Semantic Analyser for the C Language

National Institute of Technology Karnataka Surathkal

Date: 21st October 2020

Submitted To:
Prof. P. Santhi Thilagam
CSE Dept, NITK

Group Members:
Niranjan S Y 181CO136
Rajath C Aralikatti 181CO241
Shruthan R 181CO250
Varun N R 181CO134

Abstract

Semantic Analysis is the third phase of Compiler Design. Semantic Analysis makes sure that declarations and statements of a program are semantically correct. This phase has production rules as in the syntax phase. But it also has annotations under each production rule. These annotations are basically codes which perform a set of actions when the production rule is reduced. The Semantic Analyzer has a collection of procedures which is called by the parser as and when required by the grammar. Both the syntax tree of the previous phase and the symbol table are used to check the consistency of the given code. The main function of a semantic analyser is to check the declarations and statements of a program with their semantics, i.e, ensuring that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used. This report contains details of the Semantic Analyser for the C language, which is the third project of the course.

Objective

The objective of this phase is to build a semantic analyser for the C language using LEX and YACC scripts.

Functions

The semantic analyser supports int and float data types. It will throw an error in the following cases:

- Duplicate variable declarations in the same scope
- Undeclared variable
- Out of scope variable
- Duplicate functions declarations
- Return type mismatch
- Variable type mismatch
- Multiple function definitions
- Mismatch in either the number of arguments or the type of one or more arguments in a function call
- No subscript for array identifier
- Subscript for non-array identifier

Requirements

- GCC, the GNU Compiler Collection
- FLEX (Fast Lexical Analyser Generator)
- YACC (Yet Another Compiler-Compiler)

TABLE OF CONTENTS

1. Introduction

1.1 Semantic Analysis

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. 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 the next phase of the compilation, that is, 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 as the index to an array. The language specification may permit some type conversions called coercions. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. 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 help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

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

- Scope resolution
- Type checking
- Array-bound checking

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has a 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

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 in terms of higher-level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

The structure of our Yacc script is given below. Files are divided into three sections, separated by lines 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.

1.2 Execution Steps

The workflow is explained as under:

- Compile the script using Yacc tool
  `$ yacc sem.y`
- Compile the flex *s*cript using Flex tool
- `$ lex sem.l`
- After compiling the lex file, lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generated after compiling the yacc script.
- Compilation is done with the options –ll, –ly and -w

2. Design of Programs

2.1 Lexer Code

sem.l

```
alpha [A-Za-z]
digit [0-9]



%%
"<="                                                    return LE;
"<"                                                     return LT;
">="                                                    return GE;
">"                                                     return GT;
"++"                                                    return INCR;
"--"                                                    return DECR;
"=="                                                    return EQUAL;

[ \t]                                                           ;
\n  {yylineno++                                                ;}

"{" {block_start(); return '{';}
"}" {block_end(); return '}';}

"int"   {yylval.ival = INT;                             return INT;}
"float" {yylval.ival = FLOAT;                           return FLOAT;}
"void"  {yylval.ival = VOID; return VOID;}
"unsigned int" {yylval.ival = UNSIGNED_INT;             return
UNSIGNED_INT;}
"long int" {yylval.ival = L_INT;                        return L_INT;}
"short int" {yylval.ival = S_INT;                       return S_INT;}
"else"  {return ELSE;}
"if"    return IF;
"while"                                                 return WHILE;
```

```
"for"                                                   return FOR;
"return"                                                return RETURN;
"printf"                                                return PRINT;

^"#include ".+                                          return PREPROC;

{alpha}({alpha}|{digit})* {yylval.str=strdup(yytext);   return ID;}
{digit}+    {yylval.str=strdup(yytext);                 return
INT_CONST;}
{digit}+\.{digit}+  {yylval.str=strdup(yytext);         return
FLOAT_CONST;}

\/\/.*
;
\/\*(.*\n)*.*\*\/
;

\".*\"                                                  return STRING;
.                                                       return yytext[0];
%%
```

