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

Zahraa Selim - 120210083

May 19, 2025

Contents

1 Introduction

This project implements a compiler for the Classroom Object-Oriented Language (COOL), as specified in the COOL Reference Manual. COOL is a statically typed, object-oriented language designed for educational purposes, featuring classes, single inheritance, automatic memory management, and a rich expression-based syntax. The compiler, written in Java, processes COOL source files (‘.cl‘) and generates MIPS assembly code (‘.asm‘), along with visualizations of the parse tree, abstract syntax tree (AST), and control flow graph (CFG). This document details each phase of the compiler, explaining the features and functionalities implemented.

The compiler pipeline consists of:

- **Lexical Analysis**: Tokenizing COOL source code using ANTLR.
- **Syntax Analysis**: Parsing tokens into a parse tree, ensuring syntactic correctness.
- **AST Construction**: Building an abstract syntax tree from the parse tree.
- **Semantic Analysis**: Performing type checking and validating program semantics.
- **Intermediate Code Generation**: Generating Three-Address Code (TAC).
- **Visualization**: Producing visualizations of parse trees, ASTs, and CFGs as PNG images.
- **Optimization**: Applying optimizations to the TAC.
- **Code Generation**: Translating optimized TAC to MIPS assembly.

Each phase is implemented as a modular component, ensuring maintainability and extensibility. The following sections describe each phase in detail, highlighting the implemented features and their alignment with the COOL specification.

2 Lexical Analysis

2.1 COOL Lexical Structure

The lexer, implemented using ANTLR, tokenizes COOL source code according to the lexical structure defined in Section 10 of the COOL Reference Manual. It recognizes integers, identifiers, strings, comments, keywords, whitespace, and special notation.

2.1.1 Integers, Identifiers, and Special Notation

Integers are non-empty sequences of digits (0–9). Identifiers are categorized into object identifiers (starting with a lowercase letter) and type identifiers (starting with an uppercase letter), consisting of letters, digits, and underscores. The special identifiers `self` and `SELF_TYPE` are handled explicitly.

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

2.1.2 Strings

Strings are enclosed in double quotes and may contain escape sequences (`\b`, `\t`, `\n`, `\f`). Unterminated strings and invalid characters (e.g., null or EOF) trigger error messages.

```
STRING : '"' (STRING_CONTENT | ESCAPE_SEQUENCE)* '"';
fragment STRING_CONTENT : ~["\\\n\r\u0000];
fragment ESCAPE_SEQUENCE : '\\' (['"\\] | 'b' | 't' | 'n' | 'f');
UNTERMINATED_STRING : '"' (STRING_CONTENT | ESCAPE_SEQUENCE)*  ('\n' | EOF) {
    System.err.println("Error: Unterminated string detected!");
};
```

2.1.3 Comments

COOL supports single-line comments (starting with `--`) and nested multi-line comments (enclosed in `(*` and `*)`). Both are skipped during tokenization.

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

2.1.4 Keywords

COOL keywords (e.g., `class`, `if`, `while`) are case-insensitive, except for `true` and `false`, which must start with lowercase `t` or `f`. The lexer uses fragments to handle case insensitivity.

```
CLASS : C L A S S;
ELSE  : E L S E;
TRUE  : 't' R U E;
FALSE : 'f' A L S E;
fragment A : [aA];
fragment B : [bB];
// ... other fragments
```

2.1.5 Whitespace

Whitespace includes spaces, tabs, newlines, form feeds, carriage returns, and vertical tabs, all skipped during tokenization.

```
WS : [ \t\r\n\f\u000B]+ -> skip;
```

2.1.6 Operators and Punctuation

Operators and punctuation (e.g., `+`, `<-`, `.`) are defined as tokens to support the separated lexer/parser grammar.

```
PLUS      : '+' ;
ARROW     : '<-';
DOT       : '.' ;
// ... other operators
```

2.2 Implementation Details

The lexer is defined in `CoolLexer.g4`, which generates a Java lexer class using ANTLR. It handles all lexical elements specified in the COOL manual, including error reporting for invalid strings. The lexer integrates with the parser via a token stream, ensuring robust tokenization for downstream phases.

3 Syntax Analysis

3.1 COOL Syntax

The parser, implemented using ANTLR, constructs a parse tree from the token stream, enforcing the syntax defined in Section 11 of the COOL Reference Manual. COOL's syntax includes class definitions, features (attributes and methods), expressions, and control structures.

