Course Project – CS3520

Points: 60
Due Date: Friday 12/18/2020, 11:59 pm

For the course project, you will create a parser and a pretty-printing walker for the DOT language with an industrial-strength tool, ANTLR (ANother Tool for Language Recognition). ANTLR [1] is a powerful parser generator for reading, processing, executing, or translating structured text or binary files.

Part I: Get started with ANTLR4 and DOT. Please refer to [2] for installing the tool and familiarize yourself with it by trying the given example in the tutorial. More information about Dot can be found at [4].

Part II: Build a parser using ANTLR4 that prints each recognized rule by modifying the given grammar. The output should be the same as that of program4, except the error messages. The parser generated by ANTLR4 is able to detect all syntax errors that do not conform to the grammar. The ANTLR grammar that defines DOT will be provided. Please note that the ANTLR4 grammar uses a different set of meta-symbols than the EBNF grammar we learned. In the ANTLR4 grammar, both EBNF grammar and the regular expression are using the same meta-symbols, such as “?” indicates optional, “*” indicates 0 or more repetitions, and “+” indicates 1 or more.

Part III: Build a pretty-printing walker for the parser generated in part II, which can recognize any DOT code and output the same code in a formatted way. For example, given the following input file as digraph trees {subgraph t {0 -> "1" [label = "A"];0 -> "2" [label = "B"];}} The **walker** should be able to print out the code in a structured thus more readable way as

```
digraph trees {
  subgraph t {
    0 -> "1" [label = "A"];
    0 -> "2" [label = "B"];
  }
}
```

The input to ANTLR4 is the DOT.g4 file. ANTLR4 can automatically generate a parser for DOT based on the grammar in DOT.g4. You need to first figure out how ANTLR4 works, and then start to work on the two tasks. You can finish the two tasks by completing the listener java file generated by ANTLR4. An example of using the parser listener can be found at [3].

Submission:

1) The Java files you need to submit include DOTLexer.java, DOTParser.java, DOTListener.java, DOTBaseListener.java, DOTPrettyPrintListener.java, and Main.java. Among all the files, Main.java is provided by the instructor and DOTLexer.java, DOTParser.java, DOTListener.java, DOTBaseListener.java will be generated by ANTLR4 automatically. You need to modify DOTBaseListener.java for task II and create class DOTPrettyPrintListener that inherits class DOTBaseListener for task III.
2) A simple report explaining your solution with some screenshots showing the output of your code.
3) Submit the Java files and the report to a dropbox in Canvas.

Additional notes:

1. This project can be done individually or in a group of two. If you work in a group, make sure workload is divided equally.
2. Design a set of test cases for the parser and test your work before your submission. You can use the test cases provided for program4.
3. Make comments to your code whenever necessary.
4. You will lose points if your final submission is after the due time:
   -5 if your final submission is by 11:59 PM, December 18
   -10 if your final submission is by 11:59 PM, December 19
   -15 if your final submission is by 11:59 PM, December 20
   Completion credit after 11:59 PM, December 20
5. For all programs and course project in this class, you must turn in a running solution in order to pass the course.

Useful links:

[1]. Antlr4: http://www.antlr.org/
[2]. Get started with Antlr4: https://github.com/antlr/antlr4/blob/4.9/doc/getting-started.md
[3]. Antlr4 parser listener: http://theendian.com/blog/antlr-4-lexer-parser-and-listener-with-example-grammar/
[4]. The Dot language: https://graphviz.gitlab.io/_pages/doc/info/lang.html.
[5]. Antlr Development tools: https://www.antlr.org/tools.html and https://github.com/jknack/antlr4ide
[6]. Antlr API https://www.antlr.org/api/Java/index.html

Sample Input and Output:

```
digraph trees {rankdir=LR;subgraph t {0 -> "1" [label = "A"];0 -> "2" [label
= "B"];}SUBGRAPH u {Animal -> Cat [label = "feline", shape="record"];Animal ->
Dog1 [label = "canine"];}}
```

Sample Output:

```
Start recognizing a digraph
Start recognizing a cluster
Start recognizing a property
Finish recognizing a property
Start recognizing a subgraph
Start recognizing a cluster
Start recognizing an edge statement
Start recognizing a property
Finish recognizing a property
Finish recognizing an edge statement
Start recognizing an edge statement
Start recognizing a property
Finish recognizing a property
Finish recognizing an edge statement
Finish recognizing a cluster
Finish recognizing a subgraph
Start recognizing a subgraph
Start recognizing a cluster
Start recognizing an edge statement
Start recognizing a property
Finish recognizing a property
Start recognizing a property
Finish recognizing a property
Finish recognizing an edge statement
Start recognizing an edge statement
Start recognizing a property
Finish recognizing a property
Finish recognizing an edge statement
Finish recognizing a cluster
Finish recognizing a subgraph
Finish recognizing a cluster
Finish recognizing a digraph

Pretty-printed code:

digraph trees {
  rankdir=LR;
  subgraph t {
    0 -> "1" [label = "A"];
    0 -> "2" [label = "B"];
  }
  SUBGRAPH u {
    Animal -> Cat [label = "feline", shape = "record"];
    Animal -> Dog1 [label = "canine"];
  }
}
```