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 detailed 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. The compiler also generates visualizations, including parse trees, abstract syntax trees (ASTs), and control flow graphs (CFGs), to facilitate debugging and learning. This report provides a comprehensive overview of the compiler's design, implementation, and features across its pipeline.

1.2 Background

COOL supports classes, single inheritance, automatic memory management, and a rich expression-based syntax (Sections 3–7 [1]). Its design emphasizes type safety, object-oriented programming, and compiler construction principles. The compiler pipeline includes modular phases for lexical analysis, syntax analysis, semantic checking, intermediate code generation, optimization, and code generation, with visualizations to inspect intermediate representations.

The pipeline comprises:

- **Lexical Analysis**: Converts source code into tokens using ANTLR.
- **Syntax Analysis**: Builds a parse tree to enforce syntactic rules.
- **AST Construction**: Creates an AST 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**: Enhances TAC efficiency.
- **Code Generation**: Translates TAC to MIPS assembly.

1.3 Implementation

The compiler is implemented in Java, using ANTLR 4 for lexical and syntax analysis and custom Java classes for other phases. Phases are organized into packages (e.g., `ast`, `tac`, `visualization`) for modularity. The main workflow in `Main.java` orchestrates the pipeline, processing multiple `. Cardinality (.cl)` files and generating corresponding outputs.

1.4 Features

- **Type Safety**: Enforces COOL's static typing rules (Section 4).
- **Object-Oriented Support**: Supports classes, inheritance, and dispatch (Sections 3, 7.4).

- **Visualizations**: Produces PNG images for debugging.
- **SPIM Compatibility**: Generates executable MIPS code.
- **Modularity**: Facilitates extension and maintenance.

1.5 Example

The following COOL program is used throughout this report to illustrate each phase:

```
class Main inherits IO {
    x : Int <- 42;
    main() : Object {
        let y : Int <- x + 1 in
        if y <= 50 then out_int(y) else abort() fi
    };
};
```

2 Lexical Analysis

2.1 Purpose

Lexical analysis converts COOL source code into a token stream, as specified in Section 10 [1]. This phase simplifies parsing by breaking input into units like keywords, identifiers, and operators, handling COOL's lexical rules, including case sensitivity and nested comments.

2.2 Background

COOL's lexical structure includes integers, identifiers, strings, comments, keywords, whitespace, and symbols. The lexer, implemented in `CoolLexer.g4` using ANTLR 4, generates a Java lexer class. ANTLR's robust grammar language and error handling streamline development. Challenges include case-insensitive keywords (except `true`/`false`) and nested comments.

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`). Lexer rules are:

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

2.3.2 Strings

Strings (Section 10.2) are double-quoted ASCII sequences with escape sequences (`\n`, `\t`) and a 1024-character limit. Unterminated strings trigger errors:

```
STRING  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());
} ;
```

2.3.3 Comments

COOL supports single-line (`-`) and nested multi-line (`(* *)`) comments (Section 10.3), skipped during tokenization:

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

2.3.4 Keywords

Keywords (e.g., `class`, `if`) are case-insensitive, except `true` and `false` (Section 10.4):

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

2.3.5 Whitespace

Whitespace (spaces, tabs, newlines, etc.) is ignored (Section 10.5):

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

2.3.6 Operators and Punctuation

Operators and punctuation (e.g., `+`, `<-`, `.`) are defined as tokens (Section 10.6):

```
PLUS      : '+' ;
MINUS     : '-' ;
TIMES     : '*' ;
DIVIDE    : '/' ;
TILDE     : '~' ;
LT        : '<' ;
LEQ       : '<=' ;
EQ        : '=' ;
```

```
LPAREN    : '(' ;
RPAREN    : ')' ;
LBRACE    : '{' ;
RBRACE    : '}' ;
COLON     : ':' ;
COMMA     : ',' ;
ARROW     : '<-' ;
DOT       : '.' ;
SEMICOLON : ';' ;
AT        : '@' ;
CASEASSIGN: '=>' ;
```

2.4 Implementation

The lexer, defined in `CoolLexer.g4`, generates a Java lexer class integrated with the parser. Key features include:

- **Error Handling**: Detects unterminated strings with informative messages.
- **Case Insensitivity**: Uses fragments for keyword matching.
- **Nested Comments**: Handles nesting with non-greedy matching.
- **Token Stream**: Produces tokens for syntax analysis.

2.5 Example

For the sample program, the lexer produces tokens like `CLASS`, `TYPE(Main)`, `ID(x)`, `INT(42)`, `PLUS`, reflecting COOL's lexical rules.

2.6 Design Decisions

- **ANTLR Choice**: Selected for robust grammar support and error handling.
- **Error Reporting**: Prioritized user-friendly diagnostics for educational use.
- **Modularity**: Separated lexer rules for maintainability.

3 Syntax Analysis

3.1 Purpose

Syntax analysis constructs a parse tree from the token stream, ensuring adherence to COOL's grammar as defined in Section 11 [1]. It detects syntactic errors, such as missing semicolons or malformed expressions, before semantic analysis.

3.2 Background

COOL's grammar specifies programs as sequences of class definitions, each containing attributes, methods, and a rich set of expressions (Sections 3, 7, 11 [1]). The parser, implemented in `CoolParser.g4` using ANTLR 4, generates a labeled parse tree for subsequent AST construction. Key challenges include encoding operator precedence for expressions, supporting `SELF_TYPE` in type annotations, and handling nested constructs like let and case expressions.

3.3 Syntax Structure

The grammar, defined in `CoolParser.g4`, outlines the syntactic structure of COOL programs. Below are the key rules, with labels for parse tree nodes:

3.3.1 Program and Classes

A program consists of one or more class definitions (Section 3):

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

3.3.2 Features

Class features are either methods or attributes (Section 3.1):

```
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)
    ;
```

Methods include parameter lists and a body expression, while attributes may have optional initializers.

3.3.3 Expressions

Expressions (Section 7) cover a wide range of constructs, with operator precedence encoded by rule ordering (Section 11.1):

```
expr
    : expr (AT TYPE)? DOT ID LPAREN exprList? RPAREN                    # Dispatch
    | ID LPAREN exprList? RPAREN                                        #
      MethodCall
    | IF expr THEN expr ELSE expr FI                                    # IfElse
    | WHILE expr LOOP expr POOL                                         # While
    | LBRACE (expr SEMICOLON)+ RBRACE                                   # Block
    | LET letDecl (COMMA letDecl)* IN expr                              # Let
```

```
    | CASE expr OF caseBranch+ ESAC                                  # Case
    | NEW (TYPE | SELF_TYPE)                                         # New
    | TILDE expr                                                     # Negation
    | ISVOID expr                                                    # Isvoid
    | expr TIMES expr                                                # Multiplication
    | expr DIVIDE expr                                               # Division
    | expr PLUS expr                                                 # Addition
    | expr MINUS expr                                                # Subtraction
    | expr LT expr                                                   # LessThan
    | expr LEQ expr                                                  # LessThanEqual
    | expr EQ expr                                                   # Equal
    | NOT expr                                                       # Not
    | ID ARROW expr                                                  # Assignment
    | LPAREN expr RPAREN                                             # Parentheses
    | ID                                                             # Identifier
    | INT                                                            # Integer
    | STRING                                                         # String
    | TRUE                                                           # True
    | FALSE                                                          # False
    ;
```

The rule order ensures higher precedence for operators like `*` and `/` over `+` and `-`, and for unary operators (`~`, `isvoid`, `not`) over binary operators.

