ANTLR4: A Comprehensive Guide

Chabane Chaoche Rabah & Foul Rami Abdledjali

April 20, 2025

Table of Contents

Contents

1 ANTLR4

What is ANTLR4?

ANTLR4 (ANother Tool for Language Recognition, version 4) is an open-source parser generator that uses an adaptive LL(*) parsing algorithm to analyze context-free grammars.

It rejects any left-recursive rule at compile time, since it's a top-down parser.

ANTLR4 is available in multiple languages (Java, Python, C#, Go, etc.)

It generates the lexer, which comes with its own default message log when analyzing tokens. It also generates the parser and the AST, which can be viewed either in the terminal or through a Swing GUI window.

In ANTLR4, both the lexer tokens and grammar rule definitions reside in a single .g4 file. For semantic analysis and processing, separate Java files are referenced and invoked from within this .g4 file.

Note

ANTLR4 can accept a rule of the form $A \to AaA|a$, as it is neither left nor right recursive.

However, it does not accept directly left-recursive rules like $A \to Aa|a$, since it is a top-down parser.

2 Installation ANTLR4

Installation

We followed the instructions from this video
The person in the video created a GitHub repository to make the installation easier: repository, which includes the ANTLR4 `.jar` dependency and a `bin/` folder containing three `.bat` scripts:

- `antlr4.bat`: Compiles the `.g4` grammar file , produces a log file and generates four Java files:
  - **Lexer**: Recognizes individual tokens from the input stream.
  - **Parser**: Defines the grammar rules and structure for interpreting token sequences.
  - **BaseListener**: A default listener class with empty method implementations for parse tree traversal.
  - **Listener**: An interface containing method signatures triggered during parsing.
- `compile.bat`: Compiles all the generated and custom Java files using the ANTLR4 `.jar` as a dependency.
- `grun.bat`: Launches the lexer and parser, and displays the resulting AST (Abstract Syntax Tree) either in the terminal or in a Swing-based GUI window.

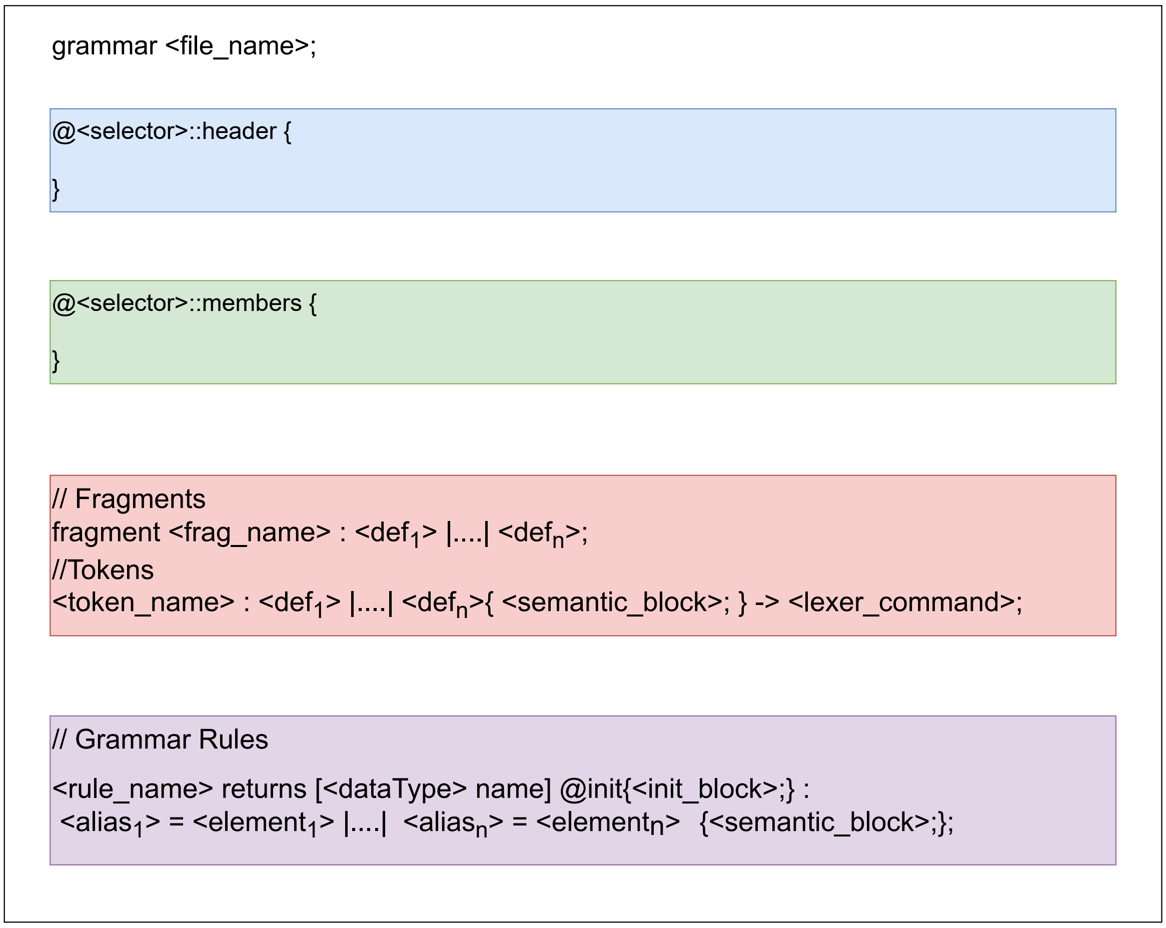
The author also recommends adding the path to the `bin/` folder in the system environment variables to make the `.bat` scripts accessible from anywhere.

