CSE422: Programming Languages and Compilers
COOL Compiler in Java
Project Report

Zahraa Selim (120210083)

May 19, 2025

Contents

1 Introduction

1.1 Purpose

The Classroom Object-Oriented Language (COOL) is a statically typed, object-oriented language designed for educational purposes, as outlined in the COOL Reference Manual [1]. This project implements a compiler in Java that translates COOL source files (`.cl`) into MIPS assembly code (`.asm`) executable on the SPIM simulator. Beyond code generation, the compiler produces visualizations (parse trees, abstract syntax trees, and control flow graphs) to aid debugging and understanding. This report provides a comprehensive overview of the compiler's design, implementation, and features, detailing each phase of the compilation process.

1.2 Background

COOL supports classes, single inheritance, automatic memory management, and a rich expression-based syntax (Sections 3–7 [1]). Its educational design emphasizes type safety, object-oriented programming, and compiler construction principles. The compiler pipeline mirrors a traditional compiler, with modular phases that handle lexical analysis, parsing, semantic checking, intermediate code generation, optimization, and final code generation. Visualizations enhance the ability to inspect intermediate representations, making the compiler a valuable learning tool.

The pipeline includes:

- **Lexical Analysis**: Converts source code into tokens using ANTLR.
- **Syntax Analysis**: Builds a parse tree to enforce syntactic rules.
- **AST Construction**: Creates an abstract syntax tree for semantic analysis.
- **Semantic Analysis**: Ensures type safety and semantic correctness.
- **Intermediate Code Generation**: Produces Three-Address Code (TAC).
- **Visualization**: Generates parse tree, AST, and CFG images.
- **Optimization**: Improves TAC efficiency.
- **Code Generation**: Translates TAC to MIPS assembly.

1.3 Implementation

The compiler is implemented in Java, leveraging ANTLR 4 for lexical and syntax analysis, and custom Java classes for subsequent phases. Each phase is encapsulated in dedicated packages (e.g., `ast`, `tac`, `visualization`), promoting modularity and maintainability. The main workflow in `Main.java` orchestrates the pipeline, processing multiple `.cl` files and producing corresponding outputs.

1.4 Features

- **Type Safety**: Enforces COOL's static typing rules (Section 4).

- **Object-Oriented Support**: Handles classes, inheritance, and dispatch (Sections 3, 7.4).
- **Visualizations**: Produces PNG images for debugging.
- **SPIM Compatibility**: Generates executable MIPS code.
- **Modularity**: Allows easy extension and maintenance.

2 Lexical Analysis

2.1 Purpose

Lexical analysis transforms COOL source code into a stream of tokens, as specified in Section 10 of the COOL Reference Manual [1]. This phase is critical as it simplifies subsequent parsing by breaking the input into meaningful units like keywords, identifiers, and operators. The lexer must handle COOL's lexical rules, including case sensitivity, nested comments, and error conditions like unterminated strings.

2.2 Background

COOL's lexical structure includes integers, identifiers, strings, comments, keywords, whitespace, and special symbols. The lexer, implemented in `CoolLexer.g4` using ANTLR 4, generates a Java lexer class that produces tokens for the parser. ANTLR was chosen for its robust grammar definition language and automatic error handling, which streamline lexer development. Challenges include handling case-insensitive keywords (except `true`/`false`) and nested multi-line comments, which require careful rule design.

2.3 Lexical Structure

2.3.1 Integers and Identifiers

Integers are non-empty digit sequences (0–9) without leading zeros, except for 0 (Section 10.1). Identifiers include object identifiers (lowercase initial, e.g., `xcar`) and type identifiers (uppercase initial, e.g., `Cons`), with special identifiers `self` and `SELF_TYPE`. The lexer rules in `CoolLexer.g4` are:

```
INT       : [0-9]+ ;
ID        : [a-z][a-zA-Z0-9_]* ;
TYPE      : [A-Z][a-zA-Z0-9_]* ;
SELF      : 'self' ;
SELF_TYPE : 'SELF_TYPE' ;
```

These rules ensure integers exclude invalid formats (e.g., `0123`) and identifiers match COOL's naming conventions.