3.3.4 Additional Constructs

Supporting constructs include:

```
exprList
    : expr (COMMA expr)*
    ;
letDecl
    : ID COLON (TYPE | SELF_TYPE) (ARROW expr)?
    ;
caseBranch
    : ID COLON (TYPE | SELF_TYPE) CASEASSIGN expr SEMICOLON
    ;
```

`exprList` defines comma-separated argument lists, `letDecl` specifies let variable declarations, and `caseBranch` defines branches for case expressions.

3.4 Implementation

The parser, defined in `CoolParser.g4` and built on ANTLR 4, generates a Java parser class that constructs a labeled parse tree from tokens produced by `CoolLexer`. The grammar uses parser rules with labels (e.g., `# Dispatch`, `# Method`) to annotate parse tree nodes, facilitating AST construction. Key implementation aspects include:

- **Token Integration**: Relies on `CoolLexer`'s token vocabulary (e.g., `CLASS`, `TYPE`, `SELF_TYPE`), ensuring seamless lexer-parser integration.
- **Error Handling**: Employs ANTLR's `BailErrorStrategy` to halt parsing on the first syntactic error, providing immediate feedback with line numbers for issues like missing semicolons or unmatched braces.
- **Parse Tree Construction**: Generates a tree with labeled nodes (e.g., `IfElse`, `Addition`) that capture the hierarchical structure of COOL programs, enabling precise mapping to AST nodes.
- **Precedence Handling**: Encodes operator precedence through rule ordering, ensuring expressions like `a + b * c` are parsed as `a + (b * c)`.

The generated parser class processes input incrementally, building the parse tree in a single pass, which is then traversed by `ASTBuilder.java` to create the AST.

3.5 Example

For the sample program:

```
class Main inherits IO {
    x : Int <- 42;
    main() : Object {
        let y : Int <- x + 1 in
        if y <= 50 then out_int(y) else abort() fi
    };
};
```

The parser constructs a parse tree as follows:

- `program`: Contains a single `class` node.
- `class`: Labeled as `class`, with `TYPE(Main)`, `INHERITS IO`, and features `attribute(x)` and `method(main)`.
- `feature (Attribute)`: For `x : Int <- 42`, includes `ID(x)`, `TYPE(Int)`, and `expr` (`Integer` with `INT(42)`).
- `feature (Method)`: For `main`, includes `ID(main)`, empty `formal` list, `TYPE(Object)`, and `expr` (a `Let` node).
- `expr (Let)`: Contains a `letDecl` (`ID(y) : TYPE(Int) ARROW expr(Addition)`), where the initializer is `expr(Identifier(x)) PLUS expr(Integer(1))`.
- `expr (IfElse)`: Includes `expr(LessThanEqual)` (`ID(y) LEQ INT(50)`), `then` branch (`Dispatch` for `out_int(y)`), and `else` branch (`Dispatch` for `abort()`).

This tree captures the syntactic structure, with labeled nodes reflecting the grammar rules applied.

3.6 Design Decisions

- **ANTLR Usage**: Chosen for its robust grammar specification, automatic parser generation, and error handling capabilities.
- **Precedence Encoding**: Ordered expression rules to match COOL's precedence (Section 11.1), simplifying parsing without explicit precedence declarations.
- **Error Strategy**: Adopted `BailErrorStrategy` for immediate error reporting, suitable for educational use where quick feedback is critical.
- **Labeled Rules**: Used ANTLR labels to annotate parse tree nodes, streamlining the transition to AST construction.

4 Abstract Syntax Tree Construction

4.1 Purpose

AST construction transforms the parse tree into a semantic representation suitable for type checking, code generation, and visualization, as specified in Sections 3, 7, and 11 of the COOL Reference Manual [1].

4.2 Background

The AST, implemented in `ASTBuilder.java`, uses a visitor pattern to traverse the parse tree generated by `CoolParser`. It handles `SELF_TYPE` in type annotations and maintains line numbers for precise error reporting, facilitating subsequent compiler phases.

4.3 AST Node Hierarchy

The `ast` package defines a comprehensive hierarchy of nodes to represent the abstract syntax tree (AST) for COOL programs, as specified in Sections 3, 7, and 11 of the COOL Reference Manual [1]. Each node type corresponds to a syntactic construct in the language, enabling semantic analysis, intermediate code generation, and visualization. The hierarchy is rooted in `ASTNode`, an abstract base class defined in `ASTNode.java`, which provides a line number for error reporting and an `accept` method for the visitor pattern, as specified in `ASTVisitor.java`. The complete node hierarchy is as follows:

- `ProgramNode`: Represents the entire COOL program, containing a list of class definitions.
- `ClassNode`: Represents a class definition, including its name, optional parent class (for inheritance), and a list of features (attributes and methods).
- `FeatureNode`: An abstract base class for class features, with two derived types:
  - `MethodNode`: Represents a method, including its name, formal parameters, return type (including `SELF_TYPE`), and body expression.
  - `AttributeNode`: Represents an attribute, including its name, type, and optional initialization expression.

- `FormalNode`: Represents a method parameter, including its name and type (including `SELF_TYPE`).
- `ExpressionNode`: An abstract base class for all expression nodes, with the following derived types:
  - `DispatchNode`: Represents a method dispatch (e.g., `obj.method(args)`), including the receiver expression, optional static type for dispatch, method name, and argument expressions.
  - `MethodCallNode`: Represents an implicit `self` method call (e.g., `method(args)`), including the method name and argument expressions.
  - `IfElseNode`: Represents a conditional expression (`if-then-else`), including the condition, then branch, and else branch expressions.
  - `WhileNode`: Represents a loop (`while-loop-pool`), including the condition and body expressions.
  - `BlockNode`: Represents a block of expressions enclosed in braces, containing a list of expressions executed sequentially.
  - `LetNode`: Represents a let expression, including a list of variable declarations (`LetDeclNode`) and a body expression.
  - `LetDeclNode`: Represents a variable declaration within a let expression, including the variable name, type, and optional initialization expression.
  - `CaseNode`: Represents a case expression, including the expression to evaluate and a list of case branches (`CaseBranchNode`).
  - `CaseBranchNode`: Represents a branch in a case expression, including the variable name, type, and body expression.
  - `NewNode`: Represents object creation (`new Type`), including the type of the object to instantiate.
  - `UnaryOpNode`: Represents a unary operation (e.g., `~`, `isvoid`, `not`), including the operator and operand expression.
  - `BinaryOpNode`: Represents a binary operation (e.g., `+`, `*`, `<`), including the operator and two operand expressions.
  - `AssignmentNode`: Represents an assignment (`identifier <- expr`), including the variable name and assigned expression.
  - `IdNode`: Represents an identifier reference (e.g., a variable like `x` or `self`).
  - `IntNode`: Represents an integer constant (e.g., `42`).
  - `StringNode`: Represents a string constant (e.g., `"hello"`).
  - `BoolNode`: Represents a boolean constant (`true` or `false`).

