National Institute of Technology Karnataka, Surathkal

COMPILER DESIGN PROJECT - III
Semantic Analyser for C language

Submitted to:

Dr. P. Santhi Thilagam

Ms. Sushmitha

Mrs. Uma Priya

Date: 29/03/2018

Project team:

Yeshwanth R (15CO154)

Shivananda D (15CO148)

ABSTRACT

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), with the latter often having a binary form known as object code. When executing, the compiler first parses all 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 generated 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 from 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 performs 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, number 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.

CONTENTS

1. **Introduction** 3
2. **Codes** 5
3. **Implementation** 25
4. **Test Cases** 26
5. **Results** 31
6. **Conclusion** 31
7. **Future Work** 31
8. **References** 32

INTRODUCTION

Semantic Analysis

Semantic analysis is the task of ensuring that the declarations and statements of a program are Semantically correct, i.e that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used. Semantic analysis can compare information in one part of a parse tree to that in another part (e.g compare reference to variable agrees with its declaration, or that parameters to a function call match the function definition). Implementing the semantic actions is conceptually simpler in recursive descent parsing because they are simply added to the recursive procedures. Some of the functions of Semantic analysis are that it maintains and updates the symbol table, check source programs for semantic errors and warnings like type mismatch, global and local scope of a variable, re-definition of variables, usage of undeclared variables.

Structure of Lex Program

The script written by us is a program that generates lexical analyzers ("scanners" or "lexers"). Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

The structure of our flex script is intentionally similar to that of a yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

Definition section

%%

Rules section

%%

C code section

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

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.

Structure of Yacc Program

A YACC source program is structurally similar to a LEX one.

Declarations

%%

Rules

%%

Routines

- The declaration section may be empty. Moreover, if the routines section is omitted, the second %% mark may be omitted also.
- Blanks, tabs and newlines are ignored except that they may not appear in names.

Declarations Section

- Declarations of tokens. Yacc requires token names to be declared as such using the keyword %token
- Declaration of the start symbol using the keyword %start.
- C declarations: included files, global variables, types.
- C code between %{ and % }

Rules Section

A rule has the form:
Nonterminal : sentential form

| sentential form
……
| sentential form
;

Actions may be associated with rules and are executed when the associated sentential form is matched.

CODES

Lex file: parser.l

```
%{
int yylineno;
%}

alpha [A-Za-z]
digit [0-9]
und [_]

%%
[ \t] ;
\n    {yylineno++;}
"{"    {push(); return '{';}
"}"    {pop(); return '}';}
";"     {   return(';'); }
","     {   return(','); }
":"     {   return(':'); }
"="         {   return('='); }
"("     {   return('('); }
")"     {   return(')'); }
("[")       {   return('['); }
("]")       {   return(']'); }
"."     {   return('.'); }
"&"         {   return('&'); }
"!"     {   return('!'); }
"~"         {   return('~'); }
"-"     {   return('-'); }
"+"         {   return('+'); }
"*"         {   return('*'); }
"/"     {   return('/'); }
"%"         {   return('%'); }
"<"         {   return('<'); }
">"         {   return('>'); }
"^"         {   return('^'); }
"|"     {   return('|'); }
```

```
"?"     {   return('?'); }
int    {yylval.ival = INT; return INT;}
float    {yylval.ival = FLOAT; return FLOAT;}
void    {yylval.ival = VOID; return VOID;}
else    {return ELSE;}
do    {return DO;}
if    {return IF;}
for     {return FOR;}
struct     {return STRUCT;}
^"#include ".+    return PREPROC;
while    return WHILE;
return     return RETURN;
printf    return PRINT;
{alpha}({alpha}|{digit}|{und})* {yylval.str=strdup(yytext); return ID;}
[+-]?{digit}+    {yylval.str=strdup(yytext);return NUM;}
[+-]?{digit}+\.{digit}+     {yylval.str=strdup(yytext); return REAL;}
"<="            return LE;
">="           return GE;
"=="           return EQ;
"!="           return NE;
"++"           return INC;
"--"           return DEC;
\/\/.* ;
\/\*(.*\n)*.*\*\/ ;
\".*\"    return STRING;
.    return yytext[0];
%%
```

YACC file: parser.y

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "symbolTable.c"
int i=1,k=-1,k1=-1,l=-1,d=-1;
int j=0;
```

```
char curfunc[100];
int stack[100];
int top=0;
int plist[100],flist[100], par_list[100][20];
int end[100];
int arr[10];
int ct=0,c=0,b;
int loop = 0;
int errc=0;
int type=0;
extern int yylineno;
%}

%token<ival> INT FLOAT VOID
%token<str> ID NUM REAL
%token WHILE IF RETURN PREPROC LE STRING PRINT FUNCTION DO ARRAY ELSE
STRUCT STRUCT_VAR FOR GE EQ NE INC DEC
%right '='

%type<str> assignment assignment1 consttype assignment2
%type<ival> Type

%union
{
   int ival;
   char *str;
}

%%

start : Function start
   | PREPROC start
   | Declaration start
   |
   ;

