Assignment 03: Semantic Analysis & Intermediate Code Generation

Course: Principles of Compiler Design

Name: Abraham Addisu

ID: 1504850

1. Introduction to Semantic Analysis

Semantic analysis is the phase of compilation that bridges the gap between syntax (structure) and code generation. While the parser ensures the code follows grammar rules, the semantic analyzer ensures it follows the rules of the language, such as type compatibility and variable declaration.

Key Objectives Addressed:

- Symbol Table Management: Creating a registry for variables and their types.
- Type Checking: Verifying operands in arithmetic expressions.
- Semantic Error Detection: Reporting mismatches between int and float.

2. Type Checking Implementation for Arithmetic Expressions

Following the requirement to implement a type checker for integers and floats, the logic follows a specific Attribute Grammar approach where types are propagated up the Abstract Syntax Tree (AST).

The C++ Semantic Analyzer Code

```
#include <iostream>

#include <string>

#include <map>


using namespace std;


// 1. Symbol Table for Scope Management

map<string, string> symbolStack;


// 2. Type Checking Rules
```

```
string checkTypes(string type1, string type2) {
    if (type1 == "int" && type2 == "int") return "int";
    if (type1 == "float" || type2 == "float") {
        // Objective: Implicit Type Promotion (Promotion of int to float)
        return "float";
    }
    return "error";
}

int main() {
    // Input: x = 5 + 2.5
    symbolStack["x"] = "float"; // Variable declaration in Symbol Table

    string operand1Type = "int";   // Derived from '5'
    string operand2Type = "float"; // Derived from '2.5'

    cout << "--- Semantic Analysis Start---" << endl;

    string resultType = checkTypes(operand1Type, operand2Type);

    if (resultType == "error") {
        cout << "SEMANTIC ERROR: Incompatible types found!" << endl;
    } else {
        cout << "Result of (int + float) is: " << resultType << endl;
        cout << "Action: Implicit Conversion (intToFloat) applied." << endl;
    }

    return 0;
}
```

---

3. Semantic Error Detection & Validation

Based on the assignment objectives, the checker must handle subtle program errors.

Test Case	Left Operand	Right Operand	Result / Error	Objective Met
10 + 5	int	int	int	Type Matching
5 + 2.5	int	float	float	**Type Promotion**
x + 2	undefined	int	**Error**	**Undeclared Variable**

4. Intermediate Representation (Three-Address Code)

As per Objective 10, the AST is converted into Three-Address Code (TAC) to simplify optimization.

For the expression x = 5 + 2.5:

1. t1 = intToFloat(5) *(Implicit Conversion)*
2. t2 = t1 + 2.5
3. x = t2

5. Summary of Learning Outcomes

- Symbol Tables: Constructed a simple map to manage variable types.
- Static Type Checking: Enforced rules to ensure integers and floats interact correctly.
- Optimization Preparation: Defined TAC which allows for future local optimizations like Constant Folding.