This hierarchy captures all syntactic constructs of COOL, enabling robust processing for type checking, code generation, and visualization. Each node type implements the visitor pattern via `ASTNode.accept`, as defined in `ASTVisitor.java`, allowing modular and extensible operations on the AST.

4.4 Implementation

The AST is constructed by `ASTBuilder.java`, which extends `CoolParserBaseVisitor<ASTNode>` to traverse the parse tree produced by `CoolParser`. It converts parse tree nodes into corresponding AST nodes, preserving semantic information and omitting syntactic tokens (e.g., semicolons, braces). The visitor pattern ensures each parse tree context is mapped to the appropriate AST node type. Key methods include:

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

- `visitProgram`: Iterates over class contexts, constructing `ClassNode` instances and aggregating them into a `ProgramNode` with the program's line number.
- `visitClass`: Extracts the class name, optional parent class, and features, creating a `ClassNode`. Features are processed by visiting `feature` contexts to produce `MethodNode` or `AttributeNode` instances.
- `visitMethod` and `visitAttribute`: For methods, collects formal parameters (`FormalNode`), return type (supporting `SELF_TYPE`), and body expression. For attributes, captures name, type, and optional initializer expression.
- `visitDispatch` and `visitMethodCall`: Constructs `DispatchNode` for explicit receiver calls (e.g., `obj.method(args)`) and `MethodCallNode` for implicit `self` calls, handling argument lists.
- `visitIfElse`, `visitWhile`, `visitLet`, `visitCase`: Builds nodes for control flow and scoping constructs, recursively processing sub-expressions and declarations (e.g., `LetDeclNode`, `CaseBranchNode`).
- `visitNegation`, `visitIsvoid`, `visitNot`: Creates `UnaryOpNode` instances with operators (`~`, `isvoid`, `not`) and operand expressions.
- `visitMultiplication`, `visitAddition`, etc.: Constructs `BinaryOpNode` instances for operators (e.g., `*`, `+`, `<=`), linking left and right operand expressions.
- `visitIdentifier`, `visitInteger`, `visitString`, `visitTrue`, `visitFalse`: Produces leaf nodes (`IdNode`, `IntNode`, `StringNode`, `BoolNode`) for basic expressions, handling string literal parsing (e.g., removing quotes).

The `ASTBuilder` ensures all nodes inherit from `ASTNode`, which provides a `lineNumber` field for diagnostics and an `accept` method for visitor-based processing. The construction process is deterministic, with each parse tree context mapping to a single AST node, ensuring a streamlined representation for downstream phases.

4.5 Features

- **Class Support**: Captures class definitions, inheritance, and features (attributes and methods).
- **Expression Coverage**: Supports all COOL expression types, including dispatches, conditionals, loops, and operators.
- **Error Reporting**: Embeds line numbers in every node via `ASTNode`, enabling precise error diagnostics.
- **Extensibility**: Leverages the visitor pattern through `ASTVisitor`, facilitating modular AST processing.

4.6 Example

For the sample program:

```
1 class Main inherits IO {
2     x : Int <- 42;
3     main() : Object {
4         let y : Int <- x + 1 in
5         if y <= 50 then out_int(y) else abort() fi
6     };
7 };
```

The `ASTBuilder` constructs the following AST:

- `ProgramNode` (line 1): Contains a single `ClassNode`.
- `ClassNode(Main)` (line 1): Has parent `IO`, features:
  - `AttributeNode(x)` (line 2): Type `Int`, initializer `IntNode(42)`.
  - `MethodNode(main)` (line 3): Return type `Object`, no formals, body as `LetNode`.
- `LetNode` (line 4): Contains one `LetDeclNode(y)` (type `Int`, initializer `BinaryOpNode(+)` with `IdNode(x)` and `IntNode(1)`) and body `IfElseNode`.
- `IfElseNode` (line 5): Condition `BinaryOpNode(<=)` (`IdNode(y)`, `IntNode(50)`), then branch `DispatchNode` (`IdNode(self)`, method `out_int`, argument `IdNode(y)`), else branch `DispatchNode` (`IdNode(self)`, method `abort`, no arguments).

The `ASTBuilder` maps parse tree contexts (e.g., `AdditionContext`, `DispatchContext`) to these nodes, preserving line numbers and semantic structure.

4.7 Design Decisions

- **Visitor Pattern**: Adopted for modular traversal of parse tree contexts, enabling clean mapping to AST nodes.
- **Type Handling**: Explicitly supports `SELF_TYPE` in `visitMethod`, `visitAttribute`, `visitFormal`, and `visitNew`, aligning with COOL's typing rules.
- **Streamlined AST**: Eliminates syntactic tokens (e.g., `SEMICOLON`, `LBRACE`) to focus on semantic content.
- **Line Number Tracking**: Integrates `lineNumber` in `ASTNode` for all nodes, enhancing error reporting.

5 Semantic Analysis

5.1 Purpose

Semantic analysis ensures type safety and semantic correctness, preventing runtime errors as specified in Sections 4, 5, 6, and 12 of the COOL Reference Manual [1]. It validates inheritance hierarchies, method signatures, and expression types, ensuring compliance with COOL's static typing rules.

5.2 Background

Implemented in `SemanticAnalyzer.java`, this phase leverages `ClassTable.java` and `SymbolTable.java` to manage class metadata and variable scopes, respectively. Key challenges include handling `SELF_TYPE`'s dynamic typing, detecting inheritance cycles, and ensuring method override consistency across class hierarchies.

