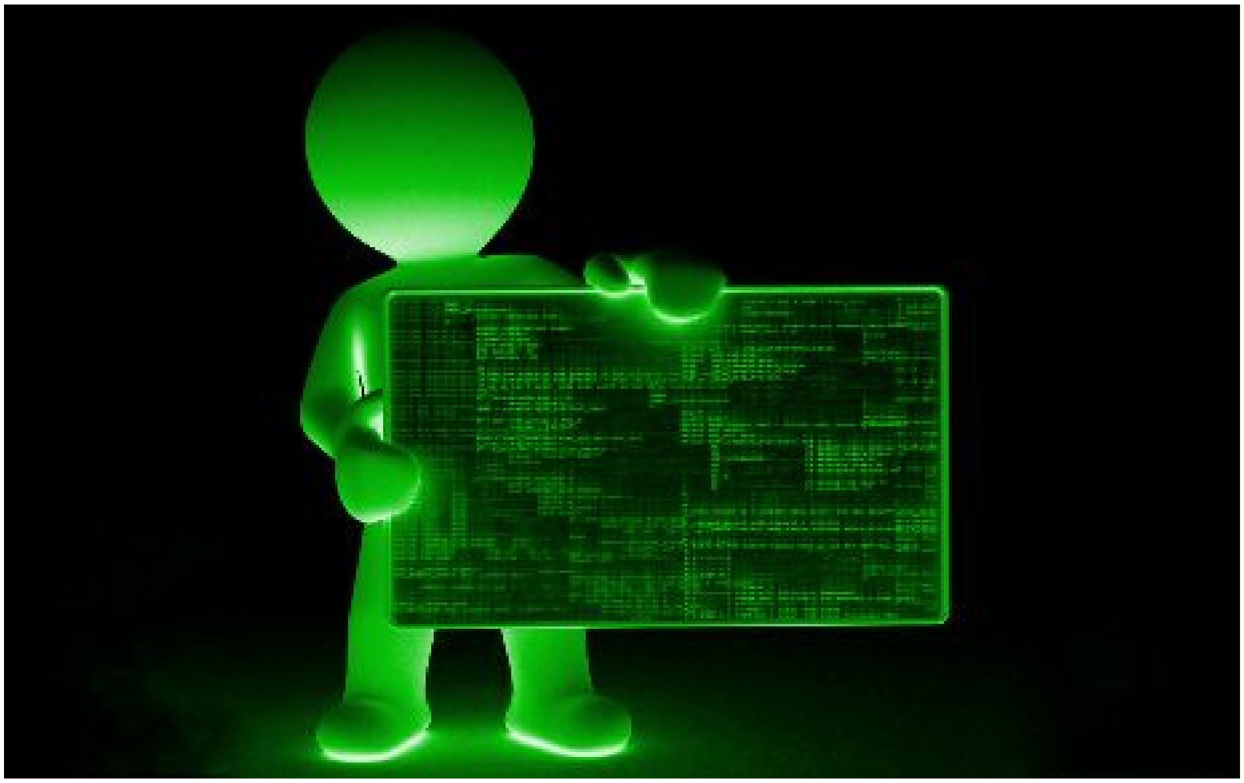


Semantic Analyzer

COMPILER DESIGN

PROJECT - 3

28.03.2018

Submitted by

Saurabh Yadav, 15CO145

Kiran S, 15CO125

Table of Contents

Phase 3 - Semantic Analyzer

Introduction

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

CFG + semantic rules = Syntax Directed Definitions

For example:
int a = “value”; should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. 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

Semantic Errors

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.

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.
Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

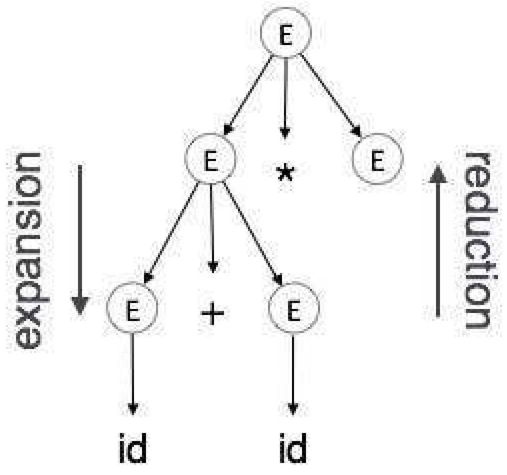
Example:
E → E + T { E.value = E.value + T.value }
The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.
Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.
Expansion : When a non-terminal is expanded to terminals as per a grammatical rule

Reduction : When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

Semantic analysis uses Syntax Directed Translations to perform the above tasks.

Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).

Semantic analyzer attaches attribute information with AST, which are called Attributed AST.



