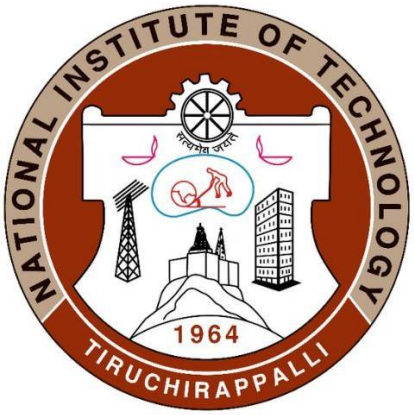
NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI



CSPC62

COMPILER DESIGN

TOPIC: C++ Compiler

LAB REPORT-3

Sub Topic : Semantic Analyzer

DONE BY:

S.No	Name	RollNo
1.	Hema Sai Dorababu K	106121065
2.	Chetan Reddy T	106121139
3.	Harshith Babu B	106121029

Introduction:

Semantic analysis acts as the semantic verifier within a compiler. It goes beyond the structural correctness checked by syntax analysis and delves into the code's meaning within the programming language's context. Imagine it as the inspector ensuring the code not only follows the grammar but also adheres to the language's logic and rules.

Type Checking:

A core function of semantic analysis is type checking. The compiler meticulously examines operations to ensure they are performed on compatible data types. Mixing data types like adding an integer and a string would raise a red flag here. Type checking involves:

- **Type Compatibility:** Verifying that operands (values used in expressions) have compatible types for the operation being performed. For example, adding two integers is allowed, but adding an integer and a boolean wouldn't be.
- **Type Conversion:** In some cases, the compiler might be able to perform implicit type conversion to make operations work. For instance, converting an integer to a float before adding them. However, semantic analysis might raise warnings for such conversions, as they can sometimes lead to unexpected behavior.
- **Type Propagation:** As the code is analyzed, type information is propagated throughout the program. This means the compiler tracks the types of variables and expressions throughout their use, ensuring consistency.

Identifier Management: Keeping Track of Names

Semantic analysis plays a crucial role in managing identifiers (variable names, function names, etc.). It ensures:

- **Variable Declaration:** Variables are declared (given a type) before they are used. This prevents errors like using a variable before it's assigned a value.
- **Scope Rules:** Variables are only accessible within their declared scope (e.g., a local variable declared within a function can't be accessed from outside that function). Semantic analysis enforces these rules.
- **Uniqueness:** Identifiers within the same scope must be unique. This ensures the code is clear and avoids confusion.

Beyond the Basics: Unveiling Other Semantic Errors

The eagle eye of semantic analysis goes beyond just type mismatches. It can detect various other semantic errors that could lead to program malfunction. These include:

- **Using Undeclared Variables:** Attempting to use a variable that hasn't been declared beforehand.
- **Misusing Reserved Keywords:** Using keywords with special meanings in the language for unintended purposes.
- **Incorrect Function Calls:** Calling functions with the wrong number or type of arguments.
- **Control Flow Issues:** Checking for errors in control flow structures like loops and conditionals. For example, using a variable before it's assigned a value within a loop.

Tools of the Trade: Symbol Table and Abstract Syntax Tree (AST)

To perform its duties effectively, semantic analysis utilizes several tools:

- **Symbol Table:** This data structure acts as a central registry, keeping track of identifiers, their types, scope information, and other relevant details. It's like a reference book for the compiler, allowing it to quickly look up information about variables and functions.
- **Abstract Syntax Tree (AST):** Often used as input for semantic analysis. The AST represents the program's structure in a tree-like form, making it easier to traverse the code and analyze its meaning. By examining the AST, the compiler can understand the relationships between different parts of the code.

The Outcome: Paving the Way for Further Stages

The result of semantic analysis can be an annotated version of the AST or the symbol table itself, enriched with type information and other semantic details. This information is crucial for subsequent compiler phases like code generation and optimization. It ensures these later stages work with a well-defined and semantically correct representation of the program.

In Conclusion:

Semantic analysis plays a vital role in ensuring a program's correctness and functionality. It bridges the gap between the code's structure and its intended meaning, laying the groundwork for a successful compilation process.

Question-1:

Syntax directed translation:

program: headers function_declarations

| EOL;

Explanation:

- This rule defines the structure of a program.
- It consists of headers followed by function declarations, separated by the "|" symbol, indicating alternatives.
- Alternatively, it can end with an EOL token if there are no headers or function declarations in the program.

function_declarations:

function_declaration function_declarations

| EOL function_declarations

| EOL

| ;

Explanation:

- Describes the structure of function declarations within the program.

- It can either be a function_declaration followed by more function_declarations or just end-of-line tokens.
- It can also end with an empty statement represented by ";" if there are no function declarations.

headers:

```
header
| header headers
| EOL headers
| EOL;
```

Explanation:

- Specifies the syntax for headers in the program.
- It can include one or more header declarations followed by more headers, separated by end-of-line tokens.
- Alternatively, it can end with an EOL token if there are no more headers.

function_declaration:

```
datatype identifier LPAREN parameter_list RPAREN LBRACE statement_list RBRACE {
    $2->dtype = $1->dtype;
    $2->value = $4->value;
    addEntry($2->value_str, $2->dtype, $2->scope, $4->value);
    addFunctionValue($2->value_str, $4->value);
};
```

Explanation:

- Defines the syntax for declaring a function.
- It starts with the data type of the function's return value followed by the function name, parameter list, and function body enclosed in braces.
- After parsing, it sets the data type and value of the function identifier ($2) to match the return type and parameter count.
- It then adds the function entry and its value to the symbol table.

parameter_list:

```
datatype identifier COMMA parameter_list {
    $2->dtype = $1->dtype;
    $2->value = $4->value + 1;
    addEntry($2->value_str, $2->dtype, $2->scope, $2->value);
}
```

```
| datatype identifier {
    $2->dtype = $1->dtype;
    $$->value = 1;
    addEntry($2->value_str, $2->dtype, $2->scope, $2->value);
}
| { $$->value = 0; };
```

Explanation:

- Describes the syntax for a list of parameters in a function declaration.
- It can either be a single parameter or multiple parameters separated by commas.
- Each parameter consists of a datatype followed by an identifier.
- After parsing, it sets the data type and value of each parameter identifier ($2) and adds it to the symbol table.

```
statement_list:
  statement statement_list
  | EOL statement_list
  | EOL
  |;
```

Explanation:

- Specifies the structure of a list of statements within a block.
- It can consist of one or more statements followed by more statements or end-of-line tokens.
- Alternatively, it can end with an empty statement if there are no more statements in the block.

```
statement:
  declaration_statement
  | assignment_statement SEMICOLON
  | for_statement
  | if_statement
  | if_else_statement
  | while_statement
  | cout_statement
  | cin_statement
  | return_statement
  | function_call SEMICOLON
  | error SEMICOLON
```

| EOL ;

Explanation:

- Defines the syntax for different types of statements.
- It can be a declaration statement, assignment statement, loop statement, control statement, input/output statement, return statement, function call, error statement, or end-of-line token.

function_call:

identifier LPAREN arg_list RPAREN {

$1->value = $3->value;

checkfnArguments($1);

}

| identifier LPAREN RPAREN ;

Explanation:

- This rule represents a function call.
- It consists of an identifier (function name), followed by a left parenthesis, an argument list (if any), and a right parenthesis.
- After parsing, the function name is stored in $1, and if arguments are present, their count is stored in $3.
- The checkfnArguments function is called to verify the correctness of the function call.

arg_list:

identifier { $$->value = 1; }

| identifier COMMA arg_list { $$->value = $3->value + 1; };

Explanation:

- This rule represents a list of arguments in a function call.
- It can have one or more arguments separated by commas.
- If there is only one argument, its count is set to 1.
- If there are more than one argument, the count is incremented by 1 for each additional argument.

