CS4100/5100 COMPILER DESIGN PROJECT
PART 3: SYNTAX ANALYZER- Part A
Fall 2021

The third step of the compiler project is the development of a recursive descent Parser (Syntax Analyzer) using the GetNextToken in Lexical Analyzer developed in Part 2. If the instructions for the lexical analyzer were followed completely, the parser task will be very straight-forward; otherwise, there may be difficulties which can only be overcome by fixing Part 1 to work properly. Part 3 of the project is broken into two smaller pieces, Part A and Part B, which will be submitted separately..

1) **Verify functionality, and fix any errors in your Lexical Analyzer.** If there were any tokens not recognized, or you generated wrong token codes, or it got hung up somewhere, fix every error before starting the parser.

2) Use the provided **partial** CFG-A in modified BNF as the basis for your Part A parser, and translate the CFG into a recursive descent parser, as discussed in class. There will be a parameterless method which returns an **int** for EACH non-terminal in your CFG. **In anticipation of the code generation in Part 4, make each non-terminal function return an integer value, even though the return values will be ignored for the syntax task.** Follow the examples given in class, they are provided to demonstrate the simplest way to implement the parser. **Initially test your program on very small test data. Implement incrementally, a piece at a time, and get the bugs out along the way before adding new features, as presented in class.**

3) For Part A, code for main.java and an initial syntactic.java class will be provided. The interface for class syntactic.java includes:

syntactic(String filename, boolean traceOn)

This is the constructor for the syntax analyzer. It calls the *Lexical* constructor like main.java for *Lexical* to get the lexical analyzer initialized.

int parse()

This is the only public method of syntactic. It calls GetNextToken to get the first token from the file, then calls the non-terminal method *Program()* in order to start the parsing process.

The code for these methods, and the first few non-terminal methods and utility functions, will be posted in the sample syntactic.java file on Canvas.

SPECIFICATIONS:

1) SWITCHABLE TEST MODE ENTRY/EXIT MESSAGES. Controlled by the syntactic constructor's *traceOn* parameter, each non-terminal procedure in your syntax analyzer must print its name at entry and exit, as 'ENTERING xxx' and 'EXITING xxx'. If it operates correctly, the list of entering names will be a pre-order traversal of the parse tree for the given input. All these prints must be turned on or off by setting *traceOn. A* single method **trace(String *name,* boolean *entering*)** is provided in the sample code.

2) GRAMMAR CONTAINED IN INITIAL COMMENTS OF SYNTAX CLASS. Include the CFG being implemented in the comments at the top of the SYNTAX class code.

3) GENERATE INTELLIGIBLE DIAGNOSTICS. In addition to the lexical analyzer's error checking for long identifiers, and unexpected end of file within a comment, your Part A syntax analyzer must generate meaningful diagnostics for the following errors, ***directly under the source line which caused the error***, including the line number:

1) Any syntax error detectable from the grammar (*xxx expected, but yyy found***)**. A sample method for this is included in the sample code.

2) 'Undefined character' token codes returned from GetNextToken. Lexical should take care of this, as well as the errors below.

3) All GetNextToken-detected errors from Lexical

4) RECOVER FROM ERRORS. For Part A, there will be **no Error Recovery**, other than halting; parsing should stop when a global *ERROR* flag is set. NOTE that this requires popping the recursion stack to stop processing by making ALL non-terminal procedures check a global error status at entry, before they continue, so they can exit immediately. This capability will be expanded in Part B. ***See the example code for details.***

5) TURN IN RUNS USING THE PROVIDED GOOD AND BAD DATA. The 'official' error-free and error-laden test files will be published on Canvas. Students should generate their own test data as part of the program development process. See the grading sheet for details.

6) TURN IN PROGRAM CODE LISTING. The program source code must be submitted in Canvas as java files, nicely structured and demonstrating good software engineering practice. **See the assignment and/or grading sheet for features which will be evaluated!**