# Semantic Analysis in a Mini Compiler

## 1. Overview

Semantic analysis ensures that the program adheres to the logical rules of the language after the code has passed through **lexical analysis** and **parsing**. While lexical analysis identifies tokens (such as keywords, variables, and operators) and parsing ensures that the program follows the syntactical rules (grammar), semantic analysis checks for issues like variable declarations, type compatibility, and function correctness.

The key tasks of **semantic analysis** include:

- **Type Checking**: Ensures operations involving variables are valid with respect to their types.
- **Scope Checking**: Ensures variables and functions are declared before use.
- **Function Call Checking**: Ensures that functions are called with the correct arguments.
- **Declaration Checking**: Ensures that variables and functions are declared before they are used.

This documentation outlines the **semantic analysis** component in the context of a **mini compiler** written in **C#**. The integration of **Lexer** and **Parser** is assumed as a part of the compiler pipeline.

---

## 2. Components of the Semantic Analyzer

The following classes are involved in performing semantic analysis:

### SymbolTable Class

The `SymbolTable` class stores information about declared variables and functions. It allows semantic analysis to look up variable types, function declarations, and function argument types.

**Code for SymbolTable:**
csharp
Copy 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)>();

    // Adds 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.");
        }
    }

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

    // Adds 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.");
        }
    }

    // Retrieves information about a function from the symbol table
    public (List<string> parameterTypes, string returnType)
GetFunctionInfo(string name)
    {
        return functions.ContainsKey(name) ? functions[name] :
(null, null);
    }
}
```

---

## SemanticAnalyzer Class

The `SemanticAnalyzer` class performs the core semantic analysis. It checks variable declarations, usage, expression types, and function call correctness.

**Code for SemanticAnalyzer:**

csharp

Copy code

```
public class SemanticAnalyzer
{
    private SymbolTable symbolTable;

    public SemanticAnalyzer()
    {
        symbolTable = new SymbolTable();
    }

    // Analyzes a variable declaration
    public void AnalyzeVariableDeclaration(string varName, string
varType)
    {
        // Ensure variable is not already declared
        if (symbolTable.GetVariableType(varName) != null)
        {
            Console.WriteLine($"Error: Variable '{varName}' is
already declared.");
        }
        else
        {
            symbolTable.AddVariable(varName, varType);
        }
    }
    // Analyzes variable usage (checks if the variable is declared)
    public void AnalyzeVariableUsage(string varName)
    {
        // Ensure the variable has been declared before use
        if (symbolTable.GetVariableType(varName) == null)
        {
            Console.WriteLine($"Error: Variable '{varName}' is used
before declaration.");
        }
    }

    // Analyzes an expression and checks 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;
        }

        // Type checking: check for valid types based on the
operation
        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.");
        }
    }

    // Analyzes a function call and checks argument types and count
    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 the Lexer and Parser

In a typical mini compiler setup, the **Lexer** breaks the source code into tokens, and the **Parser** arranges those tokens into an Abstract Syntax Tree (AST) based on the syntax rules of the language. After parsing, **semantic analysis** checks the meaning and correctness of the program using the **SymbolTable** and **SemanticAnalyzer** classes.

## Integration with Lexer and Parser

### Lexer

The **Lexer** class takes the source code as input and generates a sequence of tokens. It identifies keywords, operators, identifiers, literals, and other elements.

### Parser

The **Parser** class uses these tokens to create an **AST** representing the program structure.

## Example Integration:

Assuming we have parsed the tokens, we can use the `SemanticAnalyzer` to validate the parsed data:

**Example Usage of Semantic Analyzer with Parsed Tokens:**
csharp
Copy code
```
public class Compiler
{
    private SemanticAnalyzer semanticAnalyzer;

    public Compiler()
    {
        semanticAnalyzer = new SemanticAnalyzer();
    }

    // Function to process declarations and perform semantic
analysis
    public 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);
            }
        }
    }

    // Function to analyze expressions or statements
    public 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);
            }
        }
    }
}
```

In this example, the `Compiler` class orchestrates the **semantic analysis** phase by analyzing **declarations** and **statements**.

---

## 4. Example of Full Integration

### Lexer (simplified)

csharp
Copy code

```
public class Lexer
{
    public List<Token> Tokenize(string source)
    {
        // Tokenizing logic: Returns a list of tokens
        // (not fully implemented for brevity)
    }
}
```

### Parser (simplified)

csharp
Copy code

```
public class Parser
{
    public List<ParsedStatement> Parse(List<Token> tokens)
    {
        // Parsing logic: Converts tokens into a list of parsed
statements
        // (not fully implemented for brevity)
    }
}
```

### Compiler Workflow

1. The **Lexer** tokenizes the source code.
2. The **Parser** creates an Abstract Syntax Tree (AST) based on the tokens.
3. The **SemanticAnalyzer** performs semantic checks on the parsed AST.

csharp
Copy code

```
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)
    {
        // Tokenize the source code
        var tokens = lexer.Tokenize(sourceCode);

        // Parse the tokens into statements
        var statements = parser.Parse(tokens);

        // Perform semantic analysis
        AnalyzeDeclarations(statements);
        AnalyzeStatements(statements);
    }

    // Analyze declarations and statements using semantic analysis
    private void AnalyzeDeclarations(List<ParsedStatement>
statements) { ... }
    private void AnalyzeStatements(List<ParsedStatement> statements)
{ ... }
}
```

---

## 5. Conclusion

Semantic analysis is a crucial phase of the compiler pipeline that ensures the logical correctness of the program. By checking for type mismatches, undeclared variables, and incorrect function calls, the **SemanticAnalyzer** helps detect errors early in the compilation process. When integrated with the **Lexer** and **Parser**, it can validate a wide range of program correctness issues before code generation.