if_statement:

if_x LPAREN E RPAREN LBRACE statement_list RBRACE

| if_x LPAREN E RPAREN statement

| if_statement EOL ;

Explanation:

- This rule represents an if statement.
- It can have multiple variations:
- Condition enclosed in parentheses followed by a block of statements.
- Condition enclosed in parentheses followed by a single statement.
- Continuation of an if statement on a new line.

else_statement:

else_x LBRACE statement_list RBRACE

| else_x statement

| else_statement EOL ;

Explanation:

- This rule represents an else statement.
- It can have multiple variations:
- Block of statements following the else keyword.
- Single statement following the else keyword.
- Continuation of an else statement on a new line.

if_else_statement:

if_statement else_statement ;

Explanation:

- This rule represents an if-else statement.
- It combines an if statement followed by an else statement.

while_statement:

while_x LPAREN E RPAREN LBRACE statement_list RBRACE ;

Explanation:

- This rule represents a while loop.
- It consists of the while keyword followed by a condition in parentheses and a block of statements.

for_statement:

for_x LPAREN declaration_statement E SEMICOLON E RPAREN LBRACE statement_list RBRACE ;

Explanation:

- This rule represents a for loop.
- It consists of the for keyword followed by initialization, condition, and increment/decrement expressions within parentheses, and a block of statements enclosed in braces.

return_statement:

return_x E SEMICOLON

| return_x SEMICOLON ;

Explanation:

- This rule represents a return statement.
- It can have two variations:
- Return keyword followed by an expression and a semicolon.
- Return keyword followed by a semicolon.

cout_statement:

cout insert_statement SEMICOLON ;

Explanation:

- This rule represents a cout (console output) statement.
- It consists of the cout keyword followed by an insert statement and a semicolon.

insert_statement:

insert E insert_statement

| insert string insert_statement

| insert E

| insert string ;

Explanation:

- This rule represents the content to be inserted into the output stream in a cout statement.
- It can have several variations:
- Expression to be inserted followed by more insertions.
- String literal to be inserted followed by more insertions.

- Single expression to be inserted.
- Single string literal to be inserted.

cin_statement:

cin extract_statement SEMICOLON ;

Explanation:

- This rule represents a cin (console input) statement.
- It consists of the cin keyword followed by an extract statement and a semicolon.

extract_statement:

extract identifier extract_statement

| extract identifier ;

Explanation:

- This rule represents the extraction of input from the console.
- It can extract values into one or more identifiers.

declaration_statement:

datatype id_list SEMICOLON {

$2->dtype = $1->dtype;

};

Explanation:

- This rule represents a variable declaration statement.
- It consists of a datatype followed by a list of identifiers and ends with a semicolon.
- The datatype of the identifiers is set to the datatype of the declaration.

id_list:

id_list COMMA identifier {

$3->dtype = $<node>0->dtype;

$1->dtype = $<node>0->dtype;

addEntry($3->value_str, $3->dtype, $3->scope, $3->value);

}

```
| id_list {$1->dtype = $<node>0->dtype;} COMMA dec_assignment {
    $3->dtype = $<node>0->dtype;
  }
  | dec_assignment {
    $1->dtype = $<node>0->dtype;
  }
  | identifier {
    $1->dtype = $<node>-1->dtype;
    addEntry($1->value_str, $1->dtype, $1->scope, $1->value);
  };
```

Explanation:

- This rule represents a list of identifiers in a declaration statement.
- It can have multiple identifiers separated by commas.
- Each identifier is associated with the datatype of the declaration.

```
dec_assignment:
  identifier assignmentop E {
    $1->value = $3->value;
    $1->dtype = $<node>0->dtype;
    addEntry($1->value_str, $1->dtype, $1->scope, $1->value);
  };
```

Explanation:

- This rule represents a declaration statement with assignment.
- It assigns a value to a declared variable and adds an entry for the variable.

```
assignment_statement:
  identifier assignmentop E {
    $1->value = $3->value;
    checkid($1);
  };
```

Explanation:

- This rule represents an assignment statement.
- It assigns a value to an existing variable and checks its validity.

E:

```
identifier assignmentop E {
    $1->value = $3->value;
    checkid($1);
    checkType($1, $3);
}
| E comparisionop E {
    checkType($1, $3);

    if (strcmp($2->value_str, "<=") == 0) {
        $$->value = $1->value <= $3->value;
    } else if (strcmp($2->value_str, ">=") == 0) {
        $$->value = $1->value >= $3->value;
    } else if (strcmp($2->value_str, "==") == 0) {
        $$->value = $1->value == $3->value;
    } else if (strcmp($2->value_str, "!=") == 0) {
        $$->value = $1 ->value != $3->value;
    }
    else if (strcmp($2->value_str, "<") == 0) {
        $$->value = $1->value < $3->value;
    }
    else if (strcmp($2->value_str, ">") == 0) {
        $$->value = $1->value > $3->value;
    }
}
| E PLUS T {
    checkType($1, $3);
}
```

```
| E MINUS T {
    checkType($1, $3);
}
| T {
    $$->value = $1->value;
};
```

Explanation:

- This rule represents an expression (E) in the grammar.
- It handles various arithmetic and logical operations between identifiers, literals, and expressions.
- It ensures type compatibility and performs the necessary operations.

T:

```
T MUL F {
    checkType($1, $3);
}
| T DIV F {
    checkType($1, $3);
}
| F {
    $$->dtype = $1->dtype;
    $$->value = $1->value;
};
```

Explanation:

- This rule represents the term (T) in the grammar.
- It handles multiplication and division operations between factors.
- It ensures type compatibility and performs the necessary operations.

F:

```
number {
    $$->dtype = $1->dtype;
    $$->value = $1->value;
```

```
}
| character {
   $$->dtype = $1->dtype;
}
| LPAREN E RPAREN {
   $$->dtype = $2->dtype;
}
| identifier {
   checkid($1);
   $$->dtype = $1->dtype;
}
| unary identifier {
   checkid($2);
   $$->dtype = $2->dtype;
}
| identifier unary {
   checkid($1);
   $$->dtype = $1->dtype;
}
| string {
   $$->dtype = $1->label;
};
```

Explanation:

- This rule represents a factor (F) in the grammar.
- It can be a number, character, expression enclosed in parentheses, identifier, unary operation on an identifier,
- unary operation followed by an identifier, or a string literal.
- It ensures type compatibility and performs necessary actions based on the factor.

