# Semantic Analysis in a Mini Compiler

## 1. Overview

Semantic analysis ensures that the source code follows the logical rules of the language after the code has been tokenized (lexical analysis) and parsed (syntax analysis). The semantic analysis phase checks for issues such as undeclared variables, type mismatches, and incorrect function calls.

The primary tasks of semantic analysis include:

- **Type Checking**: Ensures that variables and operations use compatible types.
- **Scope Checking**: Ensures that variables and functions are declared before they are used.
- **Function Call Checking**: Ensures the correct number and types of arguments are passed to functions.
- **Declaration Checking**: Ensures that variables and functions are declared before use.

This document provides the complete **C# code** for performing semantic analysis in a **mini compiler**.

---

## 2. Classes Involved in Semantic Analysis

### SymbolTable Class

The `SymbolTable` class stores information about variables and functions declared in the program. This allows semantic analysis to verify variable types, function signatures, and other declarations.

**SymbolTable Class Code:**

```
public class SymbolTable
{
    private Dictionary<string, string> variables = new Dictionary<string, string>();
    private Dictionary<string, (List<string> parameterTypes, string returnType)> functions =
new Dictionary<string, (List<string>, string)>();

    // Add a variable to the symbol table
    public void AddVariable(string name, string type)
    {
        if (!variables.ContainsKey(name))
        {
```

```
            variables.Add(name, type);
        }
        else
        {
            Console.WriteLine($"Error: Variable '{name}' is already declared.");
        }
    }

    // Get the type of a variable
    public string GetVariableType(string name)
    {
        return variables.ContainsKey(name) ? variables[name] : null;
    }

    // Add a function to the symbol table
    public void AddFunction(string name, List<string> parameterTypes, string returnType)
    {
        if (!functions.ContainsKey(name))
        {
            functions.Add(name, (parameterTypes, returnType));
        }
        else
        {
            Console.WriteLine($"Error: Function '{name}' is already declared.");
        }
    }

    // Get function information
    public (List<string> parameterTypes, string returnType) GetFunctionInfo(string name)
    {
        return functions.ContainsKey(name) ? functions[name] : (null, null);
    }
}
```

---

## SemanticAnalyzer Class

The `SemanticAnalyzer` class is responsible for analyzing the semantic correctness of variable declarations, function calls, and expressions. It uses the **SymbolTable** to ensure variables are declared, types are compatible, and function calls are valid.

**SemanticAnalyzer Class Code:**

```
public class SemanticAnalyzer
{
    private SymbolTable symbolTable;

    public SemanticAnalyzer()
```

```
    {
        symbolTable = new SymbolTable();
    }

    // Analyze variable declaration
    public void AnalyzeVariableDeclaration(string varName, string varType)
    {
        if (symbolTable.GetVariableType(varName) != null)
        {
            Console.WriteLine($"Error: Variable '{varName}' is already declared.");
        }
        else
        {
            symbolTable.AddVariable(varName, varType);
        }
    }

    // Analyze variable usage
    public void AnalyzeVariableUsage(string varName)
    {
        if (symbolTable.GetVariableType(varName) == null)
        {
            Console.WriteLine($"Error: Variable '{varName}' is used before declaration.");
        }
    }

    // Analyze expressions and check for type compatibility
    public void AnalyzeExpression(string leftVar, string rightVar, string operation)
    {
        string leftType = symbolTable.GetVariableType(leftVar);
        string rightType = symbolTable.GetVariableType(rightVar);

        if (leftType == null || rightType == null)
        {
            Console.WriteLine("Error: Variables used in expression are not declared.");
            return;
        }

        if (operation == "+" || operation == "-" || operation == "*" || operation == "/")
        {
            if (leftType != rightType)
            {
                Console.WriteLine($"Error: Type mismatch in expression '{leftVar} {operation}
{rightVar}' (Expected the same type for both operands).");
            }
        }
        else
        {
```

```
            Console.WriteLine("Error: Unsupported operation.");
        }
    }

    // Analyze function calls and check arguments
    public void AnalyzeFunctionCall(string functionName, List<string> argumentTypes)
    {
        var (parameterTypes, returnType) = symbolTable.GetFunctionInfo(functionName);

        if (parameterTypes == null)
        {
            Console.WriteLine($"Error: Function '{functionName}' is not declared.");
            return;
        }

        if (parameterTypes.Count != argumentTypes.Count)
        {
            Console.WriteLine($"Error: Function '{functionName}' expects
{parameterTypes.Count} arguments, but {argumentTypes.Count} were provided.");
            return;
        }

        for (int i = 0; i < parameterTypes.Count; i++)
        {
            if (parameterTypes[i] != argumentTypes[i])
            {
                Console.WriteLine($"Error: Argument type mismatch for parameter {i + 1} in
function '{functionName}'. Expected {parameterTypes[i]}, but got {argumentTypes[i]}.");
            }
        }
    }
}
```

