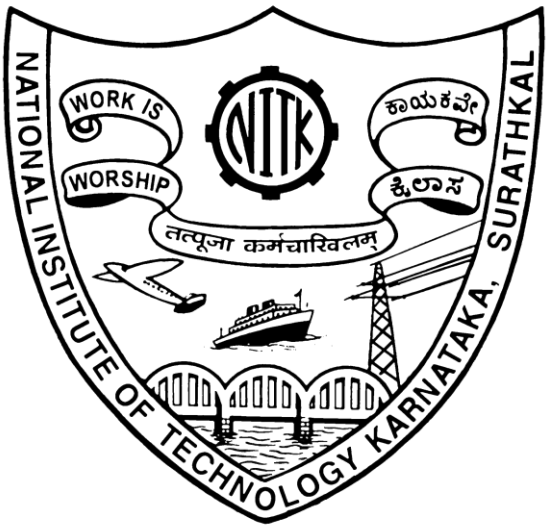
Semantic Analyzer for the C Language



National Institute of Technology Karnataka Surathkal

Date: 15/03/2017

Submitted To: Uma Priya

Group Members:

Sheetal Shalini (14CO142)

Navya R S (14CO126)

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. Lexical analysis, Syntax analysis, Semantic analysis and Intermediate Code generation are the four phases of the frontend of the compiler. Syntax analyzer will just create parse tree. Semantic Analyzer will check actual meaning of the statement parsed in parse tree. 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). Semantic analysis is also called context sensitive analysis. Semantic analysis is mainly a process in compiler construction after parsing to gather necessary semantic information from the source code. It is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or definite assignment (requiring all local variables 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 checked for semantic errors and warning/error messages are given in case of multiple definitions, type mismatch in assignments, expressions and re-declaration of functions, usage of undefined variables, return type checking with the function type. Scope of the variables are also handled and all these details are updated in the symbol table.*

Contents:

Introduction

Semantic Analysis

Syntax analyzer will just create parse tree. Semantic Analyzer will check actual meaning of the statement parsed in parse tree. 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). Semantic analysis is used for the following: 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.

Flex Script

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.

Yacc Script

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.

THE DECLARATIONS SECTION may contain the following items.

- 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

;

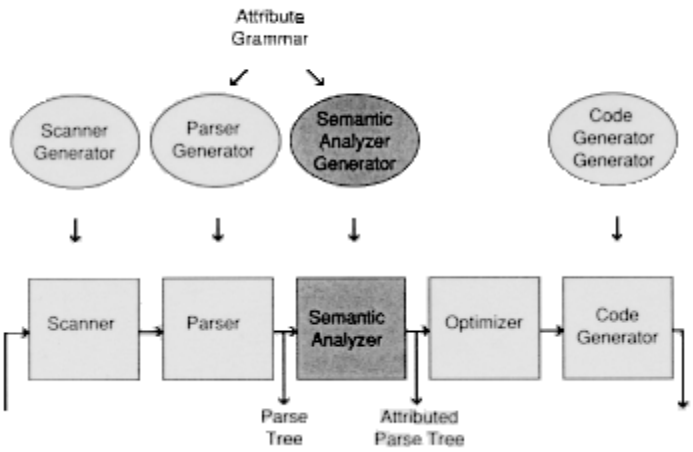
Actions may be associated with rules and are executed when the associated sentential form is matched.

C Program

This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned in account. Finally, a file called lex.yy.c is generated, which when executed recognizes the tokens present in the C program which was given as an input. The yacc script is run along with the lex script to parse the C code given as input and specify the syntactic and semantic errors (if any) or tell the user that the parsing has been successfully completed.

Design of Programs

Flow :



Parser checks the C code for syntactical errors and semantic errors and specifies them if present. Otherwise, it displays ‘parsing complete’.

Codes:

Lex Code : (parser.l file)

```
alpha [A-Za-z]
digit [0-9]
%%
[ \t] ;
\n  {yylineno++;}
"{" {open1(); return '{';}
"}" {close1(); 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;
struct  return STRUCT;
^"#include ".+  return PREPROC;
while   return WHILE;
return  return RETURN;
printf  return PRINT;
{alpha}({alpha}|{digit})* {yylval.str=strdup(yytext); return ID;}
{digit}+    {yylval.str=strdup(yytext);return NUM;}
{digit}+\.{digit}+  {yylval.str=strdup(yytext); return REAL;}
```

```
"<="        return LE;
\/\/.* ;
\/\*(.*\n)*.*\*\/ ;
\".*\"  return STRING;
.   return yytext[0];
%%
```

Yacc Code (parser.y)

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include "s_symbol.c"
    int g_addr = 100;
int i=1;
int j=8;
int stack[100];
int index1=0;
int end[100];
int arr[10];
int gl1,gl2,ct=0,c=0,b;
%}