5.3 Implementation

The semantic analysis phase is orchestrated by `SemanticAnalyzer.java`, which implements the `ASTVisitor<String>` interface to traverse the abstract syntax tree (AST) and infer expression types. It relies on two core components:

- **ClassTable**: Defined in `ClassTable.java`, it maintains a repository of class metadata, including class definitions (`classes`), inheritance relationships (`inheritance`), method signatures (`methods`), and attribute signatures (`attributes`). It initializes built-in classes (`Object`, `IO`, `Int`, `String`, `Bool`) with their standard methods (e.g., `Object.abort`, `String.concat`) and attributes. The `addClass` method populates this metadata for user-defined classes, checking for redefinition errors (e.g., redefining `IO`). The `checkInheritance` method validates inheritance by detecting undefined parent classes, illegal inheritance from basic classes (`Int`, `Bool`, `String`), and cyclic inheritance. The `checkFeatureRedefinition` method ensures attributes are not redefined in subclasses and that overridden methods maintain identical signatures.
- **SymbolTable**: Defined in `SymbolTable.java`, it manages nested scopes for variables using a list of `Map<String, String>` objects, where each map represents a scope mapping variable names to their types. The `enterScope` and `exitScope` methods handle scope nesting for blocks like methods and let expressions. The `put` method adds variables to the current scope, checking for illegal `self` bindings and redefinitions within the same scope. The `lookup` method searches scopes from innermost to outermost to resolve variable types.

The main entry point, `SemanticAnalyzer.analyze`, performs a two-pass analysis:

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

In the first pass, it populates `ClassTable` with class definitions and validates inheritance and feature redefinitions. In the second pass, it traverses the AST using the visitor pattern, performing type checking for each node. Key visitor methods include:

- `visit(ClassNode)`: Enters a new scope, adds attributes to `SymbolTable`, and visits features, ensuring attributes are not named `self`.

- `visit(MethodNode)`: Enters a method scope, adds parameters to `SymbolTable`, checks the body expression's type against the declared return type using `ClassTable.conforms`, and reports type mismatches.
- `visit(DispatchNode)`: Verifies the receiver's type, checks method existence in `ClassTable`, validates argument counts and types, and handles static dispatch and `SELF_TYPE` returns.
- `visit(IfElseNode)`: Ensures the condition is `Bool`, computes the least common ancestor of then and else branch types using `ClassTable.join`.
- `visit(LetNode)` and `visit(CaseNode)`: Manage nested scopes and validate bindings, checking for distinct case branch types.

The `ClassTable.conforms` method implements type conformance rules, resolving `SELF_TYPE` to the current class and checking inheritance chains. The `ClassTable.join` method computes the least common ancestor for expressions like `if-then-else`. Errors are collected in a `List<String>` across all components, with line numbers for precise diagnostics.

5.4 Features

- **Type Checking**: Validates all expressions (Section 12.2), including dispatches, conditionals, let bindings, and operators, ensuring type conformance.
- **Inheritance Validation**: Detects cycles, undefined parents, and illegal inheritance from built-in classes (Section 3.2).
- **Method Redefinition**: Ensures overridden methods have matching signatures (Section 6).
- **Error Reporting**: Collects detailed errors with line numbers for undefined types, type mismatches, and invalid `self` usage.
- **Built-in Classes**: Initializes standard classes with their methods, ensuring correct semantics (Section 9).

5.5 Example

For the sample program:

```
class Main inherits IO {
    x : Int <- 42;
    main() : Object {
        let y : Int <- x + 1 in
        if y <= 50 then out_int(y) else abort() fi
    };
};
```

Semantic analysis proceeds as follows:

1. `ClassTable` adds `Main` with parent `IO`, attribute `x : Int`, and method `main`. It validates inheritance and checks for feature redefinition.

2. `SymbolTable` creates a scope for `Main`, adding `x : Int`. For `main`, it enters a new scope.
3. Type checking:
   - `let y : Int <- x + 1`: `SymbolTable.lookup` finds `x : Int`, `visit(BinaryOpNode)` verifies `x + 1` yields `Int`, and `ClassTable.conforms` ensures `Int` matches `y`'s type.
   - `y <= 50`: Confirms `y : Int` and `50 : Int`, returning `Bool`.
   - `out_int(y)`: `ClassTable.findMethod` locates `IO.out_int`, verifies `y : Int` conforms to the parameter type, and returns `SELF_TYPE` (resolved to `Main`).
   - `abort()`: Finds `Object.abort`, returning `Object`.
   - `if-then-else`: Ensures condition is `Bool`, uses `ClassTable.join` to compute `Object` as the result type.
4. No errors are reported, confirming type safety.

5.6 Design Decisions

- **Two-Pass Analysis**: Separates class metadata collection from type checking to handle forward references.
- **Error Accumulation**: Collects all errors for comprehensive reporting, enhancing debugging.
- **SELF_TYPE Handling**: Dynamically resolves `SELF_TYPE` using `currentClass`, supporting COOL's dynamic typing (Section 5.5).
- **Modular Design**: Separates `ClassTable` and `SymbolTable` for clear responsibility division and reusability.

6 Intermediate Code Generation

6.1 Purpose

This phase generates Three-Address Code (TAC) from the AST, producing a platform-independent intermediate representation suitable for optimization and backend code generation, as outlined in Section 13 [1].

6.2 Background

TAC, implemented in `TACGenerator.java`, uses three-operand instructions to represent COOL programs in a simplified, linear form. It supports COOL's object-oriented features (e.g., method dispatch, attribute access), control flow constructs, and dynamic typing with `SELF_TYPE`, while managing temporaries and labels for efficient translation.

6.3 TAC Representation

The `intermediate` package defines the infrastructure for TAC generation:

- TACProgram: Defined in TACProgram.java, organizes TAC instructions by class and method in a nested map (Map<String, Map<String, List<TACInstruction»>). The addMethod method stores instructions for a specific class and method, and toString generates a human-readable representation of the program.

- TACInstruction: An abstract base class in TACInstruction.java, defines the instruction format with an Opcode, optional result variable, and line number. Supported opcodes include:

  – ASSIGN: Assigns a value to a variable (e.g., x = y).
  – BINARY: Performs a binary operation (e.g., x = y + z).
  – UNARY: Performs a unary operation (e.g., x = ~y).
  – GOTO: Unconditional jump to a label (e.g., goto L0).
  – IF: Conditional jump (e.g., if x goto L0).
  – CALL: Method invocation (e.g., x = call Class.method(args)).
  – RETURN: Returns a value (e.g., return x).
  – LABEL: Defines a jump target (e.g., L0:).
  – PARAM: Declares a method parameter (e.g., param x).
  – LOAD: Loads an attribute (e.g., x = load attr).
  – STORE: Stores a value into an attribute (e.g., store x -> attr).

Each TACInstruction subclass implements toString to produce a readable instruction format, facilitating debugging and control flow graph (CFG) construction.

6.4 Implementation

The TAC is generated by TACGenerator.java, which implements ASTVisitor<String> to traverse the AST and produce instructions stored in TACProgram. The main entry point is:

```
public TACProgram generate(ProgramNode node) {
    node.accept(this);
    return program;
}
```

The TACGenerator maintains state for the current class (currentClass), method (currentMethod), and instruction list (currentInstructions), using ClassTable and SymbolTable for type and scope information. It generates unique temporaries (newTemp) and labels (newLabel) to manage variables and control flow. Key visitor methods include:

- visitProgram: Iterates over classes, delegating to visitClass.

