A Cool language Compiler Front-end using ANTLR

Ahmed Gamea

120210114

May 20, 2025

Submitted to:

Eng. Sara Helal

Eng. Esraa Abdelrazek

1 Introduction

1.1 Overview

The **CoolC-ANTLR project** is a lexer and parser for the **Cool (Classroom Object-Oriented Language)**, built using **ANTLR** and **Java**. This project is designed to analyze Cool language programs by first breaking them down into tokens through a lexical analyzer and then validating their syntax with a parser. The compiler supports fundamental Cool language features such as *class definitions, expressions, and control structures.*

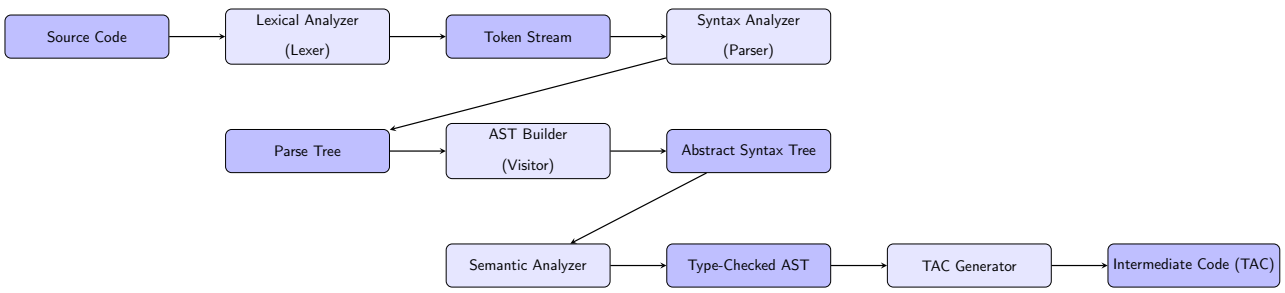
1.2 Technology Used

- **ANTLR** – To generate the lexer and parser from grammar rules.
- **Java** – To handle input processing, tokenization, and syntax validation.
- **Git** – For version control and collaboration.
- **IntelliJ IDEA** – Used as the primary development environment.

1.3 System Workflow

The COOL compiler follows a modular and sequential pipeline that processes the input source code through several well-defined stages:

1. The **Lexical Analyzer (Lexer)** tokenizes the raw source code into a stream of tokens.
2. The **Syntax Analyzer (Parser)** builds a parse tree from the token stream according to COOL grammar rules.
3. The **AST Builder** traverses the parse tree and constructs an abstract syntax tree (AST) using a visitor-based approach.
4. The **Semantic Analyzer** checks type correctness, scoping rules, and inheritance hierarchy to validate the AST.
5. The **TAC Generator (TacGen)** converts the semantically validated AST into three-address code, an intermediate representation.



This modular design promotes clarity and extensibility. Each stage can be tested, replaced, or enhanced independently—making the compiler suitable for educational use and future extensions like optimization or backend code generation.

2 Lexical Analysis

The lexical analyzer scans the input source code and converts the sequence of lexemes into a sequence of tokens.

Lexeme: A sequence of characters that matches a pattern in the grammar (e.g., `class`, `Main`, `123`).

Token: A pair consisting of a token type (e.g., `CLASS`, `TYPE_IDENTIFIER`, `INT`) and its associated lexeme.

The grammar file `LexicalAnalyzer.g4` defines:

- **Keywords:** The language has case-sensitive (`SELF`, `SELF_TYPE`) and case-insensitive keywords (`CLASS`, `INHERITS`, `IF`, etc.). To ensure case insensitivity, keywords are defined using character sets that allow both uppercase and lowercase variants (e.g., `CLASS` is defined as `[Cc][Ll][Aa][Ss][Ss]`).
- **Identifiers:** Identifiers are used for variable names, class names, and function names. They are divided into:
  - `TYPE_IDENTIFIER`: Begins with an uppercase letter and is used for class names defined as

