CS4100/5100 COMPILER DESIGN PROJECT
PART 3: SYNTAX ANALYZER- Part A
Fall 2020

The third step of the compiler project is the development of a recursive descent Parser (Syntax Analyzer) using the GetNextToken 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) **Fix any errors in your Lexical Analyzer.** If there were any tokens not recognized, or the line reading and echo did not work, or it got hung up somewhere, fix every error before starting the parser.

2) Use the provided **partial** CFG 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 procedure/method/function 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, which will be ignored for the time being.** Follow the examples given in class, they are provided to demonstrate the simplest way to implement the parser. NOTE: If you have taken the class before, THROW AWAY any old code you have and start from scratch. REUSING BAD OLD CODE WILL TAKE LONGER THAN GENERATING NEW, CORRECT CODE THE FIRST TIME, GUARANTEED. **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.**

SPECIFICATIONS:

1) SWITCHABLE TEST MODE ENTRY/EXIT MESSAGES. 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 syntax tree for the given input. All these prints must be turned on or off by setting a single boolean variable to **true** to activate the 'trace mode'. HINT: Create a single method **DEBUG(boolean *entering*, String *name*)** that prints 'Entering *name*' if *entering* is true, else 'Exiting *name*'.

2) GRAMMAR IN INITIAL COMMENTS OF MAIN. Include the CFG in the comments at the top of the main program module.

3) USE THE OUTPUT TO DISPLAY THE SYNTAX TREE. Using the captured text of the console output, illustrate the syntax tree of the input program, using the 'Entering xxx' printlns, allowing the student (and instructor) to check the correctness of the parser. See grading sheet for details on which test file to use for this. This will be discussed further in class.

4) 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***),**
2) 'Undefined character' token codes returned from GetNextToken
3) All GetNextToken-detected errors from Lexical

5) 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.

6) TURN IN RUNS USING THE PROVIDED GOOD AND BAD DATA. About 1 week prior to the due date, the 'official' error-free and error-laden test files will be placed on Canvas. Prior to that, students should generate their own test data as part of the program development process. See the grading sheet for details.

7) TURN IN PROGRAM CODE LISTING. The program source code must be submitted in Canvas as a PDF in neat form, nicely structured, using syntax highlighted printing and good software engineering practice. **See the grading sheet for features which will be evaluated!**