COMPILER DESIGN LAB (CO351)

REPORT

Project 3 : Semantic Analyzer for C-language

Submitted by :-

(1) Patel Harsh Rakeshkumar (15CO233)

(2) Shah Harsh Kamlesh (15CO243)

INDEX

ABSTRACT

The semantic analyzer will have following functionalities :-

- We are using YACC to build semantic analyzer upon the CFG defined in YACC file for the parser.
- We include semantic rules for the grammar production to do the semantic analysis.
- Input file will be a C code and will not give any error in output if it follows the semantic rules.
- The semantic analyzer makes the following changes to the symbol table:-
    - It assigns a type to the identifiers.
    - In includes nesting level for the variables.
    - It deletes variables from the symbol table at the end of the scope for the variable.
    - It has a flag to check for a function definition.
    - It stores the number of arguments and return type of the function.
    - It also stores the size of the variables based on their type.
- The semantic analyzer checks for the following errors:-
    - Redeclaration a variable
    - Undeclared variable
    - Accessing an out of scope variable
    - Type mismatch for expressions
    - Integer expression for while loop
    - Function structure mismatch
    - Function argument type mismatch
    - Return type mismatch
- The semantic analyzer checks for complex expressions and handle normal declarations and definitions.
- The semantic analyzer defines rules for do-while, while and for loops. It also gives rules for if-else ladder and switch-case statements.
- The semantic analyzer can analyze function definitions and function calls.

SEMANTIC ANALYZER

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

CFG + semantic rules = Syntax Directed Definitions

The following tasks should be performed in semantic analysis:

- Scope resolution
- Type checking

Semantic Errors :-

Following is a list of general semantic errors to be handled by a semantic analyzer :-

- 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.

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 information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

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

For example :-
Take the grammar production below :-
E → E + T { E.value = E.value + T.value }

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.
Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : **synthesized** attributes and **inherited** attributes.

Synthesized attributes

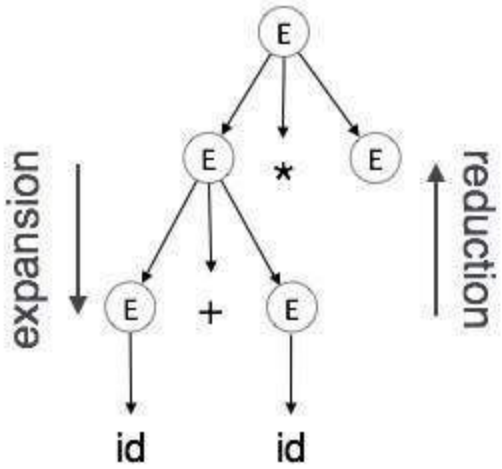
These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:
S → ABC
If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.
As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,
S → ABC
A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Expansion : When a non-terminal is expanded to terminals as per a grammatical rule

Reduction : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction



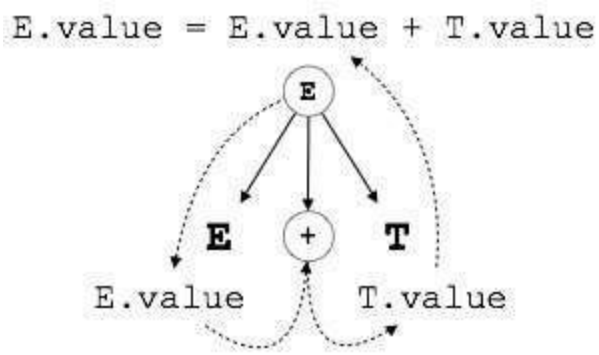
occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses **Syntax Directed Translations** to perform the above tasks. Semantic analyzer receives AST (**Abstract Syntax Tree**) from its previous stage (syntax analysis).

S and L Attributed SDT

S-Attributed SDT

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).

