****

**Lab Terminal # 04**

**Submitted by:** Isha Eman

## **Reg #:** fa21-bcs-046

**Explain the function performing semantic analyzer in your mini compiler code .**

Semantic analysis is the process of ensuring that the program has meaning and is logically correct. It checks whether the program follows the rules of the language in terms of how types, variables, and expressions are used. Unlike syntax analysis, which checks the structure of the code, semantic analysis focuses on the meaning behind the code.

#### **Purpose**:

The purpose of semantic analysis is to verify:

1. **Correct variable usage**: Ensures that variables are declared before they are used.
2. **Type compatibility**: Ensures that operations and assignments are type-appropriate (e.g., assigning an integer to a string is an error).
3. **Other logical errors**: Checks for other context-specific errors, like undeclared variables, invalid function calls, etc.

#### **Process**:

1. **AST Traversal**: The SemanticAnalyzer.cs file takes the Abstract Syntax Tree (AST) created during parsing and traverses it node by node.
2. **Checking Declarations**: For each variable or function, the analyzer checks if it has been declared before it is used.
   * For example, if a variable x is used in an expression but hasn't been declared earlier, an error is raised.
3. **Type Checking**: The analyzer checks if the types involved in operations or assignments are compatible.
   * Example: If an integer is assigned to a string variable, it will raise a type compatibility error.

**Code:-**

using System;

using System.Collections.Generic;

public class SemanticAnalyzer

{

private Dictionary<string, string> \_symbolTable; // A symbol table to track declared variables

public SemanticAnalyzer()

{

\_symbolTable = new Dictionary<string, string>(); // Initialize the symbol table

}

public void Analyze(List<AstNode> astNodes)

{

foreach (AstNode node in astNodes)

{

AnalyzeNode(node); // Analyze each node in the AST

}

}

private void AnalyzeNode(AstNode node)

{

switch (node)

{

case AssignmentNode assignmentNode:

// Check if the variable is declared

if (!\_symbolTable.ContainsKey(assignmentNode.VarName))

{

throw new Exception($"Variable '{assignmentNode.VarName}' is not declared.");

}

// Analyze the expression assigned to the variable

AnalyzeNode(assignmentNode.Expression);

// Check for type compatibility (optional, depending on how types are stored and checked)

string declaredType = \_symbolTable[assignmentNode.VarName];

string expressionType = assignmentNode.Expression.GetType().Name; // Assuming expression provides its type

if (declaredType != expressionType)

{

throw new Exception($"Type mismatch: Cannot assign {expressionType} to {declaredType}.");

}

break;

case VariableNode variableNode:

// Check if the variable is declared

if (!\_symbolTable.ContainsKey(variableNode.VarName))

{

throw new Exception($"Variable '{variableNode.VarName}' is not declared.");

}

break;

case BinaryOpNode binaryOpNode:

// Analyze both sides of the binary operation

AnalyzeNode(binaryOpNode.Left);

AnalyzeNode(binaryOpNode.Right);

// Check for type compatibility (e.g., checking types for arithmetic operations)

string leftType = binaryOpNode.Left.GetType().Name;

string rightType = binaryOpNode.Right.GetType().Name;

// Example type compatibility check for arithmetic operations

if (leftType != "int" || rightType != "int")

{

throw new Exception($"Invalid types for binary operation: {leftType} and {rightType}. Both must be 'int'.");

}

break;

// Handle other cases for additional node types

// case SomeOtherNodeType nodeType:

// AnalyzeNode(nodeType);

// break;

}

}

// Method to simulate variable declaration and storing in the symbol table

public void DeclareVariable(string varName, string varType)

{

if (\_symbolTable.ContainsKey(varName))

{

throw new Exception($"Variable '{varName}' is already declared.");

}

\_symbolTable[varName] = varType;

}

}

// Example AST Node Types

public abstract class AstNode { }

public class AssignmentNode : AstNode

{

public string VarName { get; set; }

public AstNode Expression { get; set; }

}

public class VariableNode : AstNode

{

public string VarName { get; set; }

}

public class BinaryOpNode : AstNode

{

public AstNode Left { get; set; }

public AstNode Right { get; set; }

}

#### **Examples**:

1. **Undeclared Variable**:

**Code**:

x = 10; // Error: x is undeclared

* + **Error Message**: "Error: Variable 'x' is undeclared."

1. **Type Compatibility Error**:

**Code**:

string name = "John";

name = 10; // Error: Cannot assign an integer to a string

* + **Error Message**: "Error: Cannot assign an in teger to a string."