2.3.2 Strings

Strings (Section 10.2) are double-quoted ASCII sequences supporting escape sequences (`\n`, `\t`) and limited to 1024 characters. Unterminated strings trigger errors. The lexer rules handle content and errors:

```
STRING : '"' (STRING_CONTENT | ESCAPE_SEQUENCE)* '"' ;
fragment STRING_CONTENT : ~["\\\n\r] ;
fragment ESCAPE_SEQUENCE : '\\' (['"\\] | 'b' | 't' | 'n' | 'f') ;
UNTERMINATED_STRING : '"' (STRING_CONTENT | ESCAPE_SEQUENCE)* ('\n' | EOF) {
    System.err.println("Error: Unterminated string at line " + getLine());
} ;
```

The error rule ensures user-friendly diagnostics, a critical feature for educational use.

2.3.3 Comments

COOL supports single-line (-) and nested multi-line ((* *)) comments, skipped during tokenization (Section 10.3). The lexer uses non-greedy matching for nested comments:

```
SINGLECOMMENT : '--' ~[\r\n]* -> skip ;
MULTICOMMENT  : '(*' .*? '*)' -> skip ;
```

Non-greedy matching (.*?) ensures proper nesting, a challenge resolved by leveraging ANTLR's parsing capabilities.

2.3.4 Keywords

Keywords (e.g., `class`, `if`) are case-insensitive, except `true` and `false`, which require lowercase initials (Section 10.4). Fragments in `CoolLexer.g4` handle case insensitivity:

```
CLASS : C L A S S ;
TRUE  : 't' R U E ;
FALSE : 'f' A L S E ;
fragment C : [cC] ;
fragment L : [lL] ;
// ... other fragments
```

This approach avoids duplicating rules for each case variation, improving maintainability.

2.3.5 Whitespace and Operators

Whitespace (spaces, tabs, newlines) is ignored (Section 10.5), and operators (e.g., +, <-) are tokenized individually:

```
WS : [ \t\r\n\f\u000B]+ -> skip ;
PLUS  : '+' ;
ARROW : '<-' ;
DOT   : '.' ;
```

2.4 Implementation

The lexer, defined in `CoolLexer.g4`, produces tokens integrated with the parser via a token stream. ANTLR generates a Java class (`CoolLexer.java`) that processes input character by character, applying the defined rules. Design decisions include:

- **Error Handling**: Custom error messages for unterminated strings enhance usability.

- **Case Insensitivity**: Fragments reduce rule complexity for keywords.
- **Nested Comments**: Non-greedy matching ensures correct parsing, a non-trivial task due to potential nesting depth.
- **Performance**: Skipping whitespace and comments at the lexer level reduces parser workload.

Challenges included ensuring `true/false` case sensitivity and handling edge cases like empty strings or invalid escape sequences, resolved through rigorous testing against COOL's specification.

2.5 Features

- **Comprehensive Tokenization**: Supports all COOL lexical elements.
- **Error Reporting**: Detects and reports unterminated strings.
- **Efficiency**: Skips irrelevant tokens (whitespace, comments).
- **Integration**: Seamlessly feeds tokens to the parser.

2.6 Example

For the input:

```
class Main {
    main() : Object {
        let x : Int <    \item 42 in x + 1
    };
};
```

the lexer produces tokens like `CLASS`, `TYPE(Main)`, `ID(main)`, `LET`, `ID(x)`, `INT(42)`, `PLUS`, `INT(1)`, reflecting COOL's lexical structure.

3 Syntax Analysis

3.1 Purpose

Syntax analysis constructs a parse tree from the token stream, ensuring the program adheres to COOL's grammar (Section 11 [1]). This phase validates the syntactic structure, detecting errors like missing semicolons or malformed expressions before semantic analysis.