Function : Type ID '('')' CompoundStmt {
   if ($1!=returntype_func(ct))
   {
        printf("\nError : Type mismatch : Line %d\n",printline()); errc++;
   }
```

```
        if (!(strcmp($2,"printf") && strcmp($2,"scanf") && strcmp($2,"getc") &&
strcmp($2,"gets") && strcmp($2,"getchar") && strcmp ($2,"puts") &&
strcmp($2,"putchar") && strcmp($2,"clearerr") && strcmp($2,"getw") &&
strcmp($2,"putw") && strcmp($2,"putc") && strcmp($2,"rewind") &&
strcmp($2,"sprint") && strcmp($2,"sscanf") && strcmp($2,"remove") &&
strcmp($2,"fflush")))
        {printf("Error : Redeclaration of %s : Line %d\n",$2,printline()); errc++;}
    else
    {
        insert($2,FUNCTION);
        insert($2,$1);
    }
    }
        | Type ID '(' parameter_list ')' CompoundStmt  {
    if ($1!=returntype_func(ct))
    {
        printf("\nError : Type mismatch : Line %d\n",printline()); errc++;
    }

        if (!(strcmp($2,"printf") && strcmp($2,"scanf") && strcmp($2,"getc") &&
strcmp($2,"gets") && strcmp($2,"getchar") && strcmp ($2,"puts") &&
strcmp($2,"putchar") && strcmp($2,"clearerr") && strcmp($2,"getw") &&
strcmp($2,"putw") && strcmp($2,"putc") && strcmp($2,"rewind") &&
strcmp($2,"sprint") && strcmp($2,"sscanf") && strcmp($2,"remove") &&
strcmp($2,"fflush")))
        {printf("Error : Redeclaration of %s : Line %d\n",$2,printline());errc++;}
    else
    {
        insert($2,FUNCTION);
        insert($2,$1);
                for(j=0;j<=k;j++)
                {insertp($2,plist[j],par_list[j]);}
                //insertpa($2,par_list[j]);}
                k=-1;k1=-1;
                /*for(j=0;j<=k1;j++)
                {insertpa($2,par_list[j]);}
                k1=-1;*/
    }
    }
    ;

parameter_list : parameter_list ',' parameter
                | parameter
```

```
            ;

parameter : Type ID {plist[++k]=$1; strcpy(par_list[++k1], $2);
insert($2,$1);insertscope($2,i);}
        ;

Type : INT
   | FLOAT
   | VOID
   ;

CompoundStmt : '{' StmtList '}'
   ;

StmtList : StmtList stmt
   | CompoundStmt
   |
   ;

stmt : Declaration
   | if
        | for
   | while
   | dowhile
   | RETURN consttype ';' {

                        if(!(strspn($2,"0123456789")==strlen($2)))
                                storereturn(ct,FLOAT);
                        else
                                storereturn(ct,INT);
                        ct++;
                    }
   | RETURN ';' {storereturn(ct,VOID); ct++;}
        | RETURN ID ';' {
                    int sct=returnscope($2,stack[top-1]);    //stack[top-1] - current
scope
            int type=returntype($2,sct);
                    if (type==259) storereturn(ct,FLOAT);
                    else storereturn(ct,INT);
                    ct++;
                    }
   | ';'
   | PRINT '(' STRING ')' ';'
```

```
    | CompoundStmt
    ;

dowhile : DO CompoundStmt WHILE '(' expr1 ')' ';'
    ;

if : IF '(' expr1 ')' CompoundStmt
    | IF '(' expr1 ')' CompoundStmt ELSE CompoundStmt
    ;

for : FOR '(' expr1 ';' expr1 ';' expr1 ')' '{' {loop=1;} StmtList {loop=0;} '}'
        ;

while : WHILE '(' expr1 ')' '{' {loop=1;} StmtList {loop=0;} '}'
    ;

expr1 : expr1 LE expr1
        | expr1 GE expr1
        | expr1 NE expr1
        | expr1 EQ expr1
        | expr1 INC
        | expr1 DEC
        | expr1 '>' expr1
        | expr1 '<' expr1
    | assignment1
    ;

assignment : ID '=' consttype
    | ID '+' assignment
    | ID ',' assignment
    | consttype ',' assignment
    | ID
    | consttype
    | '-'consttype
    ;

assignment1 : ID '=' assignment1
    {
        int sct=returnscope($1,stack[top-1]);
        int type=returntype($1,sct);
        if((!(strspn($3,"0123456789")==strlen($3))) && type==258)
                {printf("\nError : Type Mismatch : Line %d\n",printline()); errc++;}
```