3.2 ANTLR Parser Grammar

The parser grammar, defined in `CoolParser.g4`, specifies the structure of COOL programs. A program consists of one or more class definitions, each containing features. Expressions cover a wide range of constructs, including dispatches, conditionals, loops, and let bindings.

```
program
    : class+
    ;

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

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

expr
    : expr (AT TYPE)? DOT ID LPAREN exprList? RPAREN                # Dispatch
    | TILDE expr                                                    # Negation
    | ISVOID expr                                                   # Isvoid
    | expr TIMES expr                                               # Multiplication
    | ID ARROW expr                                                 # Assignment
    | IF expr THEN expr ELSE expr FI                                # IfElse
    | LET letDecl (COMMA letDecl)* IN expr                          # Let
    | NEW TYPE                                                      # New
    | ID                                                            # Identifier
    | INT                                                           # Integer
    | TRUE                                                          # True
    // ... other expression rules
```

3.3 Precedence and Associativity

The parser respects the operator precedence defined in the COOL manual (Section 11.1), from highest to lowest: `.`, `@`, `~`, `isvoid`, `* /`, `+ -`, `<= < =`, `not`, `<-`. All binary operators are left-associative, except assignment (`<-`), which is right-associative, and comparisons, which are non-associative.

3.4 Implementation Details

The parser uses a `BailErrorStrategy` to halt on the first syntax error, ensuring immediate feedback for invalid programs. The generated parse tree serves as input to the AST construction phase, preserving the syntactic structure for semantic analysis.

4 Abstract Syntax Tree (AST) Construction

4.1 Purpose

The AST construction phase transforms the parse tree into an abstract syntax tree, representing the program's structure in a form suitable for semantic analysis. The AST abstracts away syntactic details, focusing on the program's logical components (classes, features, expressions).

4.2 Implementation

The `ASTBuilder` class, a visitor over the parse tree, constructs AST nodes defined in the `ast` package. Key node types include:

- **ProgramNode**: Represents the entire program, containing a list of classes.
- **ClassNode**: Represents a class with its name, optional parent, and features.
- **FeatureNode**: Base class for attributes (`AttributeNode`) and methods (`MethodNode`).
- **ExpressionNode**: Base class for expressions (e.g., `BinaryExpressionNode`, `IfNode`, `NewNode`).

```
// Example: ASTBuilder visiting a class
public class ASTBuilder extends CoolParserBaseVisitor<Node> {
    @Override
    public Node visitClass(CoolParser.ClassContext ctx) {
        String className = ctx.TYPE(0).getText();
        String parentName = ctx.INHERITS() != null ? ctx.TYPE(1).getText() : null;
        List<FeatureNode> features = new ArrayList<>();
        for (CoolParser.FeatureContext featureCtx : ctx.feature()) {
            features.add((FeatureNode) visit(featureCtx));
        }
        return new ClassNode(className, parentName, features);
    }
}
```

4.3 Features

The AST supports all COOL constructs, including: - Class definitions with single inheritance. - Attributes with optional initialization. - Methods with formal parameters and return types. - Expressions such as dispatches, conditionals, loops, let bindings, and case statements.

The AST is designed for extensibility, with a clear hierarchy of node types facilitating semantic analysis and code generation.

5 Semantic Analysis

5.1 Purpose

Semantic analysis ensures the program adheres to COOL's type system and semantic rules (Sections 4, 12 of the manual). It performs type checking, validates inheritance, and detects errors such as undefined types or type mismatches.

5.2 Implementation

The `SemanticAnalyzer` class performs semantic analysis by traversing the AST. It maintains:

- **ClassTable**: Maps class names to their definitions, including parent classes and features.
- **SymbolTable**: Tracks variable scopes and their types, supporting nested scopes for let and method bodies.

```
public class SemanticAnalyzer {
    private ClassTable classTable;
    private SymbolTable symbolTable;

    public List<String> analyze(ProgramNode program) {
        List<String> errors = new ArrayList<>();
        buildClassTable(program, errors);
        checkInheritance(errors);
        checkFeatures(errors);
        return errors;
    }
}
```

5.3 Key Features

- **Type Checking**: Implements the conformance relation (`C` $\leq$ `P`) and type rules for all expressions (Section 12.2). Handles `SELF_TYPE` for flexible typing in inherited classes. - **Inheritance Validation**: Ensures no cycles in the inheritance graph and that parent classes are defined (Section 3.2). - **Method Redefinition**: Enforces that overridden methods have identical signatures (Section 6). - **Error Reporting**: Collects and reports semantic errors (e.g., type mismatches, undefined identifiers) to an error file.

The analyzer supports COOL's type safety guarantee, ensuring no runtime type errors occur (Section 4.2).

6 Intermediate Code Generation

6.1 Purpose

The intermediate code generation phase translates the AST into Three-Address Code (TAC), a platform-independent representation suitable for optimization and code generation.

6.2 Implementation

The `TACGenerator` class generates TAC by visiting AST nodes. The TAC representation includes instructions like assignments, binary operations, and control flow statements.

```
public class TACGenerator {
    private ClassTable classTable;
    private SymbolTable symbolTable;
    private int tempCounter;

    public TACProgram generate(ProgramNode program) {
        TACProgram tacProgram = new TACProgram();
        for (ClassNode classNode : program.getClasses()) {
            for (FeatureNode feature : classNode.getFeatures()) {
                if (feature instanceof MethodNode) {
                    tacProgram.addMethod(generateMethod((MethodNode) feature));
                }
            }
        }
        return tacProgram;
    }
}
```

6.3 Features

- Generates TAC for all expression types, including arithmetic, conditionals, loops, and dispatches. - Manages temporary variables for intermediate results. - Preserves method and class structure, facilitating CFG visualization and optimization.

