Assignment 6: Build a Type Checker for Assignment Statements

This assignment focuses on implementing logic to verify that the right-hand side (RHS) expression's type is compatible with the left-hand side (LHS) variable in an assignment statement (e.g., x = expr;). The type checker must detect illegal assignments and handle implicit conversions (coercion) where allowed, aligning with the objectives of semantic analysis: implementing type checking rules, detecting semantic errors like type mismatches, and **ensuring the** program is prepared for code generation.

1. Theoretical Explanation

In compiler design, semantic analysis follows syntax analysis and ensures the program adheres to the language's type rules. For assignment statements:

- **LHS:** Typically a variable (identifier) or l-value (e.g., array element, pointer dereference). Its type is looked up from the symbol table.
- **RHS**: An expression whose type is computed recursively (e.g., via abstract syntax tree (AST) traversal).
- **Compatibility Check**: The RHS type must match the LHS type exactly or be implicitly convertible (coercion). Examples:
    - Allowed: int x = 5; (exact match), float y = 3; (implicit int-to-float coercion).
    - Disallowed: int z = "hello"; (string to int mismatch), char c = 3.14; (float to char may lose data, depending on rules).
- **Implicit Conversions (Coercion)**: Defined by a type hierarchy or promotion rules (e.g., int → float, char → int). Widening conversions (no data loss) are usually allowed; narrowing (potential loss) may warn or error.
- **Error Detection**: If incompatible, report semantic errors with details (line number, types involved) to aid debugging.
- **Integration**: This is part of a semantic pass over the AST. It uses the symbol table for LHS lookup and recursive expression type computation for RHS.
- **Benefits**: Prevents runtime errors (e.g., type mismatches), enables optimizations, and prepares for intermediate code generation (e.g., inserting casts in TAC).

Key steps in implementation:

1. Lookup LHS variable type from symbol table.
2. Compute RHS expression type (recursive if nested).
3. Check compatibility using a coercion matrix or rules.
4. If compatible, proceed (possibly insert implicit cast in AST/IR); else, error.

This ties into broader objectives: building symbol tables, static type checking, detecting undeclared variables/type mismatches, and preparing for IR generation.

2. Sample Implementation (C++)

Below is a complete, self-contained C++ program implementing a basic type checker for assignment statements. It simulates:

- A symbol table (std::map<std::string, Type>).
- Expression type computation (simple recursive function).
- Coercion rules (widening allowed: char → int → float; no narrowing without explicit cast).
- Error reporting.

Types are enums: UNKNOWN, CHAR, INT, FLOAT, STRING (for simplicity).

The program processes sample assignments, checks types, and outputs results or errors.bool checkAssignment(const std::string& varName, const Expression& expr, int lineNum) {