2.2 Parser Code

sem.y

```
%{
    #include <stdio.h>
    #include <stdlib.h>

    struct symbol_table_structure
    {
        int sno;
        char token[100];
        int type[100];
        int tn;
        int addr;
        float fvalue;
        int scope;
        int arrFlag;
        int funcFlag;
        int fType[100];
        int numParams;
```

```
}Symbol_Table[100];

int var_addr = 0;
int scope_incrementer=1;
int j=8;
int scope_stack[100];
int index1=0;
int end[100];
int return_types[10];
int gl1,gl2,curr_type=0,c=0,b;
int type=258;
int fname[100];
int nP;
int fTypes[100];
int fTypes2[100];
int temptype;
int it;
int temp;

int n=0, return_types[10];

void storereturn( int curr_type, int returntype )
{
    return_types[curr_type] = returntype;
    return;
}

void insertscope(char *a,int s)
{
    int i;
    for (i=0;i<n;i++)
    {
        if (!strcmp(a,Symbol_Table[i].token))
        {
            Symbol_Table[i].scope=s;
            break;
        }
    }
}

int returntype_func(int ct)
{
```

```
    return return_types[ct-1];
}

int isArray(char* a)
{
    int i;
    for (i=0;i<=n;i++)
    {
        if (!strcmp(a,Symbol_Table[i].token))
        {
            if (Symbol_Table[i].arrFlag==1)
                return Symbol_Table[i].fvalue;
                    else
                        return 0;
        }
    }
    return 0;
}

int retNumParams(char* a)
{
    int i;
    for (i=0;i<=n;i++)
    {
        if (!strcmp(a,Symbol_Table[i].token))
        {
            return Symbol_Table[i].numParams;
        }
    }
    return 0;
}

void getParams(char* a)
{
    int i;
    for (i=0;i<=n;i++)
    {
        if (!strcmp(a,Symbol_Table[i].token))
        {
            for (int j=0; j<Symbol_Table[i].numParams; j++)
                fTypes[j] = Symbol_Table[i].fType[j];
        }
```

```
        }
        return 0;
    }
    int returnScope(char *a,int cs)
    {
        //printf("\nString is: %s", a);
        int i;
        int max = 0;
        for (i=0;i<=n;i++)
        {
            if (!strcmp(a,Symbol_Table[i].token))
            {
                if (Symbol_Table[i].scope>=max)
                    max = Symbol_Table[i].scope;
            }
        }
        return max;
    }

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

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

    void update_value(char *a,char *b,int sc)
```

```
    {
        int i,j,k;
        int max=0;
        for (i=0;i<=n;i++)
        {
            if (!strcmp(a,Symbol_Table[i].token)   && sc>=Symbol_Table[i].scope)
            {
                if (Symbol_Table[i].scope>=max)
                    max=Symbol_Table[i].scope;
            }
        }
        for (i=0;i<=n;i++)
        {
            if (!strcmp(a,Symbol_Table[i].token)   && max==Symbol_Table[i].scope)
            {
                float temp=atof(b);
                for (k=0;k<Symbol_Table[i].tn;k++)
                {
                    if
(Symbol_Table[i].type[k]==258||Symbol_Table[i].type[0]==269)
                        Symbol_Table[i].fvalue=(int)temp;
                    else
                        Symbol_Table[i].fvalue=temp;
                }
            }
        }
    }

    void insert(char *name, int type, int addr, int arrFlag, int funcStatus)
    {
        int i;
        if (lookup(name))
        {
            strcpy(Symbol_Table[n].token,name);
            if (funcStatus == 1)
                Symbol_Table[n].funcFlag=1;
            Symbol_Table[n].tn=1;
            Symbol_Table[n].type[Symbol_Table[n].tn-1]=type;
            Symbol_Table[n].addr=addr;
            Symbol_Table[n].sno=n+1;
            Symbol_Table[n].arrFlag = arrFlag;
            n++;
```

```
        }
        else
        {
            if (funcStatus == 1)
                Symbol_Table[n].funcFlag=1;
            for (i=0;i<n;i++)
            {
                if (!strcmp(name,Symbol_Table[i].token))
                {
                    Symbol_Table[i].tn++;
                    Symbol_Table[i].type[Symbol_Table[i].tn-1]=type;
                    break;
                }
            }
        }

        return;
    }
    void insertFunc(char *name, int type, int addr, int arrFlag, int params[100],
int numParams)
    {
        int i;
        if (lookup(name))
        {
            strcpy(Symbol_Table[n].token,name);