Lex Code

```
%{
int nest_cmt=0;
extern void insert(char *yytext,char type,char *data_type,struct
token_list *lexeme);
char *temp;
extern char data_type[20],const_type[20];
extern int nest,scope;
%}

alpha [a-zA-Z_]
digit [0-9]
exp [Ee][+-]?{digit}+
singlecomm (\/\/.*)
```

```
comm_beg (\/\*)
comm_end (\*\/)

%x COMM

%%

<*>\n {yylineno = yylineno + 1;}

{singlecomm} {}

{comm_beg} {
BEGIN(COMM);
nest_cmt++;
}

{comm_end} { }

<COMM>{comm_beg} {nest_cmt++;}

<COMM>{comm_end} {
if(nest_cmt>0) nest_cmt--;
if(nest_cmt==0) BEGIN(INITIAL);
}

<COMM>. {}

[ \t] ;
```

```
"auto" {return AUTO;}

"break" {return BREAK;}

"case" {return CASE;}

"char" {return CHAR;}

"const" {return CONST;}

"continue" {return CONTINUE;}

"default" {return DEFAULT;}

"do" {return DO;}

"double" {return DOUBLE;}

"else" {return ELSE;}

"enum" {return ENUM;}

"extern" {return EXTERN;}

"float" {return FLOAT;}

"for" {return FOR;}

"goto" {return GOTO;}

"if" {return IF;}

"int" {return INT;}

"long" {return LONG;}

"register" {return REGISTER;}

"return" {return RETURN;}

"short" {return SHORT;}

"signed" {return SIGNED;}

"sizeof" {return SIZEOF;}

"static" {return STATIC;}

"struct" {return STRUCT;}

"switch" {return SWITCH;}

"typedef" {return TYPEDEF;}

"union" {return UNION;}
```

```
"unsigned" {return UNSIGNED;}

"void" {return VOID;}

"volatile" {return VOLATILE;}

"while" {return WHILE;}

printf {return PRINTF;}

scanf {return SCANF;}



"#include"(.)* ;

"#define"(.)* ;


{digit}+ {yylval.s=strdup(yytext); strcpy(const_type,"int"); return
NUM;}

{digit}*"."{digit}+({exp})? {yylval.s=strdup(yytext);
strcpy(const_type,"double"); return NUM;}

{digit}+"."{digit}*({exp})? {yylval.s=strdup(yytext);
strcpy(const_type,"double"); return NUM;}

{alpha}({alpha}|{digit})* {yylval.s=strdup(yytext); return ID;}


\"(\\.|[^\\"])*\" {return STRING;}



"++" return INC;

"--" return DEC;

"<=" return LE;

">=" return GE;

"==" return EQ;

"!=" return NE;

">" return GT;

"<" return LT;
```

```
"." return DOT;

"{" { nest++;
return yytext[0];
}

"}" {
nest--;
return yytext[0];
}

";" {strcpy(data_type,"-");  strcpy(const_type,"-");return
yytext[0];}

. return yytext[0];
%%
```