SEMANTIC ANALYZER

Lexer.l:

```
%option yylineno
%{
    #include "temp2.tab.h"
    #include<stdio.h>
    #include<stdlib.h>
    #include <string.h>
    #include <stdarg.h>
    #include "TreeNode.h"
    int st[100];
    int top,count,currscope,up,declared = 0;
    char decl[20];
    int flag = 0;
    void installID(char *text);
    void display();
    struct entry
    {
        char arr[20];
        int scope;
        char dtype[10];
        int value;
    };
    struct entry symbolTable[100];

    TreeNode *createNode(char *label, int value, char *value_str,char *dtype,int scope,
int num_children, ...) {
        TreeNode *newNode = (TreeNode *)malloc(sizeof(TreeNode));
        newNode->label = label;
        newNode->value = value;
        newNode->value_str = (char*)malloc(sizeof(char) * strlen(value_str) + 1);
        strcpy(newNode->value_str,value_str);
        // strcpy(newNode->dtype, );
        if(strcmp(label,"datatype") == 0) newNode->dtype = newNode->value_str;
        else newNode->dtype = "NULL";

        newNode->scope = scope;
        newNode->num_children = num_children;
        newNode->children = NULL;
        if(num_children == 0) return newNode;
        newNode->children = malloc(sizeof(TreeNode*) * num_children);

        va_list args;
        va_start(args, num_children);

        for (int i = 0; i < num_children; i++) {
            newNode->children[i] = va_arg(args, TreeNode*);
        }
```

```
        va_end(args);

        return newNode;
    }
%}

/* regular definitions */
delim [ \t]
ws {delim}+
letter [a-zA-Z]
digit [0-9]
id {letter}({letter}|{digit})*
comparisionop (<=|>=|==|!=|<|>)
logicalop (&&|[|][|])
leftshift (<<)
rightshi (>>)
plus [+]
minus [-]
mult [*]
div [/]
num ({digit}+)
float ({num}\.{num})
arithmeticop ({plus}|{minus}|{mult}|{div})
increment {plus}{plus}
decrement {minus}{minus}
assignop =
string (\"(\\.|[^\\"])*\")
character (\'(\\.|[^\\'])*\')
keyword (if|else|const|while|for|int|float|return|void|main|char|"long
long"|double|short|long|unsigned|signed|define|struct|enum|typedef|sizeof|sta
c|register|auto|break|case|con
nue|default|do|goto|switch|cout|cin|endl|bool|using|namespace|std|include|iostream|vector|
map|set|queue|stack|push_back|pop_back|pop|push|top|front|priority_queue)
inval (({digit}+{id}))
comment (\/\/.*|\/\*([^*]|(\*+[^*/]))*\*+\/[\n]*)

%%
[\n] {printf("newline\n");return EOL;}
#include<[^>]+> {printf( "header:%s\n",yytext); yylval.node = createNode("header",-
1,yytext,"NULL",0,0); printf("done");  return header;};
#include\"([^\"]+)\" {printf( "header:%s\n",yytext); yylval.node = createNode("header",-
1,yytext,"NULL",0,0);  return header;};
if { printf("if\n");  yylval.node = createNode("if",-1,yytext,"NULL",0,0);  return if_x; }
else { printf("else\n");  yylval.node = createNode("else",-1,yytext,"NULL",0,0);  return
else_x; }
while { printf("while\n"); yylval.node = createNode("while",-1,yytext,"NULL",0,0);  return
while_x; }
for {printf("for\n"); yylval.node = createNode("for",-1,yytext,"NULL",0,0);  return for_x;
}
return { printf("return \n"); yylval.node = createNode("return",-
1,yytext,"NULL",0,0);  return return_x; }
printf { printf("printf\n"); yylval.node = createNode("printf",-
1,yytext,"NULL",0,0);  return printf_x; }
```

```
cout { printf("cout\n"); yylval.node = createNode("cout",-1,yytext,"NULL",0,0);  return
cout; }
cin { printf("cin\n"); yylval.node = createNode("cin",-1,yytext,"NULL",0,0);  return cin;
}
"<<" { printf("insert\n");  yylval.node = createNode("<<",-1,yytext,"NULL",0,0);  return
insert; }
">>" { printf("extract\n"); yylval.node = createNode(">>",-1,yytext,"NULL",0,0);  return
extract; }
{assignop} {printf("assignop:%s\n", yytext); yylval.node = createNode("assignop",-
1,yytext,"NULL",0,0);  return assignmentop; }
{comparisionop} { ; printf("compop:%s\n", yytext); strcpy(yylval.str,yytext); yylval.node
= createNode("comparisionop",-1,yytext,"NULL",0,0);  return comparisionop; }
{logicalop} { printf("logicalop:%s\n", yytext); return logicalop; }
int|float|double|char|string|"long long"|short|long { printf("datatype:%s\n",
yytext);declared = 1; yylval.node = createNode("datatype",-1,yytext,yytext,0,0);  return
datatype; }
{num} { printf("num:%s\n", yytext); yylval.node =
createNode("number",atoi(yytext),"NULL","NULL",0,0); return number; }
{increment} {  ; printf("unary:%s\n", yytext); yylval.node = createNode("unary",-
1,yytext,"NULL",0,0);  return unary; }
{decrement} {  ; printf("unary:%s\n", yytext); yylval.node = createNode("unary",-
1,yytext,"NULL",0,0);  return unary; }
{id} { if(declared == 1) {installID(yytext);}  printf("id:%s\n", yytext); yylval.node =
createNode("id",-1,yytext,"NULL",currscope,0); return identifier; }
{character} { printf("character:%s\n", yytext);  yylval.node = createNode("character",-
1,yytext,"NULL",0,0); return character;}
{string} { printf("string:%s\n", yytext); yylval.node = createNode("string",-
1,yytext,"NULL",0,0);  return string; }
\n* ;
{ws} ;
{plus} { yylval.node = createNode("plus",-1,yytext,"NULL",0,0); return PLUS;}
{minus} { yylval.node = createNode("minus",-1,yytext,"NULL",0,0); return MINUS;}
{mult} { yylval.node = createNode("mult",-1,yytext,"NULL",0,0); return MUL;}
{div} { yylval.node = createNode("div",-1,yytext,"NULL",0,0); return DIV;}
"{" { printf("Lbrace:%s\n",yytext); currscope++; yylval.node = createNode("{",-
1,yytext,"NULL",0,0);  return LBRACE; }
"}" { currscope--;printf("Rbrace:%s\n",yytext); yylval.node = createNode("}",-
1,yytext,"NULL",0,0);   return RBRACE; }
"(" {printf ("LPAREN: %s, line: %d\n", yytext, yylineno); currscope++; yylval.node =
createNode("LPAREN",-1,yytext,"NULL",0,0); return LPAREN;}
")" {printf ("RPAREN: %s, line: %d\n", yytext, yylineno); currscope--; yylval.node =
createNode("RPAREN",-1,yytext,"NULL",0,0); return RPAREN;}
"[" {printf ("LBRACKET: %s, line: %d\n", yytext, yylineno); yylval.node = createNode("[",-
1,yytext,"NULL",0,0);  return yytext[0];}
"]" {printf ("RBRACKET: %s, line: %d\n", yytext, yylineno); yylval.node = createNode("]",-
1,yytext,"NULL",0,0);  return yytext[0];}
";" { declared = 0; printf("semicolon :%s,line:%d\n",yytext,yylineno);  yylval.node =
createNode(";",-1,yytext,"NULL",0,0);  return SEMICOLON; }
"," { printf("comma :%s,line:%d\n",yytext,yylineno);  yylval.node = createNode(",",-
1,yytext,"NULL",0,0);  return COMMA; }
{comment} { };
. { return yytext[0]; }
%%
```

```
void installID(char *text)
{
    int present = 0;
    for (int i = 0; i <= up; i++)
    {
        if (strcmp(symbolTable[i].arr, text) == 0 && symbolTable[i].scope == currscope)
        {
            present = 1;
            break;
        }
    }
    if (!present)
    {
        strcpy(symbolTable[up].arr, text);
        symbolTable[up].scope = currscope;
        strcpy(symbolTable[up].dtype, "null");
        up++;
    }
}

void display()
{
    printf("\nSymbol Table\n");
    printf("Symbol\t\t\tscope\t\t\tdtype\t\t\tvalue\n");
    for (int i = 0; i < up; i++)
    {
        printf("%s\t\t\t%d\t\t\t%s\t\t\t%d\n", symbolTable[i].arr,symbolTable[i].scope,
symbolTable[i].dtype,symbolTable[i].value);
    }
}
int yywrap()
{
    return 1;
}
```

