Programming Assignment #1 **(FINAL VERSION)**
Scanning and Parsing
Due Date: February 21, 5pm

This programming assignment has 2 components: a scanner (lexical analyzer) and a parser (syntax analyzer) for a simple calculator programming language. Much of the parser will be written for you, so the focus will be on scanning or lexical analysis. A first version of the parser will be provided and you will be asked to revise (or rewrite, if you wish) the program to accommodate an additional construct. You may use either Java or C to implement this programming assignment.

Consider the following sample program for this simple calculator language (SimpCalc):

```
// this program calculates the roots of the following quadratic equation:
// 5.5x^2 + 10x - 3
discriminant := 10**2 - (4*5.5*(-3));    b^2-4ac
IF discriminant >= 0:
   root1 := (-10 + SQRT(discriminant))/(2*5.5);   (-b+-sqrt(b^2-4ac))/2a
   root2 := (-10 - SQRT(discriminant))/(2*5.5);
   PRINT("roots are",root1,root2);
ELSE // discriminant is negative
   PRINT("no real roots");
ENDIF;
PRINT("end of program");
```

Recall that the goal of lexical analysis is to scan source code, filter out white spaces and comments, identify lexical errors, and most importantly, break up the code (which is a stream of characters) into *lexical tokens*, the most basic elements of a program. For SimpCalc, the possible tokens are as follows (most of which are present in the sample code given above):

1. Identifier	
2. Number	
3. String	
4. Assign	:=
5. Semicolon	;
6. Colon	:
7. Comma	,
8. LeftParen	(
9. RightParen	)
10. Plus	+
11. Minus	-
12. Multiply	*
13. Divide	/
14. Raise	**
15. LessThan	<
16. Equal	=
17. GreaterThan	>
18. LTEqual	<=
19. NotEqual	!=
20. GTEqual	>=
21. EndOfFile	

Keywords are separate tokens, although they are initially recognized as identifiers. The possible keywords are: PRINT, IF, ELSE, ENDIF, SQRT, AND, OR, NOT (case sensitive). This results in 29 different token ids.

SimpCalc supports three kinds of statements: assignment statements, print statements and if statements. If statements may be nested and may or may not contain else blocks. Refer to the following specific details about SimpCalc:

1. Comments begin with the character sequence // and all characters up to the end of the text line should be ignored.
2. Identifiers start with a letter or underscore followed by any number of letters, digits and underscores. Letters may be upper or lower case.
3. Identifiers and keywords are case sensitive.
4. Numbers (numeric literals) are either whole numbers (any sequence of one or more digits) or floating-point numbers that follow simple decimal number notation (one or more digits, followed by a dot, followed by one or more digits) or exponential notation (a whole number or decimal number followed by an e or E, followed by an optional +/- sign, followed by a whole number).
5. Strings are delimited by double quotes (″) and allow any printable keyboard character within these quotes. Strings may not span multiple lines.
6. Assignment statements take the form **Identifier := <Expression>;**
7. An expression supports the arithmetic operators +, -, *, /, ** (exponent), and unary – (negation), while allowing parentheses for grouping and the SQRT function call. Operator precedence is as follows: parentheses, negation, exponent, multiply/divide, add/subtract. Left-to-right associativity applies at each precedence level. Basic operands are numbers and identifiers.
8. Print statements take the form **PRINT(<Argument>, …);** allowing one or more arguments, where each argument is either an expression or a string.
9. If statements take one of two forms:
   IF <Condition>: <Block> ENDIF; or,
   IF <Condition>: <Block> ELSE <Block> ENDIF;
   where a condition is (*for now*), **<Expression> <RelationalOperator> <Expression>** and a block is any sequence of statements.
   Relational operators are any of the six operators listed above (tokens 15 – 20).

Deliverables:

1. A DFA that recognizes all possible tokens of SimpCalc (to be submitted as a separate homework assignment); a state diagram of your DFA is due on 9 February 2018, 5pm, through Canvas.
2. A scanner module that recognizes SimpCalc tokens and passes on these tokens to a separate module, one token at a time. The module should support a getToken method (or gettoken function in C) and a getLineNumber method (or getlinenumber function in C). Sources for tester programs and support files will be provided in both programming platforms. This assignment is also an exercise in separate compilation, so in general, you should not modify these files.
3. A revised parser module. A recursive descent parser module that supports the program constructs detailed above will be provided to you (in both Java and C). You are expected to rewrite this module (you may also subclass the original module, if you are using Java), to support slightly more complex conditions for if statements. Complex conditions are either AND-ed relational expressions or OR-ed relational expressions (For simplicity, I have decided to render the NOT token useless). This requires adding a methods/functions to the recursive descent parser.