3.2 Background

COOL's grammar defines programs as class sequences, with classes containing attributes and methods, and expressions supporting complex constructs like dispatches and conditionals. The parser, implemented in `CoolParser.g4` using ANTLR 4, generates a labeled parse tree for AST construction. ANTLR was chosen for its ability to handle complex grammars and generate robust parsers. Challenges include encoding operator precedence and handling COOL's unique constructs like `SELF_TYPE`.

3.3 Syntax Structure

3.3.1 Program and Classes

A COOL program is one or more class definitions (Section 3), each with optional inheritance and features. The grammar in `CoolParser.g4` defines:

```
program : class+ ;
class   : CLASS TYPE (INHERITS TYPE)? LBRACE feature* RBRACE SEMICOLON ;
```

This structure ensures classes are well-formed and supports inheritance.

3.3.2 Features

Attributes and methods (Section 3.1) are defined as:

```
feature : ID LPAREN (formal (COMMA formal)*)? RPAREN COLON (TYPE | SELF_TYPE) LBRACE expr RBRACE
    SEMICOLON  # Method
        | ID COLON (TYPE | SELF_TYPE) (ARROW expr)? SEMICOLON
                                                   # Attribute ;
formal  : ID COLON (TYPE | SELF_TYPE) ;
```

Attributes may include initialization, and methods support multiple parameters.

3.3.3 Expressions

Expressions (Section 7) include dispatches, conditionals, loops, and operators. The `expr` rule handles precedence:

```
expr : expr (AT TYPE)? DOT ID LPAREN exprList? RPAREN  # Dispatch
     | IF expr THEN expr ELSE expr FI                 # IfElse
     | expr PLUS expr                                 # Addition
     | ID                                             # Identifier
     | INT                                            # Integer
    // ... other expression types
;
```

Precedence is encoded via rule ordering (Section 11.1), with `.` having the highest precedence.

3.4 Implementation

The parser, defined in `CoolParser.g4`, generates `CoolParser.java`, which consumes tokens from `CoolLexer.java`. Key design decisions:

- **Error Strategy**: `BailErrorStrategy` halts on syntax errors, providing immediate feedback.
- **Labeled Alternatives**: Labels (e.g., `# Dispatch`) simplify AST construction.
- **Precedence Handling**: Rule structure ensures correct operator precedence and associativity.
- **Token Integration**: `tokenVocab=CoolLexer` links lexer and parser.

Challenges included managing COOL's complex expression grammar and ensuring unambiguous parsing of nested constructs, addressed through careful rule design and testing.

3.5 Features

- **Complete Grammar**: Supports all COOL constructs.
- **Error Detection**: Reports syntax errors with line numbers.
- **Precedence Accuracy**: Correctly handles operator precedence.
- **Parse Tree Output**: Facilitates AST construction.

3.6 Example

For:

```
let x : Int <    \item 42 in if x < 50 then x + 1 else x fi
```

the parser builds a tree with nodes for `let`, `IfElse`, `BinaryOp(<)`, and `BinaryOp(+)`, preserving syntactic hierarchy.

4 Abstract Syntax Tree Construction

4.1 Purpose

AST construction transforms the parse tree into a semantic representation, abstracting syntactic details for type checking and code generation (Sections 3, 7, 11 [1]). The AST simplifies subsequent phases by focusing on program meaning.

4.2 Background

The AST represents COOL programs hierarchically, with nodes for classes, features, and expressions. Implemented in `ASTBuilder.java`, it uses a visitor pattern to traverse the parse tree. The design prioritizes extensibility and error reporting, with challenges including handling `SELF_TYPE` and maintaining line numbers for diagnostics.

4.3 AST Structure

The AST, in the `ast` package, uses `ASTNode` (`ASTNode.java`) as the base class, with fields for line numbers and a visitor method. Key nodes include:

- `ProgramNode`: List of `ClassNodes`.
- `ClassNode`: Name, parent, features.
- `MethodNode`: Parameters, return type, body.
- `AttributeNode`: Name, type, optional initialization.
- `ExpressionNode`: Subclasses like `DispatchNode`, `LetNode`, `BinaryOpNode`.