%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
%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());
    }

    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,g_addr);
        insert($2,$1,g_addr);
        g_addr+=4;
    }
    }
    ;

Type : INT
    | FLOAT
    | VOID
    ;

CompoundStmt : '{' StmtList '}'
    ;

StmtList : StmtList stmt
    | CompoundStmt
    |
    ;

stmt : Declaration
    | if
    | while
    | dowhile
    | RETURN consttype ';' {
                    if(!(strspn($2,"0123456789")==strlen($2)))
                        storereturn(ct,FLOAT);
                    else
                        storereturn(ct,INT); ct++;
                    }
    | RETURN ';' {storereturn(ct,VOID); ct++;}
    | ';'
    | PRINT '(' STRING ')' ';'
    | CompoundStmt
    ;

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

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

while : WHILE '(' expr1 ')' CompoundStmt
    ;

expr1 : expr1 LE expr1
    | assignment1
    ;

assignment : ID '=' consttype
    | ID '+' assignment
    | ID ',' assignment
    | consttype ',' assignment
    | ID
```

```
    | consttype
    ;

assignment1 : ID '=' assignment1
    {
        int sct=returnscope($1,stack[index1-1]);
        int type=returntype($1,sct);
        if((!(strspn($3,"0123456789")==strlen($3))) && type==258)
            printf("\nError : Type Mismatch : Line %d\n",printline());
        if(!lookup($1))
        {
            int currscope=stack[index1-1];
            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());
                }
    | assignment2
    | consttype ',' assignment1
    | ID  {
        if(lookup($1))
            printf("\nUndeclared Variable %s : Line %d\n",$1,printline());
        }
    | consttype
    ;

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

exp : ID {
    if(c==0)
    {
        c=1;
        int sct=returnscope($1,stack[index1-1]);
        b=returntype($1,sct);
    }
    else
    {
        int sct1=returnscope($1,stack[index1-1]);
        if(b!=returntype($1,sct1))
            printf("\nError : Type Mismatch : Line %d\n",printline());
    }
    }
    | 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());
        if(!lookup($2))
        {
            int currscope=stack[index1-1];
            int previous_scope=returnscope($2,currscope);
            if(currscope==previous_scope)
                printf("\nError : Redeclaration of %s : Line
%d\n",$2,printline());
            else
            {
                insert_dup($2,$1,g_addr,currscope);
                check_scope_update($2,$4,stack[index1-1]);
                g_addr+=4;
            }
        }
        else
        {
            int scope=stack[index1-1];
            insert($2,$1,g_addr);
            insertscope($2,scope);
            check_scope_update($2,$4,stack[index1-1]);
            g_addr+=4;
        }
    }

    | assignment1 ';'  {
                if(!lookup($1))
                {
                    int currscope=stack[index1-1];
                    int scope=returnscope($1,currscope);
                    if(!(scope<=currscope && end[scope]==0) || 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 '[' assignment ']' ';' {
                        insert($2,ARRAY,g_addr);
                        insert($2,$1,g_addr);
                        g_addr+=4;
                    }
    | ID '[' assignment1 ']' ';'
    | STRUCT ID '{' Declaration '}' ';' {
                        insert($2,STRUCT,g_addr);
                        g_addr+=4;
                        }
    | STRUCT ID ID ';' {
```

```
                insert($3,STRUCT_VAR,g_addr);
                g_addr+=4;
                }
    | 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 open1()
{
    stack[index1]=i;
    i++;
    index1++;
    return;
}
void close1()
{
    index1--;
    end[stack[index1]]=1;
    stack[index1]=0;
    return;
}
```

Symbol Table Code (symbol_table.c)

```
#include<stdio.h>
#include<string.h>
struct sym
```

```
{
    int sno;
    char token[100];
    int type[100];
    int tn;
    int addr;
    float fvalue;
    int scope;
}st[100];
int n=0,arr[10];
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];
    }
}

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 addr)
{
    int i;
    if(lookup(name))
    {
        strcpy(st[n].token,name);
        st[n].tn=1;
        st[n].type[st[n].tn-1]=type;
        st[n].addr=addr;
        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 addr,int s_c)
{
    strcpy(st[n].token,name);
    st[n].tn=1;
    st[n].type[st[n].tn-1]=type;
    st[n].addr=addr;
    st[n].sno=n+1;
    st[n].scope=s_c;
    n++;
    return;
}