Yacc Code

```
%{
#include <stdio.h>
#include <stdlib.h>
#include<string.h>
int count=0,mainscope=9999,flag=0,total=0,call=0,state=0;
extern char *yytext;
extern char *temp;
struct token_list
{
     char *name,token_type[25],dt[20],dimension[20];
     struct token_list *next;
     int nest,scope,parameter;
}*header,*footer;
char data_type[20],const_type[20],func_type[20],fname[20];
void lookup(char *yytext,char type,char *data_type);
void insert(char *yytext,char type,char *data_type,struct token_list
*lexeme);
void check(char *yytext);
int nest=0,scope=0;
int yyerror(char*);
void returntypecheck(char *yytext);
int yylex();
void checkforvoid();
void copy();
void parametercheck(char *s);
void arrdim(char *arr,char *dim);
void typeforvoid();
void typecheck(char *v1, char * v2);

%}

%token AUTO BREAK CASE CHAR CONST CONTINUE DEFAULT DO DOUBLE ELSE
ENUM EXTERN
%token FLOAT FOR GOTO IF INT LONG REGISTER RETURN SHORT SIGNED SIZEOF
STATIC
%token STRUCT SWITCH TYPEDEF UNION UNSIGNED VOID VOLATILE WHILE
%token PRINTF SCANF DOT INC DEC

%union
{
     char *s;
```

```
}

%token <s>ID <s>NUM <s>STRING

%right '='
%left AND OR
%left '<' '>' LE GE EQ NE LT GT
%left '+' '-'
%left '*' '/' '%'
%right DEC
%left INC

%nonassoc IFX
%nonassoc ELSE

%%

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

/* Declaration block */
Declaration: Type Assignment ';'
      | Type ArrayUsage ';'
      | error
      ;
```

```
/* Assignment block */

Assignment: identifier '=' arith {typecheck($<s>1,$<s>3);} ','
Assignment

      | identifier '=' arith    {typecheck($<s>1,$<s>3);}

      | identifier '=' FunctionCall

      | identifier '=' ArrayUsage

      | ArrayUsage '=' arith

      | identifier ',' Assignment

      | identifier ',' ArrayUsage

      | '(' Assignment ')'

      | '-' '(' Assignment ')'

      | '-' number

      | number

      | identifier

      ;


identifier:ID   {lookup($1,'i',data_type);}

;


number:NUM {/*lookup($1,'c',const_type);*/}

;


arith: ID '+' arith  {check($1);}

      | ID '-' arith  {check($1);}

      | ID '*' arith  {check($1);}

      | ID '/' arith  {check($1);}

      | '-' ID   {check($2);}

      | NUM '+' arith {check($1);}

      | NUM '-' arith {check($1);}

      | NUM '*' arith {check($1);}
      | NUM '/' arith {check($1);}

      | ID      {check($1); $<s>$=$1;}

      | NUM

;


/* Function Call Block */

FunctionCall : ID'('')'         {check($1);parametercheck($1);
call=0;}

      | ID'('argslis')'    {check($1);parametercheck($1); call=0;}

      ;


argslis:argslis ',' args

|     args

;


args: number          {call++;}

|     identifier {call++;}

;

/* Array Usage */

ArrayUsage : identifier'['Assignment']' {if(atoi($<s>3)<=0 ||
atof($<s>3)>atoi($<s>3)) yyerror("Semantic Error Undefined Array
dimension");

                                else arrdim($<s>1,$<s>3);

                            }

      //|   identifier '[' ']'

      ;


/* Function block */

Function: Type ID {strcpy(func_type,data_type);
lookup($2,'f',data_type);} {nest++;} '(' ArgListOpt ')' {nest--;
if(state==0) copy();
```

```
                                                                            else
state=0;total=0;} CompoundStmt
      ;
ArgListOpt: ArgList
      |
      ;
ArgList:  ArgList ',' Arg
      | Arg
      ;
Arg:  Type identifier {typeforvoid(); total++;}
      ;
CompoundStmt:   '{' StmtList '}'
      ;
StmtList:  StmtList Stmt
      |
      ;
Stmt:WhileStmt
      | Declaration
      | ForStmt
      | IfStmt
      | PrintFunc
      | ScanFunc
      | ReturnStmt
      | CompoundStmt
      | DoWhileStmt
      | FunctionCall ';'
      | Expr ';'
      | ';'
      ;
```

```
ReturnStmt: RETURN Expr ';' {returntypecheck($<s>2);}
      | RETURN ';'    {checkforvoid();}
;

/* Type Identifier block */
Type: INT        {strcpy(data_type,"int");}
      | LONG INT {strcpy(data_type,"int");}
      | LONG LONG INT {strcpy(data_type,"int");}
      | FLOAT         {strcpy(data_type,"float");}
      | CHAR          {strcpy(data_type,"char");}
      | DOUBLE   {strcpy(data_type,"double");}
      | LONG DOUBLE   {strcpy(data_type,"double");}
      | VOID          {strcpy(data_type,"void");}
      ;

FE:
|     Expr
;

/* Loop Blocks */
WhileStmt: WHILE '(' Expr ')' Stmt
      ;

/* For Block */
ForStmt: FOR '(' FE ';' FE ';' FE ')' Stmt
      ;

DoWhileStmt: DO Stmt WHILE '(' Expr ')' ';'
;
```

```
/* IfStmt Block */
IfStmt : IF '(' Expr ')' Stmt   %prec IFX
|    IF '(' Expr ')' Stmt ELSE Stmt
;

/* Print Function */
PrintFunc : PRINTF '(' str ')' ';'
     | PRINTF '(' str ',' var ')' ';'
     ;

str:STRING
{/*strcpy(const_type,"string");lookup(yytext,'s',const_type);*/}
;

var: identifier
| ArrayUsage
|    var ',' identifier
| var ',' ArrayUsage
;

ScanFunc : SCANF '(' str ',' scanvar ')' ';'