The `ASTVisitor` interface (`ASTVisitor.java`) supports extensible processing.

4.4 Implementation

`ASTBuilder.java` extends `CoolParserBaseVisitor<ASTNode>` to construct the AST. A key method is:

```
public ProgramNode visitProgram(CoolParser.ProgramContext ctx) {
    List<ClassNode> classes = new ArrayList<>();
    for (CoolParser.ClassContext classCtx : ctx.class_()) {
        classes.add((ClassNode) visit(classCtx));
    }
    return new ProgramNode(classes, ctx.start.getLine());
}
```

4.4.1 Design Decisions

- **Visitor Pattern**: Enables modular processing for semantic analysis and code generation.
- **Line Numbers**: Stored in each node for precise error reporting.
- **Type Support**: Explicit handling of `SELF_TYPE` in declarations.
- **Simplification**: Omits syntactic tokens (e.g., semicolons) to focus on semantics.

Challenges included mapping complex parse tree nodes to streamlined AST nodes and ensuring `SELF_TYPE` was correctly represented, resolved through a robust node hierarchy.

4.5 Features

- **Semantic Focus**: Abstracts syntactic details.
- **Extensibility**: Visitor pattern supports new analyses.
- **Error Support**: Line numbers aid diagnostics.
- **COOL Compliance**: Handles all language constructs.

4.6 Example

For:

```
class Main inherits IO {
    x : Int <    \item 42;
    main() : Object {
        let y : Int <    \item x + 1 in out_int(y)
    };
};
```

the AST includes a `ProgramNode` with a `ClassNode(Main)`, containing an `AttributeNode(x)` and a `MethodNode(main)` with a `LetNode` and `MethodCallNode(out_int)`.

5 Semantic Analysis

5.1 Purpose

Semantic analysis ensures type safety and semantic correctness, preventing runtime errors (Sections 4, 5, 6, 12 [1]). It validates inheritance, method signatures, and expression types, reporting errors for violations.

5.2 Background

COOL's type system requires static type checking, with rules for conformance, `SELF_TYPE`, and method overriding. Implemented in `SemanticAnalyzer.java`, this phase uses symbol tables and class metadata to track scopes and types. Challenges include handling `SELF_TYPE`'s dynamic typing and detecting inheritance cycles.

5.3 Implementation

The `SemanticAnalyzer` uses `ClassTable` (`ClassTable.java`) for class metadata and `SymbolTable` (`SymbolTable.jav`
for variable scopes. The main method is:

```
public List<String> analyze(ProgramNode program) {
    for (ClassNode classNode : program.classes) {
        classTable.addClass(classNode);
    }
    classTable.checkInheritance();
    errors.addAll(classTable.getErrors());
    program.accept(this);
    return errors;
}
```

5.3.1 Key Components

- **ClassTable**: Stores class names, parents, methods, and attributes, checking for cycles and illegal inheritance (e.g., `BASIC` classes).
- **SymbolTable**: Manages nested scopes for variables, updated for `let` and method parameters.
- **Type Checking**: Infers expression types using Section 12.2 rules, ensuring conformance.

5.3.2 Design Decisions

- **Two-Pass Analysis**: First builds `ClassTable`, then checks expressions to handle forward references.
- **Error Accumulation**: Collects all errors for comprehensive reporting.
- **Visitor Pattern**: Traverses AST for type inference.

Challenges included implementing `SELF_TYPE` conformance and ensuring method override rules (Section 6), resolved through detailed type inference algorithms.

5.4 Features

- **Type Safety**: Prevents runtime type errors.
- **Inheritance Validation**: Detects cycles and illegal parents.
- **Method Checking**: Ensures override compatibility.
- **Detailed Errors**: Provides line numbers and context.

5.5 Example

For:

```
let y : Int <    \item x + 1 in if y <= 50 then out_int(y) else abort() fi
```

the analyzer verifies `x : Int`, computes `x + 1 : Int`, `y <= 50 : Bool`, and checks `out_int` and `abort` dispatches, determining the `if` type as `Object`.

