained from the previous phase. We added functions `insertIdentifierNestVal()`, `insertFuncArgsCount()`, `getFuncArgsCount()`, `deleteData()`, `identifierInScope()`, `isIdentifierAFunc()`, `isIdentifierAnArray()`, `isIdentifierAlreadyDeclared()`, `isFuncRedeclared()`, `isFuncDeclared()`, `areFuncArgsNotVoid()`, `getFirstCharOfIDDatatype()`, 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 the 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 the entire C, language is written. The grammar productions do 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.

3 Test Cases

Test Case 1: Variable used without declaration

```
// Variable e not defined
#include<stdio.h>
```

```c
int main() {


    int a = 10;
    int b = 20;
    int c = 30;

    int d = a + b + c + e;
    printf("%d", d);




    return 0;
}
```

Output

```
ameya@earth:~/ALL_WORK/c_compiler_lex_yacc/Semantic_Analyser$ ./bin/sem_analyser < test/test1.c
e
Undeclared
```

Result: Passed

Test Case 2: Variable redeclared in the same scope

```c
// Variable c defined in the function again.
#include<stdio.h>

int add(int a, int b, int c) {
    int c = a + b;
```

```
    return c;
}


int main() {

    int a = 20;
    int b = 30;
    int c = 0;

    int res = add(a, b, c);
    printf("%d", res);

    return 0;
}
```

Output:

```
ameya@earth:~/ALL_WORK/c_compiler_lex_yacc/Semantic_Analyser$ ./bin/sem_analyser < test/test2.c
Identifier is already declared!
```

Result: Passed

Test Case 3: Scope violation

```
// Variable defined outside the scope.
#include <stdio.h>

int main() {

    if(3 > 2){
      int a = 5;
      printf("%d", a);
```

```
    }

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

Output:

```
ameya@earth:~/ALL_WORK/c_compiler_lex_yacc/Semantic_Analyser$ ./bin/sem_analyser < test/test3.c
a
Undeclared
```

Result: Passed

Test Case 4: Number of formal and actual parameters is different

```
// Semantic error - Number of formal and actual parameters
is different
#include<stdio.h>
void functionPow2(int a){
    a = a + a;
}
int main(){
    int a;
    int b;
    a = 2;
    b = 4;
    functionPow2(a, b);
}
```

Output:

```
ameya@earth:~/ALL_WORK/c_compiler_lex_yacc/Semantic_Analyser$ ./bin/sem_analyser < test/test4.c
14 Number of arguments in function call doesn't match number of parameters )
Status: Parsing Failed - Invalid
```

Result: Passed

Test Case 5: Type mismatch of formal and actual parameters

```c
// Semantic error - Type mismatch of formal and actual
parameters
#include<stdio.h>

void functionPow2(int a, double b){
    a = a + a;
}

int main(){
    double a;
    int b;
    a = 2.0;
    b = 4;
    functionPow2(a, b);
}
```

Output:

```
ameya@earth:~/ALL_WORK/c_compiler_lex_yacc/Semantic_Analyser$ ./bin/sem_analyser < test/test5.c
Type mismatch
```

Result: Passed

Test Case 6: Correct Code