Parser.y:

```
%{
    #include "TreeNode.h"
    #include<stdio.h>
    #include<stdlib.h>
    #include<string.h>
    int yylex();
    int yyerror(const char *s);
    int yyparse();
    extern void display();
    struct entry{
        char arr[20];
        int scope;
        char dtype[10];
        int value;
    };
    extern struct entry symbolTable[100];
```

```
    TreeNode *head = NULL;
    struct TreeNode *create_Node(char *label, int value, char *value_str,char *dtype, int
num_children, ...) {
        struct TreeNode *newNode = (struct TreeNode *)malloc(sizeof(struct TreeNode));
        newNode->label = label;
        newNode->value = value;
        newNode->value_str = (char*)malloc(sizeof(char) * strlen(value_str) + 1);
        strcpy(newNode->value_str,value_str);
        newNode->num_children = num_children;
        newNode->dtype= dtype;
        newNode->scope = -1;
        newNode->children = NULL;
        if(num_children<1) return newNode;
        newNode->children = malloc(sizeof(TreeNode*) * num_children);

        va_list args;
        va_start(args, num_children);

        for (int i = 0; i < num_children; i++) {
            newNode->children[i] = va_arg(args, struct TreeNode*);
        }

        va_end(args);

        return newNode;
    }

    void printTree(TreeNode *root,int level){
        if(root == NULL){
            return;
        }
        for(int i = 0;i<level;i++){
            printf("    ");
        }
        if(root->value ==-1 && strcmp(root->dtype,"NULL") == 0){
            if(strcmp(root->value_str,"NULL") == 0){
                printf("|--%d.%s\n",level, root->label);
            }else{
                printf("|--%d.%s\n",level, root->value_str);
            }
        }
        else{
            printf("|--%d.(%s,%d,%s,%s)\n",level, root->label,root->value,root-
>value_str,root->dtype);
        }

        for(int i = 0;i<root->num_children;i++){
            printTree(root->children[i],level+1);
        }
    }
    void addEntry(char *name, char *dtype,int scope,int value){
        for(int i = 0;i<100;i++){
            if(strcmp(symbolTable[i].arr,name) == 0 && symbolTable[i].scope == scope){
```

```
                //printf("Variable: %s, dtype:%s  \n",name,symbolTable[i].dtype);

                if(strcmp(symbolTable[i].dtype,"null") != 0 ){
                    printf("Variable %s is already declared\n",name);
                    yyerror("Variable already declared");
                    break;
                    return;
                }
                symbolTable[i].value = value;
                strcpy(symbolTable[i].dtype,dtype);
                printf("Variable: %s, dtype: %s \n",name,symbolTable[i].dtype);
                // symbolTable[i].value = 0;

                break;
            }
        }
    }
    void addFunctionValue(char *name, int value){
        for(int i = 0;i<100;i++){
            if(strcmp(symbolTable[i].arr,name) == 0){
                symbolTable[i].value = value ;
                break;
            }
        }
    }
    struct Pair{
        char *dtype;
        int scope;
    };
    struct Pair make_pair(char* dtype, int scope) {
        struct Pair p;
        p.dtype = dtype;
        p.scope = scope;
        return p;
    }

    struct Pair checkIdentifier(char *name){
        for(int i = 0;i<100;i++){
            if(strcmp(symbolTable[i].arr,name) == 0){
                return make_pair(symbolTable[i].dtype,symbolTable[i].scope);
                break;
            }
        }
        return make_pair("NULL",-1);
    }
    int checkEntry(char *name,char *dtype,int scope,int value){
        for(int i = 0;i<100;i++){
            if(strcmp(symbolTable[i].arr,name) == 0 && symbolTable[i].scope <= scope &&
strcmp(symbolTable[i].dtype,dtype) == 0){
                symbolTable[i].value = value;
                return 1;
                break;
            }
        }
```

```
        return 0;
    }
    void checkid(TreeNode *root){
        struct Pair p = checkIdentifier(root->value_str);
        if(p.dtype == "NULL"){
            printf("Variable %s is not declared\n",root->value_str);
            yyerror("Variable not declared");
            return ;
        }else{
            root->dtype = p.dtype;
        }
        if(checkEntry(root->value_str,root->dtype,root->scope,root->value) == 0){
            printf("Variable %s,%s,%d is not declared\n",root->value_str,root->dtype,root->scope);
            yyerror("Variable not declared");
            return;
        };
    }
    void checkType(TreeNode* a, TreeNode* b){
        if((a->dtype != b->dtype) && (strcmp(a->dtype, "string") == 0 || strcmp(b->dtype, "string") == 0)){
            yyerror("Type mismatch");
            return;
        }
    }
    void checkfnArguments(TreeNode* a){
        for(int i = 0;i<100;i++){
            if(strcmp(symbolTable[i].arr,a->value_str) == 0){
                if(symbolTable[i].value != a->value){
                    yyerror("Number of arguments mismatch");
                    return;
                }
            }
        }
    }
%}

%union {
    int num;
    char *str;
    struct TreeNode *node;
}

%start program
%token EOL
%error-verbose
%token<node> PLUS MINUS MUL DIV number if_x else_x while_x for_x return_x printf_x main_x assignmentop comparisionop logicalop datatype unary identifier string character cout cin insert extract header LBRACE RBRACE LPAREN RPAREN SEMICOLON COMMA
%type<node> E T F function_call arg_list assignment_statement dec_assignment statement_list function_declarations function_declaration declaration_statement id_list insert_statement extract_statement if_statement else_statement if_else_statement while_statement for_statement return_statement cout_statement cin_statement statement headers parameter_list program
```

```
/* rules */
%%

program: headers function_declarations {  $$ = create_Node("program", -1, "NULL","NULL",
2,$1,$2);if(!head) head = $$;}
    | EOL{$$ = NULL; head = $$;}
    |;
function_declarations:
    function_declaration function_declarations { $$ = create_Node("function_declarations",
-1, "NULL","NULL", 2,$1,$2); }
    | EOL function_declarations { $$ = create_Node("function_declarations", -1,
"NULL","NULL", 1,$2); }
    | EOL { $$ = NULL; }
    | ;



headers: header {$$ = create_Node("headers", -1, "NULL","NULL",1,$1);}
    | header headers { $$ = create_Node("headers", -1,"NULL","NULL", 2, $1, $2);}
    | EOL headers { $$ = create_Node("headers", -1, "NULL","NULL",1, $2);}
    | EOL{ $$ = NULL; };

function_declaration: datatype identifier LPAREN parameter_list RPAREN LBRACE
statement_list RBRACE {
    $2->dtype = $1->dtype;
    $$ = create_Node("function_declaration", -1, "NULL","NULL", 8, $1,$2,$3, $4, $5, $6,
$7, $8);
    $$->value = $4->value;
    addEntry($2->value_str,$2->dtype,$2->scope,$4->value);
    addFunctionValue($2->value_str, $4->value);
};

parameter_list: datatype identifier COMMA parameter_list {
    $$ = create_Node("parameter_list", -1, "NULL","NULL",4, $1, $2, $3, $4);
    $2->dtype = $1->dtype;
    $$->value = $4->value + 1;
    addEntry($2->value_str,$2->dtype,$2->scope,$2->value);
     }
 | datatype identifier {
    $2->dtype = $1->dtype;
    $$ = create_Node("parameter_list", -1, "NULL","NULL",2, $1, $2);
    $$->value = 1;
    addEntry($2->value_str,$2->dtype,$2->scope,$2->value);
 }
    | { $$ = create_Node("parameter_list", -1, "NULL","NULL",0); $$->value = 0 };


statement_list:
    statement statement_list { $$ = create_Node("statement_list", -1, "NULL","NULL", 2,
$1, $2);}
    |   EOL statement_list { $$ = create_Node("statement_list", -1, "NULL","NULL", 1,
$2);}
    |   EOL {$$ = NULL;}
    |
```

```
;


statement: declaration_statement { $$ = create_Node("statement", -1, "NULL","NULL",1,$1);
}
    | assignment_statement SEMICOLON {printf("assignment_statement\n"); $$ =
create_Node("statement", -1,"NULL","NULL",2, $1, $2);}
    | for_statement {printf("for_statement\n"); $$ = create_Node("statement", -1,
"NULL","NULL",1,$1); }
