Project Introduction and Grammar Analysis

1. Introduction

This project presents the design and implementation of a multi-phase plagiarism detection system for C-like programming languages. The primary objective is to detect structural similarity between two source code files, even when superficial modifications are introduced to disguise copying.

Traditional plagiarism detection techniques rely heavily on token-based comparison. While such methods are effective for identifying direct textual similarity, they are highly sensitive to cosmetic changes such as variable renaming, formatting differences, and minor syntactic alterations. As a result, token-level comparison often fails to detect deeper structural equivalence between programs.

2. Purpose of the Grammar

To construct Abstract Syntax Trees, the system uses an ANTLR-defined grammar named 'Expr'. The grammar defines a simplified C-like language that captures essential programming constructs while remaining manageable for structural analysis.

- Function declarations
- Variable declarations
- Assignment statements
- Control flow structures (if, while, for)
- Return statements
- Expression statements
- Arithmetic, relational, and logical expressions
- Unary operations
- Function calls
- Basic primitive types (int, float, char, void)

3. AST Design

The grammar is converted into an Abstract Syntax Tree using the following node types:

- PROGRAM
- FUNCTION
- PARAM
- BLOCK
- FUNCTION_CALL
- VAR_DECL
- ASSIGN
- RETURN
- IF

- WHILE
- FOR
- BIN_OP
- UNARY_OP
- IDENTIFIER
- CONSTANT

The AST intentionally simplifies syntactic details and focuses only on structural constructs. Identifiers and constants are normalized to reduce sensitivity to superficial differences.

4. Grammar Simplifications and Limitations

4.1 No Arrays or Pointers

The grammar does not support array declarations, array indexing, pointer types, or pointer arithmetic. Programs heavily using arrays or pointer-based manipulation cannot be fully represented.

4.2 No Increment/Decrement Operators

Operators such as i++, i--, ++i, and --i are not included. Updates must be written explicitly as assignment expressions (e.g., i = i + 1).

4.3 No Switch Statements

The grammar does not support switch, case, or default constructs.

4.4 No Complex Type System

Only primitive types (int, float, char, void) are supported. Structs, enums, typedefs, and user-defined types are not included.

4.5 No Preprocessor Directives

The grammar does not support #include, #define, or macros. Only simple comments are skipped.

4.6 No Object-Oriented Features

The grammar is strictly procedural and does not support classes, inheritance, or access modifiers.

4.7 Limited Function Call Semantics

Function calls are supported syntactically but no type checking, symbol resolution, or argument validation is performed.

4.8 No Semantic Analysis

The system performs only syntactic structural comparison and does not include type checking, scope analysis, data flow analysis, or control flow validation.

4.9 No Advanced Error Recovery

The grammar assumes syntactically valid input and does not implement advanced error recovery mechanisms.