;

scanvar:'&'identifier
|    scanvar ',' '&'identifier
|    '&'identifier'['Assignment']'
|    scanvar ',' '&'identifier'['Assignment']'
;
```

```
/*Expression Block*/
Expr: Expr LE Expr
      | Expr GE Expr
      | Expr NE Expr
      | Expr EQ Expr
      | Expr GT Expr
      | Expr LT Expr
      | INC ID
      | DEC ID   %prec INC
      | ID INC   %prec DEC
      | Assignment
      | ArrayUsage
      ;
%%
#include"lex.yy.c"
void typecheck(char *v1, char * v2)
{
      struct token_list *lexeme1,*lexeme2,*ptr;
      ptr =lexeme1=lexeme2= header;
      int i;
      for(i=0;i<count;i++,ptr=ptr->next)
      {
            if(!(strcmp(lexeme1->name,v1)==0 && lexeme1->nest<=nest &&
(lexeme1->scope==0 || lexeme1->scope==scope))) lexeme1=ptr;
            if(!(strcmp(lexeme2->name,v2)==0 && lexeme2->nest<=nest &&
(lexeme2->scope==0 || lexeme2->scope==scope))) lexeme2=ptr;
            if((strcmp(lexeme1->name,v1)==0 && lexeme1->nest<=nest &&
(lexeme1->scope==0 || lexeme1->scope==scope)) &&
(strcmp(lexeme2->name,v2)==0 && lexeme2->nest<=nest &&
(lexeme2->scope==0 || lexeme2->scope==scope)))
            {
                if(strcmp(lexeme1->dt,lexeme2->dt)!=0) printf("Line
%d : Semantic Error Type Mismatch %s=%s\n",yylineno,v1,v2);
                break;
            }
        }
}
void typeforvoid()
{
        if(strcmp(data_type,"void")==0) printf("Line %d : Semantic
Error Data Type cannot be void\n",yylineno);
}
void arrdim(char *arr,char *dim)
{
        struct token_list *lexeme,*ptr;
        ptr = header;
        int i;
        for(i=0;i<count;i++,ptr=ptr->next)
        {
            lexeme=ptr;
            if(strcmp(lexeme->name,arr)==0 && lexeme->scope==scope &&
lexeme->nest==nest)
            {
                strcpy(lexeme->token_type,"Array");
                strcpy(lexeme->dimension,dim);
                break;
            }
        }
}
void parametercheck(char *s)
{
        struct token_list *lexeme,*ptr;
        ptr = header;
        int i;
        for(i=0;i<count;i++,ptr=ptr->next)
        {
            lexeme=ptr;
            if(strcmp(lexeme->name,s)==0)
            {
                if(lexeme->parameter!=call) printf("Line %d :
Semantic Error Parameter Mismatch\n",yylineno);
                break;
            }
        }
}
void checkforvoid()
{
        if(strcmp(func_type,"void")!=0) printf("Line %d : Semantic
Error Type Mismatch\n",yylineno);
}
void returntypecheck(char *yytext)
{
        struct token_list *lexeme,*ptr;
        ptr = header;
        int i;
        for(i=0;i<count;i++,ptr=ptr->next)
        {
            lexeme=ptr;
            if(strcmp(lexeme->name,yytext)==0 && lexeme->nest<=nest &&
(lexeme->scope==0 || lexeme->scope==scope) &&
strcmp(lexeme->dt,func_type)==0) break;
```

```
	}
	if(i==count) printf("Line %d : Semantic Error Type Mismatch 
%s\n",yylineno,yytext);
}
void check(char *yytext)
{
	struct token_list *lexeme,*ptr;
	ptr = header;
	int i;
	for(i=0;i<count;i++,ptr=ptr->next)
	{
		lexeme=ptr;
		if(strcmp(lexeme->name,yytext)==0 && lexeme->nest<=nest && 
(lexeme->scope==0 || lexeme->scope==scope)) break;
	}
	if(i==count) printf("Line %d : Semantic Error Undeclared 
variable %s\n",yylineno,yytext);
}

void lookup(char *yytext,char type,char *data_type)
{
	struct token_list *lexeme,*ptr;
	ptr = header;
	int i;
	for(i=0;i<count;i++,ptr=ptr->next)
	{
		lexeme = ptr;
		if(strcmp(lexeme->name,yytext)==0 && (lexeme->scope==0 || 
lexeme->scope==scope))
		{
```

```
                if(strcmp(data_type,"-")!=0 && lexeme->scope==scope
&& lexeme->nest==nest)
                {
                        printf("Line %d : Semantic Error Redeclaration
of variable %s\n",yylineno,yytext);
                        if(type=='f') state=1;
                        break;
                }
                else if(strcmp(data_type,"-")==0 && (lexeme->scope==0
|| lexeme->scope==scope) && lexeme->nest<=nest) break;
            }
        }
        if(i==count)
        {
            if(strcmp(data_type,"-")==0) printf("Line %d : Semantic
Error Undeclared Variable %s\n",yylineno,yytext);
            else insert(yytext,type,data_type,footer);
        }
}
void insert(char *yytext,char type,char *data_type,struct token_list
*lexeme)
{
        int len1 = strlen(yytext);
        char token_type[25];
        struct token_list *temp;
        switch(type)
        {
            case 'k':
                strcpy(token_type,"Keyword");
                break;
            case 'i':
                strcpy(token_type,"Identifier");
                break;

            case 's':
                strcpy(token_type,"String Literal");
                break;
            case 'c':
                strcpy(token_type,"Constant");
                break;
            case 'f':
                strcpy(token_type,"Function");
                scope++;