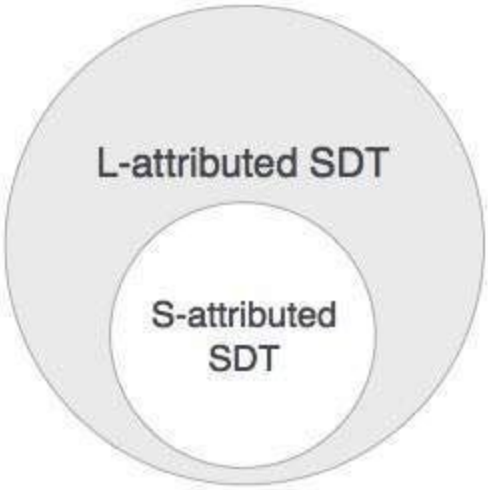


As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-Attributed SDT

This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.
In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production
S → ABC
S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

COMPILING INSTRUCTIONS

To compile the code goto the semantic directory and use the following command:

yacc -d parser.yacc && lex lexical.l && cc y.tab.c lex.yy.c && ./a.out

CODE

The following code block contains the yacc file code which has the CFG and the semantic rules are defined for the production rules.

```
%token AUTO BREAK CASE CHAR CONTINUE DEFAULT DO DOUBLE ELSE ENUM EXTERN FLOAT FOR GOTO IF
INT LONG REGISTER RETURN SHORT SIGNED SIZEOF STATIC STRUCT SWITCH TYPEDEF UNION UNSIGNED
VOID VOLATILE WHILE
%token IDENTIFIER STRING_LITERAL ASSIGN_OP REL_OP ADD_OP MUL_OP INCDEC_OP EQU_OP  LAND LOR
BAND BXOR BOR NOT_OP
%token INTVALUE FLOATVALUE
%expect 2

%{
```

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <limits.h>

extern FILE *yyin;
extern int lineno;
extern int comment_nesting;
int isfunction=0;
int scope=0;

int flag = 0;
int errorFlag=0;
struct stable{
    char name[100];
    char type[50];
    long long int scope;
        int fundef;
        int countargs;
        char argslist[20];
        int storage;
};
struct ctable{
    char name[100];
    char type[50];
};
extern struct stable symbol_table[100000];
extern struct ctable constant_table[100000];
extern char yyval[100];
extern char yycons[100];
extern char yystr[100];
extern char currtype[100],prevtype[100],currid[100],previd[100];
char fnname[100];
char fnlist[20];
int fncount=0;
char fncname[100];
char fnclist[20];
int fnccount=0;
char* getcurrid();
char* getcurrtype();
char* getprevid();
char* getprevtype();
char gettype(char *name);


    void insert_symbol_table()
    {
```

```
       int k=hash_cal(yyval);
               struct stable temp;
               strcpy(temp.name,yyval);
              temp.countargs=0;
              char* tmp = getcurrtype();
              strcpy(temp.type,tmp);
              if(tmp[0]=='i')
              {
               temp.storage=4;
              }
              else if(tmp[0]=='f')
              {
               temp.storage=8;
              }
              else
              {
               temp.storage=0;
              }
               if(isfunction==1)
              {
              temp.fundef=1;
              }
              else
              {
              temp.fundef=0;
              }
               int tempscope=scope;
               temp.scope=tempscope;
               int i=0;
               int flag=0;
               for(i=0;i<1000;i++)
               {
                      if(symbol_table[k].name[0]=='\0')
                      {
                             break;
                      }
                      else
                      {

                             if(strcmp(yyval,symbol_table[k].name)==0 &&
temp.scope==symbol_table[k].scope)
                             {
                                    flag=1;
                                    printf("Error:redeclaration\n");
                                    break;
                             }
                             k=(k+1)%1000;
                      }
               }
```

```
            if(flag==0)
            {
                   symbol_table[k]=temp;
            }
}
    int ck_symbol_table()
{
    int k=hash_cal(yyval);
            struct stable temp;
            strcpy(temp.name,yyval);
            strcpy(temp.type,"identifier");

            int tempscope=scope;
            temp.scope=tempscope;
            int i=0;
            int flag=0;

            for(i=0;i<1000;i++)
            {
                   if(strcmp(yyval,symbol_table[i].name)==0)
           {

                  return(1);
           }
            }
           printf("%s -->",yyval);
           return(0);