            Symbol_Table[n].tn=1;
            Symbol_Table[n].type[Symbol_Table[n].tn-1]=type;
            Symbol_Table[n].addr=addr;
            Symbol_Table[n].sno=n+1;
            Symbol_Table[n].arrFlag = arrFlag;
            Symbol_Table[n].funcFlag = 1;
            for (int j=0; j<numParams; j++)
                Symbol_Table[n].fType[j] = params[j];
            Symbol_Table[n].numParams = numParams;
            n++;
        }
        else
        {
            for (i=0;i<n;i++)
            {
                if (!strcmp(name,Symbol_Table[i].token))
                {
```

```
                    Symbol_Table[i].tn++;
                    Symbol_Table[i].type[Symbol_Table[i].tn-1]=type;
                    break;
                }
            }
        }

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

    void print()
    {
        int i,j;

printf("\n----------------------------------------------------------------------
--------------------");
        printf("\n\t\t\t\t\tSYMBOL TABLE\n");

printf("----------------------------------------------------------------------
-----------------");
        printf("\nToken\tValue\tScope\tisArray\tArrayDim\tType\tReturn
Type\tArguments\n");
        for (i=0;i<n;i++)
        {
            if (Symbol_Table[i].type[0]==258 || Symbol_Table[i].type[0]==261||
Symbol_Table[i].type[0]==262|| Symbol_Table[i].type[0]==263)

printf("%s\t%d\t%d\tFalse\t-\t",Symbol_Table[i].token,(int)Symbol_Table[i].fvalue
,Symbol_Table[i].scope);
            else
```

```
            {
                if (Symbol_Table[i].arrFlag)

printf("%s\t-\t%d\tTrue\t%d\t",Symbol_Table[i].token,Symbol_Table[i].scope,
(int)Symbol_Table[i].fvalue);
                else if (Symbol_Table[i].type[0]==274)

printf("%s\t-\t%d\tFalse\t-\t",Symbol_Table[i].token,Symbol_Table[i].scope);
                else

printf("%s\t%.1f\t%d\tFalse\t-\t",Symbol_Table[i].token,Symbol_Table[i].fvalue,Sy
mbol_Table[i].scope);
            }

            // if (Symbol_Table[i].funcFlag == 1)
            //  printf("\tFUNCTION");

            for (j=0;j<Symbol_Table[i].tn;j++)
            {
                if (Symbol_Table[i].type[j]==258)
                    printf("\tINT");
                else if (Symbol_Table[i].type[j]==259)
                    printf("\tFLOAT");
                else if (Symbol_Table[i].type[j]==274)
                    printf("\tFUNCTION");
                else if (Symbol_Table[i].type[j]==269)
                    printf("\tARRAY");
                else if (Symbol_Table[i].type[j]==260)
                    printf("\tVOID");
                else if (Symbol_Table[i].type[j]==261)
                    printf("\tUNSIGNED INT");
                else if (Symbol_Table[i].type[j]==263)
                        printf("\tLONG INT");
                else if (Symbol_Table[i].type[j]==262)
                        printf("\tSHORT INT");
            }
            printf("\t");
            for (int j=0;j<Symbol_Table[i].numParams;j++)
            {
                if (Symbol_Table[i].fType[j]==258)
                    printf("INT,");
                else if (Symbol_Table[i].fType[j]==259)
```

```
                    printf("FLOAT,");
                else if (Symbol_Table[i].fType[j]==274)
                    printf("FUNCTION,");
                else if (Symbol_Table[i].fType[j]==269)
                    printf("ARRAY,");
                else if (Symbol_Table[i].fType[j]==260)
                    printf("VOID,");
                else if (Symbol_Table[i].fType[j]==261)
                    printf("UNSIGNED INT,");
                else if (Symbol_Table[i].fType[j]==263)
                        printf("LONG INT,");
                else if (Symbol_Table[i].fType[j]==262)
                        printf("SHORT INT,");
            }
            printf("\n");
        }
        return;
    }
%}

%token<ival> INT FLOAT VOID UNSIGNED_INT S_INT L_INT
%token<str> ID INT_CONST FLOAT_CONST
%token WHILE FOR IF RETURN PREPROC STRING PRINT FUNCTION ARRAY ELSE
%token INCR DECR
%token EQUAL LE LT GE GT

%left '=' ','
%left '+' '-'
%left '*' '/'
%left INCR DECR
%left LE LT GE GT EQUAL
%left IF ELSE

%type<str> secondary_assignment consttype assignment_exp
%type<ival> Type

%union {
        int ival;
        char *str;
    }

%%
```

