National Institute of Technology Karnataka Surathkal

Project 3: Semantic Analyzer for C Language

Date:28/03/2018

Submitted To: Ms.Sushmita G

Group Members:

S. No	Name	E-mail	Roll Number
1	Naladala Indukala	15co230.indukala@nitk.edu.in	15CO230
2	R. Aparna	15co236.aparna@nitk.edu.in	15CO236

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. A lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of regular expressions. A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. Syntactic analysis, or parsing, is needed to determine if the series of tokens given are appropriate in a language - that is, whether or not the sentence has the right shape/form. However, not all syntactically valid sentences are meaningful. Further semantic analysis has to be done.Semantic Analyzer will check actual meaning of the statement parsed in parse tree. It can compare information in one part of a parse tree to that in another part (e.g., checkif reference to variable agrees with its declaration; check if parameters of a function call match the parameters of function definition).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, 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 error messages are given in case of multiple definitions, type mismatch in assignment, re-declaration of variables, usage of undeclared variables. The following tasks are also performed: function return type checking, parameter matching of function call and function definition. The symbol table contains details like function-definition flag, level of nesting of variables, scope and value of the variables. The function parameters table contains function details like number of parameters and type of parameters.

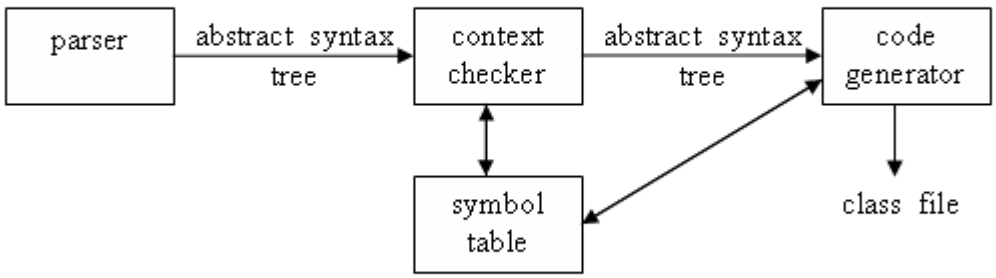
Table of Contents

Introduction

Semantic Analysis

Semantic analysis or context sensitive analysis is a process in compiler construction, which gathers necessary semantic information from the source code. 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., check if reference to variable agrees with its declaration; check if parameters of a function call match the parameters of function definition).It is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. Semantic analysis logically follows the parsing phase and logically precedes the code generation phase. Semantic analyser performs scope resolution, type checking, and array dimension checking. It gives appropriate error message in case of:

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



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 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.

THE DECLARATIONS SECTION: may contain the following items.

- Declarations of tokens. Yacc requires token names to be declared 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:

Non-terminal : sentential form

| sentential form

.....

| sentential form

;

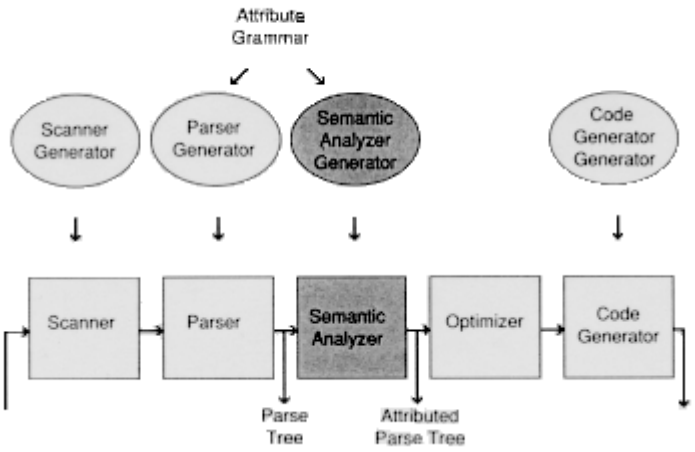
Actions may be associated with rules and are executed when the associated sentential form is matched.