    | if_statement {printf("if_statement\n"); $$ = create_Node("statement", -1,
"NULL","NULL",1,$1); }
    | if_else_statement {printf("if_else_statement\n"); $$ = create_Node("statement", -1,
"NULL","NULL",1,$1); }
    | while_statement {printf("while_statement\n"); $$ = create_Node("statement", -1,
"NULL","NULL",1,$1); }
    | cout_statement {printf("cout_statement\n"); $$ = create_Node("statement", -1,
"NULL","NULL",1,$1); }
    | cin_statement {printf("cin_statement\n"); $$ = create_Node("statement", -1,
"NULL","NULL",1,$1); }
    | return_statement {printf("return_statement\n"); $$ = create_Node("statement", -1,
"NULL","NULL",1,$1); }
    | function_call SEMICOLON {printf("function_call\n"); $$ = create_Node("statement", -
1, "NULL","NULL",2,$1,$2); }
    | error SEMICOLON { $$ = create_Node("error", -1, "NULL","NULL",0); }
    | EOL { $$ = NULL; };

function_call: identifier LPAREN arg_list RPAREN {
        $$ = create_Node("function_call", -1, "NULL","NULL", 4, $1, $2, $3, $4);
        $1->value = $3->value;
        checkfnArguments($1);
    }
    | identifier LPAREN RPAREN { $$ = create_Node("function_call", -1, "NULL","NULL", 3,
$1, $2, $3); }
    ;

arg_list: identifier { $$ = create_Node("arg_list", -1, "NULL","NULL", 1, $1); $$->value =
1; }
    | identifier COMMA arg_list { $$ = create_Node("arg_list", -1, "NULL","NULL", 3, $1,
$2, $3); $$->value = $3->value + 1; }
    ;

if_statement: if_x LPAREN E RPAREN LBRACE statement_list RBRACE { $$ =
create_Node("if_statement", -1,"NULL","NULL", 7,$1,$2,$3,$4,$5,$6,$7);}
| if_x LPAREN E RPAREN  statement { $$ = create_Node("if_statement", -1,"NULL","NULL",
5,$1,$2,$3,$4,$5);}
| if_statement EOL  { $$ = create_Node("if_statement", -1,"NULL","NULL", 1,$1);};


else_statement: else_x LBRACE statement_list RBRACE { $$ = create_Node("else_statement", -
1, "NULL","NULL", 3, $1, $2, $3); }
 | else_x statement { $$ = create_Node("else_statement", -1, "NULL","NULL", 2, $1, $2); }
 | else_statement EOL { $$ = create_Node("else_statement", -1, "NULL","NULL", 1, $1); }
 ;
```

```
if_else_statement: if_statement else_statement { $$ = create_Node("if_else_statement", -1,
"NULL","NULL", 2, $1, $2); };

while_statement: while_x LPAREN E RPAREN LBRACE statement_list RBRACE { $$ =
create_Node("while_statement", -1, "NULL","NULL", 7, $1, $2, $3, $4, $5, $6, $7); };

for_statement: for_x LPAREN declaration_statement E SEMICOLON E RPAREN LBRACE
statement_list RBRACE { $$ = create_Node("for_statement", -1, "NULL","NULL", 9, $1, $2,
$3, $4, $5, $6, $7, $8, $9); };


return_statement: return_x E SEMICOLON { $$ = create_Node("return_statement", -
1,"NULL","NULL", 2, $1, $2); }
| return_x SEMICOLON { $$ = create_Node("return_statement", -1, "NULL","NULL", 2, $1,$2);
};

cout_statement: cout insert_statement SEMICOLON { $$ = create_Node("cout_statement", -
1,"NULL","NULL", 2, $1, $2); };

insert_statement: insert E insert_statement { $$ = create_Node("insert_statement", -
1,"NULL","NULL", 3, $1, $2, $3); }
    | insert string insert_statement { $$ = create_Node("insert_statement", -1,
"NULL","NULL", 3, $1, $2, $3); }
    | insert E { $$ = create_Node("insert_statement", -1, "NULL","NULL", 2, $1, $2); }
    | insert string { $$ = create_Node("insert_statement", -1, "NULL","NULL", 2, $1, $2);
}

cin_statement: cin extract_statement SEMICOLON { $$ = create_Node("cin_statement", -1,
"NULL","NULL", 3, $1, $2, $3); };

extract_statement: extract identifier extract_statement { $$ =
create_Node("extract_statement", -1, "NULL","NULL", 3, $1, $2, $3); }
| extract identifier { $$ = create_Node("extract_statement", -1, "NULL","NULL", 2, $1,
$2); }
;

declaration_statement: datatype id_list SEMICOLON { $2->dtype = $1->dtype;  $$ =
create_Node("declaration_statement", -1, "NULL","NULL", 3, $1, $2, $3); };

id_list: id_list COMMA identifier {
    $$ = create_Node("id_list", -1, "NULL","NULL", 3, $1, $2, $3);
    $3->dtype = $<node>0->dtype;
    $1->dtype = $<node>0->dtype;
    addEntry($3->value_str,$3->dtype,$3->scope,$3->value);
}
| id_list {$1->dtype = $<node>0->dtype;} COMMA dec_assignment {
    $$ = create_Node("id_list", -1, "NULL","NULL", 3, $1, $3, $4);
    $3->dtype = $<node>0->dtype;
}
| dec_assignment {
    $$ = create_Node("id_list", -1, "NULL","NULL", 1, $1);
    $1->dtype = $<node>0->dtype;
}
| identifier {
```

```
    $$ = create_Node("id_list", -1, "NULL","NULL", 1, $1);
    $1->dtype = $<node>-1->dtype;
    addEntry($1->value_str,$1->dtype,$1->scope,$1->value);
}

dec_assignment:
    identifier assignmentop E {
        $$ = create_Node("dec_assignment", -1, "NULL","NULL", 3, $1, $2, $3);
        checkType($1,$3);
        $1->value = $3->value;
        $1->dtype = $<node>0->dtype;
        addEntry($1->value_str,$1->dtype,$1->scope,$1->value);

    }

assignment_statement:
    identifier assignmentop E  {
        $$ = create_Node("assignment_statement", -1, "NULL","NULL", 3, $1, $2, $3);
        $1->value = $3->value;
        checkid($1);
        checkType($1,$3);
    }
;

E: identifier assignmentop E  { $$ = create_Node("E", -1,  "NULL","NULL", 3, $1, $2, $3);
$1->value = $3->value;  checkid($1); checkType($1,$3); }
    |   E comparisionop E { $$ = create_Node("E", -1,  "NULL","NULL", 3, $1, $2, $3);
checkType($1,$3);

            if (strcmp($2->value_str, "<=") == 0) {
                $$->value = $1->value <= $3->value;
            } else if (strcmp($2->value_str, ">=") == 0) {
                $$->value = $1->value >= $3->value;
            } else if (strcmp($2->value_str, "==") == 0) {
                $$->value = $1->value == $3->value;
            } else if (strcmp($2->value_str, "!=") == 0) {
                $$->value = $1->value != $3->value;
            }
            else if (strcmp($2->value_str, "<") == 0) {
                $$->value = $1->value < $3->value;
            }
            else if (strcmp($2->value_str, ">") == 0) {
                $$->value = $1->value > $3->value;
            }
    }
    |   E PLUS T { $$ = create_Node("E", $1->value+$3->value, "NULL","NULL", 3, $1, $2,
$3); checkType($1,$3); }
    |   E MINUS T { $$ = create_Node("E", $1->value-$3->value, "NULL","NULL", 3, $1, $2,
$3); checkType($1,$3); }
    |   T { $$ = create_Node("E", $1->value, "NULL","NULL", 1, $1);  $$->value = $1-
>value; $$->dtype = $1->dtype; }
    ;
```

```
T: T MUL F { $$ = create_Node("T", $1->value*$3->value, "NULL","NULL", 3, $1, $2, $3);
checkType($1,$3); }
    |   T DIV F { $$ = create_Node("T", $1->value/$3->value, "NULL","NULL", 3, $1, $2,
$3);checkType($1,$3);  }
    |   F { $$ = create_Node("T", $1->value, "NULL","NULL", 1, $1); $$->dtype = $1-
>dtype;  $$->value = $1->value;}
    ;

F: number { $$ = create_Node("F", -1, "NULL","NULL", 1, $1);  $$->dtype = $1->dtype; $$-
>value = $1->value;  }
    | character { $$ = create_Node("F", -1, "NULL","NULL", 1, $1);  $$->dtype = $1-
>dtype;  }
    | LPAREN E RPAREN { $$ = create_Node("F",$2->value, "NULL","NULL", 3, $1, $2,
$3);  $$->dtype = $2->dtype;  }
    | identifier { $$ = create_Node("F", -1, "NULL","NULL", 1, $1); checkid($1);  $$-
>dtype = $1->dtype; }
    | unary identifier { $$ = create_Node("F", -1, "NULL","NULL", 2, $1, $2);
checkid($2);  $$->dtype = $2->dtype;  }
    | identifier unary { $$ = create_Node("F", -1, "NULL","NULL", 2, $1, $2);
checkid($1);  $$->dtype = $1->dtype; }
    | string { $$ = create_Node("F", -1, "NULL","NULL", 1, $1); $$->dtype = $1->label;}
    ;

%%
#include <ctype.h>
int yyerror(const char *s)
{
    extern int yylineno;
    // valid = 0;
    if(yylineno != 27){
        printf("Line no: %d \n The error is: %s\n",yylineno,s);
    }
}
extern FILE *yyin;
int main(int argc,char **argv){
    if(argc<2)
    {
        printf("Usage: %s <filename>\n",argv[0]);
        return 1;
    }
    FILE *fp = fopen(argv[1], "r");
    if(fp==NULL)
    {
        printf("Error: File not found\n");
        return 1;
    }
    extern char *yytext;
    yyin=fp;
    yyparse();
    // yyparse();
    printf("---------------\n");
    printTree(head,0);
    printf("---------------\n");
    display();
```

```
}
```

