Programming Languages
Programming Project I

This programming project concerns the material covered in the first three weeks of the course, i.e. material on *abstract machines*, *description of programming languages*, and *lexical analysis and parsing*. You can choose one or more of the following object-oriented languages for implementing your programs: C++, Java or Python (for these languages, either a compiler exists for most operating systems or an abstract machine/interpreter).

The programming task

Your programming task is twofold:

1) Write a compiler that compiles a string from the language *L* to simple stack-based intermediate code (called *S*) for an abstract machine (implemented in part 2). A string from the language *L* consists of an arithmetic expression in *infix* notation[1] (ending in a newline).

   The context-free grammar *G* for *L* is:

   Expr->Term | Term**+**Expr
   Term->Factor | Factor*****Term
   Factor->**int** | **(**Expr**)**

   Here, *Expr* is the starting symbol, *Expr*, *Term*, and *Factor* are non-terminals and **+**, *****, **int**, **(**, and **)** are terminals (tokens). **int** is a token representing an integer.

   S, our stack-based intermediate code, consists of the following four commands:

   ```
   PUSH op     // pushes the operand op onto the stack
   ADD         // pops the two top elements from the stack, adds their values
               // and pushes the result back onto the stack
   MULT        // pops the two top elements from the stack, multiplies their
               // values and pushes the result back onto the stack
   PRINT       // prints the value currently on top of the stack
   ```

2) Implement the abstract machine for *S*, i.e. a machine which allows the execution of code written in S.

1 http://en.wikipedia.org/wiki/Infix_notation

The implementation

You must use the following guidelines for the implementation (steps 1-4 correspond to the compiler, step 5 to the interpreter).

1. Write the class **Token**, which contains both a lexeme and a token code:

   ```
   String lexeme;
   TokenCode tCode;

   where TokenCode is:
   enum TokenCode { INT, PLUS, MULT, LPAREN, RPAREN, ERROR,
   END}
   ```

   (See an explanation for ERROR and END in 2)

2. Implement the class **Lexer,** a lexical analyzer. It only needs a single public method, *nextToken()*, which scans the **standard input** (stdin), looking for patterns that match one of the tokens from 1).

   Note that lexemes which are operators (PLUS, MULT) contain only one letter whereas the lexemes for the token INT can contain more than one digit. The lexical analyzer returns a token with TokenCode=**ERROR** if some illegal lexeme is found and TokenCode=**END** denoting the end of an expression.

3. Implement the class **Parser**, the syntax analyzer (parser). This should be a **top-down recursive-descent parser** for the grammar *G* above, i.e. a program which recognizes valid infix expressions. The output of the parser is the stack-based intermediate code *S*, corresponding to the given expression, written to **standard output** (stdout). If the expression is not in the language (or if an ERROR token is returned by the Lexer) then the parser should output the error message "Syntax error!" (at the point when the error is recognized) and immediately quit.

   The parser should have at least two (private) member variables, one of type **Lexer**, the other of type **Token** (for the current token). It should only have a single public method, *parse()*, for initiating the parse – other methods are private.

4. Write a main class, **Compiler**, which only does the following:

   ```
   Lexer myLexer = new Lexer();
   Parser myParser = new Parser(myLexer);
   myParser.parse();
   ```

5. Implement the (main) class, **Interpreter**. This class reads *S*, the stack-based intermediate code, from **standard input** (stdin), interprets it and executes it "on-the-fly". The result is written to standard output. The main program of the

Interpreter should correspond to the general function of an interpreter, i.e. the *fetch-decode-execute* cycle.

If the Interpreter encounters an invalid operator, or if there are not sufficiently many arguments for an operator on the stack, then it should write out the error message: "Error for operator: nameOfOperator" (where nameOfOperator is the operator in question) and immediately quit.

Running/Testing

When running the Compiler, the user inputs an infix expression and the corresponding intermediate code is written to stdout. For example (assuming a Java implementation):

```
java Compiler
2*(13+7)
PUSH 2
PUSH 13
PUSH 7
ADD
MULT
PRINT
```

Here, the user has input the infix expression "2*(13+7)" and the Compiler has written the above intermediate code to stdout.

The output of the Compiler can, of course, be redirected to a file, and, subsequently, interpreted by the Interpreter:

```
java Compiler > expr.code
2*(13+7)

java Interpreter < expr.code
40
```

You also need to make sure that it is possible to run your program in the following manner:

```
java Compiler | java Interpreter
```

The Compiler reads from standard input and writes to standard output. Then, the Interpreter reads (from standard input) the output generated by the Compiler and finally writes the result to standard output. You should make a script which runs the above sequence, such that a user, who wants to use your programs as a calculator, can issue a single command like *calc*:

```
calc
2*(13+7)
40
```

Another example:

```
calc
3*(12+7)*2+8
122
```

The case of invalid expressions:

```
java Compiler
2 +
PUSH 2
Syntax error!

calc
2 +
Error for operator: Syntax
```

Note that in both of the examples above, the text shown after the user input ("2 +") is the output generated by the compiler and the interpreter, respectively.

How to submit the project

- In Mooshak: Return all your source files (see below) in a single .zip file (the shell script is not needed).
- In MySchool: Return all your source files (see below), executable files, and a shell script (like *calc*) which runs the Compiler and the Interpreter in the sequence shown above.

  The source files are:

  1) **Java implementation:**
     Token.java, Lexer.java, Parser.java, Compiler.java (main program), Interpreter.java (main program)

  2) **Python implementation:**
     Token.py, Lexer.py, Parser.py, Compiler.py (main program), Interpreter.py (main program)

  3) **C++ implementation**
     Token.h, Lexer.h, Lexer.cpp, Parser.h, Parser.cpp, Compiler.cpp (main program), Interpreter.cpp (main program)