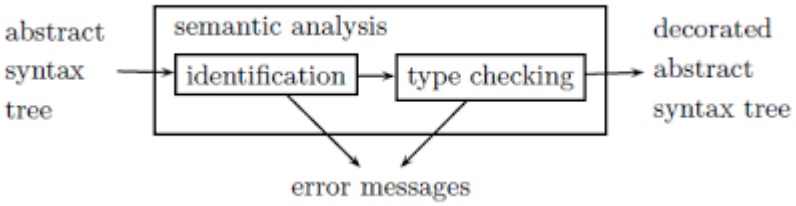
C Program

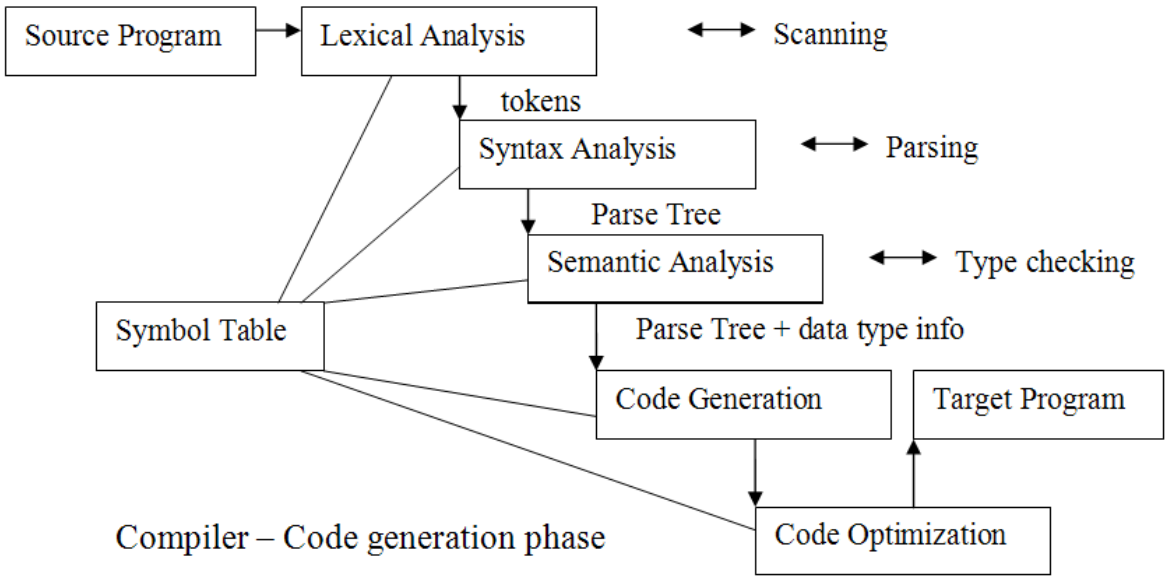
This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned in account. Finally, a file called lex.yy.c is generated, which when executed recognizes the tokens present in the C program which was given as an input. The 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 : (scanner.l file)

```
%{
int count=0;
%}

alpha [A-Za-z]
digit [0-9]
sign [+-]

%%
[ \t] ;
\n {yylineno++;}
"{" {open1(); count++; return '{';}
"}" {close1(); count--; 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 returnRETURN;
printf return PRINT;
{alpha}({alpha}|{digit})* {yylval.str=strdup(yytext); return ID;}
{sign}?{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>
int index1=0;
int function=0;
#include "symbol.c"
inti=1;
int j=8;
int stack[100];
int end[100];
intarr[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
%right '='
```

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

%union {
intival;
char *str;
}

%%

start : Function start
| PREPROC start
| Declaration start
| Function1 start
|
;

Declaration1 : Type ID ';'
{ if(!lookup($2))
{
intcurrscope=stack[index1-1];
intprevious_scope=returnscope($2,currscope);
if(currscope==previous_scope)
printf("\nError : Redeclaration of %s : Line %d\n",$2,printline());
else
{
insert_dup($2,$1,currscope,nesting());
}
}
else
{
int scope=stack[index1-1];
insert($2,$1,0,nesting());
insertscope($2,scope);
}
} ;