}

    int sym_update_fundef()
{

            int i=0;
            int flag=0;

            for(i=0;i<1000;i++)
            {
                   if(strcmp(fnname,symbol_table[i].name)==0&&symbol_table[i].fundef==1)
           {
                  symbol_table[i].countargs=fncount;
                  strcpy(symbol_table[i].argslist,fnlist);
                  return(1);
           }
            }
           return(0);


}
```

```
        int sym_ck_funcall()
    {
        // printf("Checking %s %s ",fncname,fnclist);
                int i=0;
                int flag=0;
                for(i=0;i<1000;i++)
                {

if(strcmp(fncname,symbol_table[i].name)==0&&symbol_table[i].fundef==1&&strcmp(fnclist,symbol
_table[i].argslist)==0)
               {
                       return(1);
               }
                }
               return(0);


    }
        void copyfnname()
    {
        int k=hash_cal(yyval);
                struct stable temp;
                strcpy(fnname,yyval);

    }

        void copyfncname()
    {
        int k=hash_cal(yyval);
                struct stable temp;
                strcpy(fncname,yyval);

    }
    void insert_constant_table()
    {
        int k=hash_cal(yycons);
         struct ctable temp;
         strcpy(temp.name,yycons);
         strcpy(temp.type,"numeric constant");
         int i=0;
         int flag=0;
         for(i=0;i<1000;i++)
         {
                if(constant_table[k].name[0]=='\0')
                {
                       break;
                }
                else
                {
```

```
                    if(strcmp(yycons,constant_table[k].name)==0)
                    {
                            flag=1;
                            break;
                    }
                    k=(k+1)%1000;
            }
     }
     if(flag==0)
     {
            constant_table[k]=temp;
     }
}
void insert_constant_table_str()
{
    int k=hash_cal(yystr);
     struct ctable temp;
     strcpy(temp.name,yystr);
     strcpy(temp.type,"string constant");
     int i=0;
     int flag=0;
     for(i=0;i<1000;i++)
     {
            if(constant_table[k].name[0]=='\0')
            {
                    break;
            }
            else
            {
                    if(strcmp(yystr,constant_table[k].name)==0)
                    {
                            flag=1;
                            break;
                    }
                    k=(k+1)%1000;
            }
     }
     if(flag==0)
     {
            constant_table[k]=temp;
     }
}

    void delete_sym(int n)
    {
    int i;
    for(i=0;i<1000;i++)
    {
           if(symbol_table[i].name[0]!='\0'&&symbol_table[i].scope==n)
```

```
                {
                printf("ENTRY DELETED::%s   %s  %d
\n",symbol_table[i].name,symbol_table[i].type,symbol_table[i].scope);
                symbol_table[i].name[0]='\0';
                symbol_table[i].type[0]!='\0';

                }
        }
        }

%}

%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%right ASSIGN_OP '='
%left LOR
%left LAND
%left BOR
%left BXOR
%left BAND
%left EQU_OP
%left REL_OP
%left ADD_OP
%left MUL_OP
%right NOT_OP
%left INCDEC_OP

%start PROGRAM

%%

PROGRAM
        :EXTERNAL_DECLARATION
        |PROGRAM EXTERNAL_DECLARATION
        ;

EXTERNAL_DECLARATION
        :FUNCTION_DEFINITION
        |DECLARATION
        ;

DECLARATION
        : DECLARATION_SPECIFIER INIT_DECLARATOR_LIST ';'
        ;

DECLARATION_SPECIFIER
        : TYPE_SPECIFIER
        ;
```

```
TYPE_SPECIFIER
        : VOID
        | INT
        | FLOAT
        ;

INIT_DECLARATOR_LIST
        : INIT_DECLARATOR
        | INIT_DECLARATOR_LIST ',' INIT_DECLARATOR
        ;

