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/virtual machine).

The programming task

Your programming task is twofold:

1) Write a compiler that compiles a string from the small programming 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 a list of statements, where each statement is either an assignment statement or a print statement.

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

   Statements -> Statement **;** Statements | **end**
   Statement -> **id** = Expr | **print id**
   Expr- > Term | Term + Expr | Term – Expr
   Term -> Factor | Factor ***** Term
   Factor -> **int** | **id** | **(** Expr **)**

   The non-terminals are: *Statements* (starting symbol), *Statement*, *Expr*, *Term*, and *Factor*. Terminals (tokens) are: **; end id = print + - * int ( )**

   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
SUB         // pops the two top elements from the stack, subtracts the first
            // value retrieved from the second value,
            // 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
ASSIGN      // pops the two top elements from the stack, assigns the first
            // element (a value) to the second element (a variable)
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.

The implementation

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

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

```
String lexeme;
TokenCode tCode;
```

where TokenCode is:

```
enum TokenCode { ID, ASSIGN, SEMICOL, INT, PLUS, MINUS,
MULT, LPAREN, RPAREN, PRINT, END, ERROR}
(See an explanation for ERROR in 2)
```

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

Note that the lexemes corresponding to the tokens PLUS, MINUS, MULT, LPAREN, RPAREN, ASSIGN, SEMICOL contain only a single letter. The patterns for the lexemes for the other tokens are:

INT = [0-9]+	*(non-empty, finite sequences of digits)*
ID = [A-Za-z]+	*(non-empty, finite sequences of upper- and lower-case letters)*
END = end	
PRINT = print	

The lexical analyzer returns a token with TokenCode=**ERROR** if some illegal lexeme is found.

3. Implement the class **Parser**, the syntax analyzer (parser). This should be a **top-down recursive-descent parser** for the grammar *G* above. The output of the parser is the stack-based intermediate code *S*, corresponding to the given program, 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.

   Use a Stack class to process the intermediate code and use a Map class to store the values of identifiers.

Running/Testing

When running the Compiler, the input is a program in the language L. The output is the corresponding intermediate code, written to stdout. For example, let us assume the following program written in L in the file program.l

```
var = 3;
b = 4 * (7-var);
print b;
end
```

Then (given a Java implementation), the following command:

```
java Compiler < program.l
```

writes the following intermediate code to standard output:

```
PUSH var
PUSH 3
ASSIGN
PUSH b
PUSH 4
PUSH 7
PUSH var
SUB
MULT
ASSIGN
PUSH b
PRINT
```

Note that when an identifier is used in an expression (on the right hand side of an assignment statement), the value of the identifier should be used.
The output of the Compiler can, of course, be redirected to a file, and, subsequently, interpreted by the Interpreter:

```
java Compiler < program.l > program.code

java Interpreter < program.code

16
```

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

```
java Compiler < program.1 | 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 programmer is able to run a program with a single command like *execute*:

```
execute program.l
16
```

Example cases of invalid program:

1) Assume the following program.l:

```
var = 3 + ;
print var;
end

java Compiler < program.l
PUSH var
PUSH 3
Syntax error!
```

2) Assume the following program.l:

```
var = 3 ! ;
print var;
end

java Compiler < program.l
PUSH var
PUSH 3
Syntax error!
```

How to submit the project

- In Mooshak: Submit your Compiler source files (see below) in a single .zip file and the Interpreter separately (the shell script is not needed).
- In MySchool: Return all your source files (see below), executable files, and a shell script (like *execute*) 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)