Function1 : Type ID '(' ')' ';' {
insert($2,FUNCTION,0,nesting());
```

```
insert($2,$1,0,nesting());
}
| Type ID '(' Argument ')' ';' {
insert($2,FUNCTION,0,nesting());
insert($2,$1,0,nesting());
} ;

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")))
printf("Error : Type mismatch in redeclaration of %s : Line
%d\n",$2,printline());
else
{
if(!lookup($2))
{
st[location($2)].flag=1;
}
else
{
insert($2,FUNCTION,1,nesting());
insert($2,$1,1,nesting());
}
}
}
| Type ID '(' Argument ')' 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")))
printf("Error : Type mismatch in redeclaration of %s : Line
```

```
%d\n",$2,printline());
else
{
if(!lookup($2))
{
function++;
st[location($2)].flag=1;
}
else
{
insert_func_name(function,$2);
function++;
insert($2,FUNCTION,1,nesting());
insert($2,$1,1,nesting());
}

}
}
;

Argument : Type ID ',' Argument {insert_func_param(function,$1);
int scope=stack[index1-1]+1;
insert($2,$1,0,nesting()+1);
insertscope($2,scope);}
| Type ID {insert_func_param(function,$1);int scope=stack[index1-1]+1;
insert($2,$1,0,nesting()+1);
insertscope($2,scope);};

Type : INT
| FLOAT
| VOID
;

CompoundStmt : '{' StmtList '}'
;

StmtList : StmtListstmt
| 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
| function_call
;

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

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

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

expr1 : expr1 LE expr1
| assignment1
;

constant_list: consttype ',' consttype;

constant_list2: consttype ',' consttype ',' consttype;

function_call: ID '=' ID '(' constant_list ')' ';' {int
k=lookup_func($3);if(k==-1)
printf(" \nUndefined function : Line %d\n",printline());
else
{
if(number_param(k)!=2)
```

```
printf("\nNumber of parameters is invalid : Line %d\n",printline());
}
}
|
ID '=' ID '(' constant_list2 ')' ';' {int k=lookup_func($3);if(k==-1)
printf(" \nUndefined function : Line %d\n",printline());
else
{
if(number_param(k)!=3)
printf("\nNumber of parameters is invalid : Line %d\n",printline());
}
}
|
ID '=' ID '(' consttype ')' ';' {int k=lookup_func($3);if(k==-1)
printf(" \nUndefined function : Line %d\n",printline());
else
{
if(number_param(k)!=1)
printf("\nNumber of parameters is invalid : Line %d\n",printline());
}
}
;

assignment1 : ID '=' assignment1
{
intsct=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))
{
intcurrscope=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;
intsct=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 ';'
{ if(!lookup($2))
{
intcurrscope=stack[index1-1];
intprevious_scope=returnscope($2,currscope);
if(currscope==previous_scope)
printf("\nError : Redeclaration of %s : Line %d\n",$2,printline());
else
{
insert_dup($2,$1,currscope,nesting());
}
}
else
{
int scope=stack[index1-1];
insert($2,$1,0,nesting());
insertscope($2,scope);
}
}
| Type ID '=' consttype ';'
{
if( (!(strspn($4,"0123456789")==strlen($4))) && $1==258)
printf("\nError : Type Mismatch : Line %d\n",printline());
if(!lookup($2))
{
intcurrscope=stack[index1-1];
intprevious_scope=returnscope($2,currscope);
if(currscope==previous_scope)
printf("\nError : Redeclaration of %s : Line %d\n",$2,printline());
else
```

```
{
insert_dup($2,$1,currscope,nesting());
check_scope_update($2,$4,stack[index1-1]);
}
}
else
{
int scope=stack[index1-1];
insert($2,$1,0,nesting());
insertscope($2,scope);
check_scope_update($2,$4,stack[index1-1]);
}
}
| assignment1 ';' {
if(!lookup($1))
{
intcurrscope=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 '[' NUM ']' ';' {
insert($2,ARRAY,0,nesting());
int scope=stack[index1-1];
insertscope($2,scope);
insert($2,$1,0,nesting());
if((int)(atof($4))<1)
{
printf("\nError : Array of size less than 1 : Line %d\n",printline());
}
else
storevalue($2,$4,stack[index1-1]);
}
```