    ```
    TYPE_IDENTIFIER : [A-Z][a-zA-Z0-9_]* | SELF_TYPE
    ```

  - `OBJECT_IDENTIFIER`: Begins with a lowercase letter and is used for variable and method names defined as

    ```
    OBJECT_IDENTIFIER : [a-z][a-zA-Z0-9_]* | SELF
    ```

- **Literals:** The language supports several types of literals:
  - `INT`: Represents integer values composed of one or more digits defined as (INT : [0-9]+).
  - `STRING`: Defined using double quotes and allows escape sequences.
  - `TRUE` and `FALSE`: Boolean values, which must start with a lowercase letter so defined as (e.g., TRUE : [t] [Rr] [Uu] [Ee]).
- **Symbols and Operators:** The lexer recognizes a set of symbols and operators that form the syntax of the language, including:
  - **Punctuation:** `;`, `:`, `,`, `.`, `@`, `{}`, `()`, etc defined as (e.g., `SEMICOLON`: `';'`)

- **Operators:** Arithmetic operators (+, -, *, /), relational operators (=, <, <=), etc.

- **Comments:** The language supports both single-line and multi-line comments:
  - SINGLECOMMENT: Starts with - - and extends to the end of the line defined as

    ```
    SINGLECOMMENT: '--' ~[\r\n]* -> skip
    ```

  - MULTICOMMENT: Starts with (* and extends until *). The rule is designed to support nested comments defined as

    ```
    MULTICOMMENT : '(*' (MULTICOMMENT | .)*? '*)' -> skip
    ```

- **Whitespace Handling:** Spaces, newlines, and tabs are ignored using the WS rule to prevent them from affecting parsing.

3 Parsing and Syntax Analysis

The parser takes the token stream from the lexical analyzer and verifies whether it conforms to the Cool language syntax though building the parse tree representation of the program and generates an output file {input_file}-ast containing the parse tree representation.

The grammar consists of multiple rules defining the structure of a Cool program. The parser uses the token definitions from the lexer via the following directive:

```
options {
  tokenVocab = LexicalAnalyzer;
}
```

3.0.1 Grammar Rules

- **Program Structure:** A Cool program consists of one or more class definitions, ending with EOF:

  ```
  prog  : classDefine+ EOF;
  ```

- **Class Definitions:** Each class can inherit from another class and contains attributes and methods:

  ```
  classDefine : CLASS TYPE_IDENTIFIER (INHERITS TYPE_IDENTIFIER)?
                LBRACE classBody RBRACE SEMICOLON;
  ```

- **Class Body:** The body of a class consists of multiple attributes and methods in any order:

```
classBody: (attr | method)*;
```

- **Attributes:** Attributes define instance variables and may have an initialization expression:

```
attr: OBJECT_IDENTIFIER COLON TYPE_IDENTIFIER (ASSIGN expr)? SEMICOLON;
```

- **Methods:** Methods include a list of parameters and a body enclosed in braces:

```
method: OBJECT_IDENTIFIER LPAREN (parm (COMMA parm)*)? RPAREN COLON TYPE_IDENTIFIER
        LBRACE expr RBRACE SEMICOLON;
```

- **Parameters:** Parameters define method arguments with a name (identifier) and a type:

```
parm: OBJECT_IDENTIFIER COLON TYPE_IDENTIFIER;
```

- **Expressions:** Expressions form the core of computations and include:
  - Assignments
  - Method calls (both short and long)
  - Control flow constructs (if-else, while loops, case expressions)
  - Arithmetic and logical operations
  - Object instantiation (`new`)

3.1 Importance of Rule Order

ANTLR processes rules sequentially, so rule ordering is crucial to avoid ambiguity. Rules that match similar patterns should be ordered carefully:

- Keywords must be defined before identifiers to prevent misclassification. Becasue Longer, more specific rules (e.g., `SELF_TYPE`) should precede general ones (`TYPE_IDENTIFIER`).
- Expressions are ordered to handle precedence (e.g., multiplication and division before addition and subtraction).