```c
// Correct Code v2

#include<stdio.h>
```

```c
void addFunc(int a, int b)
{
    int ans;
    ans = a + b;
}

int main()
{
    int a;
    int b;
    a = 2;
    b = 4;
    if(a < b)
    {
      int c;
      c = a + b;
    }
    addFunc(a, b);
}
```

Output:

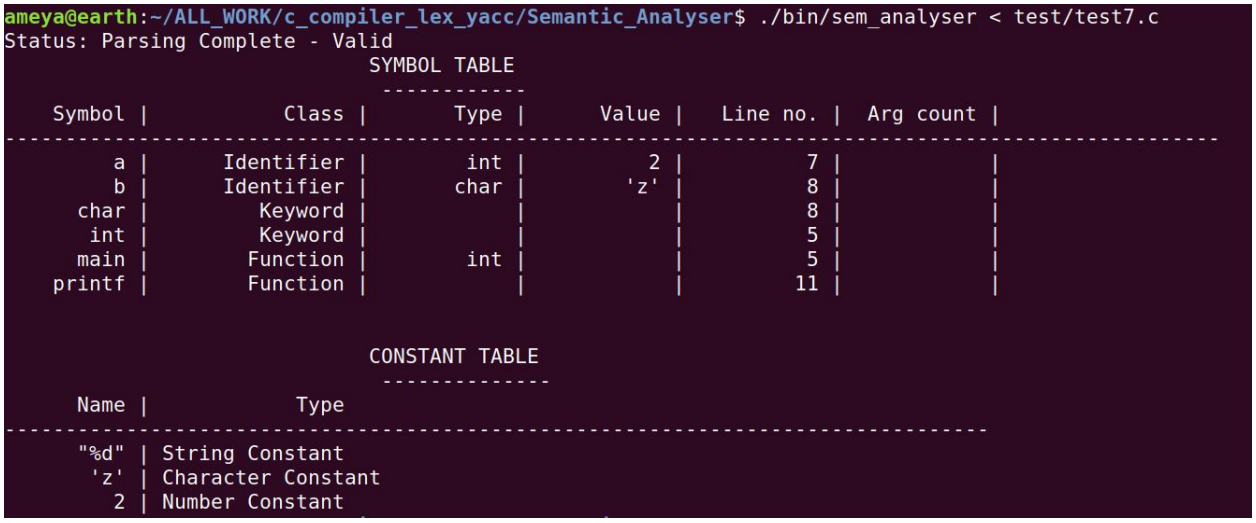
```
ameya@earth:~/ALL_WORK/c_compiler_lex_yacc/Semantic_Analyser$ ./bin/sem_analyser < test/test6.c
Status: Parsing Complete - Valid
                        SYMBOL TABLE
                         ------------
   Symbol |           Class |       Type |      Value |   Line no. |  Arg count |
-------------------------------------------------------------------------------------------------
        a |      Identifier |        int |            |          5 |            |
        b |      Identifier |        int |            |          5 |            |
        a |      Identifier |        int |          2 |         13 |            |
        b |      Identifier |        int |          4 |         14 |            |
        c |      Identifier |        int |            |          0 |            |
       if |         Keyword |            |            |         17 |            |
      int |         Keyword |            |            |          5 |            |
     main |        Function |        int |            |         11 |            |
      ans |      Identifier |        int |            |          7 |            |
  addFunc |        Function |       void |            |          5 |          2 |
     void |         Keyword |            |            |          5 |            |


                        CONSTANT TABLE
                         --------------
     Name |           Type
-----------------------------------------------------------------------------
        2 | Number Constant
        4 | Number Constant
```

Result: Passed

Test Case 7: Correct Code - Multiple datatypes

```c
// Correct Code - Multiple datatypes
#include<stdio.h>
int main()
{
     int a;
     char b;
     a = 2;
     b = 'z';
     printf("%d", &a);
}
```

Output:



```
ameya@earth:~/ALL_WORK/c_compiler_lex_yacc/Semantic_Analyser$ ./bin/sem_analyser < test/test7.c
Status: Parsing Complete - Valid
                            SYMBOL TABLE
                             ------------
   Symbol |           Class |       Type |      Value |   Line no. |  Arg count |
-------------------------------------------------------------------------------------------------
        a |      Identifier |        int |          2 |          7 |            |
        b |      Identifier |       char |        'z' |          8 |            |
     char |         Keyword |            |            |          8 |            |
      int |         Keyword |            |            |          5 |            |
     main |        Function |        int |            |          5 |            |
   printf |        Function |            |            |         11 |            |