6 Intermediate Code Generation

6.1 Purpose

Intermediate code generation produces Three-Address Code (TAC) from the AST, serving as a platform-independent representation for optimization and code generation (Section 13 [1]). TAC simplifies complex expressions into three-operand instructions.

6.2 Background

TAC is a low-level representation where each instruction has at most three operands, making it ideal for optimization and translation to assembly. Implemented in `TACGenerator.java`, this phase handles COOL's object-oriented features and control flow. Challenges include translating complex expressions (e.g., nested dispatches) and managing temporary variables.

6.3 TAC Structure

TAC, defined in `TACProgram.java`, organizes instructions by class and method. Instructions include `ASSIGN`, `BINARY`, `CALL`, `JUMP`, etc. Example:

```
public class AssignTAC extends TACInstruction {
    private String source;
    public AssignTAC(String result, String source, int lineNumber) {
        super(Opcode.ASSIGN, result, lineNumber);
        this.source = source;
    }
}
```

6.4 Implementation

`TACGenerator.java` traverses the AST, generating instructions. A key method is:

```
public String visitMethodCallNode(MethodCallNode node) {
    List<String> argResults = new ArrayList<>();
    for (ExpressionNode arg : node.args) {
        argResults.add(arg.accept(this));
    }
    String result = newTemp();
    currentInstructions.add(new CallTAC(result, currentClass + "." + node.method, argResults,
        node.lineNumber));
    return result;
}
```

6.5 Design Decisions

- **Temporary Variables**: Generated for intermediate results to ensure evaluation order.
- **Control Flow**: Labels and jumps handle conditionals and loops.
- **Object-Oriented Support**: Instructions for attribute access and dispatch.
- **Modularity**: Separate classes for each instruction type.

Challenges included handling `SELF_TYPE` dispatches and ensuring correct control flow, addressed through careful instruction design and testing.

6.6 Features

- **Full Expression Support**: Translates all COOL expressions.
- **Control Flow**: Accurate jumps and labels.
- **Temporary Management**: Prevents register conflicts.
- **COOL Compliance**: Supports object-oriented features.

6.7 Example

For:

```
let y : Int <    \item x + 1 in if y <= 50 then out_int(y) else abort() fi
```

TAC includes:

```
t0 = load x
t1 = t0 + 1
y = t1
t2 = y <= 50
if t2 goto L0
goto L1
L0: t3 = call Main.out_int(y)
...
```

7 Visualization

7.1 Purpose

Visualization generates PNG images of parse trees, ASTs, and CFGs to aid debugging and understanding (Sections 11, 13 [1]). These graphics provide insights into the compiler's intermediate representations.

7.2 Background

Visualizations are implemented in the `visualization` package, using Graphviz to convert `.dot` files to PNGs. This phase enhances the compiler's educational value by making internal structures visible. Challenges include representing complex trees and graphs clearly.

7.3 Implementation

Key classes:

- `ParseTreeVisualizer.java`: Traverses parse trees to generate `.dot` files.
- `ASTVisualizer.java`: Represents AST nodes and edges.
- `CFGVisualizer.java`: Builds CFGs from TAC basic blocks.
- `DotToImageConverter.java`: Executes Graphviz `dot` commands.

Example for parse tree:

```java
public class ParseTreeVisualizer {
    public void visualize(ParseTree tree, String outputFile) {
        dot.append("digraph ParseTree {\n");
        visit(tree, 0);
        dot.append("}\n");
        try (PrintWriter writer = new PrintWriter(outputFile)) {
            writer.println(dot.toString());
        }
    }
}
```

7.3.1 Design Decisions

- **Graphviz Integration**: Leverages `dot` for high-quality graphics.
- **Hierarchical Representation**: Ensures clear node relationships.
- **Error Handling**: Manages I/O and Graphviz errors.

