PRINCIPLES OF COMPILER DESIGN

Individual Assignment 03 Semantic Rules & Error Detection

QUESTION - 58

Name: Abrham Abebaw
Course Code: SEng4031
Id: BDU1504862
Section: B

Table of Contents

Topic Area: Semantic Analysis

Assignment No.: 58

58. Enforce const correctness in assignments. You must design semantic checks ensuring that const qualified variables, parameters, and objects are not modified. The assignment includes checking indirect modifications via pointers or references.

1. Introduction

Semantic analysis is a crucial phase of compiler design that ensures a program is **meaningful and logically correct**, beyond syntactic correctness. One of the most important semantic constraints enforced by modern compilers is **const-correctness**.

Const-correctness guarantees that variables, parameters, and objects declared as const **cannot be modified**, either directly or indirectly. This rule improves:

- Program safety
- Predictability
- Optimization opportunities
- Error detection at compile time

This assignment focuses on designing **semantic rules and error detection mechanisms** that prevent illegal modifications of const-qualified entities.

2. What Is Const-Correctness?

A const qualifier specifies that the value of a variable **must remain unchanged** after initialization.

Key Principle

If an entity is declared const, **any attempt to modify it must be rejected during semantic analysis.**

This includes:

- Direct assignments
- Modifications via pointers or references

- Mutating const objects
- Passing const arguments to non-const parameters

3. Role of Semantic Analysis

During semantic analysis, the compiler:

1. Builds **symbol tables**
2. Tracks **type qualifiers** (const, volatile, etc.)
3. Validates **assignments and expressions**
4. Detect **illegal operations**

Const-correctness is **not** checked during lexical or syntax analysis, it is a **semantic rule**.

4. Symbol Table Design for Const Enforcement

Each symbol table entry must store:

Attribute	Description
Name	Variable or function name
Type	Data type (int, float, object, pointer)
Const Flag	Indicates whether entity is const
Scope	Local, global, block, function
Indirection Level	Pointer/reference depth

Example Symbol Table Entry

Name: x

Type: int

Const: true

Scope: Local

5. Semantic Rules for Const-Correctness

Rule 1: No Assignment to Const Variables

Semantic Rule

If LHS is const-qualified, assignment is illegal.

`Invalid Code`

```
const int x = 10;

x = 20;  // Semantic Error
```

Compiler Error

Error: Cannot assign to const variable 'x'

Rule 2: Const Parameters Cannot Be Modified

Invalid Code

```
void update (const int a) {

    a = 5; // Error

}
```

Explanation

- Parameters are part of the symbol table
- const applies to the parameter scope
- Assignment violates const-correctness

Rule 3: Indirect Modification via Pointers

Invalid Code

```
const int x = 10;

int* p = &x;

*p = 20; // Semantic Error
```

Why This Is Illegal?

- x is const

- p indirectly modifies x
- Compiler must track **pointer target qualifiers**

Semantic Check

If (*p) points to const → modification forbidden

Rule 4: Const Pointers vs Pointer to Const

Declaration	Meaning	Can Modify Value?	Can Modify Pointer?
const int* p	Pointer to const int	NO	YES
int* const p	Const pointer	YES	NO
const int* const p	Const pointer to const	NO	NO

Invalid Code

```
const int* p = &x;

*p = 30; // Error
```

Rule 5: Const Objects and Member Functions

Invalid Code

```
class A {

public:

    int x;

};


const A obj;

obj.x = 10; // Semantic Error
```

Explanation

- obj is const
- All its data members become read-only

Rule 6: Const Methods in Objects

```
class A {
public:
    int x;
    void update() const {
        x = 5;   // Error
    }
};
```

Reason

- const method promises not to modify object state
- Compiler enforces this using semantic rules

6. Attribute Grammar for Const Checking

Attributes Used

- type
- isConst
- isLValue

Grammar Rule

Assignment → LHS = RHS

Semantic Rule

if LHS.isConst == true

→ error ("Assignment to const")

7. Error Detection Strategy

Compiler Steps

1. Look up LHS in symbol table
2. Check const qualifier
3. Trace pointer/reference targets
4. Reject illegal writes
5. Generate descriptive error message

8. Example: Complete Semantic Violation Scenario

```
const int x = 10;

const int* p = &x;

int y = 5;

p = &y; // Valid

*p = 20; // Error
```

Detected Errors

- *p refers to const memory
- Indirect modification detected

9. Why Const-Correctness Matters

✓ Prevents accidental bugs
✓ Improves compiler optimizations
✓ Enables safe parallel execution
✓ Enhances API reliability
✓ Supports advanced features (closures, generics)

10. Learning Outcomes Achieved

After enforcing const-correctness:

- ✓ Construct symbol tables with qualifiers
- ✓ Perform static semantic checks
- ✓ Detect subtle program errors
- ✓ Prepare AST for IR generation
- ✓ Improve program safety before code generation

11. Conclusion

Const-correctness enforcement is a **core semantic analysis responsibility**. By integrating const tracking into symbol tables, attribute grammars, and assignment rules, a compiler can detect both **direct and indirect illegal modifications** early in the compilation process. This ensures correctness, safety, and optimization readiness.