INIT_DECLARATOR
        : DECLARATOR
        | DECLARATOR '=' EXPRESSION {if($3==-1) printf("Invalid Expression\n");}
        ;

DECLARATOR
        : DECID
        | '(' DECLARATOR ')'
        ;

FUNCTION_DEFINITION
        : DECLARATION_SPECIFIER FUNDECID FND_OPN  FND_CLS COMPOUND_STATEMENT
{sym_update_fundef();fncount=0;fnlist[0]='\0';}
        | DECLARATION_SPECIFIER FUNDECID FND_OPN DEFINITION_SPECIFIER_LIST FND_CLS
COMPOUND_STATEMENT   {sym_update_fundef();fncount=0;fnlist[0]='\0';}
        ;

FND_OPN
        : '(' {scope++;}
        ;

FND_CLS
        : ')' {scope--;}
        ;

DEFINITION_SPECIFIER_LIST
        :VARIABLE_DECLARATION
{fnlist[fncount++]=getcurrtype()[0];fnlist[fncount]='\0';if(getcurrtype()[0]=='v'){printf("E
RROR:function argument cannot be void.\n");}}
        | DEFINITION_SPECIFIER_LIST ',' VARIABLE_DECLARATION
{fnlist[fncount++]=getcurrtype()[0];fnlist[fncount]='\0';if(getcurrtype()[0]=='v'){printf("E
RROR:function argument cannot be void.\n");}}
        ;
VARIABLE_DECLARATION
        :DECLARATION_SPECIFIER IDENTIFIER {insert_symbol_table();}
        ;
COMPOUND_STATEMENT
```

```
        :OP_BRACE  STATEMENT_LIST CL_BRACE
        |OP_BRACE CL_BRACE
        ;

STATEMENT_LIST
        :STATEMENT_LIST STATEMENT
        |STATEMENT
        ;



STATEMENT
        : LABELED_STATEMENT
        | COMPOUND_STATEMENT
        | EXPRESSION_STATEMENT
        | ITERATION_STATEMENT
        | SELECTION_STATEMENT
        | JUMP_STATEMENT
        | DECLARATION
        | error ';'
        ;



LABELED_STATEMENT
        : CASE EXPRESSION ':' STATEMENT {if($2==-1) printf("Invalid Expression\n");}
        | DEFAULT ':' STATEMENT
        ;

EXPRESSION_STATEMENT
        : ';'
        | EXPRESSION ';' {if($1==-1) printf("Invalid Expression\n");}
        ;

EXPRESSION
        : EXPID ASSIGN_OP EXPRESSION { if($3==1 && $1==$3)$$=1; else if($3==2 && $1==$3)$$=2;
else $$=-1;}
        | EXPID '=' EXPRESSION {if($3==1 && $1==$3)$$=1; else if($3==2 && $1==$3)$$=2; else
$$=-1;}
        | EXPRESSION EQU_OP EXPRESSION {if($1==1 && $3==1)$$=1; else if($1==2 && $3==2)$$=2;
else $$=-1;}
        | EXPRESSION REL_OP EXPRESSION {if($1==1 && $3==1)$$=1; else if($1==2 && $3==2)$$=2;
else $$=-1;}
        | EXPRESSION ADD_OP EXPRESSION {if($1==1 && $3==1)$$=1; else if($1==2 && $3==2)$$=2;
else $$=-1;}
        | EXPRESSION MUL_OP EXPRESSION {if($1==1 && $3==1)$$=1; else if($1==2 && $3==2)$$=2;
else $$=-1;}
        | EXPRESSION LAND EXPRESSION {if($1==1 && $3==1)$$=1; else if($1==2 && $3==2)$$=2;
else $$=-1;}
```