7 Visualization

7.1 Purpose

The visualization phase generates graphical representations of the parse tree, AST, and CFG, aiding in debugging and understanding the compiler's output. Visualizations are output as PNG images, converted from Graphviz '.dot' files.

7.2 Implementation

Three visualizer classes handle different aspects:

- **ParseTreeVisualizer**: Generates a '.dot' file for the parse tree using ANTLR's parse tree structure.
- **ASTVisualizer**: Traverses the AST to produce a '.dot' file representing the abstract syntax.
- **CFGVisualizer**: Constructs a control flow graph from the TAC, outputting a '.dot' file.

The `DotToImageConverter` class converts '.dot' files to PNG images using Graphviz's `dot` command.

```
public class DotToImageConverter {
    public static void convertToPng(String dotFilePath, String outputPngPath) {
        ProcessBuilder pb = new ProcessBuilder("dot", "-Tpng", dotFilePath, "-o",
            outputPngPath);
        Process process = pb.start();
        int exitCode = process.waitFor();
        if (exitCode != 0) {
            throw new IOException("Graphviz dot command failed");
        }
```

```
    }
}
```

7.3 Features

- Produces PNG images for parse trees, ASTs, and CFGs, stored in the `output` directory. - Supports visualization of complex program structures, including nested expressions and control flow. - Integrates seamlessly with the compiler pipeline, invoked after each relevant phase.

8 Optimization

8.1 Purpose

The optimization phase improves the TAC to enhance the efficiency of the generated code, focusing on reducing instruction count and improving runtime performance.

8.2 Implementation

The `ProgramOptimizer` class applies optimizations to the TAC program, such as constant folding and dead code elimination.

```
public class ProgramOptimizer {
    public TACProgram optimize(TACProgram program) {
        TACProgram optimized = new TACProgram();
        for (TACMethod method : program.getMethods()) {
            optimized.addMethod(optimizeMethod(method));
        }
        return optimized;
    }
}
```

8.3 Features

- Implements basic optimizations (e.g., constant folding, removing unreachable code). - Preserves program semantics, ensuring correctness. - Outputs optimized TAC to a separate file (e.g., `program.opt.tac`).

9 Code Generation

9.1 Purpose

The code generation phase translates optimized TAC into MIPS assembly code, executable on a MIPS simulator like SPIM (Section 2 of the manual).

9.2 Implementation

The `CodeGenerator` class converts TAC instructions into MIPS assembly, leveraging the class table for method dispatch and object layout.

```
public class CodeGenerator {
    private ClassTable classTable;

    public AssemblyProgram generate(TACProgram tacProgram) {
        AssemblyProgram asmProgram = new AssemblyProgram();
        for (TACMethod method : tacProgram.getMethods()) {
            asmProgram.addFunction(generateFunction(method));
        }
        return asmProgram;
    }
}
```

9.3 Features

- Generates MIPS assembly for all TAC instructions, including arithmetic, control flow, and method calls. - Supports COOL's object model, including attribute initialization and method dispatch. - Outputs assembly to `.asm` files, compatible with SPIM.

10 Main Workflow

10.1 Implementation

The `Main` class orchestrates the compiler pipeline, processing all `.cl` files in the `input` directory and producing outputs in the `output` directory.

```
public class Main {
    public static void main(String[] args) {
        Path inputDir = Paths.get("input");
        Path outputDir = Paths.get("output");
        // Reset output directory
        Files.createDirectories(outputDir);
        // Process each .cl file
        for (Path inputFile : getCoolFiles(inputDir)) {
            CharStream input = CharStreams.fromPath(inputFile);
            CoolLexer lexer = new CoolLexer(input);
            CoolParser parser = new CoolParser(new CommonTokenStream(lexer));
            ParseTree tree = parser.program();
            // Generate visualizations, AST, TAC, etc.
        }
    }
}
```

10.2 Features

- Processes multiple input files, generating separate outputs for each. - Handles errors gracefully, logging issues to `.errors` files. - Integrates all compiler phases, ensuring a cohesive workflow.

11 Conclusion

This COOL compiler implements a complete pipeline for compiling COOL programs into MIPS assembly, with robust support for lexical analysis, syntax analysis, semantic checking, intermediate code generation, visualization,

optimization, and code generation. The visualization feature, enhanced by automatic conversion of '.dot' files to PNG images, provides valuable insights into the compilation process. The implementation adheres closely to the COOL Reference Manual, ensuring correctness and type safety while offering extensibility for future enhancements.