```
            else if (type==273)    {printf("\nError : Type Mismatch : Line %d\n",printline());errc++;}
        if(!lookup($1))
        {
                int currscope=stack[top-1];
                int scope=returnscope($1,currscope);
                if((scope<=currscope && end[scope]==0) && !(scope==0))
                        check_scope_update($1,$3,currscope);
        }
    }

    | ID ',' assignment1
    {
        if(lookup($1))
                 printf("\nUndeclared Variable %s : Line %d\n",$1,printline()); errc++;
    }
    | assignment2
    | consttype ',' assignment1
    | ID
    {
        if(lookup($1))
                 { printf("\nUndeclared Variable %s : Line %d\n",$1,printline()); errc++; }
    }
    | ID '=' ID '(' paralist ')'                //function call
        {
                int sct=returnscope($1,stack[top-1]);
        int type=returntype($1,sct);
                //printf("%s",$3);
                int rtype;
                rtype=returntypef($3); int ch=0;
                //printf("%d",rtype);
        if(rtype!=type)
                 { printf("\nError : Type Mismatch : Line %d\n",printline()); errc++;}
        if(!lookup($1))
        {
          for(j=0;j<=l;j++)
                {ch = ch+checkp($3,flist[j],j);}
                if(ch>0) { printf("\nError : Parameter Type Mistake or Function undeclared : Line %d\n",printline()); errc++;}
                l=-1;
        }
```

```
   }
   | ID '(' paralist ')'                //function call without assignment
   {
            int sct=returnscope($1,stack[top-1]);
        int type=returntype($1,sct); int ch=0;
        if(!lookup($1))
        {
          for(j=0;j<=l;j++)
            {ch = ch+checkp($1,flist[j],j);}
            if(ch>0) { printf("\nError : Parameter Type Mistake or Required Function undeclared : Line %d\n",printline()); errc++;}
            l=-1;
        }
            else {printf("\nUndeclared Function %s : Line %d\n",$1,printline());errc++;}
   }
   | consttype
   ;

paralist : paralist ',' param
        | param
        ;

param : ID
   {
            if(lookup($1))
                {printf("\nUndeclared Variable %s : Line %d\n",$1,printline());errc++;}
            else
            {
                int sct=returnscope($1,stack[top-1]);
                flist[++l]=returntype($1,sct);
            }
   }
   ;

assignment2 : ID '=' exp {c=0;}
        | ID '=' '(' exp ')'
        ;

exp : ID
   {
```

```
        if(c==0)                                    //check compatibility of
mathematical operations
        {
                c=1;
                int sct=returnscope($1,stack[top-1]);
                b=returntype($1,sct);
        }
        else
        {
                int sct1=returnscope($1,stack[top-1]);
                if(b!=returntype($1,sct1))
                        {printf("\nError : Type Mismatch : Line
%d\n",printline());errc++;}
        }
   }
   | exp '+' exp
   | exp '-' exp
   | exp '*' exp
   | exp '/' exp
   | '(' exp '+' exp ')'
   | '(' exp '-' exp ')'
   | '(' exp '*' exp ')'
   | '(' exp '/' exp ')'
   | consttype
   ;

consttype : NUM
   | REAL
   ;

Declaration : Type ID '=' consttype ';'
   {
        if( (!(strspn($4,"0123456789")==strlen($4))) && $1==258)
                {printf("\nError : Type Mismatch : Line %d\n",printline());errc++;}
               else if ($1==273)    {printf("\nError : Type Mismatch : Line
%d\n",printline());errc++;}
        if(!lookup($2))
        {
                int currscope=stack[top-1];
                int previous_scope=returnscope($2,currscope);
                if(currscope==previous_scope)
                        {printf("\nError : Redeclaration of %s : Line
%d\n",$2,printline());errc++;}
```

```
                else
                {
                        insert_dup($2,$1,currscope);
                        check_scope_update($2,$4,stack[top-1]);
                }
        }
        else
        {
                int scope=stack[top-1];
                insert($2,$1);
                insertscope($2,scope);
                check_scope_update($2,$4,stack[top-1]);
        }
   }
   | assignment1 ';'
   {
        if(!lookup($1))
        {
                int currscope=stack[top-1];
                int scope=returnscope($1,currscope);
                int type=returntype($1,scope);
                  if(!(scope<=currscope && end[scope]==0) || scope==0 &&
type!=271)
                        {printf("\nError : Variable %s out of scope : Line
%d\n",$1,printline());errc++;}
        }
        else
                  {printf("\nError : Undeclared Variable %s : Line
%d\n",$1,printline());errc++;}
   }
        | Type ID ';'
        {
                if(!lookup($2))
        {
                int currscope=stack[top-1];
                int previous_scope=returnscope($2,currscope);
                if(currscope==previous_scope)
                        {printf("\nError : Redeclaration of %s : Line
%d\n",$2,printline());errc++;}
                else
                {
                        insert_dup($2,$1,currscope);
                }
```

```
        }
        else
        {
                int scope=stack[top-1];
                insert($2,$1);
                insertscope($2,scope);
        }
   }
   | Type ID '[' assignment ']' ';' {
                        int itype;
                        if(!(strspn($4,"-0123456789")==strlen($4))) { itype=259; }
else itype = 258;
                        if(itype!=258)
                        { printf("\nError : Array index must be of type int : Line
%d\n",printline());errc++;}
                        if(atoi($4)<=0)
                        { printf("\nError : Array index must be of type int > 0 : Line
%d\n",printline());errc++;}
                        if(!lookup($2))
                {
                 int currscope=stack[top-1];
                 int previous_scope=returnscope($2,currscope);
                 if(currscope==previous_scope)
                                {printf("\nError : Redeclaration of %s : Line
%d\n",$2,printline());errc++;}
                 else
                 {