```
       | EXPRESSION LOR EXPRESSION {if($1==1 && $3==1)$$=1; else if($1==2 && $3==2)$$=2;
else $$=-1;}
       | EXPRESSION BOR EXPRESSION {if($1==1 && $3==1)$$=1; else if($1==2 && $3==2)$$=2;
else $$=-1;}
       | EXPRESSION BAND EXPRESSION {if($1==1 && $3==1)$$=1; else if($1==2 && $3==2)$$=2;
else $$=-1;}
       | EXPRESSION BXOR EXPRESSION {if($1==1 && $3==1)$$=1; else if($1==2 && $3==2)$$=2;
else $$=-1;}
       | NOT_OP EXPRESSION {if($2==1)$$=1; else if($2==2)$$=2; else $$=-1;}
       | INCDEC_OP EXPID {$$=$2;}
       | EXPID INCDEC_OP {$$=$1;}
       | '(' EXPRESSION ')' {$$=$2;}
       | EXPID  {$$=$1;}
       | INTVALUE {insert_constant_table(); $$=1;}
       | FLOATVALUE {insert_constant_table(); $$=2;}
       | FUNCTION_CALL {$$ = $1; if($1==-1)$$=3;}
       | STRING_LITERAL {insert_constant_table_str(); $$=-1;}
       ;

FUNCTION_CALL
       : EXPID FNC_OPN  FUNCTION_CALL_CONTD
{$$=$1;if(!sym_ck_funcall()){printf("Error:function defination does not match any existing
function defination\n");}fnccount=0;fnclist[0]='\0';}
       ;
FNC_OPN
       : '('   {copyfncname();}
       ;
FUNCTION_CALL_CONTD
       : ')'
       |  EXPRESSION_LIST ')'
       ;
EXPRESSION_LIST
       : EXPRESSION_LIST ',' EXPRESSION {if($3==-1) printf("Invalid Expression\n");else
if($3==1){fnclist[fnccount++]='i';fnclist[fnccount]='\0';}else
if($3==2){fnclist[fnccount++]='f';fnclist[fnccount]='\0';}}
       | EXPRESSION {if($1==-1) printf("Invalid Expression\n");else
if($1==1){fnclist[fnccount++]='i';fnclist[fnccount]='\0';}else
if($1==2){fnclist[fnccount++]='f';fnclist[fnccount]='\0';}}
       ;



SELECTION_STATEMENT
       : IF '(' EXPRESSION ')' STATEMENT  %prec LOWER_THAN_ELSE {if($3==-1) printf("Invalid
Expression\n");}
       | IF '(' EXPRESSION ')' STATEMENT ELSE STATEMENT {if($3==-1) printf("Invalid
Expression\n");}
       | SWITCH '(' EXPRESSION ')' STATEMENT {if($3==-1) printf("Invalid Expression\n");}
       ;
```

```
ITERATION_STATEMENT
       : WHILE '(' EXPRESSION ')' STATEMENT { if($3==-1) printf("Invalid Expression\n");else
if($3!=1){printf("While should have a interger expression\n");}}
       | DO STATEMENT WHILE '(' EXPRESSION ')' ';' {if($5==-1) printf("Invalid
Expression\n");}
       | FOR '(' EXPRESSION_STATEMENT EXPRESSION_STATEMENT ')' STATEMENT
       | FOR '(' EXPRESSION_STATEMENT EXPRESSION_STATEMENT EXPRESSION ')' STATEMENT
{if($5==-1) printf("Invalid Expression\n");}
       ;

JUMP_STATEMENT
       : CONTINUE ';'
       | BREAK ';'
       | RETURN ';'   {if(gettype(fnname)!='v'){printf("Error Return type wrong.\n");}}
       | RETURN EXPRESSION {if($2==-1) printf("Invalid Expression\n");else if(($2==1&&
gettype(fnname)!='i') || ($2==2 && gettype(fnname)!='f') ){printf("Error Return type
wrong.\n");}}
       ;

OP_BRACE
    :'{' {scope++;}
    ;

CL_BRACE
    :'}' {delete_sym(scope);scope--;}
    ;


DECID
    :IDENTIFIER {insert_symbol_table();}
    ;

