Queenie Leung

CS 326

Interpreter 1

09/27/23

Source provided from Interpreter 1 in Assignment: <https://github.com/antlr/grammars-v4>

Note: Interpreter.g4 is

1. Your original solution, before an Internet search or any help, whatsoever (5 points)

See interpreter.g4 (original) file

A computer screen shot of a program

Description automatically generated

1. A list of the three grammars you chose from the source above (3 points)
2. For each grammar, place in a document the rules that are similar to yours. **DO NOT COPY THE ENTIRE GRAMMAR. IF YOU DO, YOU WILL RECEIVE ZERO POINTS FOR THE ASSIGNMENT.** The purpose is to look through the grammar and learn what regions of it match to the kinds of grammar rules you are working with so far, not to just blindly copy something from the Internet. (5 points)
3. For each rule you copied in, discuss whether, and how, it is different, from your attempt at the problem that you tried before you looked things up. (3 points)

**Note**: Questions 1 to 4 will be answered in order after each grammar listed below:

**Grammar Rule:**

**Javascript** (https://github.com/antlr/grammarsv4/blob/master/javascript/javascript/JavaScriptParser.g4)

singleExpression

:  singleExpression ('\*' | '/' | '%') singleExpression                   # MultiplicativeExpression    ;

| singleExpression ('+' | '-') singleExpression                         # AdditiveExpression

**Similar Lexer Rule:**

TIMES : '\*';

DIVIDE: '/';

MODULO: '%';  //note this is not similar to the rules I have but this is how I would define it.

PLUS: '+';

MINUS: '-';

**Similarities/Differences**: The way this was set up initially from the given assignment code is very similar however I did not know that there could be more than 2 OR in the grammar *expression*  (\* | / | % ) *expression*. That it would be able to handle modulo if parsed correctly. I thought I needed to create a separate modulo expression (i.e. *expression* (%) *expression*) or something of the sort. But Javascript is definitely very similar to how Antlr is written at least in the grammar I found for comparison.

**Grammar Rule:**

**C** (https://github.com/antlr/grammars-v4/blob/master/c/C.g4)

castExpression

    :   '\_\_extension\_\_'? '(' typeName ')' castExpression

    |   unaryExpression  //did not copy entire grammar but this expression had \* | + | / | - …etc

    |   DigitSequence // not sure what this was for as I searched but I couldn’t find its rules

    ;

multiplicativeExpression

    :   castExpression (('\*'|'/'|'%') castExpression)\* //this is using the earlier castExpression above

    ;

additiveExpression

    :   multiplicativeExpression (('+'|'-') multiplicativeExpression)\* //now this is using the multiplicative expression above

    ;

**Similar Lexer Rule:**

TIMES : '\*';

DIVIDE: '/';

MODULO: '%';  //note this is not similar to the rules I have but this is how I would define it.

PLUS: '+';

MINUS: '-';

**Similarities/Differences**: It was interesting to compare grammar rules we did in our project vs Javascript and now compared to C. There are similarities but there was differences in types of ways execution was made. In the class we had expression ( + | - ) expression however I did not know you could technically create an expression of an expression which you see above there is castExpression and then in similar grammar rules below you see multiplicativeExpression using the castExpression – then again additiveExpression uses multiplicativeExpression within the grammar. It is a bit confusing at first when reading it but I believe it is to set some level of precedence here.

**Grammar Rule:**

**BASIC** (https://github.com/antlr/grammars-v4/blob/master/basic/jvmBasic.g4)

signExpression

: NOT? (PLUS | MINUS)? func\_

;

exponentExpression

: signExpression (EXPONENT signExpression)\*

;

multiplyingExpression

: exponentExpression ((TIMES | DIV) exponentExpression)\*

;

addingExpression

: multiplyingExpression ((PLUS | MINUS) multiplyingExpression)\*

;

**Similar Lexer Rule:**

PLUS: '+';

MINUS: '-';

**Similarities/Differences:** This is another example of using another expression within another expression to create a level of operator precedence. While it is like C and Javascript, it still has similarities to my original interpreter code but with minor differences. Similarities are like expression (PLUS | MINUS) expression from our given interpreter.g4 grammar. However again its like layered version of an expression to the next expression. What was also different was that they had a NOT operator similar to a ~(PLUS | MINUS) if I was to do quasi coding and the “?” after NOT? (PLUS | MINUS)? func\_ after researching means that the NOT? Is optional and the (PLUS | MINUS)?func\_ is like saying +func (positive value) or -func(negative value) that also expresses a function call. Which also incorporates a Boolean into a function call or expression from what I can understand.

1. Finally, rewrite your grammar rules to what you think, subjectively, is the best solution from amongst the various languages. Write a paragraph stating what you changed and why you changed it. (5 points)

A computer screen shot of a program

Description automatically generatedA screen shot of a computer program

Description automatically generated

Revised

Original

Changes I made were line 12 where I did not realize you could directly add parenthesis in while I was looking at the other codes from GitHub. I think this is better as it is easier to read than putting it in as a Lexer Rule from my interpreter.g4. The next line I changed is line 22 where I realized that after a Lexer rule was defined it could be used within another Lexer rule for NUMBER. Adding the \* was similar to the + at the end of the INT where there could be zero or more numbners after the decimal point. This I thought intuitively looked better and visually differentiates from the Lexer rule INT. I changed line 24 WS as I thought it was just ‘ ’ as it would be recognized from coding in C++ but after reviewing multiple parser rules in the GitHub provided I realized that \t (tab), \r (return carriage), \n (new line) were also seen as white space. I feel the change here made it more concise and adding the + made it so that everything after it would be skipped and not read by the parser. Similarly on line 26 I thought including ‘a’.. ‘z’ ‘ ’ ‘a’.. ‘z’ ‘ ’ etc. would allow for characters to be entered similar to the provided INT but changing it to not return carriage and new line ( ~[\r\n]\*) ->skip made it simplier and more concise to understand with my current coding knowledge.

1. Important: As one final inclusion, have a section in your word document that says what problems, if any, you had while completing this assignment. If you had none, that's ok. If you had difficulties, please state them in no more than 1 paragraph. The TA will compile the responses. (4 points)

Installing IntelliJ where it was working properly and knowing where to put my Antlr file and how to generate the files was an issue that I spent a couple of days researching on. I then resolved it after inquiring with the professor as to what I should do as I had issues generating the files as well as the “defaults” were not specified in the instruction as to what version of JDK should be downloaded. The instructions were detailed; however, it was on Linux where most students had PC or MAC OS which looked very different than Linux. Other issues I had were understanding what the assignment was asking for, which needed further clarification and how to submit. Cleaning the binaries was also another issue. As I did not know how to clean binaries and what exactly they are. I followed the professor’s instruction to go to ChatGPT and ask, “How to clean binaries on IntelliJ Antlr” and follow what I could understand from the instructions.