                            CONSTANT TABLE
                             --------------
     Name |            Type
-----------------------------------------------------------------------------
     "%d" | String Constant
      'z' | Character Constant
        2 | Number Constant
```

Result: Passed

Test Case 8: Correct Code - Type compatible expression

```c
// Correct Code - Type compatible expression
#include<stdio.h>

int main()
{
     int a;
     int b;
     int c;
     a = 1;
     b = 2;
     c = a + b;
}
```

Output:

```
ameya@earth:~/ALL_WORK/c_compiler_lex_yacc/Semantic_Analyser$ ./bin/sem_analyser < test/test8.c
Status: Parsing Complete - Valid
                          SYMBOL TABLE
                           ------------
   Symbol |           Class |       Type |      Value |   Line no. |  Arg count |
-------------------------------------------------------------------------------------------------
        a |      Identifier |        int |          1 |          6 |            |
        b |      Identifier |        int |          2 |          7 |            |
        c |      Identifier |        int |            |          8 |            |
      int |         Keyword |            |            |          4 |            |
     main |        Function |        int |            |          4 |            |


                          CONSTANT TABLE
                           --------------
     Name |            Type
-----------------------------------------------------------------------------
        1 | Number Constant
        2 | Number Constant
```

Result: Passed

Test Case 9: Correct Code - No scope violation

```c
// Correct code - No scope violation
#include<stdio.h>
int main(){
     int a;
```

```
    a = 1;
    if(a > 0){
        a = 2;
    }
}
```

Output:

```
ameya@earth:~/ALL_WORK/c_compiler_lex_yacc/Semantic_Analyser$ ./bin/sem_analyser < test/test9.c
Status: Parsing Complete - Valid
                        SYMBOL TABLE
                         ------------
   Symbol |          Class |       Type |      Value |   Line no. |  Arg count |
-----------------------------------------------------------------------------------------------
        a |      Identifier |        int |          0 |          5 |            |
       if |         Keyword |            |            |          7 |            |
      int |         Keyword |            |            |          4 |            |
     main |        Function |        int |            |          4 |            |


                        CONSTANT TABLE
                         --------------
     Name |           Type
-------------------------------------------------------------------------------
        0 | Number Constant
        1 | Number Constant
        2 | Number Constant
```

Result: Passed

Test Case 10: Correct Code - Formal and actual params match

```
// Correct code - Formal and actual params match
#include<stdio.h>
void function(int a, int b){
    a = a + b;
}
int main(){
    int a;
    int b;
    a = 1;
    b = 2;
    function(a, b);
```

```
}
```

Output:

```
ameya@earth:~/ALL_WORK/c_compiler_lex_yacc/Semantic_Analyser$ ./bin/sem_analyser < test/test10.c
Status: Parsing Complete - Valid
                            SYMBOL TABLE
                             ------------
   Symbol |           Class |       Type |      Value |   Line no. |  Arg count |
-------------------------------------------------------------------------------------------------
        a |      Identifier |        int |            |          3 |            |
        b |      Identifier |        int |            |          3 |            |
        a |      Identifier |        int |          1 |          7 |            |
        b |      Identifier |        int |          2 |          8 |            |
 function |        Function |       void |            |          3 |          2 |
      int |         Keyword |            |            |          3 |            |
     main |        Function |        int |            |          6 |            |
     void |         Keyword |            |            |          3 |            |


                            CONSTANT TABLE
                             --------------
     Name |            Type
-----------------------------------------------------------------------------
        1 | Number Constant
        2 | Number Constant
```

Result: Passed

4 Implementation

The yacc script takes the stream of tokens recognized by the lexer.
The following semantic error is checked in this phase :

- Undeclared variable
- Redeclaration of the variable in the same scope
- Variable out of scope
- The return type of function mismatch
- Number of parameters in a function

5 Results

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

6 Future work

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

7 References

[1] Aho, A.V., Lam, M.S., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, & Tools*, 2nd ed., Pearson
[2] cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
[3] dinosaur.compilertools.net
[3] http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf