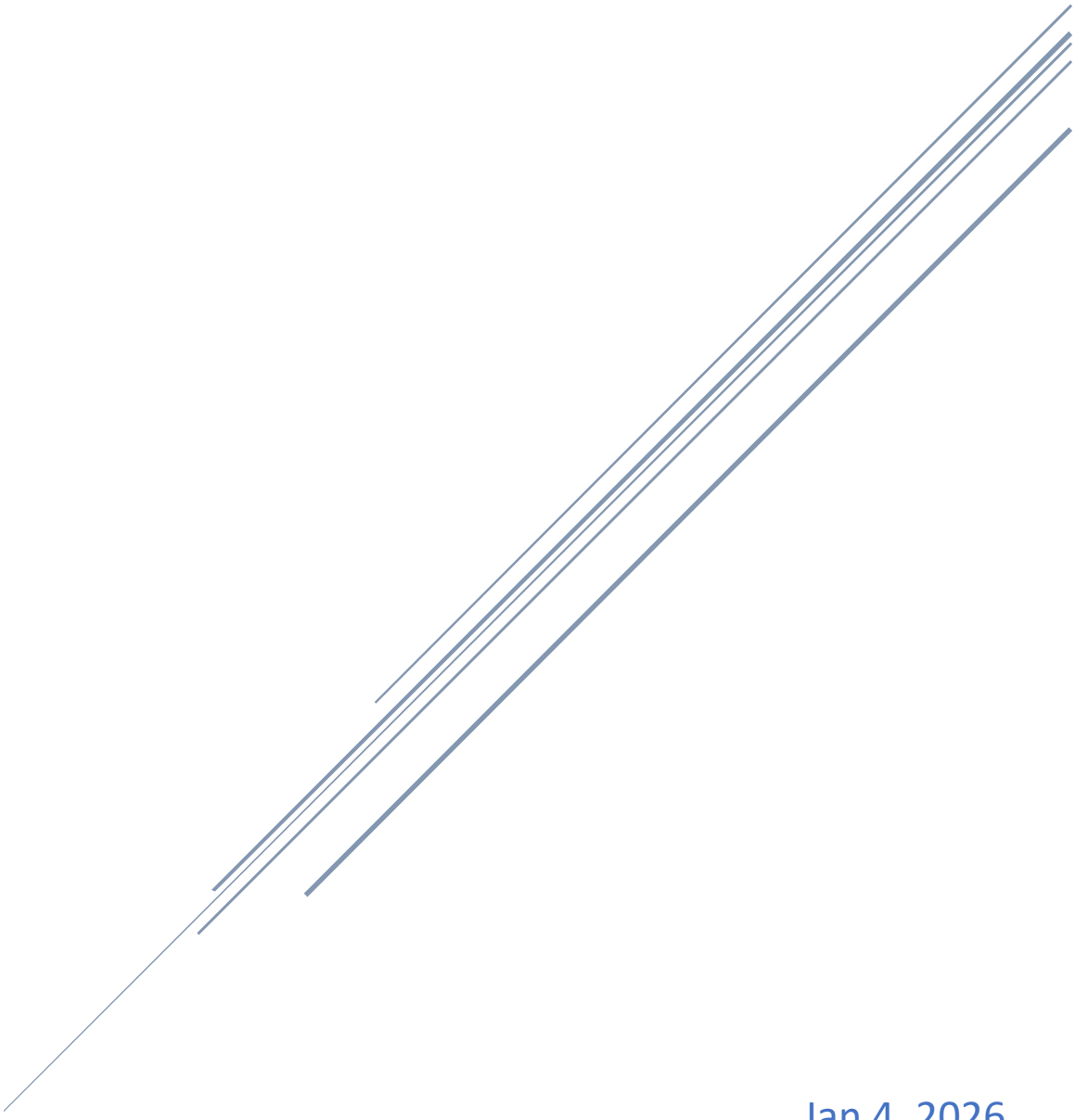
# PRINCIPLE OF COMPILER DESIGN

## Individual assignment - 3



Jan 4, 2026

Yonatan Ayisheshim [BDU1508377]

# Contents

**Principles of Compiler Design** ..... 2
**Semantic Analysis Assignment** ..... 2
**Question 56: Implement Semantic Checks for Return Type Consistency** ..... 2
**1. Introduction** ..... 2
**2. Problem Definition** ..... 2
**3. Importance of Return Type Consistency** ..... 2
**4. Semantic Rules for Return Statements** ..... 2
**5. Control Flow Considerations** ..... 3
**Example (Incorrect):** ..... 3
**6. Algorithm for Return Type Checking** ..... 3
**7. Return Type Consistency Check** ..... 4
**8. Semantic Check (Pseudocode)** ..... 4
**9. Handling Multiple Return Paths** ..... 5
**Example (Correct):** ..... 5
**10. Common Semantic Errors Detected** ..... 5
**11. Integration with Compiler Pipeline** ..... 5
**Conclusion** ..... 6
**References** ..... 6

# Principles of Compiler Design

## Semantic Analysis Assignment

### Question 56: Implement Semantic Checks for Return Type Consistency

# 1. Introduction

Semantic analysis is a critical phase in the compiler front end that ensures the program is **meaningful and logically correct**, beyond syntactic correctness. One essential semantic rule is **return type consistency**, which verifies that every function returns values compatible with its declared return type.

Failure to enforce return type consistency can lead to **runtime errors**, undefined behavior, or incorrect code generation. Therefore, modern compilers perform strict semantic checks to ensure that all return statements conform to function signatures.

# 2. Problem Definition

**Return type consistency** requires that:

1. Every function with a non-void return type must return a value.
2. The type of every return expression must be compatible with the function's declared return type.
3. All execution paths in a function must satisfy the return requirement.

Violations of these rules represent **semantic errors** and must be detected during compilation.

# 3. Importance of Return Type Consistency

Ensuring return type consistency is essential because:

- It prevents **type mismatches** during function invocation.
- It ensures correctness during **intermediate code generation**.
- It enables safe **register allocation and calling conventions**.
- It avoids undefined behavior in compiled programs.

Compilers such as GCC, Clang, and Java's `javac` enforce these rules strictly.

# 4. Semantic Rules for Return Statements

Let a function be defined as: `T f(...) { body }`

The semantic rules are:

| Rule ID | Description |
|---|---|
| **R1** | If `T ≠ void`, every execution path must end with a `return` |
| **R2** | If `T = void`, `return expr` is illegal |
| **R3** | If `T ≠ void`, `return expr` must exist |
| **R4** | Type(expr) must be compatible with `T` |
| **R5** | Implicit conversions are allowed only if defined by the language |

# 5. Control Flow Considerations

A function may contain:

- Conditional statements
- Loops
- Early returns

Therefore, **control-flow-aware analysis** is required.

### Example (Incorrect):

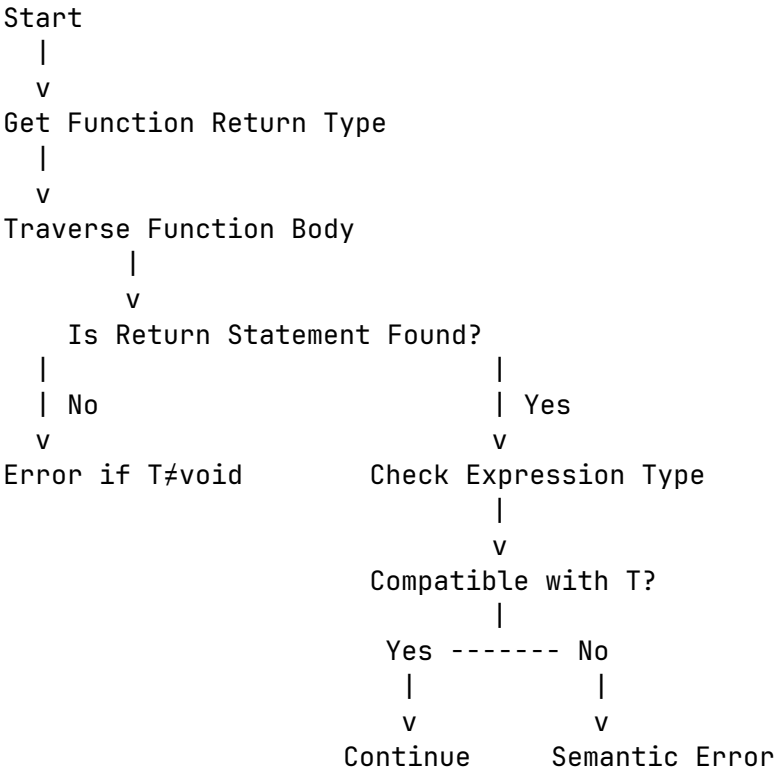
```
int f(int x) {
    if (x > 0)
        return x;
    // Missing return
}
```

This results in semantic Error: Not all paths return a value.

# 6. Algorithm for Return Type Checking

1. Retrieve function return type `T`
2. Traverse the function body AST
3. Collect all `return` statements
4. For each return:
    - Check presence of expression
    - Type-check expression against `T`
5. Analyze control-flow paths
6. Report semantic errors if violations occur

# 7. Return Type Consistency Check




# 8. Semantic Check (Pseudocode)

```
void checkReturnConsistency(FunctionNode* func) {
    Type declaredType = func->returnType;
    bool hasReturn = false;

    for (auto stmt : func->body) {
        if (stmt->isReturn()) {
            hasReturn = true;

            if (declaredType == VOID && stmt->expr != nullptr) {
                error("Void function cannot return a value");
            }

            if (declaredType != VOID && stmt->expr == nullptr) {
                error("Non-void function must return a value");
            }

            if (!isCompatible(stmt->expr->type, declaredType)) {
                error("Return type mismatch");
            }
```

```
        }
    }

    if (declaredType != VOID && !hasReturn) {
        error("Missing return statement");
    }
}
```

# 9. Handling Multiple Return Paths

### Example (Correct):

```
int max(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}
```

All execution paths return an `int`.

# 10. Common Semantic Errors Detected

| Error Type | Example |
|---|---|
| **Missing return** | Non-void function exits without return |
| **Type mismatch** | `int` function returning `float` |
| **Invalid return** | `return value;` in void function |
| **Inconsistent paths** | One branch returns, another does not |

# 11. Integration with Compiler Pipeline

Return type checking occurs:

- After **symbol table construction**
- After **type inference**
- Before **IR generation**

This ensures:

- ✓ Safe TAC generation
- ✓ Correct function epilogues
- ✓ Reliable optimizations

# Conclusion

Semantic checks for return type consistency are **mandatory** for building a correct and reliable compiler. By ensuring that all functions return values consistent with their declared types across all execution paths, the compiler prevents serious runtime failures and enforces strong type safety.

This check exemplifies how semantic analysis bridges syntax and execution correctness in compiler design.

# References


1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D.
   *Compilers: Principles, Techniques, and Tools*, Pearson.
2. Grune, D., & Jacobs, C. J.
   *Parsing Techniques: A Practical Guide*, Springer.