```
| STRUCT ID '{' Declaration1 '}' ';' {
insert($2,STRUCT,0,nesting());
}
| error
;

%%
#include "lex.yy.c"
#include<ctype.h>
int main(intargc, 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 nesting()
{
return count;
}
intprintline()
{
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>
structsym
{
intsno;
char token[100];
int type[100];
inttn;
float fvalue;
int scope;
int flag;
int nest;
}st[100];
structfunction_table
{
int type[100];
intnum;
char name[100];
```

```
}func[20];

int n=0,arr[10];

intlookup_func (char *a)
{
inti;
for(i=0;i<function;i++)
{
if(strcmp(a,func[i].name)==0)
{
return i;
}
}
return -1;
}

intnumber_param (inti)
{
return func[i].num;
}
intreturntype_func(intct)
{
return arr[ct-1];
}
void storereturn( intct, intreturntype )
{
arr[ct] = returntype;
return;
}
void insertscope(char *a,int s)
{
inti;
for(i=0;i<n;i++)
{
if(!strcmp(a,st[i].token))
{
st[i].scope=s;
break;
```

```
}
}
}
intreturnscope(char *a,intcs)
{
inti;
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)
{
inti;
for(i=0;i<n;i++)
{
if( !strcmp( a, st[i].token) )
return 0;
}
return 1;
}
int location(char *a)
{
inti;
for(i=0;i<n;i++)
{
if( !strcmp( a, st[i].token) )
return i;
}
return 1;
```

```
}
intreturntype(char *a,intsct)
{
inti;
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,intsc)
{
inti,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,ints_c)
{
inti;
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 flag, int count)
{
inti;
if(lookup(name))
{
strcpy(st[n].token,name);
st[n].tn=1;
st[n].type[st[n].tn-1]=type;
st[n].sno=n+1;
st[n].flag=flag;
st[n].nest=count;
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, inttype,ints_c, int count)

{

strcpy(st[n].token,name);

st[n].tn=1;

st[n].type[st[n].tn-1]=type;

st[n].sno=n+1;

st[n].nest=count;

st[n].scope=s_c;

n++;

return;

}

void insert_func_name(int function, char* name)

{

strcpy(func[function].name,name);

}

void insert_func_param(int function, intparam)

{

func[function].num++;

func[function].type[func[function].num-1]=param;

}

void print()

{

inti,j;

printf("\nSymbol Table\n\n");

printf("\nSNo.\tToken\tValue\tScope\tNesting\tType\n");

for(i=0;i<n;i++)

{

if(st[i].type[0]==258||st[i].type[0]==273)

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

else

printf("%d\t%s\t%.1f\t%d\t%d",st[i].sno,st[i].token,st[i].fvalue,st[i].scop
e,st[i].nest);
```