FUNDECID
    :IDENTIFIER {isfunction=1;copyfnname();insert_symbol_table();isfunction=0;}
    ;
EXPID
    :IDENTIFIER {

                if(gettype(getcurrid())=='i')
                       $$ = 1;
                else if(gettype(getcurrid())=='f')
                       $$=2;
                else
                       $$=-1;

                if(!ck_symbol_table())
```

```
                {
                        printf("ERROR:undeclared variable\n");
                }
                }
    ;




%%

int yyerror()
{
        flag = 1;
        printf("PARSING ERROR at Line Number - %d\n",lineno);
        return (1);
}

int main()
{


        yyin=fopen("test.txt","r");


        yyparse();

        if(comment_nesting!=0)
        printf("LEXICAL ERROR : Unterminated Comment\n");

        printf("\n*****Symbol Table******\n\n");
        int i;
    for(i=0;i<1000;i++)
    {
         if(symbol_table[i].name[0]!='\0')
         {
                printf("Var:%s Type:%s Size:%d Scope:%d  Isfunction:%d
args:%d\n",symbol_table[i].name,symbol_table[i].type,symbol_table[i].storage,symbol_table[i]
.scope,symbol_table[i].fundef,symbol_table[i].countargs);
         }
    }


    printf("\n*****Constant Table******\n\n");
    for(i=0;i<1000;i++)
    {
         if(constant_table[i].name[0]!='\0')
         {
                printf("%s    %s\n",constant_table[i].name,constant_table[i].type);
```

```
        }
    }

        if(!flag)
        {
        printf("\nParsing SUCCESSFUL.\n");
        }
}

char* getcurrid()
{
        return currid;
}

char* getcurrtype()
{
        return currtype;
}

char* getprevid()
{
        return previd;
}

char* getprevtype()
{
        return prevtype;
}

char gettype(char *name)
{
        int i;
        char t;
        int scp=-1;
    for (i=0;i<1001;i++){
        if(strcmp(symbol_table[i].name,name) == 0 && symbol_table[i].scope >scp){
                t =  (symbol_table[i].type[0]);
               scp=symbol_table[i].scope;
        }
    }
        return t;
}
```

TEST CASES

1) **Main test case**

INPUT

```
float fun(float x, int y)
{
      float w;

      return x;
}
int z;
void v( )
{
    return ;
}
int small()
{
      //w=5;
}
int main()
{
      int a,b;
      a=5;

    int i;
      for(i=0;;i++)
      {
    int a;
      while(a>0)
      {
            b++;
      }
      }
      v();
      {
      float a;
```

```
            1.0+fun(a,2);
        }
}
```

OUTPUT

```
ENTRY DELETED::w   float  1
ENTRY DELETED::x   float  1
ENTRY DELETED::y   int  1
ENTRY DELETED::a   int  2
ENTRY DELETED::a   float  2
ENTRY DELETED::a   int  1
ENTRY DELETED::b   int  1
ENTRY DELETED::i   int  1

*****Symbol Table******

Name        Type        Size        Scope       IsFunction    NoOfArgs
v      void        0     0     1     0
z      int         4     0     0     0
fun          float       8     0     1     2
main         int         4     0     1     0
small        int         4     0     1     0

*****Constant Table******

0     numeric constant
2     numeric constant
5     numeric constant
1.0     numeric constant

Parsing SUCCESSFUL.
```

2) Variable redeclaration

INPUT

```
int main()
```

```
{

    int a;
    a=5;
    int a;
}
```

OUTPUT

```
Error:redeclaration
ENTRY DELETED::a   int  1

*****Symbol Table******

Name        Type       Size       Scope      IsFunction    NoOfArgs
main        int        4     0    1     0

*****Constant Table******

5    numeric constant

Parsing SUCCESSFUL.
```

3) Undeclared variable

INPUT

```
int main()
{
    a=5;
    int a;
}
```

OUTPUT

```
a -->ERROR:undeclared variable
Invalid Expression
ENTRY DELETED::a   int  1
```