                         insert_dup($2,ARRAY,currscope);
                                insert_by_scope($2,$1,currscope);      //to insert type to
the correct identifier in case of multiple entries of the identifier by using scope
                                if (itype==258) {insert_index($2,$4);}
                 }
                }
                else
                {
                 int scope=stack[top-1];
                        insert($2,ARRAY);
                 insert($2,$1);
                 insert_dim($2,1);
                 insertscope($2,scope);
                        if (itype==258) {insert_index($2,$4);}
                }
```

```
            }

   |Type ID '[' assignment ']' '[' assignment ']' ';'
   {
                    int itype;
                    if(!(strspn($4,"-0123456789")==strlen($4))) { itype=259; }
else itype = 258;
                    if(itype!=258)
                    { printf("\nError : Array index must be of type int : Line
%d\n",printline());errc++;}
                    if(atoi($4)<=0)
                    { printf("\nError : Array index must be of type int > 0 : Line
%d\n",printline());errc++;}

                    int itype2;
                    if(!(strspn($7,"-0123456789")==strlen($7))) { itype2=259; }
else itype2 = 258;
                    if(itype2!=258)
                    { printf("\nError : Array index must be of type int : Line
%d\n",printline());errc++;}
                    if(atoi($7)<=0)
                    { printf("\nError : Array index must be of type int > 0 : Line
%d\n",printline());errc++;}

                    if(!lookup($2))
               {
                int currscope=stack[top-1];
                int previous_scope=returnscope($2,currscope);
                if(currscope==previous_scope)
                        {printf("\nError : Redeclaration of %s : Line
%d\n",$2,printline());errc++;}
                else
                {

                 insert_dup($2,ARRAY,currscope);
                        insert_by_scope($2,$1,currscope);     //to insert type to
the correct identifier in case of multiple entries of the identifier by using scope
                        if (itype==258) {insert_index($2,$4);}
                }
               }
               else
               {
```

```
                int scope=stack[top-1];
                        insert($2,ARRAY);
                insert($2,$1);
                insert_dim($2,2);
                insertscope($2,scope);
                        if (itype==258) {insert_index($2,$4);}
                }

    }

    | STRUCT ID '{' Declaration '}' ';' {
                                        insert($2,STRUCT);
                                        }
    | STRUCT ID ID ';' {
                        insert($3,STRUCT_VAR);
                        }
    | error
    ;



%%

#include "lex.yy.c"
#include<ctype.h>
int main(int argc, char *argv[])
{
    yyin =fopen(argv[1],"r");
    if(!yyparse()&& errc<=0)
    {
        printf("\nParsing Completed\n");
        display();
    }
    else
    {
        printf("\nParsing Failed\n");
                display();
    }
    fclose(yyin);
    return 0;
}

yyerror(char *s)
```

```
{
   printf("\nLine %d : %s %s\n",yylineno,s,yytext);
}

int printline()
{
   return yylineno;
}
void push()
{
   stack[top]=i;
   i++;
   top++;
   return;
}
void pop()
{
   top--;
   end[stack[top]]=1;
   stack[top]=0;
   return;
}
```

Symbol table file: symbolTable.c

```
#include<stdio.h>
#include<string.h>
struct sym
{
   int sno;
   char token[100];
   int type[100];
   int paratype[100];
   char paralist[100][20];
   int tn;
   int dim;
   int pn;
   float fvalue;
   int index;
   int scope;
```

```
}st[100];
int n=0,arr[10];
int tnp;

int returntype_func(int ct)
{
   return arr[ct-1];
}
void storereturn( int ct, int returntype )
{
   arr[ct] = returntype;
   return;
}
void insertscope(char *a,int s)
{
   int i;
   for(i=0;i<n;i++)
   {
        if(!strcmp(a,st[i].token))
        {
                st[i].scope=s;
                break;
        }
   }
}
int returnscope(char *a,int cs)
{
   int i;
   int max = 0;
   for(i=0;i<=n;i++)
   {
        if(!strcmp(a,st[i].token) && cs>=st[i].scope)
        {
                if(st[i].scope>=max)
                        max = st[i].scope;
        }
   }
   return max;
}
int lookup(char *a)
{
   int i;
   for(i=0;i<n;i++)
```

```
    {
        if( !strcmp( a, st[i].token) )
            return 0;
    }
    return 1;
}
int returntype(char *a,int sct)
{
    int i;
    for(i=0;i<n;i++)
    {
        if(!strcmp(a,st[i].token) && st[i].scope==sct)
            return st[i].type[0];
    }
}

int returntype2(char *a,int sct)
{
    int i;
    for(i=0;i<n;i++)
    {
        if(!strcmp(a,st[i].token) && st[i].scope==sct)
            { return st[i].type[1];}
    }
}

int returntypef(char *a)
{
    int i;
    for(i=0;i<n;i++)
    {
        if(!strcmp(a,st[i].token))
            { return st[i].type[1];}
    }
}

void insert_dim(char *name, int d)
{
    int i;
    for(i=0;i<n;i++)
    {
        if(!strcmp(name,st[i].token))
            { st[i].dim = d; }
```