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

Function
    : Type ID '(' ')' compound_stmt {

    if ($1!=returntype_func(curr_type))
    {
        printf("\nError : Type mismatch : Line %d\n",printline());
    }

    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 : Type mismatch in redeclaration of %s : Line
%d\n",$2,printline());
    else
    {
        insert($2,FUNCTION,var_addr, 0, 1);
        insert($2,$1,var_addr, 0, 0);
        var_addr+=4;
    }
    }
    | Type ID '(' param_list ')' compound_stmt {


    if ($1!=returntype_func(curr_type))
    {
        printf("\nError : Type mismatch : Line %d\n",printline());
    }

    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 : Type mismatch in redeclaration of %s : Line
%d\n",$2,printline());
    else
    {
        insertFunc($2,FUNCTION,var_addr, 0, fname, nP);
        insert($2,$1,var_addr, 0, 0);
        var_addr+=4;
    }
    };

param_list: Type ID { nP = 1; fname[nP-1] = $1; }
    | param_list ',' Type ID { nP++; fname[nP-1] = $3; };

Type
    : INT
    | FLOAT
    | VOID
  | UNSIGNED_INT
  | S_INT
  | L_INT
    ;

compound_stmt
    : '{' statement_list '}'
    ;

statement_list
    : statement_list stmt
    | compound_stmt
    |
    ;

stmt
    : Declaration
    | if_stmt
    | while_stmt
    | for_stmt
    | function_call
    | RETURN consttype ';' {
                    if (!(strspn($2,"0123456789")==strlen($2)))
```

```
                        storereturn(curr_type,FLOAT);
                    else
                        storereturn(curr_type,INT); curr_type++;
                }
    | RETURN ';' {storereturn(curr_type,VOID); curr_type++;}
    | ';'
    | PRINT '(' STRING ',' exp ')' ';'
    | PRINT '(' STRING ')' ';'
    | compound_stmt
    ;

function_call: ID '(' call_list ')' ';' {
    if (lookup($1))
        printf("\nError: Undeclared function %s : Line %d\n", $1, printline());
    else
    {
        if (retNumParams($1) == 0)
            printf("\nError : Parameter list does not match signature : Line
%d\n", printline());
        getParams($1);
    }

    for (int j=0; j<retNumParams($1); j++)
    {
        if (fTypes[j] != fTypes2[j])
            printf("\nError : Parameter list does not match signature : Line
%d\n", printline());
    }

}
| ID '(' ')' ';' {
    if (lookup($1))
        printf("\nError: Undeclared function %s : Line %d\n", $1, printline());
    else
    {
        if (retNumParams($1) != 0)
            printf("\nError : Parameter list does not match signature : Line
%d\n", printline());
    }
};
```

```
call_list : ID { temptype = returntype($1, scope_stack[index1-1]); it = 0;
fTypes2[it] = temptype; }
    | consttype { temptype = temp; it = 0; fTypes2[it] = temptype; }
    | call_list ',' ID { it++; temptype = returntype($3, scope_stack[index1-1]);
fTypes2[it] = temptype;}
    | call_list ',' consttype { temptype = temp; it++; fTypes2[it] = temptype;}
    ;

if_stmt
    : IF '(' expr1 ')' compound_stmt
    | IF '(' expr1 ')' compound_stmt ELSE compound_stmt
    ;

while_stmt
    : WHILE '(' expr1 ')' compound_stmt
    ;

for_stmt
    : FOR '(' expr1 ';' expr1 ';' expr1 ')' compound_stmt

expr1
    : expr1 LE expr1
    | expr1 LT expr1
    | expr1 GE expr1
    | expr1 GT expr1
    | expr1 EQUAL expr1
    | secondary_assignment
    |
    ;

secondary_assignment : ID '=' secondary_assignment
    {
      c=0;
        int scope_curr=returnScope($1,scope_stack[index1-1]);
        int type=returntype($1,scope_curr);
        if ((!(strspn($3,"0123456789")==strlen($3))) && type==258)
            printf("\nError : Type Mismatch : Line %d\n",printline());
        if (!lookup($1))
        {
            int curr_scope=scope_stack[index1-1];
            int scope=returnScope($1,curr_scope);
            if ((scope<=curr_scope && end[scope]==0) && !(scope==0))
```

