CS4100/5100 COMPILER DESIGN PROJECT
PART 3: SYNTAX ANALYZER- Part B
Fall 2022

Part B of this third step of the compiler project builds the rest of the recursive descent Parser from Part A to fill out the remaining language features.
.
1) **Fix any errors in your Part A.** 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 Part A recursive descent parser. This uses the same pattern as before, with 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 will still 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.

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

3) GENERATE ERROR MESSAGES. 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 syntax error detectable from the grammar (*xxx expected, but yyy found***)**,
2) All GetNextToken-detected errors from Lexical
3) Identifiers used before being declared. In the Identifier Declaration section of the input program, identifiers should be added to the symbol table as normal. After that section in the CFG is exited, a flag should be set (in SymbolTable class) so that any time an identifier not already present will trigger an **"Undeclared Identifier: xxx"** error, and add the identifier anyway. That way, the error will only appear once for any identifier that did not appear in the Declarations.

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. Follow the assignment instructions and rubric.

6) TURN IN PROGRAM CODE LISTING AND JAR. The program source code must be submitted in clean (no commented out sections of test code). **See the rubric**

Suggested implementation approach:

Implementing and testing each of the following one step at a time will likely provide the most efficient sequence for your full Syntactic class development.

1) Ensure that the <simple expression> and the non-terminals it calls are correct and work for various test cases.

2) Add the <block body> statement type, seeking a BEGIN/END block, as it is very easy to add to the Part A Assignment Statement.

3) Implement the <relexpression> [relational expression using >, <, >=, <=. =, <>] which depends upon <simple expression>

4) Implement the DOWHILE <relexpression> statement.

5) Implement the IF/ELSE statement.

6) Implement the WRITELN statement.

7) Implement the READLN statement.

8) Implement the REPEAT/UNTIL and then the FOR loops.