```
for(j=0;j<st[i].tn;j++)
{
if(st[i].type[j]==258)
printf("\tINT");
else if(st[i].type[j]==259)
printf("\tFLOAT");
else if(st[i].type[j]==271)
printf("\tFUNCTION\t%d",st[i].flag);
else if(st[i].type[j]==273)
printf("\tARRAY");
else if(st[i].type[j]==275)
printf("\tSTRUCT");
else if(st[i].type[j]==260)
printf("\tVOID");
}
printf("\n");
}
printf("\nFunctionParamaters Table\n\n");
printf("\nSNo.\tName\tNo. of Param\tParameters\n");
for(i=0;i<function;i++)
{
printf("%d\t%s\t%d",i+1,func[i].name,func[i].num);
for(j=0;j<func[i].num;j++)
{
if(func[i].type[j]==258)
printf("\tINT");
else if(func[i].type[j]==259)
printf("\tFLOAT");
}
printf("\n");
}
return;
}
```

Implementation Explanation

Files

1. **scanner.l:** Lexfile which defines all the terminals of the productions stated in the yacc file. Itcontains regular expressions.
2. **parser.y**: Yacc file is where the productions for all the loops, selection and conditional statements and expressions are mentioned. This file also contains semantic rules defined against productions.
3. **symbol_table.c**: It is the C file which generates the symbol table. It is included in the yacc file.
4. **test_1.c:** The input C code which will be parsed and checked for semantic correctness by executing the lex and yacc files.

Yacc file has productions to check the following:

- Preprocessor directives
- Function definition
- Function declaration
- Function call
- Compound statements
- Nested compound statements
- Nested if else
- Nested while
- Nested do while
- Variable definition and declaration
- Assignment operations
- Arithmetic operations
- Structures and Arrays

Our semantic analyser 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 return type of functions
- Update value of variable based on scope
- Check if function call is proper (parameter checking)

Scope

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that, the variablecannot be accessed. There are three places where variables can be declared in C programming language −

- Inside a function or a block (called local variables)
- Outside of all functions (called global variables)
- In the definition of function (called formal parameters)

Explanation of Semantic Functionalities Implementation

Type mismatch in return type of functions

- The data type of the variable/constant mentioned in the return statement is compared with the return type of the function mentioned in definition
- If both are not the same, there is a type mismatch in the return type of the function
- The data type of constant in return statement is determined using strspn(), which is used to get span of character set in string

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

Symbol table entry for functions

- Function definition flag is set to 0 if function is only declared and not defined
- If the function is defined, the flag is set to 1
- When a function has arguments, a new entry is created in the “Function Parameters Table”. For each such function, number of parameters and type of parameters is mentioned

```
Function1 : Type ID '(' ')' ';' {
            insert($2,FUNCTION,0,nesting());
            insert($2,$1,0,nesting());
            }
            | Type ID '(' Argument ')' ';' {
            insert($2,FUNCTION,0,nesting());
            insert($2,$1,0,nesting());
            } ;
```

```
Argument : Type ID ',' Argument {insert_func_param(function,$1);int scope=stack[index1-1]+1;
                        insert($2,$1,0,nesting()+1);
                        insertscope($2,scope);}
           | Type ID {insert_func_param(function,$1);int scope=stack[index1-1]+1;
                        insert($2,$1,0,nesting()+1);
                        insertscope($2,scope);};
```

Scope of variable

- Scope of variable is determined through two functions, open () and close ()
- Stack is maintained to determine scope
- open1() is called by scanner.l when it encounters {
- close1() is called by scanner.l when it encounters }
- The scope of variable is maintained in the symbol table
- Whenever a variable is encountered a program, its scope is checked to see if the variable name exists and if it is bound to a specific value

```
"{"     {open1(); count++; return '{';}
"}"     {close1(); count--; return '}';}
```

```
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);
                }
                }
```

Undeclared variable

- Whenever a variable is used, it is checked whether the variable has been declared before
- This is done through lookup () function of symbol table
- lookup () returns 0 if variable is found; else it returns 1

```
| 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());
                    }
```

Arrays

- Array with dimension less than 1 is not allowed
- Value of array in the symbol table is its dimension

```
| Type ID '[' NUM ']' ';' {
                                    insert($2,ARRAY,0,nesting());
                                    int scope=stack[index1-1];
                                    insertscope($2,scope);
                                    insert($2,$1,0,nesting());
                                    if((int)(atof($4))<1)
                                    {
                                    printf("\nError : Array of size less than 1 : Line %d\n",printline());
                                    }
                                    else
                                    storevalue($2,$4,stack[index1-1]);
                            }
```

Function Calling

- When a function is called, it is checked if the function has been declared before
- Later, it is checked if the number of input parameters being passed through function call is the same as the number of arguments in the function's definition
- If not, an error message is displayed