Question-2:

Sample code (c++):

```
#include<bits/stdc++.h>

int function(int x, int y){

}

int main(){
    int a = 5,d;
    cin>>a;
    int e = 10;
    char c = 'c';
    for(int i= 0;i<10;i++){
        a = 1;
    }

    if(b) {
        a = 10;
    }
    int z = 11;
    string s = "hello";
    c = (s*10);

    function(a,b,c);

    return 0;
}
```

Output for sample code:

header:#include<bits/stdc++.h>

datatype:int

id:function

LPAREN: (, line: 3

datatype:int

id:x

comma :,,line:3

datatype:int

id:y

RPAREN:), line: 3

Variable: y, dtype: int

Variable: x, dtype: int

Lbrace:{

Rbrace:}

Variable: function, dtype: int

datatype:int

id:main

LPAREN: (, line: 7

RPAREN:), line: 7

Lbrace:{

newline

datatype:int

id:a

assignop:=

num:5

comma :,,line:8

Variable: a, dtype: int

id:d

Variable: d, dtype: int

semicolon :;,line:8

newline

cin

extract

id:a

semicolon :;,line:9

cin_statement

newline

datatype:int

id:e

assignop:=

num:10

semicolon :;,line:10

Variable: e, dtype: int

newline

datatype:char

id:c

assignop:=

character:'c'

semicolon :;,line:11

Variable: c, dtype: char

newline

for

LPAREN: (, line: 12

datatype:int

id:i

assignop:=

num:0

semicolon :;,line:12

Variable: i, dtype: int

id:i

compop:<

num:10

semicolon :;,line:12

id:i

unary:++

RPAREN:), line: 12

Lbrace:{

newline

id:a

assignop:=

num:1

semicolon :;,line:13

assignment_statement

newline

Rbrace:}

for_statement

newline

newline

if

LPAREN: (, line: 16

id:b

RPAREN:), line: 16

Variable b is not declared

Line no: 16

The error is: Variable not declared

Lbrace:{

newline

id:a

assignop:=

num:10

semicolon :;,line:17

assignment_statement

newline

Rbrace:}

newline

datatype:int

if_statement

id:z

assignop:=

num:11

semicolon :;,line:19

Variable: z, dtype: int

newline

datatype:string

id:s

assignop:=

string:"hello"

