CS 8803-O08: ANTLR4 Guide

Jon Greene
`jgreene78@cc.gatech.edu`

Georgia Institute of Technology — Updated: September 3, 2022

Overview

This guide is meant to be a starting point for the further study of ANTLR4. It is not comprehensive.

1 ANTLR4 Introduction

You may be familiar with the programs lex and yacc, or their GNU counterparts flex and bison. When provided with a configuration or grammar file, these programs can be used to generate a lexer (lex/flex) or a parser (yacc/bison). The benefit provided is that the code is produced directly from the configuration file itself, without having to hand-code a lexer or parser.

ANTLR4 is a modern implementation of a lexer and parser generator. When provided with a configuration file consisting of a grammar and lexer specification for a language, ANTLR4 can produce source code files which may be used directly in your code to implement a lexer and LL(*) parser. Furthermore, ANTLR4 provides a method of walking the parse tree and provides both listener and visitor interfaces for easy extraction of information. Using these interfaces, rather than embedded actions, allows the programmar to keep the grammar and application code decoupled.

2 Resources

When accessing online resources, note that we use ANTLR4, not ANTLR3. The two versions are not compatible.

- Website with installation instructions and usage examples - www.antlr.org
- Github repo with example grammars for common languages - github.com/antlr/grammars-v4/
- Definitive ANTLR4 Reference Manual - www.oreilly.com/library/view/the-definitive-antlr/9781941222621/

  The reference manual is free to GA Tech Students. Chapter highlights:
  - 5 - information on designing grammars
  - 4.2, 7.2 - example of implementing listener
  - 4.3, 7.3 - example of implementing visitor
  - 7.4 - rule labels are important tools for generating clear grammars
  - 7.5 - information on sharing information between methods
  - 9 - error reporting, including altering default error messages
- LL(*) research paper (not required reading) - www.antlr.org/papers/LL-star-PLDI11.pdf

3 Installation

The Vagrant VM comes with the correct version of ANTLR4 pre-installed: ANTLR 4.9.3. If installing locally, follow the directions using the link in resources.

4 Usage

4.1 Workflow

1. Create or modify an ANTLR4 grammar file to match the target language.
2. Run `antlr` executable to generate the lexer/parser C++/Java source files.
3. Extend the generated visitor/listener base class or implement the generated visitor/listener interface to do work on the parse tree.
4. Write your code to call ANTLR4 classes to lex/parse the input file and use your visitor/listener classes.
5. Compile and test.
6. Repeat...

4.2 Grammar

ANTLR4 generates its parser and lexer directly from a text file containing the target language's grammar rules and lexical specification. The syntax is similar to BNF but with enhanced options and functionality.

- The lexer and parser rules can be in the same file, with the parser rules generally appearing first.
- Lexer rules use all UPPERCASE letters.
- Parser rules use all lowercase letters.
- The starting production rule name is arbitrary but is used in your source code to indicate the starting point of the parse.
- The name of the grammar file must match the name of the grammar. Assuming a grammar named "Example":
  - The name of the grammar file must be: `Example.g4`
  - The first line of the grammar must be: `grammar Example;`

4.3 Listener and Visitor

ANTLR4 can generate both listener and visitor interfaces. These interfaces are used to interact with the generated parse tree. You may decide to use either listeners, visitors, or both depending on the task and your implementation preferences. Their use cases largely overlap, but you may find one or the other easier to use for a given scenario.

The primary difference between them is that a visitor directly controls the process of walking the tree, whereas a listener passes control to an external "walking" mechanism that handles walking the parse tree. This leads to several implemenation differences:

- A listener requires an external "Walker" class (which is generated automatically by ANTLR4). A visitor does not require an external walker.
- A listener accesses every node via the walker. A visitor must choose which nodes to visit.
- A listener must use an external data structure to pass information between methods (such as Parse-TreeProperty). A visitor can use the call stack.
- A listener is simpler to implement so may be the best choice for simple tasks. A visitor is more complex but required if the programmer needs control over the path taken.

The rules defined in the grammar **directly** affect the API created for the visitor and listener interfaces. There will be one or more function definitions created for every grammar rule. Any changes to the grammar rules cascade to changes in the interfaces.

Info: Careful crafting of the grammar rules can result in much simpler visitor/listener interfaces.

4.3.1 Listener

The `antlr` executable will by default create both a listener interface and a base class which implements the interface. The base class contains empty stub functions for every rule, allowing for simple use.

The listener interface is grammar specific and will contain an empty "enterRuleXX" and "exitRuleXX" function for each rule in your grammar. To create a listener, simply extend the base listener class and override any of the functions with the code you want to run while entering or exiting the nodes.

Using the implemented listener class requires an external parse tree walker - either the one provided by ANTLR4, ParseTreeWalker, or one of your own design. The walker will "walk" the parse tree. As it walks, it will call your "enterRuleXX" function as it enters each rule node and will call your "exitRuleXX" function as it exits each rule node. This allows you to run pre-order or post-order code on your tree.

4.3.2 Visitor

The `antlr` executable must be called with the `-visitor` command line option in order to create a visitor interface and base class. The base visitor class not only contains stub functions for every rule, but also extends the class AbstractParseTreeVisitor<T> to provide a default visit() method implementation.

The visitor interface is grammar specific and will contain a "visitRuleXX" function with generic return type <T> for each rule in your grammar. To craft a visitor, extend the grammar specific base class and override the methods required for your application.

Using the implemented visitor is different from using the listener. When using a listener, all nodes will automatically be visited in order by the walker. Whereas when using a visitor, your code is responsible for manually calling "visit()" on any child nodes that need visiting.