```
function_call: ID '=' ID '(' constant_list ')' ';' {int k=lookup_func($3);if(k==-1)
printf(" \nUndefined function : Line %d\n",printline());
else
{
if(number_param(k)!=2)
printf("\nNumber of parameters is invalid : Line %d\n",printline());
}
}
```

Key Symbol Table Functions

lookup()
Returns 1 if variable is not present in symbol table, else returns 0

returntype_func()
Used to determine return type of function

storereturn()
Used to store return type of function

storevalue()
Used to store value of a variable; accepts variable name and variable scope as arguments

insert()
Accepts variable name and variable data type as arguments. If variable already exists in symbol table, it appends the type to the variable's entry in symbol table (used to enter return type of functions; as the function name will have two types: FUNCTION and RETURN_TYPE). Else, new entry is created for the variable.

check_scope_update()
Checks the scope of variable whose value has been changed,searches for it in the symbol table and updates the value in the symbol table

insert_dup()
Used to insert a variable in symbol table when there exists another variable with the same name but different scope in the symbol table

insert_scope()
Used to insert scope of a variable in the symbol table. This follows the insert() function, which is used to insert variable and its type in the symbol table. insert_scope() accepts top of stack as input, which is the scope of the variable.

return_scope()
Accepts current scope and variable name as input and determines what is the scope of the variable (In case of duplicate variables (having same name), it returns the maximum scope among variables of a particular name with scope less than current scope)

insert_func_name()
Used to insert function name in function parameters table

insert_func_param()
Used to insert type of parameters passed to the function. The number of parameters passed to function is also set.

lookup_func()
Returns -1 if function has not been declared; else returns position of function in the function parameters table

number_param()
Returns the number of parameters of the function by searching in the function parameters table

print ()
Used to print symbol table entries including name, scope, value, type, nesting (determined indirectly through stack contents and scope of variable, i.e., number of opening braces with no corresponding closing brace). It also prints the function parameters table.

Test Cases

Test Case #1

Input

```
1
2 #include <stdio.h>
3
4 void food()
5 {
6         return 9.8;
7 }
8
9 int gh()
10 {
11         int a=9;
12         return 8.7;
13 }
14
15
16 int main()
17 {
18         int a =5;
19         {
20                 int a=4;
21                 a=6;
22                 int x;
23         }
24         a=7;
25         return ;
26 }
27
28
```

Output

```
Error : Type mismatch : Line 7

Error : Type mismatch : Line 13

Error : Type mismatch : Line 26
Parsing done

Symbol Table


SNo.  Token   Value   Scope   Nesting Type            Flag    Function_return_type
1     food    0.0     0       0       FUNCTION        1       VOID
2     a       9       2       1       INT
3     gh      0.0     0       0       FUNCTION        1       INT
4     a       7       3       1       INT
5     a       6       4       2       INT
6     x       0       4       2       INT
7     main    0.0     0       0       FUNCTION        1       INT

Function Paramaters Table


SNo.  Name    No. of Param    Parameters
```

Test Case #2

Input

```
1
2 #include <stdio.h>
3
4 void scanf()
5 {
6         return;
7 }
8
9 int main()
10 {
11         int a =5;
12         int arr[3];
13
14         while(x <= 3)
15         {
16                 int q=0;
17                 q=2;
18                 {
19                         int x=4;
20                 }
21         }
22         return 0;
23 }
24
25
26
```

Output

```
Error : Type mismatch in redeclaration of scanf : Line 7

Undeclared Variable x : Line 14
Parsing done

Symbol Table


SNo.  Token   Value   Scope   Nesting Type           Flag    Function_return_type
1     a       5       2       1       INT
2     arr     3       2       1       ARRAY   INT
3     q       2       3       2       INT
4     x       4       4       3       INT
5     main    0.0     0       0       FUNCTION       1       INT

Function Paramaters Table


SNo.  Name    No. of Param    Parameters
```

Test Case #3

Input

```
 1
 2 #include <stdio.h>
 3 int main()
 4 {
 5         int arr[-4];
 6         int a =5;
 7         float b=6.7;
 8         int c = 4;
 9         c=6 + 7;
10         a=c + 5;
11         a=b + c;
12
13         {
14                 a=4;
15         }
16
17         printf("Depicts addition of float with int type");
18         return 0;
19 }
20
```

Output

```
Error : Array of size less than 1 : Line 5

Error : Type Mismatch : Line 11
Parsing done

Symbol Table


SNo.  Token   Value   Scope   Nesting Type          Flag    Function_return_type
1     arr     0       1       1       ARRAY   INT
2     a       4       1       1       INT
3     b       6.7     1       1       FLOAT
4     c       4       1       1       INT
5     main    0.0     0       0       FUNCTION      1       INT