semicolon :;,line:20

Variable: s, dtype: string

newline

id:c

assignop:=

LPAREN: (, line: 21

id:s

num:10

Line no: 21

The error is: Type mismatch

RPAREN:), line: 21

semicolon :;,line:21

assignment_statement

id:function

LPAREN: (, line: 23

id:a

comma :,,line:23

id:b

comma :,,line:23

id:c

RPAREN:), line: 23

Line no: 23

The error is: Number of arguments mismatch

semicolon :;,line:23

function_call

return

num:0

semicolon :;,line:25

return_statement

newline

```
Rbrace:}
Variable: main, dtype: int
----------
|--0.program
  |--1.headers
    |--2.#include<bits/stdc++.h>
  |--1.function_declarations
    |--2.(function_declaration,2,NULL,NULL)
      |--3.(datatype,-1,int,int)
      |--3.(id,-1,function,int)
      |--3.(
      |--3.(parameter_list,2,NULL,NULL)
        |--4.(datatype,-1,int,int)
        |--4.(id,-1,x,int)
        |--4.,
        |--4.(parameter_list,1,NULL,NULL)
          |--5.(datatype,-1,int,int)
          |--5.(id,-1,y,int)
      |--3.)
      |--3.{
      |--3.(parameter_list,1,NULL,NULL)
        |--4.(datatype,-1,int,int)
        |--4.(id,-1,y,int)
      |--3.}
    |--2.function_declarations
      |--3.(function_declaration,0,NULL,NULL)
        |--4.(datatype,-1,int,int)
        |--4.(id,-1,main,int)
        |--4.(
        |--4.(parameter_list,0,NULL,NULL)
        |--4.)
        |--4.{
```

```
|--4.statement_list
  |--5.statement_list
    |--6.statement
      |--7.declaration_statement
        |--8.(datatype,-1,int,int)
        |--8.(id_list,-1,NULL,int)
          |--9.(id_list,-1,NULL,int)
            |--10.(dec_assignment,-1,NULL,int)
              |--11.(id,5,a,int)
              |--11.=
              |--11.(E,5,NULL,NULL)
                |--12.(T,5,NULL,NULL)
                  |--13.(F,5,NULL,NULL)
                    |--14.(number,5,NULL,NULL)
          |--9.,
          |--9.(id,-1,d,int)
        |--8.;
    |--6.statement_list
      |--7.statement_list
        |--8.statement
          |--9.cin_statement
            |--10.cin
            |--10.extract_statement
              |--11.>>
              |--11.a
            |--10.;
        |--8.statement_list
          |--9.statement_list
            |--10.statement
              |--11.declaration_statement
                |--12.(datatype,-1,int,int)
                |--12.(id_list,-1,NULL,int)
```

```
        |--13.(dec_assignment,-1,NULL,int)
          |--14.(id,10,e,int)
          |--14.=
          |--14.(E,10,NULL,NULL)
            |--15.(T,10,NULL,NULL)
              |--16.(F,10,NULL,NULL)
                |--17.(number,10,NULL,NULL)
    |--12.;
|--10.statement_list
  |--11.statement_list
    |--12.statement
      |--13.declaration_statement
        |--14.(datatype,-1,char,char)
        |--14.(id_list,-1,NULL,char)
          |--15.(dec_assignment,-1,NULL,char)
            |--16.(id,-1,c,char)
            |--16.=
            |--16.E
              |--17.T
                |--18.F
                  |--19.'c'
        |--14.;
    |--12.statement_list
      |--13.statement_list
        |--14.statement
          |--15.for_statement
            |--16.for
            |--16.(
            |--16.declaration_statement
              |--17.(datatype,-1,int,int)
              |--17.(id_list,-1,NULL,int)
                |--18.(dec_assignment,-1,NULL,int)
```

```
        |--19.(id,0,i,int)
        |--19.=
        |--19.(E,0,NULL,NULL)
          |--20.(T,0,NULL,NULL)
            |--21.(F,0,NULL,NULL)
              |--22.(number,0,NULL,NULL)
  |--17.;
|--16.(E,1,NULL,NULL)
  |--17.E
    |--18.(T,-1,NULL,int)
      |--19.(F,-1,NULL,int)
        |--20.(Ç,-1,i,int)
  |--17.<
  |--17.(E,10,NULL,NULL)
    |--18.(T,10,NULL,NULL)
      |--19.(F,10,NULL,NULL)
        |--20.(number,10,NULL,NULL)
|--16.;
|--16.E
  |--17.(T,-1,NULL,int)
    |--18.(F,-1,NULL,int)
      |--19.(id,-1,i,int)
      |--19.++
|--16.)
|--16.{
|--16.statement_list
  |--17.statement_list
    |--18.statement
      |--19.assignment_statement
        |--20.(id,1,a,int)
        |--20.=
        |--20.(E,1,NULL,NULL)
```

```
                    |--21.(T,1,NULL,NULL)
                      |--22.(F,1,NULL,NULL)
                        |--23.(number,1,NULL,NULL)
                |--19.;
          |--14.statement_list
            |--15.statement_list
              |--16.statement_list
                |--17.statement
                  |--18.if_statement
                    |--19.if_statement
                      |--20.if
                      |--20.(
                      |--20.E
                        |--21.T
                          |--22.F
                            |--23.b
                      |--20.)
                      |--20.{
                      |--20.statement_list
                        |--21.statement_list
                          |--22.statement
                            |--23.assignment_statement
                              |--24.(id,10,a,int)
                              |--24.=
                              |--24.(E,10,NULL,NULL)
                                |--25.(T,10,NULL,NULL)
                                  |--26.(F,10,NULL,NULL)
                                    |--27.(number,10,NULL,NULL)
                            |--23.;
                      |--20.}
                |--17.statement_list
                  |--18.statement
```

```
                |--19.declaration_statement
                  |--20.(datatype,-1,int,int)
                  |--20.(id_list,-1,NULL,int)
                    |--21.(dec_assignment,-1,NULL,int)
                      |--22.(id,11,z,int)
                      |--22.=
                      |--22.(E,11,NULL,NULL)
                        |--23.(T,11,NULL,NULL)
                          |--24.(F,11,NULL,NULL)
                            |--25.(number,11,NULL,NULL)
                  |--20.;
              |--18.statement_list
                |--19.statement_list
                  |--20.statement
                    |--21.declaration_statement
                      |--22.(datatype,-1,string,string)
                      |--22.(id_list,-1,NULL,string)
                        |--23.(dec_assignment,-1,NULL,string)
                          |--24.(id,-1,s,string)
                          |--24.=
                          |--24.E
                            |--25.(T,-1,NULL,string)
                              |--26.(F,-1,NULL,string)
                                |--27."hello"
                      |--22.;
                  |--20.statement_list
                    |--21.statement_list
                      |--22.statement
                        |--23.assignment_statement
                          |--24.(id,-10,c,char)
                          |--24.=
                          |--24.(E,-10,NULL,NULL)
```

```
                |--25.(T,-10,NULL,NULL)
                  |--26.(F,-10,NULL,NULL)
                    |--27.(
                    |--27.(E,-10,NULL,NULL)
                      |--28.(T,-10,NULL,NULL)
                        |--29.(T,-1,NULL,string)
                          |--30.(F,-1,NULL,string)
                            |--31.(id,-1,s,string)
                        |--29.*
                        |--29.(F,10,NULL,NULL)
                          |--30.(number,10,NULL,NULL)
                    |--27.)
            |--23.;
          |--22.statement_list
            |--23.statement
              |--24.function_call
                |--25.(id,3,function,NULL)
                |--25.(
                |--25.(arg_list,3,NULL,NULL)
                  |--26.a
                  |--26.,
                  |--26.(arg_list,2,NULL,NULL)
                    |--27.b
                    |--27.,
                    |--27.(arg_list,1,NULL,NULL)
                      |--28.c
                |--25.)
              |--24.;
            |--23.statement_list
              |--24.statement
                |--25.return_statement
                  |--26.return
```

|--26.(E,0,NULL,NULL)

|--27.(T,0,NULL,NULL)

|--28.(F,0,NULL,NULL)

|--29.(number,0,NULL,NULL)

|--4.}

Symbol Table

Symbol	scope	dtype	value
function	0	int	2
x	1	int	-1
y	1	int	-1
main	0	int	0
a	1	int	10
d	1	int	-1
e	1	int	10
c	1	char	-10
i	2	int	-1
z	1	int	11
s	1	string	-1

Question-3:

1. **Undeclared usage of variables and functions**
   When a variable is not declared in the present code and used in the code or scope and also if the variable is declared in the scope and is used outside the present declared scope then the compiler immediately detect it as the semantic error and declare it in the output.
   - **Code:**

```
void checkid(TreeNode *root){
        struct Pair p = checkIdentifier(root->value_str);
        if(p.dtype == "NULL"){
            printf("Variable %s is not declared\n",root->value_str);
            yyerror("Variable not declared");
            return ;
        }else{
            root->dtype = p.dtype;
```

```
        }
        if(checkEntry(root->value_str,root->dtype,root->scope,root->value) == 0){
            printf("Variable %s,%s,%d is not declared\n",root->value_str,root->dtype,root->scope);
            yyerror("Variable not declared");
            return;
        };
    }
struct Pair checkIdentifier(char *name){
        for(int i = 0;i<100;i++){
            if(strcmp(symbolTable[i].arr,name) == 0){
                return make_pair(symbolTable[i].dtype,symbolTable[i].scope);
                break;
            }
        }
        return make_pair("NULL",-1);
    }
```

2. Multiple declarations of variables and functions

when a variable or function is declared multiple times in the present scope or in the whole code, compiler immediately detect it as the semantic error and declare it in the output except if the variable is declared in the previous scope and in the new scope out of the old scope if it is declared then it would not be considered as semantic error.

- **Code:**

```
void addEntry(char *name, char *dtype,int scope,int value){
        for(int i = 0;i<100;i++){
            if(strcmp(symbolTable[i].arr,name) == 0 && symbolTable[i].scope == scope){
                //printf("Variable: %s, dtype:%s  \n",name,symbolTable[i].dtype);

                if(strcmp(symbolTable[i].dtype,"null") != 0 ){
                    printf("Variable %s is already declared\n",name);
                    yyerror("Variable already declared");
                    break;
                    return;
                }
                symbolTable[i].value = value;
                strcpy(symbolTable[i].dtype,dtype);
                printf("Variable: %s, dtype: %s \n",name,symbolTable[i].dtype);
                // symbolTable[i].value = 0;

                break;
            }
        }
    }
```

3. Mismatched datatypes of variables in expressions

In programming languages, each variable has a specific data type associated with it. When variables of different data types are used together in expressions, it can lead to mismatched datatypes. These errors are detected by the compiler and declared in the output.

- **Code:**

```
void checkType(TreeNode* a, TreeNode* b){
        if((a->dtype != b->dtype) && (strcmp(a->dtype, "string") == 0 || strcmp(b->dtype, "string") == 0)){
            yyerror("Type mismatch");
            return;
        }
    }
```

4. Function call mismatch based on no of arguments

In programming languages, functions are defined with a certain number of parameters. When calling a function, it's essential to provide arguments that match the number and types of parameters declared in the function definition. If the number of arguments passed during the function call doesn't match the number of parameters expected by the function, it results in a function call mismatch based on the number of arguments.