                strcpy(fname,yytext);
                if(strcmp(yytext,"main")==0) mainscope=scope;
                if(scope>mainscope)
                {
                    printf("Line %d : Semantic Error Function main
should be last function %s\n",yylineno,yytext);
                    exit(0);
                }
                break;
    }
    temp = (struct token_list*)malloc(sizeof(struct token_list));
    temp->name = (char*)malloc((len1+1)*sizeof(char));
    strcpy(temp->name,yytext);
    strcpy(temp->token_type,token_type);
    strcpy(temp->dt,data_type);
    strcpy(temp->dimension,"-");
    temp->nest=nest;
    temp->scope=scope;
```

```
	temp->parameter=0;
	temp->next = NULL;
	if(count==0) header=temp;
	else lexeme->next = temp;
	count++;
	//fprintf(yyout,"\n%35s %30s %30s %10d
%10d",temp->name,temp->token_type,temp->dt,temp->scope,temp->nest);
	footer=temp;
}
void copy()
{
	struct token_list *lexeme,*ptr;
	ptr = header;
	int i;
	for(i=0;i<count;i++,ptr=ptr->next)
	{
		lexeme=ptr;
		if(strcmp(lexeme->name,fname)==0)
		{
			lexeme->parameter=total;
			break;
		}
	}
}
int main(int argc, char *argv[])
{
	if(argc==1) printf("Arguments missing ! correct format :
./a.out filename \n");
	else
	{
```

```
            yyin = fopen(argv[1], "r");

            yyout = fopen("output.txt","w");


fprintf(yyout,"\t\t\t\t\t\t-----------------------------------------
--\n");

            fprintf(yyout,"\t\t\t\t\t\t\t\tSYMBOL TABLE\n");


fprintf(yyout,"\t\t\t\t\t\t-----------------------------------------
--\n");

            fprintf(yyout,"\t\tLexeme\t\tToken\t\t\tData Type\tScope\t
nest\t   Parameter\tDimension\n");

            int x=yyparse();

            struct token_list *lexeme,*ptr;

            ptr = header;

            int i;

            for(i=0;i<count;i++,ptr=ptr->next)

            {

                  lexeme=ptr;

                  fprintf(yyout,"\n%20s %20s %20s %13d %8d %10d
%10s",lexeme->name,lexeme->token_type,lexeme->dt,lexeme->scope,lexeme
->nest,lexeme->parameter,lexeme->dimension);

            }

            if(!x)

            {


                  printf("\nParsing complete\n");

            }

            else

                  printf("\nParsing failed\n");

            fclose(yyout);

            fclose(yyin);
```

```
    }
    return 0;
}

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

Code Explanation

Whenever a variable is declared, the compiler looks up in the symbol table to see if the variable has been declared earlier or not for checking redeclaration errors. If the variable is already declared, the compiler reports a redeclaration error. Otherwise, it inserts the newly declared variable to the symbol table with appropriate values. The compiler also checks for the scope of the variables. If a variable is declared twice in different scopes, the compiler stores both the variables separately in the symbol table under different scope values. The lookup() function is used to check for the variable in the symbol table and the function insert() is used to insert a newly declared variable in the symbol table.

Whenever the compiler encounters a variable in an expression, it looks up in the symbol table to check if the variable is declared or not. It reports the "Undeclared variable" error in the latter case. It also checks if the variable being used is present in the current scope or not. If the variable goes out of scope and is still being used in the program, an error is reported stating the same. The function check() is used for doing this job.

If a keyword is used as an identifier, the compiler detects it and reports a "reserved identifier error". Hence, keywords can't be used as identifiers in C code. This error is detected automatically as the keywords do not satisfy the grammar meant for the identifiers.

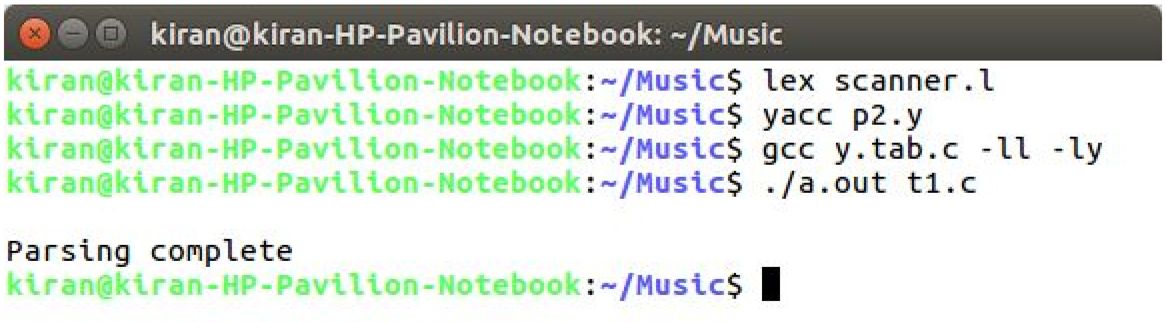
The value returned by a function is compared with the data type of the function which is retrieved from the symbol table to check type mismatch error for the function return type. Also, when a variable is assigned a value, its data type is compared with that of the RHS to see if the data types are matching or not. In case of type mismatch, errors are suitably printed. The functions typecheck() and returntypecheck() are used for checking variable type mismatch and function return type mismatch errors respectively.

Also, when a function is called, the parameters from the function call are compared with that of the parameters in the function definition. If the parameters do not match, then an error is reported on the screen. The function parametercheck() is used for comparing the formal parameters and the actual parameters.

Test Cases

Test Case - 1

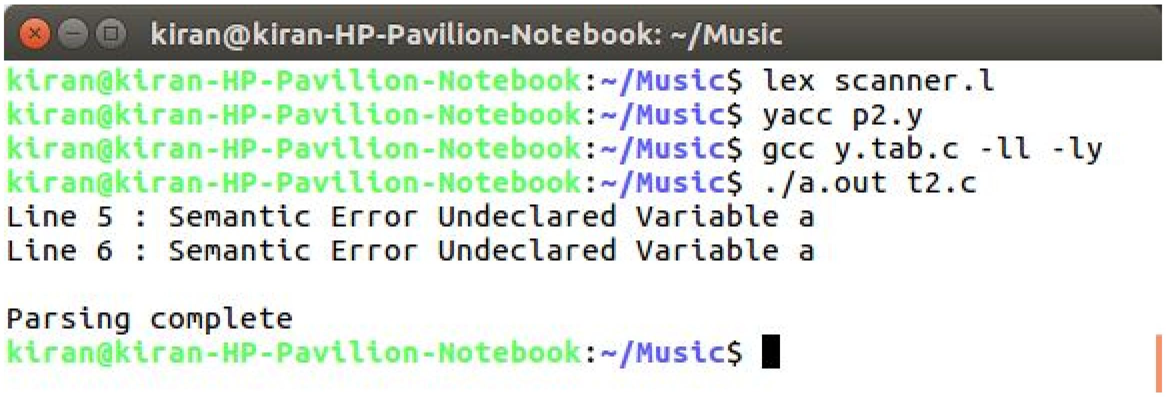
```
#include <stdio.h>