```
                update_value($1,$3,curr_scope);
        }
        if (isArray($1))
                printf("\nError: array Identfier has no subscript: Line %d\n",
printline());

        }

    | ID ',' secondary_assignment    {
                    if (lookup($1))
                        printf("\nUndeclared Variable %s : Line
%d\n",$1,printline());

                        if (isArray($1))
                                printf("\nError: array identfier has no
subscript: Line %d\n", printline());

                }
    | assignment_exp
    | consttype ',' secondary_assignment
    | ID  {
        if (lookup($1))
            printf("\nUndeclared Variable %s : Line %d\n",$1,printline());

            if (isArray($1))
                printf("\nError: Non-array variable used as an array: Line %d\n",
printline());

        }
    | exp
    | consttype
    ;

assignment_exp :  ID '[' INT_CONST ']' '=' exp {
            if (lookup($1))
                printf("\nUndeclared Variable %s : Line %d\n",$1,printline());

            if (isArray($1)==0)
                printf("\nError: Non-array variable used as an array: Line %d\n",
printline());

                float bound = isArray($1);
```

```
                if (isArray($1) && (atoi($3) >= bound || atoi($3) < 0))
                    printf("\nError: array subscript out of bounds : Line %d\n",
printline());

        }
        ;

exp : ID {
    if (isArray($1))
     printf("\nError: array identifier has no subscript: Line %d\n",
printline());

    if (c==0)
    {
        c=1;
        int scope_curr=returnScope($1,scope_stack[index1-1]);
        b=returntype($1,scope_curr);
    }
    else
    {
        int scope_curr1=returnScope($1,scope_stack[index1-1]);
        if (b!=returntype($1,scope_curr1))
            printf("\nError : Type Mismatch : Line %d\n",printline());
    }
    if (!lookup($1))
    {
        int curr_scope=scope_stack[index1-1];
        int scope=returnScope($1,curr_scope);
        if (!(scope<=curr_scope && end[scope]==0))
            printf("\nError : Variable %s out of scope : Line
%d\n",$1,printline());
    }
  else
    printf("\nError : Undeclared Variable %s : Line %d\n",$1,printline());
    }
    | ID '[' INT_CONST ']'{
        if (c==0)
        {
            c=1;
            int scope_curr=returnScope($1,scope_stack[index1-1]);
            b=returntype($1,scope_curr);
```

```
        }
        else
        {
            int scope_curr1=returnScope($1,scope_stack[index1-1]);
            if (b!=returntype($1,scope_curr1))
                printf("\nError : Type Mismatch : Line %d\n",printline());
        }
        if (!lookup($1))
        {
            int curr_scope=scope_stack[index1-1];
            int scope=returnScope($1,curr_scope);
            if (!(scope<=curr_scope && end[scope]==0))
                printf("\nError : Variable %s out of scope : Line
%d\n",$1,printline());
        }
      else
        printf("\nError : Undeclared Variable %s : Line %d\n",$1,printline());

        if (isArray($1)==0)
            printf("\nError: Non-array variable used as an array: Line %d\n",
printline());

        float bound = isArray($1);

        if (isArray($1) && (atoi($3) >= bound || atoi($3) < 0) )
            printf("\nError: array subscript out of bounds : Line %d\n",
printline());

    }
    | exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
    | '(' exp '+' exp ')'
    | '(' exp '-' exp ')'
    | '(' exp '*' exp ')'
    | '(' exp '/' exp ')'
    | consttype
    | '(' exp ')'
    | INCR exp
    | DECR exp
    | exp INCR
```

```
    | exp DECR
    ;

consttype : INT_CONST { temp = 258;}
    | FLOAT_CONST { temp = 259;}
    ;

Declaration
    : Type ID '=' consttype ';'
        {
            if ( (!(strspn($4,"0123456789")==strlen($4))) && $1==258)
                printf("\nError : Type Mismatch : Line %d\n",printline());

            if (!lookup($2))
            {
                int curr_scope=scope_stack[index1-1];
                int previous_scope=returnScope($2,curr_scope);
                if (curr_scope==previous_scope)
                    printf("\nError : Redeclaration of %s : Line
%d\n",$2,printline());
                else
                {
                    insert_dup($2,$1,var_addr,curr_scope, 0);
                    update_value($2,$4,scope_stack[index1-1]);
                    var_addr+=4;
                }
            }
            else
            {
                int scope=scope_stack[index1-1];
                insert($2,$1,var_addr, 0, 0);
                insertscope($2,scope);
                update_value($2,$4,scope_stack[index1-1]);
                var_addr+=4;
            }
        }
    | Type ID ';' {
        if (!lookup($2))
        {
            int curr_scope=scope_stack[index1-1];
            int previous_scope=returnScope($2,curr_scope);
            if (curr_scope==previous_scope)
```