```
    }
}

void check_scope_update(char *a,char *b,int sc)
{
    int i,j,k;
    int max=0;
    for(i=0;i<=n;i++)
    {
        if(!strcmp(a,st[i].token)   && sc>=st[i].scope)
        {
            if(st[i].scope>=max)
                max=st[i].scope;
        }
    }
    for(i=0;i<=n;i++)
    {
        if(!strcmp(a,st[i].token)   && max==st[i].scope)
        {
            float temp=atof(b);
            for(k=0;k<st[i].tn;k++)
            {
                if(st[i].type[k]==258)
                    st[i].fvalue=(int)temp;
                else
                    st[i].fvalue=temp;
            }
        }
    }
}
void storevalue(char *a,char *b,int s_c)
{
    int i;
    for(i=0;i<=n;i++)
    {
        if(!strcmp(a,st[i].token) && s_c==st[i].scope)
        {
            st[i].fvalue=atof(b);
        }
    }
}

void insert(char *name, int type)
```

```
{
   int i;
   if(lookup(name))
   {
        strcpy(st[n].token,name);
        st[n].tn=1;
              st[n].pn=0;
        st[n].type[st[n].tn-1]=type;
        st[n].sno=n+1;
        n++;
   }
   else
   {
        for(i=0;i<n;i++)
        {
              if(!strcmp(name,st[i].token))
              {
                    st[i].tn++;
                    st[i].type[st[i].tn-1]=type;
                    break;
              }
        }
   }

   return;
}

void insert_dup(char *name, int type, int s_c)
{
   strcpy(st[n].token,name);
   st[n].tn=1;
       st[n].pn=0;
   st[n].type[st[n].tn-1]=type;
   st[n].sno=n+1;
   st[n].scope=s_c;
   n++;
   return;
}

void insert_by_scope(char *name, int type, int s_c)
{
        int i;
   for(i=0;i<n;i++)
```

```
        {
        if(!strcmp(name,st[i].token) && st[i].scope==s_c)
        {
                        st[i].tn++;
                        st[i].type[st[i].tn-1]=type;
        }
        }
}

void insertp(char *name,int type, char *id)
{
        int i;
        for(i=0;i<n;i++)
        {
        if(!strcmp(name,st[i].token))
        {
                        st[i].pn++;
                        st[i].paratype[st[i].pn-1]=type;
                        strcpy(st[i].paralist[st[i].pn-1], id);
                        break;
        }
        }
}

void insert_index(char *name,int ind)
{
        int i;
        for(i=0;i<n;i++)
        {
        if(!strcmp(name,st[i].token) && st[i].type[0]==273)
        {
                        st[i].index = atoi(ind);
        }
   }
}

int checkp(char *name,int flist,int c)
{
        int i,j;
        for(i=0;i<n;i++)
        {
        if(!strcmp(name,st[i].token))
        {
```

```
                        if(st[i].paratype[c]!=flist)
                        return 1;
        }
        }
        return 0;
}

void display()
{
   int i,j;
   printf("\n");
                  printf("-------------------------------------------------------------Symbol
Table---------------------------------------------------------\n");

printf("-----------------------------------------------------------------------------------------
-----------------------------------------------\n");
                  printf("\nSNo\tIdentifier\tScope\t\tValue\t\tType\t\t\tDim\tParameter
type\t\tParameter list\n");

printf("-----------------------------------------------------------------------------------------
-----------------------------------------------\n\n");
   for(i=0;i<n;i++)
   {
        if(st[i].type[0]==258 || st[i].type[1]==258 || st[i].type[1]==260)

printf("%d\t%s\t\t%d\t\t%d\t",st[i].sno,st[i].token,st[i].scope,(int)st[i].fvalue,st[i]
.dim);
        else

printf("%d\t%s\t\t%d\t\t%.2f\t",st[i].sno,st[i].token,st[i].scope,st[i].fvalue,st[i].di
m);
                printf("\t");
        for(j=0;j<st[i].tn;j++)
        {
                if(st[i].type[j]==258)
                        printf("INT");
                else if(st[i].type[j]==259)
                        printf("FLOAT");
                else if(st[i].type[j]==271)
                        printf("FUNCTION");
                else if(st[i].type[j]==273)
                        printf("ARRAY");
                else if(st[i].type[j]==260)
```