int main()

{

    int a;

    a = 5;

    printf("%d",a);

    return 0;

}
```



```
kiran@kiran-HP-Pavilion-Notebook:~/Music$ lex scanner.l
kiran@kiran-HP-Pavilion-Notebook:~/Music$ yacc p2.y
kiran@kiran-HP-Pavilion-Notebook:~/Music$ gcc y.tab.c -ll -ly
kiran@kiran-HP-Pavilion-Notebook:~/Music$ ./a.out t1.c

Parsing complete
kiran@kiran-HP-Pavilion-Notebook:~/Music$ 
```

Test Case - 2

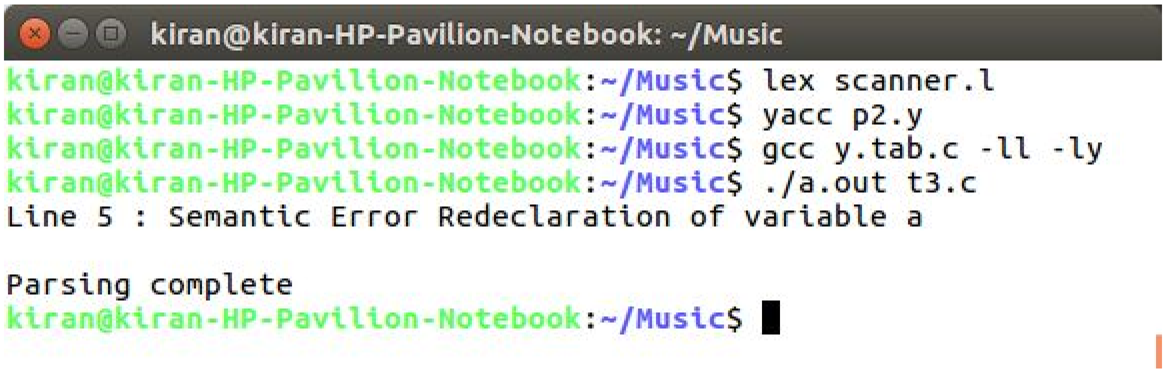
```
#include <stdio.h>

int main()