C++ code implementation

```
#include <iostream>
#include <map>
#include <string>
#include <variant>  // For flexible expression values

// Enum for data types
enum class Type { UNKNOWN, CHAR, INT, FLOAT, STRING };

// Helper to get type name as string (for error messages)
std::string typeToString(Type t) {
  switch (t) {
    case Type::CHAR: return "char";
    case Type::INT: return "int";
    case Type::FLOAT: return "float";
    case Type::STRING: return "string";
    default: return "unknown";
    }
}

// Symbol table: maps variable names to types
std::map<std::string, Type> symbolTable;

// Struct for simple expressions (literal or variable)
struct Expression {
    std::variant<int, float, char, std::string> value;  // Literal value
    std::string varName;  // If it's a variable reference
    bool isLiteral;  // Flag to distinguish

    Expression() : isLiteral(false) {}
};

// Function to compute expression type (recursive for nested, but simplified here)
Type getExpressionType(const Expression& expr) {
    if (expr.isLiteral) {
        if (std::holds_alternative<int>(expr.value)) return Type::INT;
        if (std::holds_alternative<float>(expr.value)) return Type::FLOAT;
        if (std::holds_alternative<char>(expr.value)) return Type::CHAR;
        if (std::holds_alternative<std::string>(expr.value)) return Type::STRING;
    } else {
        // Lookup variable type
        auto it = symbolTable.find(expr.varName);
        if (it != symbolTable.end()) return it->second;
    }
    return Type::UNKNOWN;  // Error or undeclared
}
```

```
// Coercion check: Can srcType be assigned to destType? (widening allowed)
bool isCompatible(Type destType, Type srcType) {
    if (destType == srcType) return true;  // Exact match

    // Widening conversions
    if (destType == Type::INT && srcType == Type::CHAR) return true;
    if (destType == Type::FLOAT && (srcType == Type::INT || srcType == Type::CHAR)) return true;

    // No narrowing (e.g., float to int) without explicit cast
    return false;
}

// Main type checker for assignment: var = expr;
// Step 1: Get LHS type (variable must be declared)
    auto it = symbolTable.find(varName);
    if (it == symbolTable.end()) {
        std::cerr << "Error (line " << lineNum << "): Undeclared variable '" << varName << "'." << std::endl;
        return false;
    }
    Type lhsType = it->second;

    // Step 2: Get RHS type
    Type rhsType = getExpressionType(expr);
    if (rhsType == Type::UNKNOWN) {
        std::cerr << "Error (line " << lineNum << "): Invalid or undeclared expression in assignment." << std::endl;
        return false;
    }

    // Step 3: Check compatibility
    if (!isCompatible(lhsType, rhsType)) {
        std::cerr << "Error (line " << lineNum << "): Type mismatch in assignment. Cannot assign
"
                  << typeToString(rhsType) << " to " << typeToString(lhsType) << "." << std::endl;
        return false;
    }

    // Success (could insert implicit cast in AST here if coercion needed)
    std::cout << "Assignment valid (line " << lineNum << "): " << varName << " = [expr of type "
<< typeToString(rhsType) << "]." << std::endl;
    return true;
}

int main() {
    // Populate symbol table (simulate declarations)
    symbolTable["x"] = Type::INT;
    symbolTable["y"] = Type::FLOAT;
    symbolTable["z"] = Type::CHAR;
    symbolTable["str"] = Type::STRING;

    // Test case 1: Exact match (int = int)
    Expression expr1;
    expr1.isLiteral = true;
    expr1.value = 42;
    checkAssignment("x", expr1, 10);

    // Test case 2: Coercion (float = int)
    Expression expr2;
    expr2.isLiteral = true;
    expr2.value = 5;
    checkAssignment("y", expr2, 20);

    // Test case 3: Mismatch (char = float) - narrowing not allowed
    Expression expr3;
    expr3.isLiteral = true;
    expr3.value = 3.14f;
    checkAssignment("z", expr3, 30);

    // Test case 4: Variable reference (float = float var)
    Expression expr4;
    expr4.isLiteral = false;
    expr4.varName = "y";
    checkAssignment("y", expr4, 40);

    // Test case 5: Undeclared var
    checkAssignment("undeclared", expr1, 50);

    // Test case 6: String mismatch
    Expression expr5;
    expr5.isLiteral = true;
    expr5.value = "hello";
    checkAssignment("x", expr5, 60);

    return 0;
}
```

Explanation of Code:

- **Symbol Table**: Stores variable types (simulate declarations).
- **Expression Type Computation**: Handles literals or variable lookups.
- **Compatibility**: Uses rules for exact match or widening (e.g., int to float allowed, but not reverse to avoid data loss).
- **Error Handling**: Prints detailed messages with line numbers.
- **Output Example** (running the program):

text

```
Assignment valid (line 10): x = [expr of type int].
Assignment valid (line 20): y = [expr of type int].
Error (line 30): Type mismatch in assignment. Cannot assign float to char.
Assignment valid (line 40): y = [expr of type float].
Error (line 50): Undeclared variable 'undeclared'.
Error (line 60): Type mismatch in assignment. Cannot assign string to int.
```

This demonstrates detection of mismatches and support for coercion.

3. Problem-Solving and Extensions

- **Edge Cases**: Handle undeclared vars (error), constants (e.g., const int = float disallowed if narrowing), arrays/pointers (extend getExpressionType for sub-types).
- **Integration with Compiler Pipeline**: In a full compiler (e.g., using Flex/Bison), hook this into the semantic phase after parsing the AST. For coercion, insert cast nodes in AST if needed (e.g., int to float).