```
                printf("VOID");
                if(st[i].tn>1 && j<(st[i].tn-1))printf(" - ");
        }
                printf("\t\t");
        for(j=0;j<st[i].pn;j++)
        {
                if(st[i].paratype[j]==258)
                        printf("INT");
                else if(st[i].paratype[j]==259)
                        printf("FLOAT");
                if(st[i].pn>1 && j<(st[i].pn-1))printf(", ");
        }
        printf("\t\t\t");
        for(j=0;j<st[i].pn;j++)
        {
                printf("%s, ",st[i].paralist[j]);
        }
        printf("\n");
   }

printf("-------------------------------------------------------------------------------------
-----------------------------------------------\n\n");
   return;
}
-------------------------------------------------------------------------------------------
```

IMPLEMENTATION

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:

1. Serial No - represents the number of entries in the symbol table
2. Identifier - specifies the name of the variables which are identifiers
3. Scope - specifies the scope of the variables
4. Value - represents the mathematical value of a variable if initialized/defined
5. Type - specifies the data type of the variable
6. Dimension - specifies the dimension if the identifier is an array
7. Parameter type - specifies the data types of function arguments/parameters
8. Parameter list - specifies the names of function parameters

The semantic analyzer built for the subset of C language reports various errors present (if any) and has the following functionalities:

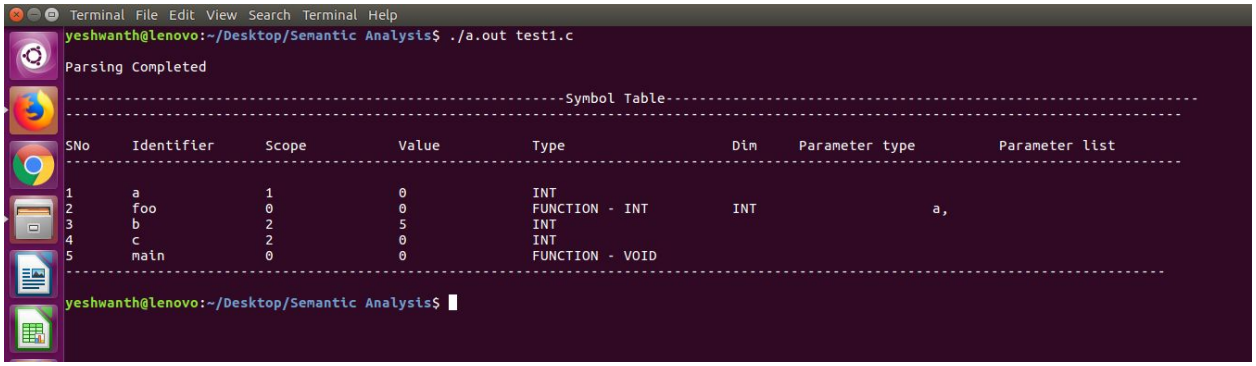
1. Checks for undeclared variables
2. Checks for redeclaration of variables
3. Type checking of variables
4. Mathematical operations compatibility
5. Checks if the return type of a function matches with the datatype of the variable/constant being returned
6. Checks if the number of parameters in a function call matches with the number of parameters in the function definition
7. Checks if datatypes of parameters in function call and its definition match
8. Checks for scope of variables and reports an error if a variable is out of scope
9. Identifies the dimensionality of arrays and checks if index of an array is a positive integer or not

TEST CASES

Test case 1 (no error)

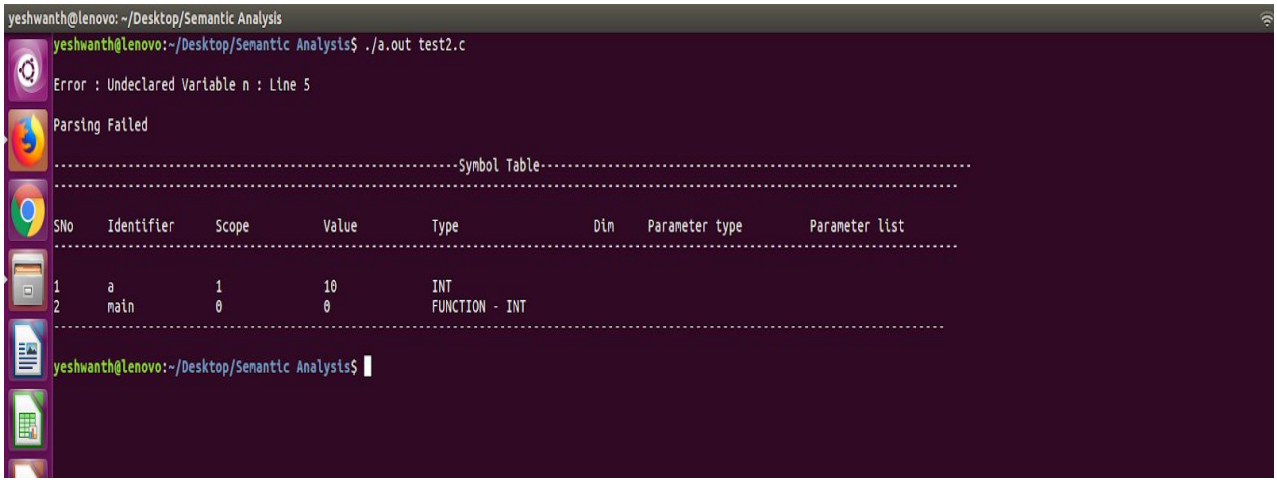
```
#include <stdio.h>
int foo(int a)
{
   a = a + 1;
   return a;
}

void main()
{
   int b;
   b=5;
   int c;
   c=foo(b);
   return;
}
```



Test case 2