{

    a = 5;

    printf("%d",a);

    return 0;

}
```



```
kiran@kiran-HP-Pavilion-Notebook: ~/Music
kiran@kiran-HP-Pavilion-Notebook:~/Music$ lex scanner.l
kiran@kiran-HP-Pavilion-Notebook:~/Music$ yacc p2.y
kiran@kiran-HP-Pavilion-Notebook:~/Music$ gcc y.tab.c -ll -ly
kiran@kiran-HP-Pavilion-Notebook:~/Music$ ./a.out t2.c
Line 5 : Semantic Error Undeclared Variable a
Line 6 : Semantic Error Undeclared Variable a

Parsing complete
kiran@kiran-HP-Pavilion-Notebook:~/Music$ 
```

Test Case - 3

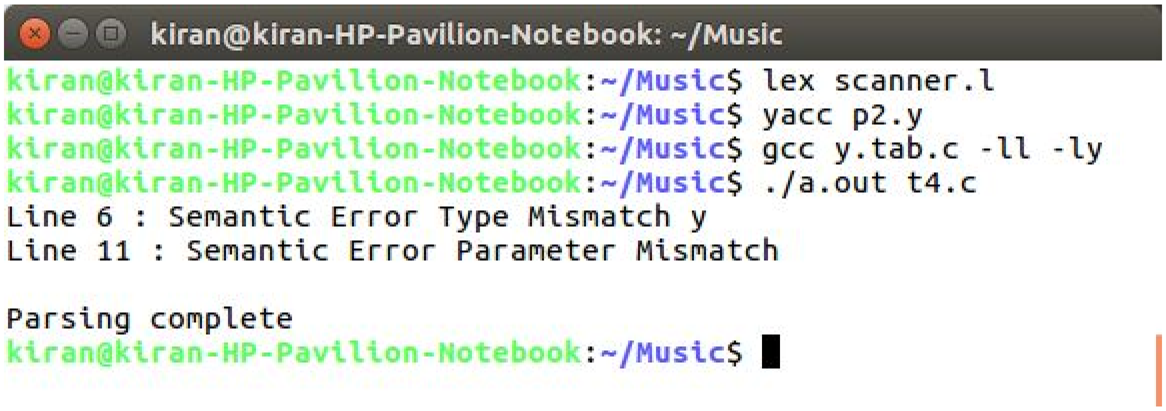
```
#include <stdio.h>

int main()

{

    int a = 5;

    float a = 6.05;

    printf("Hello World %f",a);

    return 0;

}
```



```
kiran@kiran-HP-Pavilion-Notebook: ~/Music
kiran@kiran-HP-Pavilion-Notebook:~/Music$ lex scanner.l
kiran@kiran-HP-Pavilion-Notebook:~/Music$ yacc p2.y
kiran@kiran-HP-Pavilion-Notebook:~/Music$ gcc y.tab.c -ll -ly
kiran@kiran-HP-Pavilion-Notebook:~/Music$ ./a.out t3.c
Line 5 : Semantic Error Redeclaration of variable a

Parsing complete
kiran@kiran-HP-Pavilion-Notebook:~/Music$ 
```

Test Case - 4

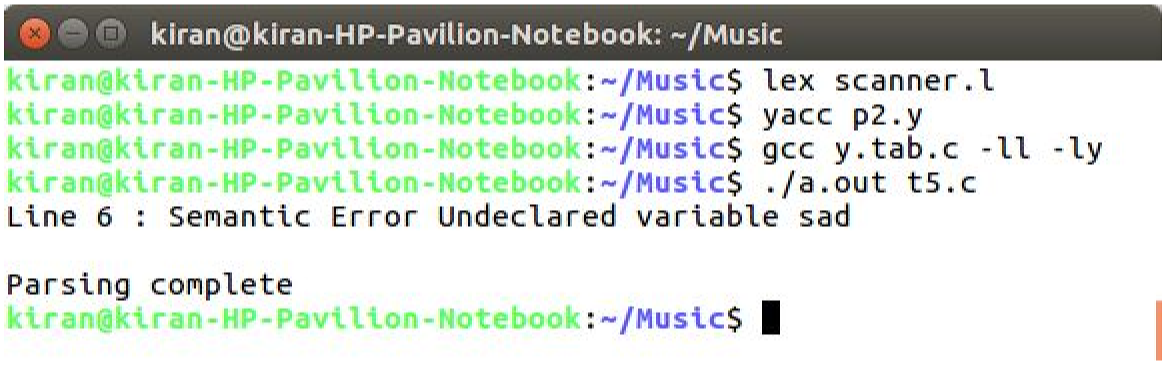
```
#include <stdio.h>

int sad(int a, int b)

