Programming Assignment: Exploring Grammars with ANTLR

CSI 310: Programming Languages

Spring 2023

Goals for this assignment

- Practice working with ANTLR grammars
- Understand operator precedence and associativity as expressed in ANTLR grammars.
- Gain a deeper understanding of grammars and parse trees.

Academic Honesty

Please read carefully the section titled “Academic Integrity” in the syllabus. If you have any questions, contact your instructor before you begin this assignment.

Setting Up

Download the zip file named `antlr_template.zip` , extract it and use it as a starting point. Rename the extracted directory to `pa2` .

Visualizing a Parse Tree with the ANTLR Plugin

1. Create your grammar file: `GrammarName.g4`, , and place it in the folder `src/main/antlr` . See the Section below for the basic skeleton of a grammar file.
2. Create an input file for testing (e.g. `input.txt`), containing some sentences that are in the language described by your grammar.
3. Edit the `build.gradle` file and change the variables in the ext section to match your grammar and input file.
4. From the command line, run `./gradlew parseTree` (macOS/Linux/Cygwin) or `gradlew parseTree` (Windows CMD).
5. If there are no errors, the ANTLR window should appear with a visual representation of the parse tree.

Assignment

For each problem below, turn in a print-out and a pdf of your grammar(s) and parse tree(s). Take a screenshot of your parse tree, and paste into a document. **Please try to not use too much paper!** Organize your work so that it is clear and easy to read. Do each problem in order and organize your papers in the same order. Scale the images so that they are not too large and place them in the document next to the associated grammars, along with your answers to each question.

1. Convert the BNF grammar (not the EBNF one) from Example 3.5 in the textbook to a equivalent ANTLR grammar. Place the ANTLR grammar in a file named `Expression.g4`. Test it on the following input sentences:
    - `A + B * C`
    - `(A+B)**C`
    - `A + B ** C`
    - `A**B/C`

    For each sentence, predict what the parse tree should be, then test it in the ANTLR Preview window.

2. Repeat the above problem, except use the EBNF version of the grammar. Convert the EBNF grammar from Example 3.5 into ANTLR notation. You'll need to use ANTLR's `*` (zero or more repetitions) and `|` operators, as well as parentheses for grouping. You'll find that the ANTLR grammar looks very similar to the EBNF form. Place this grammar into a file named ExpressionEBNF.g4.

   Repeat the above tests and verify that the parse trees are similar. Are the differences significant?

3. In the grammar Expression.g4, what is the associativity of the operators? Turn in a parse tree that demonstrates the associativity of the `*` operator and one for the `**` operator. Explain how the tree shows the associativity.

4. Using the EBNF grammar from Example 3.5 (ExpressionEBNF.g4), how is associativity of the same operators represented? Show the parse trees for both grammars that demonstrate the difference. How is associativity expressed in this grammar?

5. Implement the ambiguous grammar from Example 3.3 in a file named Ambiguous.g4. Since this grammar is ambiguous there are multiple ways that the parse trees could be generated. Do some experiments to determine how ANTLR resolves the ambiguity by default. How does this determine the associativity of the `+` and `*` operators?

6. ANTLR provides a way to change the associativity for an ambiguous grammar like this one. See this URL for the syntax:

   https://github.com/antlr/antlr4/blob/master/doc/left-recursion.md#formal-rules

   Turn in a version of this grammar that uses right-to-left associativity for the + operator, and left-to-right associativity for the * operator. Turn in a parse tree that demonstrates each.

7. With the ambiguous grammar of Example 3.3, investigate precedence. ANTLR uses a simple rule to resolve the ambiguity. What rule does it use? Recall that the way that the ambiguity is resolved determines the operator precedence. What determines the operator precedence for this grammar? How would you modify the grammar slightly (without changing the rule content), but give the * operator higher precedence? Try it in ANTLR and verify that it works. Turn in a print out of the parse trees before and after the modification for the sentence: `A = A+B*C`.

8. Update the grammar from question 1 so that it accepts multiple assignment statements that are terminated by semicolons. Turn in the grammar and an ANTLR generated parse tree (just one) for the following input:

```
A = A+B;
C= A-B;
A = A * B;
```

Hints

The basic structure for your grammar files should be as follows:

```
grammar GrammarName;

// Lexer rules
WS: [ \t\n\r]+ -> skip;

// Grammar rules
...
```

On the first line, replace GrammarName with the name of your grammar which should be the same as the file name (without the .g4 extension). The lexer rules define how to handle white-space (ignore), and define ID as a token class for the identifiers A, B, and C. You can use ID within your grammar rules.

Submission Instructions

- Create a document containing your answers to the above questions. Include (copy and paste) your grammars and pictures of your parse trees. Include written paragraphs describing the pictures and answering the questions. You can use a screen shot tool to get an image of your parse tree, however, please don't include all of the screen, just the parse tree only. Make your solutions clean, clear, and understandable. Use a mono-spaced font for all grammars.
- Turn in a print out of your solutions in class on the due date with the grade sheet stapled on top. Additionally submit your work digitally.