- visitClass: Sets currentClass and processes methods, ignoring attributes unless initialized.

- visitMethod: Creates a new instruction list, enters a scope, adds ParamTAC for parameters, generates TAC for the body, adds a ReturnTAC, and stores the method in TACProgram.

- visitAttribute: Generates StoreTAC for initialized attributes, storing the initializer's result.

- `visitDispatch` and `visitMethodCall`: Generate `CallTAC` for method invocations, handling receiver and arguments, with `MethodCallNode` prefixing the class name for implicit `self` calls.
- `visitIfElse`: Emits `IfTAC`, `GotoTAC`, and `LabelTAC` to implement conditional branching, assigning the result to a temporary if both branches yield values.
- `visitWhile`: Uses `LabelTAC`, `IfTAC`, and `GotoTAC` to create a loop structure with start, body, and end labels.
- `visitLet` and `visitCase`: Manage scopes via `SymbolTable`, generating `AssignTAC` for let initializers and simplified type-checking for case branches.
- `visitUnaryOpNode` and `visitBinaryOpNode`: Produce `UnaryTAC` and `BinaryTAC` for operators (e.g., `~`, `+`), storing results in temporaries.
- `visitAssignment`: Emits `AssignTAC` for local variables or `StoreTAC` for attributes, based on `SymbolTable.lookup`.
- `visitIdNode`, `visitIntNode`, `visitStringNode`, `visitBoolNode`: Generate `LoadTAC` for attributes or return variable names for locals, and `AssignTAC` for constants.

The generator ensures instructions include line numbers for debugging and maintains scope consistency using `SymbolTable`, producing a linear sequence of TAC instructions per method.

6.5 Features

- **Expression Translation**: Supports all COOL constructs, including dispatches, conditionals, loops, let bindings, and operators, with appropriate TAC instructions.
- **Control Flow**: Implements conditionals and loops using `IF`, `GOTO`, and `LABEL` opcodes, enabling accurate CFG construction.
- **Object-Oriented Support**: Handles method calls (`CALL`), attribute access (`LOAD`, `STORE`), and object creation (`new Type`).
- **Error Tracking**: Embeds line numbers in instructions, aiding debugging during optimization and code generation.

6.6 Example

For the sample program:

```
class Main inherits IO {
    x : Int <- 42;
    main() : Object {
        let y : Int <- x + 1 in
        if y <= 50 then out_int(y) else abort() fi
    };
};
```

The `TACGenerator` produces the following TAC for the `main` method (simplified for clarity):

```
class Main:
  method main:
    t0 = load x
    t1 = t0 + 1
    y = t1
    t2 = y <= 50
    if t2 goto L0
    t3 = call Object.abort()
    goto L1
    L0:
    t4 = call IO.out_int(y)
    t5 = t4
    goto L1
    L1:
    return t5
```

- `visitAttribute`: For `x : Int <- 42`, generates `t0 = 42; store t0 -> x` (in class initialization, not shown).
- `visitLet`: Assigns `y` via `t0 = load x; t1 = t0 + 1; y = t1`.
- `visitIfElse`: Generates `t2 = y <= 50; if t2 goto L0` for the condition, `t3 = call Object.abort()` for the else branch, and `t4 = call IO.out_int(y); t5 = t4` for the then branch, with `goto L1` and `L1:` for convergence.
- `visitMethod`: Adds `return t5` to return the if-else result.

This TAC reflects the control flow and expression evaluation, with temporaries (`t0t5`) and labels (`L0`, `L1`).

6.7 Design Decisions

- **Visitor Pattern**: Ensures modular traversal of the AST, mapping each node to TAC instructions cleanly.
- **Temporary Management**: Uses a counter-based `newTemp` for unique temporary variables, avoiding conflicts.
- **Structured Output**: Organizes instructions by class and method in `TACProgram`, facilitating CFG construction and optimization.
- **Scope Integration**: Leverages `SymbolTable` for accurate variable resolution (local vs. attribute), ensuring correct `LOAD` and `STORE` usage.

7 Visualization

7.1 Purpose

Visualization generates PNG images of parse trees, abstract syntax trees (ASTs), and control flow graphs (CFGs) to aid debugging and understanding of COOL programs, as referenced in Sections 11 and 13 [1].

7.2 Background

Implemented in the `visualization` package, this phase uses Graphviz to produce `.dot` files representing parse trees, ASTs, and CFGs, which are then converted to PNG images. Challenges include clearly representing hierarchical structures, maintaining semantic details (e.g., line numbers, types), and handling complex control flow in TAC instructions.

7.3 Implementation

The visualization phase is implemented through four key classes in the `visualization` package:

- **ParseTreeVisualizer**: Defined in `ParseTreeVisualizer.java`, generates a DOT representation of the parse tree from `CoolParser`'s `ParseTree`. It traverses the tree recursively, creating nodes for rule contexts (e.g., `program`, `class`) and terminals (e.g., `CLASS`, `ID`), labeled with rule names, token text, and line numbers. The `visualize` method writes the DOT graph to a file, using `node [shape=box]` for consistent styling.
- **ASTVisualizer**: Defined in `ASTVisualizer.java`, implements `ASTVisitor<String>` to traverse the AST and generate a DOT graph. Each node is labeled with its type (e.g., `Class:  Main`, `IfElse`) and attributes (e.g., line number, return type). The `newNodeId` method assigns unique IDs, and `addEdge` creates directed edges for parent-child relationships. The `visualize` method outputs the graph with `node [shape=box]`.
- **CFGVisualizer**: Defined in `CFGVisualizer.java`, generates a DOT graph for the CFG from a `TACProgram`. It constructs basic blocks by splitting TAC instructions at control points (e.g., `GotoTAC`, `IfTAC`, `LabelTAC`) and builds edges based on control flow (e.g., jumps, sequential execution). Each block is a node labeled with its instructions, grouped in subgraphs per class and method (e.g., `cluster_Main_main`). Nodes use `shape=box, style=filled, fillcolor=lightgrey`.
- **DotToImageConverter**: Defined in `DotToImageConverter.java`, converts `.dot` files to PNG images using the Graphviz `dot` command (`dot -Tpng input.dot -o output.png`). It handles errors via `ProcessBuilder`, verifies output file existence, and logs success or failure.

Each visualizer's `visualize` method produces a `.dot` file, which `DotToImageConverter.convertToPng` transforms into a PNG image. The process ensures robust error handling (e.g., `IOException` for file operations, Graphviz command failures) and consistent node styling across visualizations.

7.4 Features

- **Parse Tree Visualization**: Displays the syntactic structure with rule and token nodes, including line numbers and token text for precise debugging.
- **AST Visualization**: Highlights the semantic structure with nodes for classes, methods, expressions, and their attributes (e.g., types, names), aiding semantic analysis.
- **CFG Visualization**: Illustrates control flow through basic blocks, with subgraphs for each method and edges for jumps and sequential flow, supporting TAC optimization.
- **PNG Output**: Produces high-quality, portable images via Graphviz, compatible with various platforms.