```
*****Symbol Table******

Name        Type        Size        Scope       IsFunction    NoOfArgs
main        int         4     0     1     0

*****Constant Table******

5     numeric constant

Parsing SUCCESSFUL.
```

4) Variable declared out of scope

INPUT

```
int main()
{
    {
       int b;
    }
    b=2;
}
```

OUTPUT

```
ENTRY DELETED::b   int  2
b -->ERROR:undeclared variable
Invalid Expression

*****Symbol Table******

Name        Type        Size        Scope       IsFunction    NoOfArgs
main        int         4     0     1     0

*****Constant Table******

2     numeric constant
```

```
Parsing SUCCESSFUL.
```

5) Expression Mismatch

INPUT

```
int main()
{
    int a;
    a=1+2.0;
}
```

OUTPUT

```
Invalid Expression
ENTRY DELETED::a   int  1

*****Symbol Table******

Name        Type       Size       Scope      IsFunction    NoOfArgs
main        int        4     0    1     0

*****Constant Table******

1    numeric constant
2.0    numeric constant

Parsing SUCCESSFUL.
```

6) Not a Integer Expression for While Loop

INPUT

```
int main()
{
    int a;
    a=1;
    while(1.5)
    {
```

```
        a++;
    }
}
```

OUTPUT

```
While should have a interger expression
ENTRY DELETED::a   int  1

*****Symbol Table******

Name        Type        Size        Scope       IsFunction    NoOfArgs
main        int         4     0     1     0

*****Constant Table******

1     numeric constant
1.5     numeric constant

Parsing SUCCESSFUL.
```

7) Function structure mismatch

INPUT

```
int fun(int x, int y)
{
    return x+y;
}

int main()
{
    int a,b;
    a=1;
    b=2;
    fun(a,b,a);
}
```

OUTPUT

```
ENTRY DELETED::x   int  1
ENTRY DELETED::y   int  1
Error:function defination does not match any existing function
defination
ENTRY DELETED::a   int  1
ENTRY DELETED::b   int  1

*****Symbol Table******

Name        Type       Size       Scope      IsFunction    NoOfArgs
fun         int        4     0    1     2
main        int        4     0    1     0

*****Constant Table******

1    numeric constant
2    numeric constant

Parsing SUCCESSFUL.
```

8) Function argument type mismatch

INPUT

```
int fun(int x, int y)
{
    return x+y;
}

int main()
{
    int a;
    float b;
    a=1;
    b=2.3;
    fun(a,b);
}
```

OUTPUT

```
ENTRY DELETED::x   int  1
ENTRY DELETED::y   int  1
Error:function defination does not match any existing function
defination
ENTRY DELETED::a   int  1
ENTRY DELETED::b   float  1

*****Symbol Table******

Name        Type       Size       Scope      IsFunction    NoOfArgs
fun         int        4     0    1     2
main        int        4     0    1     0

*****Constant Table******

1    numeric constant
2.3    numeric constant

Parsing SUCCESSFUL.
```

9) Return type mismatch

INPUT

```
float fun(int x, int y)
{
    return x+y;
}

int main()
{
    int a,b;
    a=1;
    b=2;
    fun(a,b);
}
```

OUTPUT

```
Error Return type wrong.
ENTRY DELETED::x   int  1
ENTRY DELETED::y   int  1
ENTRY DELETED::a   int  1
ENTRY DELETED::b   int  1

*****Symbol Table******

Name        Type       Size       Scope      IsFunction    NoOfArgs
fun         float      8     0    1     2
main        int        4     0    1     0

*****Constant Table******

1    numeric constant
2    numeric constant

Parsing SUCCESSFUL.
```

CONCLUSION

We studied Semantic Rules and applied the rules to make a working Semantic Analyzer which checks the Semantical correctness of the program and the supports basic declarations and definitions, expressions, function definition and function call, looping constructs (for, while, do-while) and conditional statements (if-else ladder, switch-case). Thus, the semantic analysis unit of compiler design is studied and implemented.

THE END