void print()
{
    int i,j;
    for(i=0;i<n;i++)
    {
        if(st[i].type[0]==258)
            printf("%d %s %d
%d",st[i].sno,st[i].token,st[i].addr,(int)st[i].fvalue);
        else
            printf("%d %s %d
%f",st[i].sno,st[i].token,st[i].addr,st[i].fvalue);
        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]==269)
                printf(" FUNCTION");
            else if(st[i].type[j]==271)
                printf(" ARRAY");
            else if(st[i].type[j]==260)
                printf(" VOID");
        }
        printf(" %d ",st[i].scope);
        printf("\n");
    }
    return;
}
```

Explanation :

Files :

1. **Parser.l** : Lex file which defines all the terminals of the productions stated in the yacc file. It contains regular expressions.
2. **Parser.y** : Yacc file is where the productions for all the loops, selection and conditional statements and expressions are mentioned. This fils also contains the semantic rules defined against every production necessary.
3. Symbol_table.c : It is the C file which generates the symbol table. It is included along with the yacc file.
4. Test.c : The input C code which will be parsed and checked for semantic correctness by executing the lex and yacc files along with it.

Yacc file has productions to check the following functionalities:

- Preprocessor directives
- Function definition
- Compound statements
- Nested compound statements
- Nested if else
- Nested while
- Variable definition and declaration
- Assignment operation
- Arithmetic operations

Semantic file contains the following functionalities:

- Check if a variable is in scope
- Check for multiple variable definitions in same/different scope
- Check for undeclared variables
- Type mismatch between variables in an expression. In this case type conversion is done before storing value in symbol table.
- Type mismatch in re-declaration of a function
- Type mismatch in return type of functions
- Update value of variable based on scope

Test Cases:

Without Errors:

1. Code : (Store & Update values of variables (based on scope) in symbol table, Multiple functions)

```
#include <stdio.h>
int main()
{
    int a = 4 ;
    {
        int b=3;
        float hg=9.7;
        int c=10;
        a=5;
        {
            b=10;
            int c=11;
            a=6;
            a;
        }
        c=8;
    }
    return 0;
}
void foo()
{
    int a=678;
    return;
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$ ./a.out test.c
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     6       1       INT
2       b       104     10      2       INT
3       hg      108     9.7     2       FLOAT
4       c       112     8       2       INT
5       c       116     11      3       INT
6       main    120     0.0     0       FUNCTION        INT
7       a       124     678     4       INT
8       foo     128     0.0     0       FUNCTION        VOID
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$
```

With Errors:

1. Code : (Undeclared variable)

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

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$ ./a.out test.c

Error : Undeclared Variable b : Line 5
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     10      1       INT
2       main    104     0.0     0       FUNCTION        INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$
```

2. Code : (Redeclaration of variable in same scope)

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

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$ ./a.out test.c

Error : Redeclaration of a : Line 6
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     4       1       INT
2       b       104     9       1       INT
3       main    108     0.0     0       FUNCTION        INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$
```

3. Code : (Variable out of scope)

```
#include <stdio.h>
```

```
int main()
{
    int a =4;
    {
        int c=56;
    }
    c=8;
    a=9;
    return 0;
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$ ./a.out test.c

Error : Variable c out of scope : Line 8
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     9       1       INT
2       c       104     56      2       INT
3       main    108     0.0     0       FUNCTION        INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$
```

4. Code : (Type mismatch : assigning value of different type to a variable)

```
#include <stdio.h>
int main()
{
    int a = 5.4;
    return 0;
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$ ./a.out test.c

Error : Type Mismatch : Line 4
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     5       1       INT
2       main    104     0.0     0       FUNCTION        INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$
```

5. Code : (Type mismatch : performing operations on variables of different types)

```
#include <stdio.h>
int main()
{
    int a =5;
    float b=6.7;
```

```
    int c = 4;
    c=a+b;
    return 0;
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$ ./a.out test.c

Error : Type Mismatch : Line 7
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     5       1       INT
2       b       104     6.7     1       FLOAT
3       c       108     4       1       INT
4       main    112     0.0     0       FUNCTION        INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$
```

6. Code : (Type mismatch : Redeclaration of standard functions)

```
#include <stdio.h>
int main()
{
    int a =5;
    return 0;
}
void scanf()
{
    return;
}
```

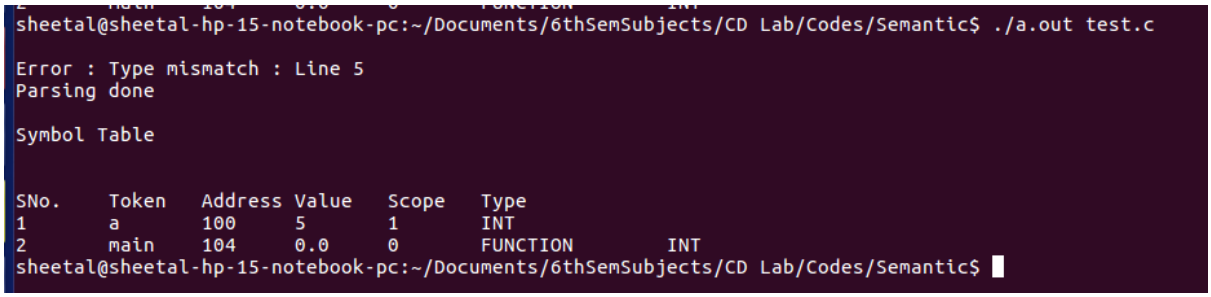
Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$ ./a.out test.c
Error : Type mismatch in redeclaration of scanf : Line 10
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     5       1       INT
2       main    104     0.0     0       FUNCTION        INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$
```

