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

Part B of this third step of the compiler project builds on the recursive descent Parser from Part A to fill out the remaining language features.

.

1) **Fix any errors in your Part A.** It is assumed that by this point, the Lexical Analyzer has been perfected. Now, Part A must be working as specified, so that the new CFG rules can be added on top of the existing code.

2) Use the provided complete CFG for the Part B parser, and add the new features into the recursive descent parser. As before, there will be a parameterless function for EACH non-terminal in the CFG. **Again, make each non-terminal function return an integer value, which will be used later for code generation.** Follow the examples given in class, as 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.**

SPECIFICATIONS- *CHANGES TO PART-A ARE NOTED*:

1) SWITCHABLE TEST MODE ENTRY/EXIT MESSAGES. *Same as in Part A*, each non-terminal procedure in your syntax analyzer must optionally 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 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 **PART B CFG** in the comments at the top of the main program module. **FOR PART B,** turn in a word-processed, 1-paragraph overall description of the program as well.

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

1) **Any use of undeclared identifiers, or use of a label as a variable, or variable as a label** (warn user, continue parsing)
2) **Any syntax error detectable from the grammar (***xxx expected, but yyy found***)**,
3) **All GetNextToken-detected errors from Lexical**
4) **Any attempt to redeclare an identifier; the program header name cannot be reused as a label or variable; a label or variable cannot be redeclared as a variable or label, respectively.**

4) RECOVER FROM ERRORS. For Part B, error recovery must consist of flushing the entire current statement, and skipping tokens to start the analyzer at the beginning of the very next possible statement starting token. NOTE that this requires popping the recursion stack back to the correct place, by making non-terminal procedures set and also check a global error status before they continue on, so they can exit early. Finding the 'end' of a statement is more than just looking for a semi-colon; the code must look for the start of a new statement.

5) TURN IN RUNS USING 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 the course website. Prior to that, students should generate their own test data as part of the program development process. **See the grading sheet for details on the required run turn-ins.**

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