{

    int x;
    float y;
    return y;
}
int main()
{
    int x = 5;
    sad(x);
    printf("Hello World");
    return 0;
}
```



```
kiran@kiran-HP-Pavilion-Notebook: ~/Music
kiran@kiran-HP-Pavilion-Notebook:~/Music$ lex scanner.l
kiran@kiran-HP-Pavilion-Notebook:~/Music$ yacc p2.y
kiran@kiran-HP-Pavilion-Notebook:~/Music$ gcc y.tab.c -ll -ly
kiran@kiran-HP-Pavilion-Notebook:~/Music$ ./a.out t4.c
Line 6 : Semantic Error Type Mismatch y
Line 11 : Semantic Error Parameter Mismatch

Parsing complete
kiran@kiran-HP-Pavilion-Notebook:~/Music$ 
```

Test Case - 5

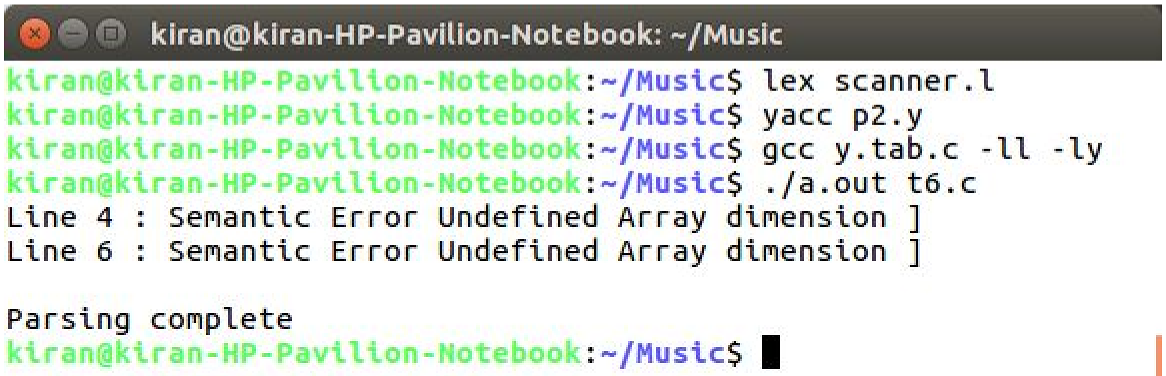
```
#include <stdio.h>
int x,a;
int main()
{
    int x = 5;
    sad();
    printf("Hello World %d",a);
    return 0;
}
```



```
kiran@kiran-HP-Pavilion-Notebook: ~/Music
kiran@kiran-HP-Pavilion-Notebook:~/Music$ lex scanner.l
kiran@kiran-HP-Pavilion-Notebook:~/Music$ yacc p2.y
kiran@kiran-HP-Pavilion-Notebook:~/Music$ gcc y.tab.c -ll -ly
kiran@kiran-HP-Pavilion-Notebook:~/Music$ ./a.out t5.c
Line 6 : Semantic Error Undeclared variable sad

Parsing complete
kiran@kiran-HP-Pavilion-Notebook:~/Music$
```

Test Case - 6

```
#include <stdio.h>

int main()

{

    int a[2.5];

    int b[-3];

    printf("Hello World %d",a[0]);

    return 0;

}
```



```
kiran@kiran-HP-Pavilion-Notebook: ~/Music
kiran@kiran-HP-Pavilion-Notebook:~/Music$ lex scanner.l
kiran@kiran-HP-Pavilion-Notebook:~/Music$ yacc p2.y
kiran@kiran-HP-Pavilion-Notebook:~/Music$ gcc y.tab.c -ll -ly
kiran@kiran-HP-Pavilion-Notebook:~/Music$ ./a.out t6.c
Line 4 : Semantic Error Undefined Array dimension ]
Line 6 : Semantic Error Undefined Array dimension ]

Parsing complete
kiran@kiran-HP-Pavilion-Notebook:~/Music$
```

Conclusion

The proposed work has been implemented successfully. We checked for the type mismatch errors (return type of function as well as type of declared variables while assigning values). We also took care of undeclared variables and reserved identifier misuse. The problem of multiple declaration of a variable in a scope is handled keeping track of the scope of each and every identifier. The number of actual and formal parameters of function are also checked and the error messages are displayed accordingly. Along with this, we enhanced the symbol table implementation. The symbol table contains the name of the token, type of the token, data type, nesting level of each identifier, scope of each identifier, number of parameters (in case of function) and array dimensions (in case of arrays). Thus, all the objectives of the project have been implemented successfully and this wraps up the Semantic Analyser in C.