4 AST Construction

4.1 Overview

Once the parser constructs the parse tree from the input token stream, a visitor-based traversal is performed to build a more concise and semantically meaningful representation known as the Abstract Syntax Tree (AST). This is handled by the `AstBuilder` class, which walks the parse tree using ANTLR's visitor pattern and constructs typed nodes defined in the `AST` class hierarchy.

4.2 AST Builder

The `AstBuilder` extends `ParserBaseVisitor<Object>` and overrides the `visit` methods for all relevant grammar productions. It handles both declarations and expressions in the COOL language.

4.3 AST Node Hierarchy

The `AST` class contains nested static classes representing all node types in the COOL language. These include:

- **Program structure:** `program`, `class_`, `feature`, `method`, `attr`
- **Expressions:** `plus`, `sub`, `mul`, `assign`, `dispatch`, `cond`, `loop`, etc.
- **Control flow:** `cond` (if-then-else), `loop` (while), `block`, `typcase`
- **Special constructs:** `let`, `static_dispatch`, `new_`, `isvoid`
- **Constants:** `int_const`, `string_const`, `bool_const`, `no_expr`
- **Support nodes:** `binding`, `formal`, `branch`, etc.

Each class stores the relevant subcomponents (e.g., child expressions, identifiers, types) and overrides a `getString()` method to generate an annotated string format of the AST, used for debugging or visualization.

4.4 Example

An example source input such as:

```
let x : Int <- 5, y : Int in x + y
```

Would be transformed by the parser into a parse tree, and by `AstBuilder` into an `AST.let` node with two `binding`s and a `plus` expression as the body.

4.5 Advantages

- The AST structure abstracts away syntactic sugar and irrelevant grammar rules, making downstream passes (semantic analysis, code generation) more straightforward.
- The typed and structured nature of the AST Modular design allows for precise scope and type checking.

5 Semantic Analysis

Semantic analysis ensures that a COOL program is semantically valid according to the language's typing and scoping rules. This section describes the key checks implemented in the semantic analyzer and outlines how each is performed.

5.1 Scoping and Class Information

The analyzer uses:

- **Scope Table:** Tracks variable declarations (attributes, method parameters, locals) via lexical scoping.
- **Class Table:** Stores class attributes and methods, resolving inheritance and conformance relations.

5.2 Class Hierarchy Validation

To validate inheritance relationships, the analyzer constructs a directed graph where each node represents a class, and each edge points from a parent to a child. Key validation steps include:

- **Forbidden Redefinitions:** Classes such as `Object`, `IO`, `Int`, `Bool`, and `String` are considered core language const ructs. Attempting to redefine any of them results in an immediate semantic error.
- **Invalid Inheritance:** Classes are not allowed to inherit from `String`, `Int`, or `Bool`. If a class declares such a parent, the analyzerflags an error and halts.
- **Undefined Parent Classes:** Before graph construction, each parent class name is checked against the known set of declared classes. An undefined parent triggers a fatal error.
- **Cyclic Inheritance Detection:** After the graph is built, a Breadth-First Search (BFS) is initiated from the root class `Object`. A boolean `visited` array marks each visited node. If a node is encountered a second time during BFS, it indicates a cycle in the inheritance hierarchy. Such cycles violate the acyclic requirement of inheritance and are reported as: `Class A or an ancestor of A is involved in an inheritance cycle`. The program exits after detecting cycles to prevent undefined behavior.

5.3 Program Entry Point

The analyzer checks for a valid program entry point by verifying:

- That a class named `Main` exists.
- That `Main` defines a parameterless `main` method.

5.4 Feature Processing

Each class's features are processed with semantic checks:

- **Attributes:** Attribute initializers are type-checked. If the expression's type does not conform to the declared type, an error is reported.
- **Methods:** Method parameters are checked for duplicate names using a local scope. The method body is type-checked, and its inferred return type is validated against the declared return type using the class table's conformance rules.