Function Paramaters Table


SNo.  Name    No. of Param    Parameters
```

Test Case #4

Input

```
1 #include <stdio.h>
2
3 int sum(int a,int b)
4 {
5         int c;
6         c= 7 + 3;
7         c=1;
8         return 9;
9 }
10
11 int main()
12 {
13         int a=5.4;
14         int b;
15         b=sum(3,4);
16         b=wum(3,4);
17         b=sum(3,4,2);
18         do
19         {
20                 int q=4;
21                 q= q + 7 ;
22
23                 printf("function call matching");
24
25         }
26         return 8;
27 }
```

Output

```
Error : Type Mismatch : Line 13

Undefined function : Line 16

Number of parameters is invalid : Line 17

Line 26 : syntax error return
Parsing done

Symbol Table


SNo.    Token   Value   Scope   Nesting Type
1       b       0       1       1       INT
2       a       0       1       1       INT
3       c       1       1       1       INT
4       sum     0.0     0       0       FUNCTION        1       INT
5       a       5       2       1       INT
6       b       0       2       1       INT
7       q       4       3       2       INT
8       main    0.0     0       0       FUNCTION        1       INT

Function Paramaters Table


SNo.    Name    No. of Param    Parameters
1       sum     2       INT     INT
```

Test Case #5

Input

```
#include <stdio.h>

int foo(int x,int y)
{
        int x;
        int m=4;
        int m;
        return 1;
}

struct qwe
{
        int r;
};


int main()
{

        int b=5;
        a,b=6;

        {
                int e=5;
                {
                        e=7;
                        int new=0;
                }
        }

        return 2;
}
```

Output

```
Error : Redeclaration of x : Line 5

Error : Redeclaration of m : Line 7

Undeclared Variable a : Line 22

Error : Undeclared Variable a : Line 22
Parsing done

Symbol Table


SNo.  Token   Value   Scope   Nesting Type            Flag    Function_return_type
1     y       0       1       1       INT
2     x       0       1       1       INT
3     m       4       1       1       INT
4     foo     0.0     0       0       FUNCTION        1       INT
5     r       0       2       1       INT
6     qwe     0.0     0       0       STRUCT
7     b       6       3       1       INT
8     e       7       4       2       INT
9     new     0       5       3       INT
10    main    0.0     0       0       FUNCTION        1       INT

Function Paramaters Table


SNo.  Name    No. of Param    Parameters
1     foo     2       INT     INT
```

Test Case #6

Input

```
#include <stdio.h>

int diff();


int find(int q,int p)
{
        int x=8;

        return 3;
}

int main()
{

        th=7;
        int arr[6];
        {
                int c=4;
        }

        if(1)
        {
                printf("Yes\n");
        }

        c=0;
        return 0;
}

```

Output

```
Error : Undeclared Variable th : Line 17

Error : Variable c out of scope : Line 28
Parsing done

Symbol Table


SNo.  Token   Value   Scope   Nesting Type           Flag    Function_return_type
1     diff    0.0     0       0       FUNCTION       0       INT
2     p       0       1       1       INT
3     q       0       1       1       INT
4     x       8       1       1       INT
5     find    0.0     0       0       FUNCTION       1       INT
6     arr     6       2       1       ARRAY   INT
7     c       4       3       2       INT
8     main    0.0     0       0       FUNCTION       1       INT

Function Paramaters Table


SNo.  Name    No. of Param    Parameters
1     find    2       INT     INT
```

Results

The lex (scanner.l) and yacc (parser.y) codes are compiled and executed by the following terminal commands to parse the given input file (test_1.c)

lexscanner.l

yaccparser.y

gccy.tab.c -ll -ly-w

./a.out test_1.c

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

Future work

The semantic rules specified in the yacc file take care of

- Multiple definitions

- Reserved identifier misuse
- Undeclared variables
- Type mismatch
- Scope resolution
- Actual and formal parameter mismatch

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

References

- Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
- https://web.cs.wpi.edu/~kal
- https://www.tutorialspoint.com