7. Code : (Type mismatch : Missing return statement in function)

```
#include <stdio.h>
int main()
{
    int a =5;
}
```

Output :



8. Code : (Type mismatch : Value returned by a function does not have same type as its return type)

```
#include <stdio.h>
int main()
{
    int a =5;
    return 9.8;
}
float foo()
{
    return 9;
}
void food()
{
    return 9.8;
}
int gh()
{
    return 8;
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$ ./a.out test.c

Error : Type mismatch : Line 6

Error : Type mismatch : Line 10

Error : Type mismatch : Line 14
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     5       1       INT
2       main    104     0.0     0       FUNCTION        INT
3       foo     108     0.0     0       FUNCTION        FLOAT
4       food    112     0.0     0       FUNCTION        VOID
5       gh      116     0.0     0       FUNCTION        INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Semantic$
```

Implementation :

Some of the productions along with their semantic rules are listed below :

1. Undeclared variable :

```
    | ID ',' assignment1    {
                    if(lookup($1))
                        printf("\nUndeclared Variable %s : Line
%d\n",$1,printline());
                }
    | assignment2
    | consttype ',' assignment1
    | ID  {
        if(lookup($1))
            printf("\nUndeclared Variable %s : Line %d\n",$1,printline());
        }
```

```
| assignment1 ';'  {
                if(!lookup($1))
                {
                    int currscope=stack[index1-1];
                    int scope=returnscope($1,currscope);
                    if(!(scope<=currscope && end[scope]==0) || 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());
                }
```

2. Redeclaration of variable :

```
Declaration : Type ID '=' consttype ';'
    {
        if( (!(strspn($4,"0123456789")==strlen($4))) && $1==258)
            printf("\nError : Type Mismatch : Line %d\n",printline());
        if(!lookup($2))
        {
            int currscope=stack[index1-1];
            int previous_scope=returnscope($2,currscope);
            if(currscope==previous_scope)
                printf("\nError : Redeclaration of %s : Line
%d\n",$2,printline());
            else
            {
                insert_dup($2,$1,g_addr,currscope);
                check_scope_update($2,$4,stack[index1-1]);
                g_addr+=4;
            }
        }
        else
        {
            int scope=stack[index1-1];
            insert($2,$1,g_addr);
            insertscope($2,scope);
            check_scope_update($2,$4,stack[index1-1]);
            g_addr+=4;
        }
    }
```

3. Variable out of scope :

```
| assignment1 ';'  {
                if(!lookup($1))
                {
                    int currscope=stack[index1-1];
                    int scope=returnscope($1,currscope);
                    if(!(scope<=currscope && end[scope]==0) || 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());
                }
```

4. Type mismatch

```
Function : Type ID '('')' CompoundStmt {
    if ($1!=returntype_func(ct))
    {
        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());
```

```
assignment1 : ID '=' assignment1
    {
        int sct=returnscope($1,stack[index1-1]);
        int type=returntype($1,sct);
        if((!(strspn($3,"0123456789")==strlen($3))) && type==258)
            printf("\nError : Type Mismatch : Line %d\n",printline());
```

```
exp : ID {
    if(c==0)
    {
        c=1;
        int sct=returnscope($1,stack[index1-1]);
        b=returntype($1,sct);
    }
    else
    {
        int sct1=returnscope($1,stack[index1-1]);
        if(b!=returntype($1,sct1))
            printf("\nError : Type Mismatch : Line %d\n",printline());
    }
```

5. Symbol table function to check scope and update value :

```
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;
            }
        }
    }
```

```
}
```

6. Symbol table structure :

```
struct sym
{
    int sno;
    char token[100];
    int type[100];
    int tn;
    int addr;
    float fvalue;
    int scope;
}st[100];
```

7. Functions to insert & return scope :

```
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;
}
```

Results :

The lex (parser.l) and yacc (parser.y) codes are compiled and executed by the following terminal commands to parse the given input file (test.c)

lex parser.l

yacc parser.y

gcc y.tab.c -ll -ly

./a.out test.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 errrors.

Future work :

The semantic rules specified in the yacc file takes care of

1. multiple definitions

2. undeclared variables

3. type mismatch

4. scope.

We will try to implement type checking between the format specifier used and the type of the variable.

References :

1. Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

2. https://web.cs.wpi.edu/~kal/

3. https://www.tutorialspoint.