```
                printf("\nError : Redeclaration of %s : Line
%d\n",$2,printline());
            else
            {
                insert_dup($2,$1,var_addr,curr_scope, 0);
                var_addr+=4;
            }
        }
        else
        {
            int scope=scope_stack[index1-1];
            insert($2,$1,var_addr, 0, 0);
            insertscope($2,scope);
            var_addr+=4;
        }
    }
    | secondary_assignment ';'  {
                if (!lookup($1))
                {
                    int curr_scope=scope_stack[index1-1];
                    int scope=returnScope($1,curr_scope);
                    if (!(scope<=curr_scope && end[scope]==0))
                        printf("\nError : Variable %s out of scope : Line
%d\n",$1,printline());
                }
                else
                    printf("\nError : Undeclared Variable %s : Line
%d\n",$1,printline());
                }

    | Type ID '[' INT_CONST ']' ';' {
                        insert($2,ARRAY,var_addr,1,0);
                        insert($2,$1,var_addr,1,0);
                        update_value($2,$4,scope_stack[index1-1]);
                        int scope=scope_stack[index1-1];
                        insertscope($2, scope);
                        var_addr+=4;
                        if (atoi($4)<=0)
                        {
                            printf("\nError: Illegal array subscript %d : Line
%d\n", atoi($4), printline());
                        }
```

```
                    }
    | error
    ;

%%

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

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

int printline()
{
    return yylineno;
}

void block_start()
{
    scope_stack[index1]=scope_incrementer;
    scope_incrementer++;
    index1++;
    return;
}
```

```
void block_end()
{
    index1--;
    end[scope_stack[index1]]=1;
    scope_stack[index1]=0;
    return;
}
```

2.3 Explanation

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

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

Also, we have writen test cases that show the following additional functionality:

- Loops (for and while)
- Functions

3. Test Cases

3.1 Without Errors:

Test 1 :

```c
#include <stdio.h>

int arr2[10];

float func(int b, float c)
{
    int a;
    int arr1[10];
    return 4.3;
}

int func2()
{
    printf("Hello");
    return 1;
}

void fn(int x)
{
    printf("Hello");
```

```
    return;
}

int gnome(long int abc, float xyz)
{
    printf("HI");
    return 6;
}

int main()
{
    int a;
    int k=5;
    int arr[5];
    long int z = 0;
    while(z < 6)
    {
        int b=5;
        while(b > 0)
        {
            int c;
            int arr3[10];
        }
        z = z + 1;
    }
    int i;
    for(i=0; i<5; i++)
    {
        printf("hello");
    }
}
```

Output :

```
PARSING DONE

------------------------------------------------------------------------------------------
                                    SYMBOL TABLE
------------------------------------------------------------------------------------------
Token   Value   Scope   isArray ArrayDim       Type    Return Type     Arguments
arr2    -       0       True    10              INT
a       0       1       False   -               INT
arr1    -       1       True    10              INT
func    -       0       False   -               FUNCTION        FLOAT   INT,FLOAT,
func2   -       0       False   -               FUNCTION        INT
fn      -       0       False   -               FUNCTION        VOID    INT,
gnome   -       0       False   -               FUNCTION        INT     LONG INT,FLOAT,
a       0       5       False   -               INT
k       5       5       False   -               INT
arr     -       5       True    5               INT
z       0       5       False   -               LONG INT
b       5       6       False   -               INT
c       0       7       False   -               INT
arr3    -       7       True    10              INT
i       0       5       False   -               INT
main    -       0       False   -               FUNCTION        INT
```

Test 2:

```c
//Program to show for loop working

#include <stdio.h>

int arr2[10];

int main()
{
    int i;
    for(i=0; i<10; i++)
    {
        printf("hello");
    }

    return 0;
}
```

Output:

```
PARSING DONE

-------------------------------------------------------------------------------------
                                    SYMBOL TABLE
-------------------------------------------------------------------------------------
Token   Value   Scope   isArray ArrayDim        Type    Return Type     Arguments
arr2    -       0       True    10              INT
i       0       1       False   -               INT
main    -       0       False   -               FUNCTION        INT
```

3.2 With Errors

Test 3:

```c
// Program having undeclared variable error


#include <stdio.h>


int main()
{
    int a = 10;
    int b = 20;
    c = a + b;
    printf("%d", c);

    return 0;
}
```

Output :

```
Error : Undeclared Variable c : Line 8

Error : Undeclared Variable c : Line 9
PARSING DONE

-------------------------------------------------------------------------------------
                                    SYMBOL TABLE
-------------------------------------------------------------------------------------
Token   Value   Scope   isArray ArrayDim        Type    Return Type     Arguments
a       10      1       False   -               INT
b       20      1       False   -               INT
main    -       0       False   -               FUNCTION        INT
```

Test 4:

```c
// Program having type mismatch error


#include <stdio.h>


int main()
{
    int a = 10;
    float b;
    float c;
    c = a + b;


    return 0;
}
```

Output :

```
Error : Type Mismatch : Line 9
PARSING DONE

-------------------------------------------------------------------------------------------
                                        SYMBOL TABLE
-------------------------------------------------------------------------------------------
Token   Value   Scope   isArray ArrayDim        Type    Return Type     Arguments
a       10      1       False   -               INT
b       0.0     1       False   -               FLOAT
c       0.0     1       False   -               FLOAT
main    -       0       False   -               FUNCTION        INT
```

Test 5:

```c
// Program having array index out of bounds error


#include <stdio.h>


int main()
{
    int arr[20];
    arr[25] = 20;


    return 0;
}
```

Output:

```
Error: array subscript out of bounds : Line 7
PARSING DONE

------------------------------------------------------------------------------------------
                                        SYMBOL TABLE
------------------------------------------------------------------------------------------
Token    Value    Scope    isArray ArrayDim        Type     Return Type      Arguments
arr      -        1        True    20              INT
main     -        0        False   -               FUNCTION         INT
```

Test 6:

```c
// Function call parameter list not matching function signature

#include <stdio.h>

void func(int a)
{
    int b = 10;
    return;
}

int main()
{
    func();
    return 0;
}
```

Output:

```
Error : Parameter list does not match signature : Line 12
PARSING DONE

------------------------------------------------------------------------------------------
                                        SYMBOL TABLE
------------------------------------------------------------------------------------------
Token    Value    Scope    isArray ArrayDim        Type     Return Type      Arguments
b        10       1        False   -               INT
func     -        0        False   -               FUNCTION         VOID     INT,
main     -        0        False   -               FUNCTION         INT
```

Test 7:

```c
// Redeclaration of variable in the same scope

#include <stdio.h>

int main()
{
    int a = 10;
    printf("%d", a);
    int a = 40;
    printf("%d", a);

    return 0;
}
```

Output:

```
Error : Redeclaration of a : Line 8
PARSING DONE

-----------------------------------------------------------------------------------------
                                        SYMBOL TABLE
-----------------------------------------------------------------------------------------
Token   Value   Scope   isArray ArrayDim       Type    Return Type     Arguments
a       10      1       False   -               INT
main    -       0       False   -               FUNCTION        INT
```

Test 8:

```c
// Variable out of scope

#include <stdio.h>

int main()
{
    {
        int a = 10;
        int b = 20;
        printf("a is %d\n", a);
        printf("b is %d\n", b);
    }
    a = 40;
```

```
    return 0;
}
```

Output:

```
Error : Variable a out of scope : Line 12
PARSING DONE

------------------------------------------------------------------------------------------
                                        SYMBOL TABLE
------------------------------------------------------------------------------------------
Token   Value   Scope   isArray ArrayDim        Type    Return Type     Arguments
a       10      2       False   -               INT
b       20      2       False   -               INT
main    -       0       False   -               FUNCTION        INT
```

Test 9:

```
// Use of non-array variable with subscript


#include <stdio.h>


int main()
{
    int a;
    a[0] = 40;


    return 0;
}
```

Output:

```
Error: Non-array variable used as an array: Line 7
PARSING DONE

------------------------------------------------------------------------------------------
                                        SYMBOL TABLE
------------------------------------------------------------------------------------------
Token   Value   Scope   isArray ArrayDim        Type    Return Type     Arguments
a       0       1       False   -               INT
main    -       0       False   -               FUNCTION        INT
```

Test 10:

```c
// Use of array variable without subscript


#include <stdio.h>


int main()
{
    int a[10];
    a = 40;


    return 0;
}
```

Output:

```
Error: array Identifier has no subscript: Line 7
PARSING DONE

------------------------------------------------------------------------------------------
                                        SYMBOL TABLE
------------------------------------------------------------------------------------------
Token   Value   Scope   isArray ArrayDim        Type    Return Type     Arguments
a       -       1       True    40              INT
main    -       0       False   -               FUNCTION        INT
```

Test 11:

```c
// Use of array of size less than 1


#include <stdio.h>


int main()
{
    int a[0];


    return 0;
}
```

Output:

```
Error: Illegal array subscript 0 : Line 6
PARSING DONE

------------------------------------------------------------------------------------------
                                    SYMBOL TABLE
------------------------------------------------------------------------------------------
Token   Value   Scope   isArray ArrayDim        Type    Return Type     Arguments
a       -       1       True    0               INT
main    -       0       False   -               FUNCTION        INT
```

Test 12:

```c
// A void function trying to return a value

#include <stdio.h>

void func()
{
    return 1;
}

int main()
{
    int a = 10;

    return 0;
}
```

Output:

```
Error : Type mismatch : Line 7
PARSING DONE

------------------------------------------------------------------------------------------
                                    SYMBOL TABLE
------------------------------------------------------------------------------------------
Token   Value   Scope   isArray ArrayDim        Type    Return Type     Arguments
func    -       0       False   -               FUNCTION        VOID
a       10      2       False   -               INT
main    -       0       False   -               FUNCTION        INT
```

4. Implementation Details

The YACC script takes the stream of tokens recognized by the lexer using yyparse(). The following semantic errors are checked by the semantic analyser:

- Undeclared variable: If the semantic analyser encounters a variable in some statement in the given C program, it first checks if the variable is already present in the Symbol Table. If not, then it throws an error saying that the variable is undeclared.
- Redeclaration of variable in same scope: For a given identifier name, the semantic analyzer tries to match it with a declaration which has the closest scope with respect to the identifier name. However, if the analyzer finds two variables in the same scope declared with the same, it throws an error.
- Variable out of scope: The semantic analyzer throws an error if there is no declaration for the variable in the current or an outer scope.
- Return type of function misatch: This can happen if a function returns a value which is of a different data-type than that found in the function's signature
- Number of parameters in a function: The semantic analyzer checks the number of parameters against the number of parameters defined in the function's signature. If a mismatch is found, it throws an error.

5. Results and Future Work

In this third phase of the compiler design project, we have built a semantic analyzer that analyzes the tokens from the source program and checks that they follow the rules of the C semantics perfectly, and gives errors otherwise. Tokens recognized by the lexer are successfully parsed in the parser. The tokens are recorded in the Symbol Table. The output displays the symbol table, which contains the set of Identifiers and constants (integer, string, float, etcetera) present in the program along with their data-types. The symbol table includes the following attributes for each entry:

- Address: To perform the required storage allocation for data symbols
- Token name: Name of the token
- Initial value
- Scope: A number given to identify the block to which the variable belongs
- isArray: A boolean flag to indicate whether a variable is an array
- ArrayDim: Number of values that can be stored in the case of an array
- Type: Data-type of the variable
- Return Type: Return type in the case of a function
- Arguments: Arguments in the case of a function

The parser generates error messages in case of any syntactic or semantic errors in the test program.

The lex and yacc scripts presented together in this report take care of all the semantic rules of C language but are not fully exhaustive in nature. Our future work would include making the script more robust by adding unary and logical operator functionality and other corner cases and thus make it more effective. Next we would work on Intermediate Code Generation.

6. References

1. A. Aho, M. Lam, R. Sethi and J. Ullman, “Compilers Principles, Tools and Techniques”
2. https://silcnitc.github.io/yacc.html
3. https://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/tutorial-large.pdf
4. https://docs.oracle.com/cd/E19504-01/802-5880/6i9k05dgt/index.html
5. https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm
6. https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/