7.5 Example

For the sample program:

```
1 class Main inherits IO {
2     x : Int <- 42;
3     main() : Object {
4         let y : Int <- x + 1 in
5         if y <= 50 then out_int(y) else abort() fi
6     };
7 };
```

The visualizations are as follows:

- **Parse Tree** (`ParseTreeVisualizer`):
  - Root: `program` (line 1), child `class`.
  - `class`: Children `CLASS`, `TYPE(Main)`, `INHERITS`, `TYPE(IO)`, `LBRACE`, two `feature` nodes, `RBRACE`, `SEMICOLON`.
  - `feature` (attribute): `ID(x)`, `COLON`, `TYPE(Int)`, `ARROW`, `expr` (`INT(42)`), `SEMICOLON`.
  - `feature` (method): `ID(main)`, `LPAREN`, `RPAREN`, `COLON`, `TYPE(Object)`, `LBRACE`, `expr` (Let), `RBRACE`, `SEMICOLON`.
  - `Let`: `LET`, `letDecl` (`ID(y)`, `TYPE(Int)`, `ARROW`, `expr(BinaryOp: +)`), `IN`, `expr(IfElse)`.
  - `IfElse`: `IF`, `expr(BinaryOp: <=)`, `THEN`, `expr(Dispatch: out_int)`, `ELSE`, `expr(Dispatch: abort)`, `FI`.
- **AST** (`ASTVisualizer`):
  - Root: `Program` (line 1), child `Class: Main`.
  - `Class: Main` (inherits `IO`, line 1): Children `Attribute: x` (type `Int`, line 2), `Method: main` (return `Object`, line 3).
  - `Attribute: x`: Child `Int: 42`.
  - `Method: main`: Child `Let` (line 4).
  - `Let`: Child `LetDecl: y` (type `Int`, initializer `BinaryOp: +` with `Id: x`, `Int: 1`), body `IfElse`.
  - `IfElse` (line 5): Children `BinaryOp: <=` (`Id: y`, `Int: 50`), `Dispatch: out_int` (`Id: self`, argument `Id: y`), `Dispatch: abort` (`Id: self`).
- **CFG** (`CFGVisualizer`, for `main` method):
  - Subgraph: `cluster_Main_main`.
  - Block 1: `t0 = load x; t1 = t0 + 1; y = t1; t2 = y <= 50`, edges to Block 2 (if true) and Block 3 (if false).
  - Block 2: `L0: t4 = call IO.out_int(y); t5 = t4`, edge to Block 4.
  - Block 3: `t3 = call Object.abort()`, edge to Block 4.
  - Block 4: `L1: return t5`.

Each visualization is output as a `.dot` file and converted to a PNG image, with nodes labeled by their type, attributes, and line numbers, and edges representing structural or control flow relationships.

7.6 Design Decisions

- **Graphviz Format**: Chosen for cross-platform compatibility and robust rendering of directed graphs in PNG format.
- **Node Labeling**: Includes semantic details (e.g., line numbers, types, instruction text) to enhance debugging and comprehension.
- **Basic Blocks**: Splits TAC instructions at control points (e.g., `GotoTAC`, `IfTAC`) in `CFGVisualizer` to clearly represent control flow.
- **Error Handling**: Implements robust file I/O and process execution checks in `DotToImageConverter` to handle Graphviz failures gracefully.

8 Optimization

8.1 Purpose

Optimization improves the efficiency of Three-Address Code (TAC) by reducing runtime overhead and resource usage, as outlined in Section 13 [1], while preserving program semantics.

8.2 Background

Implemented in the `optimization` package, primarily through `ProgramOptimizer.java`, this phase applies a sequence of optimization passes, including constant folding and loop optimizations (loop-invariant code motion and loop unrolling). Key challenges include identifying optimizable patterns (e.g., constant expressions, loop structures) and ensuring semantic equivalence, particularly for COOL's object-oriented and dynamic features.

8.3 Implementation

The optimization framework, defined in `ProgramOptimizer.java`, orchestrates multiple passes over TAC instructions, each implementing the `OptimizationPass` interface from `OptimizationPass.java`. The main entry point is:

```java
public TACProgram optimize(TACProgram program) {
    TACProgram optimizedProgram = new TACProgram();
    for (String className : classMethods.keySet()) {
        for (Map.Entry<String, List<TACInstruction>> entry : methods.entrySet()) {
            List<TACInstruction> optimizedInstructions = instructions;
            for (OptimizationPass pass : passes) {
                optimizedInstructions = pass.optimize(optimizedInstructions);
            }
            optimizedProgram.addMethod(className, methodName, optimizedInstructions);
        }
    }
    return optimizedProgram;
}
```

The optimizer initializes with three passes in `ProgramOptimizer`:

- **ConstantFolder**: Implemented in `ConstantFolder.java`, evaluates constant expressions at compile time. It tracks constants in a `Map<String, String>` and optimizes:
  - `AssignTAC`: Replaces variables with known constants (e.g., `t0 = 42`) and propagates them (e.g., `t1 = t0` becomes `t1 = 42`).
  - `BinaryTAC`: Evaluates binary operations with constant operands (e.g., `t2 = 42 + 1` becomes `t2 = 43`) using `evaluateBinary` for operators (`+`, `-`, `*`, `/`).
- **LoopOptimizer**: Implemented in `LoopOptimizer.java`, applies loop-invariant code motion (LICM) and loop unrolling, using `LoopAnalyzer` to identify loops:
  - `LoopAnalyzer` (`LoopAnalyzer.java`): Builds a control flow graph (CFG) and detects loops via back-edges (e.g., `GotoTAC` to an earlier `LabelTAC`). It constructs `Loop` objects with header, body, and exit indices.
  - `LICM`: Moves invariant instructions (e.g., assignments or operations not dependent on loop-modified variables) to a preheader label, reducing redundant computations.
  - `Unrolling`: Replicates loop body `UNROLL_FACTOR` (4) times for loops with a single exit (e.g., `IfTAC`), renaming temporaries to avoid conflicts, and adjusts control flow with a new exit label.
- **ConstantFolder (second pass)**: Re-applied after loop optimizations to capture new constant opportunities (e.g., from unrolled loop computations).

Each pass processes a method's TAC instructions, producing an optimized list that `ProgramOptimizer` stores in a new `TACProgram`. The framework ensures modularity via the `OptimizationPass` interface, allowing additional passes to be integrated easily.

8.4 Features

- **Constant Folding**: Evaluates and propagates constant expressions (e.g., integers, strings, booleans) at compile time, reducing runtime computations.
- **Loop Optimization**: Applies LICM to hoist invariant code and unrolling to reduce loop overhead, improving performance for iterative constructs.
- **Semantic Preservation**: Ensures optimizations (e.g., constant evaluation, instruction reordering) maintain program behavior, verified through CFG analysis.
- **Modular Pipeline**: Supports multiple passes with `OptimizationPass`, enabling flexible and extensible optimization strategies.

