COMPILER CONSTRUCTION

**Semantic Analysis in a Mini Compiler**

**Semantic analysis** is a critical phase of compilation that comes after **syntax analysis**. While syntax analysis ensures the code adheres to the grammar rules of the language, semantic analysis verifies that the code makes sense logically and adheres to the rules of the language semantics.

**Purpose of Semantic Analysis**

1. **Type Checking**: Ensures that operations are performed on compatible data types (e.g., you cannot add a string to an integer).
2. **Scope Resolution**: Verifies that variables and functions are declared before they are used and checks their scopes.
3. **Function and Variable Binding**: Ensures the correct function or variable is referenced.
4. **Semantic Rules Enforcement**: Checks language-specific rules, such as a return statement being present in all execution paths of a function.

**How Semantic Analysis Works**

1. **Symbol Table Creation**:
   * A symbol table is constructed to store information about identifiers (e.g., variables, functions, types) in the program.
   * Each entry contains details like the name, type, scope, and other attributes of an identifier.
2. **Traversing the Abstract Syntax Tree (AST)**:
   * The compiler traverses the AST and validates the semantics of the code using the symbol table.

### ****Example in a Mini Compiler****

#### **Input Source Code**:

int x = 10;

float y = x + 20.5;

int z = y \* 2; // Error: Type mismatch

### ****Implementation****

#### **Symbol Table Example**:

public class SymbolTable

{

private Dictionary<string, Symbol> \_table = new();

public void AddSymbol(string name, string type, int scopeLevel)

{

if (\_table.ContainsKey(name))

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

\_table[name] = new Symbol(name, type, scopeLevel);

}

public Symbol GetSymbol(string name)

{

if (!\_table.ContainsKey(name))

throw new Exception($"Error: Variable '{name}' is not declared.");

return \_table[name];

}

}

public class Symbol

{

public string Name { get; }

public string Type { get; }

public int ScopeLevel { get; }

public Symbol(string name, string type, int scopeLevel)

{

Name = name;

Type = type;

ScopeLevel = scopeLevel;

}

}

1. **Type Checking Example**:

public class SemanticAnalyzer

{

private SymbolTable \_symbolTable = new();

public void Analyze(Node node)

{

if (node is DeclarationNode declaration)

{

\_symbolTable.AddSymbol(declaration.Name, declaration.Type, declaration.ScopeLevel);

}

else if (node is AssignmentNode assignment)

{

var symbol = \_symbolTable.GetSymbol(assignment.Variable);

var expressionType = EvaluateExpression(assignment.Expression);

if (symbol.Type != expressionType)

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

}

// Recursively analyze other nodes

}

private string EvaluateExpression(Node expression)

{

// Simplified example: Determines the type of an expression

if (expression is NumberNode) return "int";

if (expression is FloatNode) return "float";

if (expression is BinaryExpressionNode binary)

{

var leftType = EvaluateExpression(binary.Left);

var rightType = EvaluateExpression(binary.Right);

if (leftType != rightType)

throw new Exception($"Type Mismatch in expression: {leftType} vs {rightType}");

return leftType; // Assuming consistent types

}

throw new Exception("Unknown expression type");

}

}

1. **Example Usage**:

class Program

{

static void Main(string[] args)

{

// AST for `int x = 10; float y = x + 20.5; int z = y \* 2;`

var program = new ProgramNode(new List<Node>

{

new DeclarationNode("x", "int", 0),

new DeclarationNode("y", "float", 0),

new AssignmentNode("y", new BinaryExpressionNode(

new VariableNode("x"),

new Token("Operator", "+"),

new FloatNode(20.5f))),

new DeclarationNode("z", "int", 0),

new AssignmentNode("z", new BinaryExpressionNode(

new VariableNode("y"),

new Token("Operator", "\*"),

new NumberNode(2)))

});

try

{

var analyzer = new SemanticAnalyzer();

analyzer.Analyze(program);

Console.WriteLine("Semantic analysis passed.");

}

catch (Exception e)

{

Console.WriteLine($"Semantic error: {e.Message}");

}

}

### ****Output****

### If No Errors:

Semantic analysis passed.

**If Errors Exist (e.g., Type Mismatch):**

##### Semantic error: Type Mismatch: Cannot assign float to int

**Common Semantic Errors**

1. **Type Mismatch**:
   * Attempting to assign a value of one type to a variable of another incompatible type.
2. **Undeclared Variables**:
   * Using a variable that has not been declared.
3. **Invalid Operations**:
   * Applying operators to incompatible types (e.g., adding a string to an integer).
4. **Scope Violations**:
   * Using a variable outside its scope.

Semantic analysis ensures the logical consistency of the program, preparing it for further optimization and code generation phases. This step is essential to prevent runtime errors caused by semantic inconsistencies.