5.5 Expression Type Checking

Expressions undergo recursive validation according to their kind:

- **Assignments:** Ensure the target variable is declared and the assigned expression type conforms to the variable's type.
- **Dispatch and Static Dispatch:** Validate that the called method exists, is invoked with the correct number and types of arguments, and that the caller expression conforms to the expected dispatch type.
- **Conditionals and Loops:** The predicate must be of type `Bool`. The result type of conditionals is determined by computing the least common ancestor (LCA) of the branch types.
- **Case Expressions:** Each branch must declare a unique and defined type. The result type is inferred as the LCA of all branch bodies.
- **Let Expressions:** Initializer expressions are validated, and their types must conform to the declared types. The variable is scoped locally for the body of the let.
- **Arithmetic and Logical Operations:** Require operands of type `Int` (for `+`, `-`, `*`, `/`) or `Bool` (for `not`, `isvoid`, etc.).
- **Equality:** If either operand is a basic type (`String`, `Int`, `Bool`), both must have the same type.
- **Instantiation:** `new` expressions are valid only for known class types.

6 Three-Address Code Generation

The `TacGen` module translates the COOL abstract syntax tree (AST) into a linear, low-level intermediate representation using three-address code (TAC). This intermediate form simplifies later stages such as interpretation, optimization, or translation to assembly.

6.1 Built-in Class Generation

Before generating code for user-defined classes, the generator emits TAC for built-in classes: `Object`, `IO`, `Int`, `Bool`, and `String`. Each class includes:

- A class declaration and inheritance specification.
- A virtual method table (vtable) associating method names to labels.
- Method implementations (e.g., `Object_abort`, `String_length`) using runtime calls.
- Declarations of external helper routines such as `_print_string` and `_halt`.

Additionally, a `__init` function initializes runtime structures, and a global `main` entry function instantiates `Main` and calls its `main` method.

6.2 Class and Method Emission

Each user-defined class emits:

- A `class` block that specifies its parent and generates a complete vtable by appending new methods to inherited ones.
- A constructor function `Class_new` which:
  - Calls the parent's constructor (if any).
  - Initializes local attributes, using default values or constant expressions.
- Method bodies labeled as `Class_method`, where:
  - Formal parameters are bound to temporary variables.
  - A scoped environment is created for local identifiers.
  - The method body is recursively translated using expression handlers.
  - The result is returned via a final TAC instruction.

6.3 Class Layout and Inheritance Support

To support correct code emission, class metadata is constructed in two passes:

- **Attribute Layouts:** Maps each attribute to an integer offset within object memory. Inherited attributes preserve their offset; new ones append to the end.
- **Method Tables:** Tracks all methods available to a class. If a method overrides a parent method, its label replaces the inherited entry.
- **Parent Mapping:** A map from each class to its parent is used for layout inheritance and constructor chaining.

6.4 Expression Translation

Expressions are lowered recursively into TAC using the `generateExpr` function. Highlights include:

- **Constants and Variables:** Constants are emitted as `tN = const x`. Variables are resolved from the current environment or read from the object's attribute section using an offset from `self`.
- **Unary and Binary Operations:** Expressions like `+`, `-`, `*`, `/`, `==`, and logical negations use standard TAC instructions and temporary registers.
- **Assignments:** The right-hand side is evaluated, and the result is either stored in a local variable or assigned to an attribute of `self`.
- **Blocks:** Sequentially evaluate each expression, returning the value of the last one.
- **Conditionals:** Emit conditional branches with fresh labels for `then`, `else`, and `end`. Use `ifFalse` instructions to control flow.
- **Loops:** Emit a test label, loop body label, and backward jump to emulate `while` constructs.
- **Dispatch (Dynamic):** The receiver is evaluated and the method's address is fetched from its vtable. Parameters are pushed in reverse order and the method is called using `call_indirect`.
- **Static Dispatch:** Uses the statically known class to compute the method label directly. Arguments are pushed, and a direct call is emitted.
- **Object Creation:** `new ClassName` translates to a call to the `ClassName_new` constructor.

6.5 Temporaries and Labels

Temporary variables (`t1`, `t2`, ...) are allocated incrementally to hold intermediate results. Similarly, fresh labels (`L1`, `L2`, ...) are generated to control conditional and loop branches, ensuring correct flow control without name collisions.