8.5 Example

For the sample program:

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

```
        if y <= 50 then out_int(y) else abort() fi
    };
};
```

The original TAC for `main` (from Section 6) is:

```
t0 = load x
t1 = t0 + 1
y = t1
t2 = y <= 50
if t2 goto L0
t3 = call Object.abort()
goto L1
L0:
t4 = call IO.out_int(y)
t5 = t4
goto L1
L1:
return t5
```

The optimizations applied by `ProgramOptimizer` include:

- **ConstantFolder (first pass)**: Since `x` is initialized to `42`, `t0 = load x` loads `42`, and `t1 = t0 + 1` becomes `t1 = 42 + 1`, folding to `t1 = 43`. Thus, `y = t1` becomes `y = 43`, and `t2 = y <= 50` becomes `t2 = 43 <= 50`, folding to `t2 = true`. The optimized TAC is:

```
        y = 43
        t2 = true
        if t2 goto L0
        t3 = call Object.abort()
        goto L1
        L0:
        t4 = call IO.out_int(43)
        t5 = t4
        goto L1
        L1:
        return t5
```

- **LoopOptimizer**: `LoopAnalyzer` finds no loops (no `WhileNode` or back-edges), so LICM and unrolling are not applied.
- **ConstantFolder (second pass)**: Since `t2 = true`, the `if t2 goto L0` is always taken, but further constant folding is limited as `out_int` and `abort` are method calls. The TAC remains largely unchanged, though dead code elimination (not implemented) could remove the unreachable `t3 = call Object.abort()` branch.

The optimized TAC simplifies constant expressions (e.g., `y = 43`, `t2 = true`), reducing runtime computations while preserving the conditional's semantics.

8.6 Design Decisions

- **Pass-Based Design**: Uses `OptimizationPass` interface for modularity, allowing sequential application of `ConstantFolder` and `LoopOptimizer`.

- **Control Flow Analysis**: Employs `LoopAnalyzer` to build CFGs and detect loops, enabling precise LICM and unrolling.
- **Safe Optimizations**: Restricts transformations to constant propagation and loop restructuring, ensuring semantic equivalence via `isInvariant` checks and temporary renaming.
- **Multiple Passes**: Re-applies `ConstantFolder` after `LoopOptimizer` to capture new optimization opportunities, balancing efficiency and simplicity.

9 Code Generation

9.1 Purpose

Code generation translates Three-Address Code (TAC) into MIPS assembly code executable by the SPIM simulator, as specified in Section 2 [1], enabling runtime execution of COOL programs.

9.2 Background

Implemented in `CodeGenerator.java`, this phase leverages `ClassTable` for type and method information and manages a limited register pool. Key challenges include implementing COOL's object-oriented features (e.g., dispatch tables, attribute access), ensuring type safety, and handling register spilling to the stack for complex expressions.

9.3 Implementation

The code generation phase, defined in `CodeGenerator.java`, translates a `TACProgram` into an `AssemblyProgram`, which organizes MIPS instructions by class and method. The main entry point is:

```
public AssemblyProgram generate(TACProgram tacProgram) {
    for (String className : tacProgram.getClassMethods().keySet()) {
        currentClass = className;
        for (Map.Entry<String, List<TACInstruction>> entry : methods.entrySet()
            ) {
            currentMethod = entry.getKey();
            currentInstructions = new ArrayList<>();
            generateMethod(entry.getValue());
            program.addMethod(className, currentMethod, currentInstructions);
        }
    }
    return program;
}
```

The `CodeGenerator` maintains state for the current class, method, and instruction list, using a register pool (`$r2` to `$r15`) and a stack offset for spilling. Key components include:

- **Register Allocation**: The `allocateRegister` method assigns TAC temporaries to registers via `registerMap`. If no registers are available in `freeRegisters`, it spills to the stack (e.g., `4($fp)`), incrementing `stackOffset`. `freeRegister` reclaims registers when temporaries are no longer needed.

- **Method Generation**: `generateMethod` emits a prologue (`push $fp; move $fp, $sp`), translates TAC instructions, and adds an epilogue (`move $sp, $fp; pop $fp; ret`) for methods without explicit returns.
- **TAC Translation**: The `translateTAC` method maps TAC opcodes to MIPS instructions:
  - `ASSIGN`: `MoveInst` (e.g., `move $r2, $r3`).
  - `BINARY`: `BinaryInst` for arithmetic (`add $r2, $r3, $r4`) or comparison (`cmp` followed by `blt/ble/beq` with labels for `<`, `<=`, `=`).
  - `UNARY`: `UnaryInst` (e.g., `neg $r2, $r3` for `~`).
  - `LABEL/GOTO/IF`: `LabelInst`, `JumpInst`, `BranchInst` (e.g., `bne $r2, L0`).
  - `CALL`: `CallInst`, pushing arguments (`PushInst`) and adjusting the stack (`PopInst`).
  - `RETURN`: `ReturnInst` with epilogue.
  - `PARAM`: `PopInst` to retrieve parameters.
  - `LOAD/STORE`: `LoadInst/StoreInst` for attributes (e.g., `lw $r2, attr($fp)`).
- **AssemblyProgram**: Defined in `AssemblyProgram.java`, stores instructions in a nested map (`class -> method -> instructions`) and formats output with `.class`, `.method`, and `.end_method` directives via `toString`.
- **Instruction**: Abstract base class in `Instruction.java`, defines `opcode` and `lineNumber`, with subclasses (e.g., `MoveInst`, `BinaryInst`) implementing `toString` for MIPS syntax.

The generator uses `ClassTable` to resolve method and attribute offsets, ensuring accurate dispatch and access. Instructions include line numbers for debugging, and stack management supports local variables and method calls.

9.4 Features

- **TAC Translation**: Maps all TAC opcodes to MIPS instructions, supporting arithmetic, control flow, and method calls.
- **Object Support**: Implements COOL's object model with attribute access (`LOAD/STORE`) and method dispatch (`CALL`).
- **Type Safety**: Uses `ClassTable` to ensure correct method and attribute resolution, preventing runtime errors.
- **SPIM Compatibility**: Produces assembly code with proper prologue/epilogue and stack management, executable on SPIM.

