For your assignment, you can use the right-to-left evaluator code as a starting point.

Your evaluator class should have the following features:

1) permit spaces between numbers and operators.

OK: 3.14 + 75 or (3.14+75) or 3.14 +75 or 3.14+ 75 etc.

Not OK: 3 . 14 or 3 . 1 4 etc.

You can decide on your own rules for spaces, but include these in the comments to the "user".

2) a negative number should have a minus sign followed *immediately* by a digit or a decimal point; further, negative numbers should usually be surrounded by parentheses.

OK: 5 + (-13) or 5 -(-13) or 3+ (-.123) etc.

If you want to permit -7 + 16, the code will need to be more complicated, so tell the "user" what is permitted.

Not OK: 5 + -13 or 5 - -13 or 5 + (- 13) or - .7 + 16 etc.

Indicating a negative number with (-x) is more natural than using _x, but it causes difficulties because we also use parentheses to enclose subexpressions, e.g. 3*(4+5). Without parentheses, this would compute (left to right) as17 instead of 27. But the dual role of parentheses requires extra attention.

3) Exponentiation should be permitted, with *highest* precedence.

Use "a ^ b" to mean a to the power b.

4) Evaluation should be left to right. This is the main change.

A) The formNum() method has to be rewritten so that it scans from left to right when it encounters the *leftmost* digit of a number or a *leading* decimal point. Of course, the method of building up the double value from the successive digits has to be adjusted since you are scanning from left to right- try some examples to discover the correct approach.

B) The evaluator() method also has to be adjusted to scan from left to right.

There are several changes:

i) When a left parenthesis is encountered it is *always* pushed onto stack A unless it signals a negative number (as in 5+ (-13.4) +2). In that case, formNum() should be called to compute the negative number and push it as a double onto B, and the corresponding right parenthesis should be skipped over.

ii) When a right parenthesis is encountered (but not the right end of a negative number) it triggers a call to evalDown() to evaluate the operations in a parenthesized subexpression until the corresponding left parenthesis is reached. The left parenthesis is also popped from A.

iii) when the current token operator is equal in precedence to the operator at the top of the stack, do *not* push the token. Instead, perform the operation at the top of the A stack using eval(), and then compare the current token operator to the new top of the A stack.

Example. Suppose we have 3.0 / 2.0 * 4.0 .

We push 3 onto B , / onto A, and 2 onto B.

If we (incorrectly) push * onto A (it has equal precedence with /), and then 4 onto B, we will compute 2 * 4 first and then 3/8.. Instead, we should compute 3./2 first and push 1.5, and later compute 1.5 * 4 = 6.0. (In right to left scanning, we always pushed the new operator if it had equal precedence with the top operator, now we do *not* push it.)

iv) push a new token operator or left parenthesis (except one indicating a negative number) on top of a left parenthesis at the top of stack A.

.