Deliverables 2 and 3 are due on 21 February, 5pm, through Canvas. Deliverable 2 is worth 70% of this assignment, while deliverable 3 is worth 30% of this assignment.

The following is an inventory of the source files that will be provided to you as well as the modules you will need to write:

Description	Java	C
Token Data Structure (provided)	Token.java: contains a class definition that encapsulates a token integer id and lexeme (String); the class also includes constants for the different valid token ids and an array of token names for printing	token.h: contains a C struct definition that represents a token record (int id; and char[] lexeme); the header file also includes constants for the different token ids. token.c: defines an array of token names for printing
Scanner Module (you will need to create this module for this assignment)	ScannerModule.java: uses the Token class and defines: a constructor (with a filename argument) that initializes the module, getToken(), and getLineNumber()	scannermodule.c, scannermodule.h: uses the token structure and defines: initscannermodule(filename), gettoken(), and getlinenum(). scannermodule.h will be provided
Scanner Tester (provided)	ScannerTester.java: contains a main method that instantiates a ScannerModule object and repeatedly calls getToken() on that object. A filename of the source file to be scanned needs to be specified upon instantiation.	scannertester.c: contains a main program that: - invokes the initscannermodule() function with a valid source filename as its argument - repeatedly calls gettoken()
Parser Module (provided)	ParserModule.java: performs recursive descent parsing through a parse() method called on a ParseModule object; it uses the ScannerModule class and calls both getToken() and getLineNumber() of that class	parsermodule.c, parsermodule.h: contains definitions for - initparsermodule(filename) - parse() code invokes the scannermodule functions initscannermodule() gettoken(), and getlinenum()
Parser Tester (provided)	ParserTester.java: contains a main method that instantiates a ParserModule object, and then calls parse() on the object	parsertester.c: contains a main program that invokes initparsermodule() and then parse()
Revised Parser Module (you will need to create this module for this assignment)	NewParserModule.java: cut and paste ParserModule.java and then revise or extend the ParserModule class to support the complex conditions feature.	newparsermodule.c: cut and paste parsermodule.c and then revise to support the complex conditions feature; you do not need to revise parsermodule.h
Revised Parser Tester (not provided)	NewParserTester.java: you will need to create a new class that instead uses NewParserModule; a very minor revision from ParserTester.java	Not applicable. parsertester.c will work accordingly as long as you link the correct object files
Makefile (provided but may require revisions)	Not applicable.	You may need to revise this file for your submitted code to compile correctly.

Sample Files and Expected Output

Sample files will be posted on Canvas. Expected output for the Scanner Tester will be a list of recognized tokens together with their corresponding lexemes. The text below shows the first few lines expected for the sample program shown at the beginning of this document:

```
Identifier  discriminant
Assign      :=
Number      10
Raise       **
Number      2
Minus       -
LeftParen   (
Number      4
Multiply    *
Number      5.5
Multiply    *
LeftParen   (
Minus       -
Number      3
RightParen  )
RightParen  )
Semicolon   ;
If          IF
Identifier  discriminant
GTEqual     >=
Number      0
Colon       :
Identifier  root1
Assign      :=
LeftParen   (
Minus       -
```

The Parser Tester will output indications that valid statements were recognized, and whether the source code is a valid SimpCalc program. For the sample program, the output is as follows:

```
Assignment Statement Recognized
If Statement Begins
Assignment Statement Recognized
Assignment Statement Recognized
Print Statement Recognized
Print Statement Recognized
If Statement Ends
Print Statement Recognized
sample1.txt is a valid SimpCalc program
```

Lexical errors (only 3 or 4 possible error types) and parse errors shall be displayed when they occur, interspersed with the output for lexical errors. For parse errors, an error message is printed on the first error and then the program aborts. Refer to the sample input and output files for examples. Since the scanner module is where file operations are carried out, file related errors occur (e.g., when a file does not exist) from that module. In this case, simply print an error message and abort the program.

Document your code sufficiently, consistent with the usual guidelines for programs in your past CS classes. Submit a zip file of all program sources and upload through Canvas the due date.