9.5 Example

For the sample program:

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

```
        if y <= 50 then out_int(y) else abort() fi
    };
};
```

The optimized TAC for `main` (from Section 8) is:

```
y = 43
t2 = true
if t2 goto L0
t3 = call Object.abort()
goto L1
L0:
t4 = call IO.out_int(43)
t5 = t4
goto L1
L1:
return t5
```

The `CodeGenerator` produces the following MIPS assembly for `main` (simplified):

```
.class Main
.method main
    push $fp
    move $fp, $sp
    move $r2, 43            # y = 43
    move $r3, 1             # t2 = true
    bne $r3, L0             # if t2 goto L0
    jal Object_abort        # t3 = call Object.abort()
    j L1
L0:
    push $r2                # push y (43)
    jal IO_out_int          # t4 = call IO.out_int(43)
    pop                     # adjust $sp
    move $r4, $v0           # t5 = t4 ($v0 is return register)
    j L1
L1:
    move $sp, $fp
    pop $fp
    ret                     # return t5 ($v0)
.end_method
.end_class
```

- **Prologue**: `push $fp; move $fp, $sp` sets up the stack frame.
- **Assignments**: `move $r2, 43` for `y = 43`, `move $r3, 1` for `t2 = true`, using `allocateRegister` to map `y` to `$r2`, `t2` to `$r3`.
- **Conditional**: `bne $r3, L0` for `if t2 goto L0`, since `t2 = true` ensures the branch is taken.
- **Calls**: `jal Object_abort` for `abort()`, `push $r2; jal IO_out_int; pop` for `out_int(43)`, with `$r2` holding `y`.
- **Return**: `move $r4, $v0` for `t5 = t4`, followed by epilogue and `ret`.

The assembly reflects register allocation (`$r2`, `$r3`, `$r4`), stack management, and SPIM-compatible syntax.

9.6 Design Decisions

- **Register Allocation**: Uses a `freeRegisters` pool and spills to stack when exhausted, balancing performance and resource constraints.
- **Dispatch Tables**: Relies on `ClassTable` for method resolution, supporting inheritance and dynamic dispatch.
- **Stack Management**: Implements prologue/epilogue and parameter passing via stack, ensuring robust method calls and local variable handling.
- **Line Number Tracking**: Embeds `lineNumber` in instructions for debugging, aligning with TAC.

10 Main Workflow

10.1 Purpose

The main workflow orchestrates the COOL compiler pipeline, processing `.cl` source files to produce assembly code, intermediate representations, visualizations, and error reports, as described in Sections 2, 11, and 13 [1].

10.2 Implementation

Implemented in `Main.java`, the workflow processes all `.cl` files in the `samples` directory recursively. The main entry point is:

```
public static void main(String[] args) {
    Path samplesDir = Paths.get("samples");
    List<Path> coolFiles = new ArrayList<>();
    Files.walk(samplesDir)
            .filter(path -> path.toString().endsWith(".cl"))
            .forEach(coolFiles::add);
    for (Path inputFile : coolFiles) {
        // Pipeline: lexing, parsing, AST, semantic analysis, TAC, CFG,
           optimization, codegen
    }
}
```

For each `.cl` file, the pipeline performs the following steps:

- **Lexing and Parsing**: Uses `CoolLexer` and `CoolParser` (from `parser` package) with ANTLR's `CharStreams` and `CommonTokenStream`. A `BailErrorStrategy` ensures parsing halts on syntax errors, producing a `ParseTree` for valid input.
- **Parse Tree Visualization**: `ParseTreeVisualizer` generates a `.parse.dot` file, converted to `.parse.png` by `DotToImageConverter`, visualizing the syntactic structure.
- **AST Construction**: `ASTBuilder` transforms the `ParseTree` into a `ProgramNode`, representing the abstract syntax.

- **AST Visualization**: `ASTVisualizer` produces a `.ast.dot` file, converted to `.ast.png`, depicting the semantic structure.
- **Semantic Analysis**: `SemanticAnalyzer` checks the AST for type and scoping errors, producing a `ClassTable` and `SymbolTable`. Errors are written to a `.errors` file, and processing stops if errors are found.
- **TAC Generation**: `TACGenerator` converts the AST to a `TACProgram`, written to a `.tac` file, using `ClassTable` and `SymbolTable` for context.
- **CFG Visualization**: `CFGVisualizer` generates a `.cfg.dot` file for the TAC, converted to `.cfg.png`, illustrating control flow.
- **Optimization**: `ProgramOptimizer` applies constant folding and loop optimizations, producing an optimized `TACProgram`, written to a `.opt.tac` file.
- **Code Generation**: `CodeGenerator` translates the optimized `TACProgram` to an `AssemblyProgram`, written to a `.asm` file, using `ClassTable` for MIPS code generation.
- **Error Handling**: Exceptions during processing (e.g., I/O errors, parsing failures) are caught, with details written to a `.errors` file alongside stack traces.

The workflow uses `java.nio.file` for file handling, ensuring robust directory traversal and output file management. Each phase's output is saved in the same directory as the input file, with filenames derived from the base name (e.g., `test.cl` produces `test.asm`, `test.tac`).

10.3 Example

For the sample program in `samples/test.cl`:

```
class Main inherits IO {
    x : Int <- 42;
    main() : Object {
        let y : Int <- x + 1 in
        if y <= 50 then out_int(y) else abort() fi
    };
};
```

The workflow generates the following files in `samples`:

- **`test.parse.dot`, `test.parse.png`**: Parse tree visualization, with nodes for `program`, `class`, `let`, `if`, etc., labeled with rule names and token text (e.g., `TYPE(Main)`, `INT(42)`).
- **`test.ast.dot`, `test.ast.png`**: AST visualization, showing `Program`, `Class: Main`, `Attribute: x`, `Method: main`, `Let`, `IfElse`, with attributes like line numbers and types.
- **`test.errors`** (if errors): Semantic error messages (e.g., type mismatches) if analysis fails; empty for the sample program.
- **`test.tac`**: TAC for `main`, e.g.:

```
class Main:
  method main:
    t0 = load x
    t1 = t0 + 1
    y = t1
    t2 = y <= 50
    if t2 goto L0
    t3 = call Object.abort()
    goto L1
    L0:
    t4 = call IO.out_int(y)
    t5 = t4
    goto L1
    L1:
    return t5
```

- **test.cfg.dot, test.cfg.png**: CFG visualization, with basic blocks for the `let`, `if`, and branches, connected by control flow edges.
- **test.opt.tac**: Optimized TAC, e.g.:

```
class Main:
  method main:
    y = 43
    t2 = true
    if t2 goto L0
    t3 = call Object.abort()
    goto L1
    L0:
    t4 = call IO.out_int(43)
    t5 = t4
    goto L1
    L1:
    return t5
```

- **test.asm**: MIPS assembly, e.g.:

```
.class Main
.method main
    push $fp
    move $fp, $sp
    move $r2, 43
    move $r3, 1
    bne $r3, L0
    jal Object_abort
    j L1
L0:
    push $r2
    jal IO_out_int
    pop
    move $r4, $v0
    j L1
L1:
    move $sp, $fp
    pop $fp
```

```
            ret
        .end_method
        .end_class
```

The workflow ensures all outputs are generated in the `samples` directory, with visualizations aiding debugging and the `.asm` file ready for SPIM execution.

11 Conclusion

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

References

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