Note

The `grun.bat` script is only meant to simplify the launching of the lexer and parser it is not the only way to run them. We can also invoke the analysis directly in Java code.

While we initially used `grun.bat`, we later switched to launching the analysis directly from our Java code when we began implementing the simple IDE.

3 Structure Of .g4 File



grammar <file_name>;

@<selector>::header {

}

@<selector>::members {

}

// Fragments
fragment <frag_name> : <def$_1$> |....| <def$_n$>;
//Tokens
<token_name> : <def$_1$> |....| <def$_n$>{ <semantic_block>; } -> <lexer_command>;

// Grammar Rules
<rule_name> returns [<dataType> name] @init{<init_block>;} :
<alias$_1$> = <element$_1$> |....| <alias$_n$> = <element$_n$> {<semantic_block>;};

3.1 Grammar Line

Grammar Line

At the beginning of the .g4 file, we define the grammar name, which must match the filename. By convention, we often name it `Expr`.

The lexer and parser `.java` file will be named based on this name if we take the name `Expr` we'll get `ExprLexer` for the lexer and `ExprParser` for the parser.

3.2 Header & Members

3.2.1 Selectors

Selectors

Selectors are used to target specific sections of the generated lexer and parser `.java` files:

- **None** Applies to both the lexer and parser.
- `parser::` Targets the parser only.
- `lexer::` Targets the lexer only.

These selectors are used in the `@header` and `@members` sections to insert code or import statements into the appropriate generated files.

3.2.2 Header

@header

The `@header` section is used to insert package declarations and import statements into the targeted files.

3.2.3 Members

@members

The `@members` section allows you to insert additional Java code into the targeted files.

Note

Both `@header` and `@members` are optional.

3.3 Tokens & Context Object

Tokens & Context

Terminal symbols (defined in the lexer) are represented as `Token` objects. These provide useful methods and attributes:

- `getText()` : Returns the matched text of the token.
- `getLine()` : Returns the line number.
- `getCharPositionInLine()` – Returns the column position of the token.
- `channel` : By default, tokens are assigned to channel 0 and processed by the parser. Other channels allow you to ignore tokens like comments and whitespace, helping to keep the grammar clean and readable.

Non-terminal symbols (defined as parser rules) are represented as `Context` objects. These are recursive and follow a composite structure, representing the parse tree hierarchy.

- `getText()` : Returns the full text matched by the rule.
- `start` : The first `Token` that triggered this rule context.
- `getTokens(token_type)` : Returns all matched tokens of a given type within this context.

3.4 Lexer

3.4.1 Syntax

Syntax

The syntax for defining regex is similar to that of many other languages and parser generators. It uses `''` for literal strings, and the ~ symbol instead of ^ to indicate negation.

All fragment and token names must start with an uppercase letter.

3.4.2 Fragment

Fragment

Fragments are reusable regular expression components that are not matched by the lexer on their own. Instead, they are used within token definitions to build more complex patterns.

3.4.3 Tokens

Tokens

Tokens are the lexical elements that the lexer identifies and matches based on defined patterns.

Semantic blocks are blocks of Java code placed inside the token rule they are executed each time the token is matched. They are useful for logging, value validation, or triggering side effects.

