**Q5) Explain the functions which performs the semantic analysis in your mini compiler.**

**Semantic Analysis in my Mini Compiler**

**Semantic analysis** is an essential phase of the compilation process, where the primary task is to ensure that the program makes sense in terms of its meaning and is logically consistent. While syntactic analysis ensures the correct structure of the code, semantic analysis checks for errors like type mismatches, undeclared variables, incorrect use of functions, and more. My semantic analysis phase uses the **Abstract Syntax Tree (AST)** generated during parsing to verify that the program adheres to the rules of the language and is semantically correct.

In my **mini compiler**, the semantic analysis phase typically performs several checks, which include:

1. **Type Checking**: Ensures that expressions and variables are used with compatible data types.
2. **Scope Checking**: Verifies that variables and functions are declared before they are used and that they are used within their correct scope.
3. **Symbol Table Management**: Keeps track of identifiers (such as variables, functions, and classes), their types, and other relevant attributes.
4. **Function Call Validation**: Checks that function calls have the correct number and types of arguments.
5. **Declaration Checking**: Verifies that every variable or function is declared before it is used.
6. **Control Flow Checking**: Ensures that the program contains no unreachable code and that control structures (like loops and conditionals) are used correctly.

**Explanation of the Semantic Analysis Function**

My **semantic analysis function** typically the AST and performs various checks to ensure that the program adheres to the semantic rules of the language. Below is a detailed breakdown of how such a function might be structured and how it performs different checks.

**1. Symbol Table**

A **Symbol Table** is a data structure that stores information about identifiers (such as variables, functions, and classes) encountered during the compilation process. The semantic analysis function interacts heavily with the symbol table to check the scope, type, and declaration status of various identifiers.

* **Symbol Table Structure**: It usually consists of key-value pairs, where:
  + The **key** is the identifier (variable or function name).
  + The **value** is an associated record or structure containing information like the data type, scope level, and whether the identifier is declared.

**Structure of Symbol Table Record:**

struct SymbolInfo {

string name; // Identifier name

string type; // Data type (e.g., int, string, etc.)

string scope; // Scope in which the identifier is defined

bool isDeclared; // Whether the identifier has been declared

};

**2. Type Checking**

One of the most critical aspects of semantic analysis is ensuring that the program's expressions are type-correct. For example, an integer cannot be added to a string in most programming languages.

* **Expression Type Checking**: For every expression in the AST, the semantic analysis function checks that the types of the operands match the expected types for the given operation.
* **Assignment Checking**: The function ensures that a variable is assigned a value of the correct type. For instance, trying to assign a string to an integer variable should raise an error.

**Type Checking Algorithm:**

* For each node in the AST that represents an operation, the function will:
  1. Check the types of operands (variables, literals, or expressions).
  2. Ensure that the operation makes sense with those types (e.g., adding two integers).
  3. Return the type of the expression (this is necessary to propagate the type up the tree).
* **Example Type Mismatch Check**: If an expression involves the addition of an integer and a string, the semantic analyzer will raise an error, indicating the type mismatch.

if (operand1.type != operand2.type) {

throw TypeMismatchError("Cannot add different types.");

}

**3. Scope Checking**

Scope checking ensures that variables and functions are used only in the context where they are valid. For example, a variable declared inside a function cannot be used outside of it.

* **Scope Rules**:
  + **Global Scope**: Variables declared outside of any function are in the global scope.
  + **Local Scope**: Variables declared inside a function are only visible within that function.
  + **Function Scope**: Parameters are only valid inside the function they are declared in.

**Scope Checking Algorithm:**

* When the semantic analysis function processes each statement in the AST, it verifies whether the identifier being accessed (such as a variable or function) is valid within its scope.
* If an identifier is used outside of its scope, the function will raise an error.

if (symbolTable.find(identifier) == symbolTable.end() ||

symbolTable[identifier].scope != currentScope) {

throw UndeclaredIdentifierError("Variable not declared in this scope.");

}

**4. Declaration Checking**

Before using a variable, function, or any other identifier, it must be **declared** in the program. The semantic analysis function checks that:

* All variables are declared before they are used.
* Functions are defined before they are called.

**Declaration Checking Algorithm:**

* The semantic analysis function keeps track of whether an identifier has been declared using the symbol table.
* If the program attempts to use an undeclared identifier, an error is raised.

if (!symbolTable[identifier].isDeclared) {

throw UndeclaredIdentifierError("Variable or function is not declared.");

}

**5. Function Call Validation**

Function calls must match the function signature in terms of the number and types of arguments passed. This check ensures that the function call adheres to the function's definition.

* **Parameter Type Checking**: The semantic analysis function compares the types of the arguments in the function call with the types of the parameters in the function's declaration.

**Function Call Validation Algorithm:**

* When processing a function call, the function verifies that:
  1. The function exists in the symbol table.
  2. The number and types of arguments match the function's parameters.

FunctionInfo func = symbolTable[functionName];

if (func.parameterCount != arguments.size()) {

throw ArgumentCountError("Incorrect number of arguments.");

}

for (int i = 0; i < arguments.size(); ++i) {

if (arguments[i].type != func.parameters[i].type) {

throw ArgumentTypeError("Argument type mismatch.");

}

}

**6. Control Flow Analysis**

In certain programming languages, the flow of control can affect the program's execution. For instance:

* **Unreachable code**: Code after a return statement or break should not be executed.
* **Loops and conditionals**: Ensuring that loops or conditionals are used correctly and that there are no infinite loops.

**Control Flow Checking Algorithm:**

* The semantic analysis function checks that there are no logical errors in the program's control flow. For example, it ensures that there is no unreachable code or that a return statement is used appropriately.

if (hasReturnStatement && !nextStatementsAreReachable) {

throw UnreachableCodeError("Code after return is unreachable.");

}

**Semantic Analysis Flow**

The semantic analysis function typically works by recursively traversing the **AST**, performing checks on each node, and utilizing the **symbol table** to validate identifiers, types, and scopes.

**Basic Algorithm:**

void semanticAnalysis(ASTNode\* node, SymbolTable& symbolTable, string currentScope) {

switch (node->type) {

case NODE\_VARIABLE\_DECLARATION:

checkVariableDeclaration(node, symbolTable, currentScope);

break;

case NODE\_ASSIGNMENT:

checkAssignment(node, symbolTable);

break;

case NODE\_FUNCTION\_CALL:

checkFunctionCall(node, symbolTable);

break;

case NODE\_EXPRESSION:

checkExpression(node, symbolTable);

break;

// Additional checks for other node types (loops, conditionals, etc.)

}

}