7.4 Features

- **Comprehensive Visuals**: Covers syntax, semantics, and control flow.
- **PNG Output**: Cross-platform compatible.
- **Debugging Aid**: Highlights structural issues.

7.5 Example

For the example program, visualizations show parse tree nodes for `class`, AST nodes for `LetNode`, and CFG blocks with control flow edges.

8 Optimization

8.1 Purpose

Optimization improves TAC efficiency, reducing runtime overhead while preserving semantics (Section 13 [1]). This phase enhances performance for SPIM execution.

8.2 Background

Optimizations include constant folding and loop optimizations, implemented in `ProgramOptimizer.java`. The goal is to minimize instruction counts and improve control flow. Challenges include ensuring semantic equivalence and identifying optimization opportunities.

8.3 Implementation

`ProgramOptimizer.java` applies multiple passes:

```
public class ProgramOptimizer {
    public ProgramOptimizer() {
        passes.add(new ConstantFolder());
        passes.add(new LoopOptimizer());
    }
}
```

- **ConstantFolder**: Evaluates constant expressions (e.g., `2 + 3 = 5`).
- **LoopOptimizer**: Applies loop-invariant code motion (LICM) and unrolling, using `LoopAnalyzer` to identify loops.

8.3.1 Design Decisions

- **Multiple Passes**: Allows iterative optimization.
- **Semantic Checks**: Ensures transformations preserve behavior.
- **Modularity**: Separate classes for each optimization.

8.4 Features

- **Constant Folding**: Reduces constant computations.
- **Loop Optimizations**: Improves loop performance.
- **Semantic Safety**: Maintains program correctness.

8.5 Example

For:

```
let x : Int <    \item 2 + 3 in while i < 10 loop out_int(x) pool
```

optimized TAC is:

```
x = 5
L_preheader: t2 = x
L0: t1 = i < 10
...
```

9 Code Generation

9.1 Purpose

Code generation translates optimized TAC to MIPS assembly for SPIM execution (Section 2 [1]). It maps TAC instructions to MIPS instructions, supporting COOL's object model.

9.2 Background

Implemented in `CodeGenerator.java`, this phase uses `ClassTable` for metadata and manages registers and stack. Challenges include implementing dispatch tables and ensuring type safety in assembly.

9.3 Implementation

`CodeGenerator.java` generates `AssemblyProgram.java` output:

```
public class CodeGenerator {
    public CodeGenerator(ClassTable classTable) {
        this.program = new AssemblyProgram();
        this.classTable = classTable;
        this.freeRegisters = new LinkedHashSet<>(Arrays.asList("$r2", "$r3", ...));
    }
}
```

9.3.1 Design Decisions

- **Register Allocation**: Uses a simple free-register pool.
- **Dispatch Tables**: Implements method dispatch for inheritance.
- **Stack Management**: Handles local variables and calls.

9.4 Features

- **Full TAC Support**: Translates all instructions.

- **Object-Oriented Features**: Supports inheritance and dispatch.
- **SPIM Compatibility**: Produces executable code.

10 Main Workflow

10.1 Purpose

The main workflow orchestrates the compiler pipeline, processing `.cl` files to produce outputs.

10.2 Implementation

`Main.java` scans `samples` and applies each phase:

```
public class Main {
    public static void main(String[] args) {
        Path samplesDir = Paths.get("samples");
        Files.walk(samplesDir).filter(path -> path.toString().endsWith(".cl")).forEach(inputFile
            -> {
            // Lexing, parsing, AST, semantic analysis, TAC, visualization, optimization, codegen
        });
    }
}
```

10.3 Features

- **Batch Processing**: Handles multiple files.
- **Error Handling**: Logs errors to `.errors` files.
- **Visualization Support**: Generates diagnostic graphics.

11 Conclusion

This COOL compiler fully implements the language specification [1], producing type-safe MIPS assembly and visualizations. Future work could explore advanced optimizations and cross-platform code generation.

References

[1] The Cool Reference Manual, CS164, University of California, Berkeley, 2025.