```
#include <stdio.h>
int main()
{
   int a=4;
   n=9;        //undeclared variable
   a=10;
   return 0;
}
```

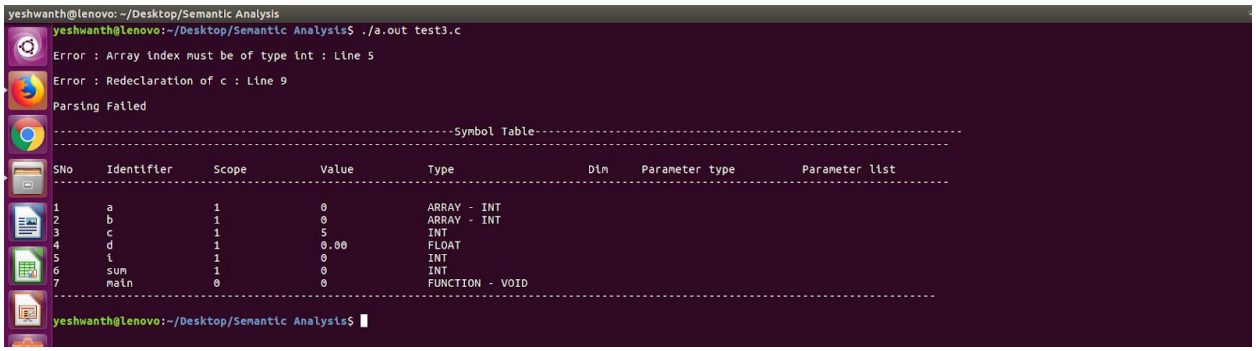


Test case 3

```
#include <stdio.h>
void main()
{
   int a[5][3];    //array index error (has to be greater than zero)
   int b[8.5];    //array index cannot be a float value
   int c=5;
   float d;
```

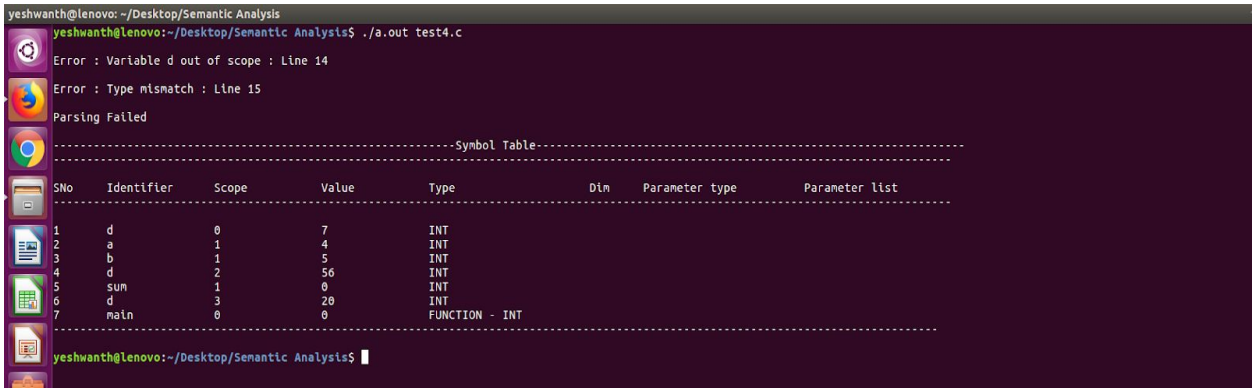
```
        int i;
        int c=3;    //redeclaration of variable 'c'
        int sum;
        sum=0;