---

# 3. Integrating with Lexer and Parser

The **Lexer** (Tokenizes the source code) and **Parser** (Builds the Abstract Syntax Tree - AST) are responsible for generating the necessary structure that the **SemanticAnalyzer** can then validate.

### Lexer Class (Simplified)

```
public class Lexer
{
    public List<Token> Tokenize(string source)
    {
        // Tokenize the source code
```

```
        // (This is a simplified example of how it might look)
        List<Token> tokens = new List<Token>();
        // Process source code and generate tokens...
        return tokens;
    }
}
```

### Parser Class (Simplified)

```
public class Parser
{
    public List<ParsedStatement> Parse(List<Token> tokens)
    {
        // Parse tokens into statements (AST)
        List<ParsedStatement> statements = new List<ParsedStatement>();
        // Process tokens and build the abstract syntax tree...
        return statements;
    }
}
```

---

# 4. Example of Full Integration

The **Compiler** class integrates the Lexer, Parser, and Semantic Analyzer.

### Compiler Class

```
public class Compiler
{
    private Lexer lexer;
    private Parser parser;
    private SemanticAnalyzer semanticAnalyzer;

    public Compiler()
    {
        lexer = new Lexer();
        parser = new Parser();
        semanticAnalyzer = new SemanticAnalyzer();
    }

    public void Compile(string sourceCode)
    {
        // Step 1: Tokenize the source code
        var tokens = lexer.Tokenize(sourceCode);

        // Step 2: Parse the tokens into statements (AST)
```

```
        var statements = parser.Parse(tokens);

        // Step 3: Perform semantic analysis
        AnalyzeDeclarations(statements);
        AnalyzeStatements(statements);
    }

    // Analyze variable declarations and function declarations
    private void AnalyzeDeclarations(List<ParsedStatement> statements)
    {
        foreach (var stmt in statements)
        {
            if (stmt is VariableDeclarationStatement decl)
            {
                semanticAnalyzer.AnalyzeVariableDeclaration(decl.VarName, decl.VarType);
            }
            else if (stmt is FunctionDeclarationStatement funcDecl)
            {
                semanticAnalyzer.SymbolTable.AddFunction(funcDecl.FuncName,
funcDecl.ParameterTypes, funcDecl.ReturnType);
            }
        }
    }

    // Analyze expressions and function calls
    private void AnalyzeStatements(List<ParsedStatement> statements)
    {
        foreach (var stmt in statements)
        {
            if (stmt is ExpressionStatement exprStmt)
            {
                semanticAnalyzer.AnalyzeExpression(exprStmt.Left, exprStmt.Right,
exprStmt.Operator);
            }
            else if (stmt is FunctionCallStatement funcCallStmt)
            {
                semanticAnalyzer.AnalyzeFunctionCall(funcCallStmt.FunctionName,
funcCallStmt.ArgumentTypes);
            }
        }
    }
}
```

---

# 5. Example Workflow

The following steps describe how the **Compiler** class processes the source code:

1. **Lexer**: Tokenizes the source code into tokens (e.g., identifiers, operators, keywords).
2. **Parser**: Converts the tokens into an Abstract Syntax Tree (AST).
3. **SemanticAnalyzer**: Checks the semantic correctness of the AST:
    - Ensures variables are declared before use.
    - Ensures type compatibility in expressions.
    - Ensures functions are called with the correct arguments.

---

# 6. Conclusion

The **SemanticAnalyzer** class ensures that the program follows the logical rules of the language. By checking for type mismatches, undeclared variables, and incorrect function calls, semantic analysis ensures the code is valid before code generation.

This approach integrates smoothly with **Lexer** and **Parser** components, making the semantic analysis phase a vital part of a mini compiler pipeline.