- **Code:**

```
void checkfnArguments(TreeNode* a){
        for(int i = 0;i<100;i++){
            if(strcmp(symbolTable[i].arr,a->value_str) == 0){
                if(symbolTable[i].value != a->value){
                    yyerror("Number of arguments mismatch");
                    return;
                }
            }
        }
    }
```

Sample code(c++) containing all the errors mentioned above:

```
#include<bits/stdc++.h>
int function(int x, int y){
}
int function(int u, int v){
}

int main(){
    int d = 10;
    int d = 20;
    int a, b , c;
    function(a,b);
    function(b,c,d);
    int e = "Hello";
    c = "hai";
    return 0;
}
```

Output:

header:#include<bits/stdc++.h>

newline

datatype:int

id:function

LPAREN: (, line: 2

datatype:int

id:x

comma :,,line:2

datatype:int

id:y

RPAREN:), line: 2

Variable: y, dtype: int

Variable: x, dtype: int

Lbrace:{

newline

Rbrace:}

Variable: function, dtype: int

newline

datatype:int

id:function

LPAREN: (, line: 4

datatype:int

id:u

comma :,,line:4

datatype:int

id:v

RPAREN:), line: 4

Variable: v, dtype: int

Variable: u, dtype: int

Lbrace:{

newline

Rbrace:}

Variable function is already declared

Line no: 5

The error is: Variable already declared

datatype:int

id:main

LPAREN: (, line: 7

RPAREN:), line: 7

Lbrace:{

newline

datatype:int

id:d

assignop:=

num:10

semicolon :;,line:8

Variable: d, dtype: int

newline

datatype:int

id:d

assignop:=

num:20

semicolon :;,line:9

Variable d is already declared

Line no: 9

The error is: Variable already declared

newline

datatype:int

id:a

comma :,,line:10

Variable: a, dtype: NULL

id:b

Variable: b, dtype: int

comma :,,line:10

id:c

Variable: c, dtype: int

semicolon :;,line:10

newline

id:function

LPAREN: (, line: 11

id:a

comma :,,line:11

id:b

RPAREN:), line: 11

semicolon :;,line:11

function_call

newline

id:function

LPAREN: (, line: 12

id:b

comma :,,line:12

id:c

comma :,,line:12

id:d

RPAREN:), line: 12

Line no: 12

The error is: Number of arguments mismatch

semicolon :;,line:12

function_call

newline

datatype:int

id:e

assignop:=

string:"Hello"

semicolon :;,line:13

Line no: 13

The error is: Type mismatch

Variable: e, dtype: int

newline

id:c

assignop:=

string:"hai"

semicolon :;,line:14

Line no: 14

The error is: Type mismatch

assignment_statement

newline

return

num:0

semicolon :;,line:15

return_statement

newline

Rbrace:}

Variable: main, dtype: int

newline

|--0.program

  |--1.headers

    |--2.#include<bits/stdc++.h>

  |--1.function_declarations

    |--2.(function_declaration,2,NULL,NULL)

      |--3.(datatype,-1,int,int)

      |--3.(id,-1,function,int)

      |--3.(

      |--3.(parameter_list,2,NULL,NULL)

        |--4.(datatype,-1,int,int)

        |--4.(id,-1,x,int)

        |--4.,

        |--4.(parameter_list,1,NULL,NULL)

```
        |--5.(datatype,-1,int,int)
        |--5.(id,-1,y,int)
    |--3.)
    |--3.{
    |--3.}
|--2.function_declarations
    |--3.function_declarations
      |--4.(function_declaration,2,NULL,NULL)
        |--5.(datatype,-1,int,int)
        |--5.(id,-1,function,int)
        |--5.(
        |--5.(parameter_list,2,NULL,NULL)
          |--6.(datatype,-1,int,int)
          |--6.(id,-1,u,int)
          |--6.,
          |--6.(parameter_list,1,NULL,NULL)
            |--7.(datatype,-1,int,int)
            |--7.(id,-1,v,int)
        |--5.)
        |--5.{
        |--5.}
      |--4.function_declarations
        |--5.(function_declaration,0,NULL,NULL)
          |--6.(datatype,-1,int,int)
          |--6.(id,-1,main,int)
          |--6.(
          |--6.(parameter_list,0,NULL,NULL)
          |--6.)
          |--6.{
          |--6.statement_list
            |--7.statement_list
              |--8.statement
```

```
        |--9.declaration_statement
          |--10.(datatype,-1,int,int)
          |--10.(id_list,-1,NULL,int)
            |--11.(dec_assignment,-1,NULL,int)
              |--12.(id,10,d,int)
              |--12.=
              |--12.(E,10,NULL,NULL)
                |--13.(T,10,NULL,NULL)
                  |--14.(F,10,NULL,NULL)
                    |--15.(number,10,NULL,NULL)
          |--10.;
      |--8.statement_list
        |--9.statement_list
          |--10.statement
            |--11.declaration_statement
              |--12.(datatype,-1,int,int)
              |--12.(id_list,-1,NULL,int)
                |--13.(dec_assignment,-1,NULL,int)
                  |--14.(id,20,d,int)
                  |--14.=
                  |--14.(E,20,NULL,NULL)
                    |--15.(T,20,NULL,NULL)
                      |--16.(F,20,NULL,NULL)
                        |--17.(number,20,NULL,NULL)
              |--12.;
          |--10.statement_list
            |--11.statement_list
              |--12.statement
                |--13.declaration_statement
                  |--14.(datatype,-1,int,int)
                  |--14.(id_list,-1,NULL,int)
                    |--15.(id_list,-1,NULL,int)
```

```
          |--16.(id_list,-1,NULL,int)
            |--17.a
          |--16.,
          |--16.(id,-1,b,int)
        |--15.,
        |--15.(id,-1,c,int)
      |--14.;
|--12.statement_list
  |--13.statement_list
    |--14.statement
      |--15.function_call
        |--16.(id,2,function,NULL)
        |--16.(
        |--16.(arg_list,2,NULL,NULL)
          |--17.a
          |--17.,
          |--17.(arg_list,1,NULL,NULL)
            |--18.b
        |--16.)
      |--15.;
    |--14.statement_list
      |--15.statement_list
        |--16.statement
          |--17.function_call
            |--18.(id,3,function,NULL)
            |--18.(
            |--18.(arg_list,3,NULL,NULL)
              |--19.b
              |--19.,
              |--19.(arg_list,2,NULL,NULL)
                |--20.c
                |--20.,
```

```
            |--20.(arg_list,1,NULL,NULL)
              |--21.d
          |--18.)
        |--17.;
      |--16.statement_list
        |--17.statement_list
          |--18.statement
            |--19.declaration_statement
              |--20.(datatype,-1,int,int)
              |--20.(id_list,-1,NULL,int)
                |--21.(dec_assignment,-1,NULL,int)
                  |--22.(id,-1,e,int)
                  |--22.=
                  |--22.(E,-1,NULL,string)
                    |--23.(T,-1,NULL,string)
                      |--24.(F,-1,NULL,string)
                        |--25."Hello"
              |--20.;
          |--18.statement_list
            |--19.statement_list
              |--20.statement
                |--21.assignment_statement
                  |--22.(id,-1,c,int)
                  |--22.=
                  |--22.(E,-1,NULL,string)
                    |--23.(T,-1,NULL,string)
                      |--24.(F,-1,NULL,string)
                        |--25."hai"
                |--21.;
              |--20.statement_list
                |--21.statement_list
                  |--22.statement
```

```
                                        |--23.return_statement
                                          |--24.return
                                          |--24.(E,0,NULL,NULL)
                                            |--25.(T,0,NULL,NULL)
                                              |--26.(F,0,NULL,NULL)
                                                |--27.(number,0,NULL,NULL)
            |--6.}
---------------
```

Symbol Table

Symbol	scope	dtype	value
function	0	int	2
x	1	int	-1
y	1	int	-1
u	1	int	-1
v	1	int	-1
main	0	int	0
d	1	int	10
a	1	NULL	-1
b	1	int	-1
c	1	int	-1
e	1	int	-1