Lexer commands on the other hand actions that follow the `->` symbol They include:

- `skip` : Ignores the matched token (commonly used for whitespace, newlines, etc.).
- `channel(HIDDEN)` : Sends the token to the `HIDDEN` channel, so the parser will ignore it and not include it in the parse tree.

ERROR_TOKEN is a default token_name provided by ANTLR4 it means it didn't match any defined token.

Note

Lexer command , semantic block and fragments are all optional.

3.5 Grammar Rules (Parser)

Rules

All non-terminal symbols (grammar rules) must start with a lowercase letter.

To access elements of a rule, we use the dollar sign `$`.

ANTLR provides shorthand for frequently used methods and fields to improve readability:

- `getText()` → `text`
- `getLine()` → `line`
- `getCharPositionInLine()` → `pos`
- `ctx` → current context (acts like `this`)

By default, associativity is left-to-right.

When multiple alternatives are defined in the same rule, priority is given from left to right. However, there is no implicit precedence between separate rule elements.

For example:

```
expr : expr (DIV | MUL) expr
     | expr (SUB | PLUS) expr ;
```

In this case:

- `DIV` and `MUL` have equal priority.
- `SUB` and `PLUS` have equal priority.
- But the first alternative has higher priority than the second, because it comes first.

The `@init` block is a code block that gets executed as soon as the rule is matched.

The `returns [type name]` clause allows you to attach additional attributes to a rule (non-terminal symbol).

You can also assign aliases to elements within a rule using `=` sign.

Note

Semantic blocks, `@init` blocks, additional attributes, and aliases are all optional.

Semantic blocks do not have to appear at the end of the rule multiple semantic blocks can be placed at different positions within the same rule.

4 Semantic

Semantic

Semantic behavior in ANTLR4 can be implemented in two main ways:

- By invoking semantic routines directly inside semantic blocks within rules in the `.g4` file.
- By using listeners and invoking semantic routines inside the corresponding listener methods.

5 Things We Added

5.1 Lexer

5.1.1 Added Keywords

Keywords

- `True`
- `False`
- `break`
- `switch`
- `default`
- `Bool`
- `Char`
- `String`

5.1.2 Token Validation

Validation

We added value validation for specific tokens:

- `float` : range must be within $[-3.4 \times 10^{38}, 3.4 \times 10^{38}]$
- `String` : maximum length of 100 characters
- `Char` : must be exactly one character

5.1.3 Float Format

Float

We added support for float literals written in compact form, such as `.21` and `33.`, in addition to the standard format.

5.2 Parser

New Rules

- **concat**: concatenation between strings and characters using the `.` operator.
- **switch-case**: follows C-style syntax.
- **elsif {}**: additional conditional branching.
- **while loop**: standard looping structure.

Altered Rules

- **Condition**: now supports boolean expressions like `a AND b`.
- **{value$_1$, ..., value$_n$}**: allows assigning a list of values to an array.
- **Expression assignment**: variables can now be initialized with full expressions, not just simple values.

New Semantic Checks

- **Assigning an array to a non-array, or vice versa**: ensures type compatibility.
- **Using array variables in expressions**: error can't use array in any expression.
- **Indexing into a non-array variable**: disallowed and triggers an error.
- **Assigning an array with more elements than the destination can hold**: raises a size mismatch error.
- **Undefined or invalid array sizes**: must be explicitly and correctly defined.
- **Arrays with elements of mismatched types**: enforces type homogeneity.
- **Arrays exceeding the maximum allowed number of elements**: triggers overflow errors.
- **Invalid expressions due to incorrect data types**: ensures expression consistency.
- **Using a constant as a loop index in `for`**: invalid a constant cannot be used as a loop variable.
- **Using an array as a loop index in `for`**: invalid .

5.3 IDE

IDE

We developed a small IDE with syntax highlighting and a simple user interface to launch the different steps of compilation and view the console log output.

The IDE was built using `JavaFX` and `RichTextFX`, a library that simplifies the creation of editor-like components. Our design and syntax-highlighter was inspired by the Astro IDE.

5.4 Desktop Deployment

5.4.1 Jlink JRE

Jlink

We bundled our application into a standalone `.jar` executable, and then used `jlink` to generate a custom JRE. This ensures that our IDE runs on any Windows machine, even without a pre-installed JDK.

5.4.2 NSIS Wizard Installer

NSIS

Finally, we created a Windows wizard-style installer using `NSIS`. It allows users to easily install the application and automatically creates a desktop shortcut.