**Project 1: Calculator**

Write an interactive desk calculator that evaluates arithmetic expressions on integers. The grammar for expressions is defined as follows:

<expression> ::= <term>

| <expression> '+' <term>

| <expression> '-' <term>

<term> ::= <factor>

| <term> '\*' <factor>

| <term> '/' <factor>

| <term> '%' <factor>

<factor> ::= <primary>

| <primary> '^' <factor>

<primary> ::= <integer>

| '(' <expression> ')'

<integer> ::= <digit> | <digit> <integer>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The / and % are, respectively, the integral division (returns the quotient) and modulus operators.

The calculator should prompt for an expression, read in a line from the standard input stream, tokenize, parse, and evaluate the expression on that line, and write the result to the standard output stream. The line of input is expected to contain one valid expression. The calculator should inform the user if the input is not recognized as a valid expression. Note that the evaluator should allow any number of whitespace characters (tabs or spaces) anywhere within an expression. The single character 'q' is interpreted as the command to exit the program.

calc> (2^3) / 4

2

calc> 1 + HJK#$#

\*\* unrecognized expression!

calc> 1+ (((1)))

2

calc> q

Bye!

What to turn in:

1. The file calculator.cpp, that contains the main function.
2. All other .cpp and .hpp files that are necessary for building the application—you may or may not have some of these.
3. A Makefile.

The first target of the Makefile should build the application, and the executable that results should be named calc. (Or calc.exe if you insist).

1. A file named report.pdf that describes how to use the calculator, and what are its limitations, if any.

Please use C++ as the implementation language. You may use whatever C++ compiler you desire, but please write portable code; do not use any operation system, compiler, or library distribution specific language or library features. The instructor or TA may test your program using any standards conforming C++ compiler he desires.

**About outside help**

The assignment is to be done individually.

* This does not mean that you should not discuss your design with others. By all means, do talk to your classmates about ways of implementing a parser and an evaluator, ways to structure the code, a tricky piece of code that you or your classmate is struggling with to get working, etc. You can of course also use the literature to find ideas for how to solve this kind of a programming task, but remember to cite your sources.
* This does mean, however, that will write your program wholly by yourself, without copying parts of it from codes that others have written.

**Grading**

The assignment will be partially graded by whether your code works or not, but even more importantly, by how you structure your code, comment it, and lay it out. Please consult and try to follow the guidance given during the lectures and the course text.

**Hints**

The grammar given above is *left-recursive*, which, if implemented as a *recursive descent parser*, will cause an infinite loop. It may be easier to base your parser on the following transformed grammar that does not contain left-recursive rules.

<expression> ::= <term> <expression'>

<expression'> ::= '+' <expression>

| '-' <expression>

| empty

<term> ::= <factor> <term'>

<term'> ::= '\*' <term>

| '/' <term>

| '%' <term>

| empty

<factor> ::= <primary>

| <primary> '^' <factor>

<primary> ::= <integer>

| '(' <expression> ')'

<integer> ::= <digit> | <digit> <integer>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Note that while avoiding infinite loops, a recursive descent parser for this grammar has another pitfall: operators are right-associative. Take this into consideration in the implementations of the parsers for the <expression> and <term> nonterminals, and make sure that the AST-nodes are constructed from left to right, so that the +, -, \*, /, and % operators are left associative.