    for(i=0;i<12;i++)
    {
         sum=sum+i;
    }
    return;
}
```



Test case 4

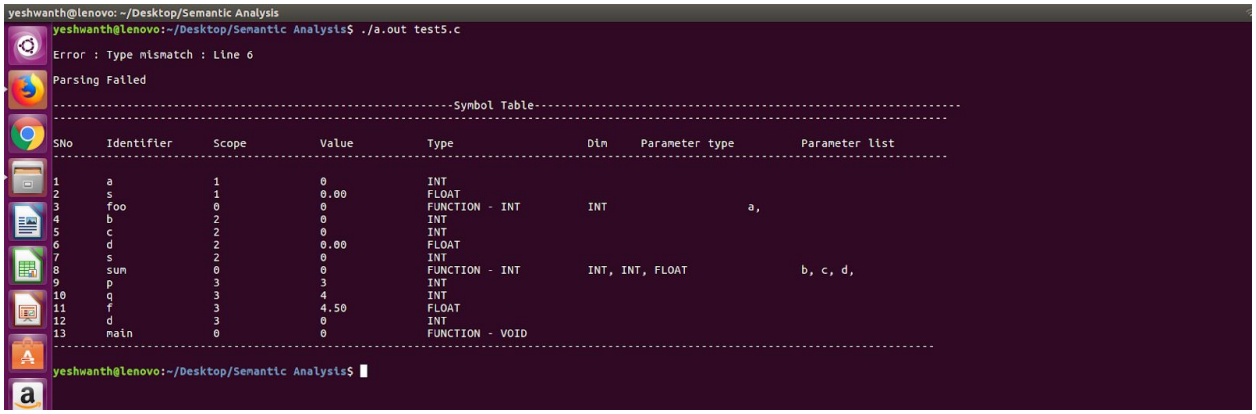
```
#include <stdio.h>
int d = 7;
int main()
{
    int a=4;
    int b=5;
    {
         int d=56;
    }
    int sum;
    {
         int d=20;
    }
    d=89;       //variable 'd' out of scope
}
```



Test case 5

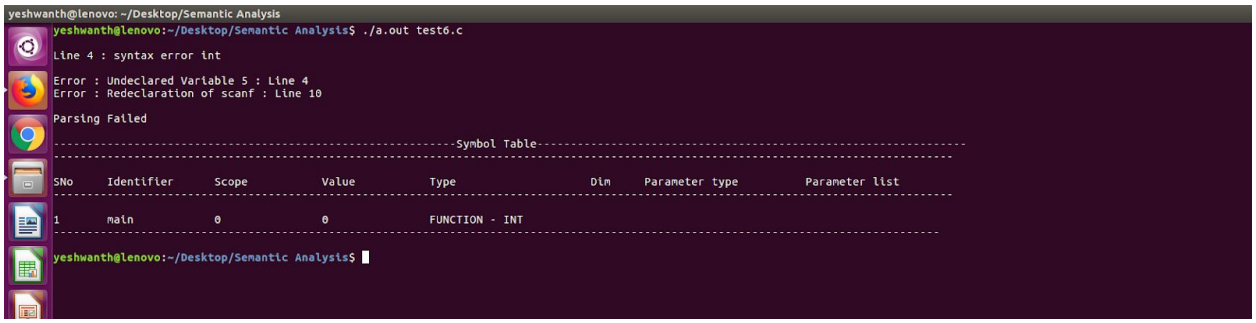
```
#include <stdio.h>
int foo(int a)
{
   float s;
   return s;          //return type mismatch
}
int sum(int b,int c,float d)
{
   int s;
   s=b+c;
   return s;
}

void main()
{
   int p=3;
   int q=4;
   float f=4.5;
   int d;
   d=sum(p,p,f);
   return;
}
```



Test case 6

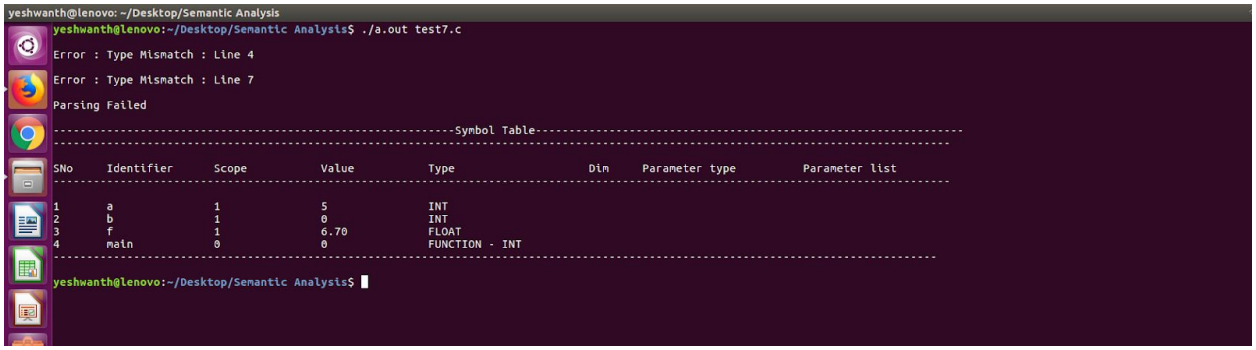
```
#include <stdio.h>
int main()
{
   int int=5;
   return 0;
}
void scanf()          //error - standard function redeclared
{
   return;
}
```



Test case 7

```
#include <stdio.h>
int main()
{
   int a = 5.4;    //type mismatch
   int b=67;
   float f=6.7;
   b=f;         //type mismatch (int = float)
```

```
    return 0;
}
```



RESULTS

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

lex parser.l
yacc parser.y
cc y.tab.h -ll
./a.out test1.c

After parsing, if there are errors then the line numbers of those errors are displayed along with a 'parsing failed' on the terminal. Otherwise, a 'parsing complete' message is displayed on the console. The symbol table with stored & updated values is always displayed, irrespective of errors.

CONCLUSION

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

FUTURE WORK

We have implemented the parser and semantic analyzer for only a subset of C language. The future work may include defining the grammar and specifying the

semantics for switch statements, predefined functions (like string functions, file read and write functions), jump statements and enumerations.

REFERENCES

1. http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf - Lex and Yacc Tutorial by Tom Nieman
2. Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
3. https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm - Compiler Design Semantic Analysis by TutorialsPoint