A Cool language Lexer and Parser using ANTLR

Ahmed Gamea

120210114

March 25, 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 compiler follows a sequential pipeline:

1. The **lexer** processes the input source code and produces a tokenized output.
2. The **parser** receives the token stream and constructs an AST.
3. The generated AST is written to an output file for further processing.

Source Code → Lexer → Token Stream → Parser → AST

This modular approach ensures flexibility, allowing for future enhancements such as semantic analysis and code generation.

2 System Architecture & Design

2.1 Project Structure

The CoolC-ANTLR compiler is organized into two main components: the **Lexical Analyzer** and the **Parser**. Each component follows a structured directory layout to ensure modularity. The project structure is as follows:

```
CoolC-ANTLR/
 LexicalAnalyzer/
    gen/
       LexicalAnalyzer.java     % Lexer implementation code
       LexicalAnalyzer.tokens   % List of token types
    grammar/
       LexicalAnalyzer.g4       % Lexer grammar rules
    src/
       Main.java                % Main program that runs the lexer
    test_cases/                 % Input test cases
 Parser/
    gen/
       LexicalAnalyzer.java
       ParserParser.java        % Parser implementation code
    grammar/
       Parser.g4                % Parser grammar rules
       LexicalAnalyzer.tokens
    src/
       Main.java                % Main program that runs the Parser
    test_cases/                 % Input test cases
```

2.2 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.

2.3 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 Abstract Syntax Tree (AST) representation of the program and generates an output file `{input_file}-ast` containing the AST 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;
}
```

2.3.1 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`)

2.4 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).