Expression Evaluation

In this assignment you will implement a program to evaluate an arithmetic expression
Worth 10% of course grade

Expression
Here are some sample expressions of the kind your program will evaluate:

```
3
Xyz
3-4*5
a-(b+A[B[2]])*d+3
A[2*(a+b)]
(varx + vary*varz[(vara+varb[(a+b)*33])])/55
```

The expressions will be restricted to the following
components: • Integer constants
• Simple (non-array) variables with integer values
• Arrays of integers, indexed with a constant or a subexpression • Addition, subtraction, multiplication, and division operators, i.e. '+','-','*','/' • Parenthesized subexpressions
Note the following:
- Subexpressions (including indexes into arrays between '[' and ']') may be nested to any level
- Multiplication and division have higher precedence than addition and subtraction
- Variable names (either simple variables or arrays) will be made up of one or more letters ONLY (nothing but letters a-z and A-Z), are case sensitive (Xyz is different from xyz) and will be unique.
- Integer constants may have multiple digits
- There may any number of spaces or tabs between any pair of tokens in the expression. Tokens are variable names, constants, parentheses, square brackets, and operators.

Implementation and Grading
You will see a project called *Expression Evaluation* with the following classes in package *app*:
- *Variable*
  This class represents a simple variable with a single value. Your implementation will create one *Variable* object for every simple variable in

the expression (even if there are multiple occurrences of the same variable). You don't have to implement anything in this class, so do not make any changes to it.

- *Array*
  This class represents an array of integer values. Your implementation will create one *Array* object for every array in the expression (even if there are multiple occurrences of the same array).
  You don't have to implement anything in this class, so do not make any changes to it.
- *Expression* This class consists of methods for various steps of the evaluation process:
  - 20 pts: *makeVariableLists* - This method populates the *vars* and *arrays* lists with *Variable* and *Array* objects, respectively, for the simple variable and arrays that appear in the expression.
    You will fill in the implementation of this method. Make sure to read the comments above the method header to get more details.
  - *loadVariableValues* - This method reads values for all simple variables and arrays arrays from a file, into the *Variable* and *Array* objects stored in the *vars* and *arrays* array lists. This method is already implemented, do not make any changes.
  - 60 pts: *evaluate* - This method evaluates the expression.
    You will fill in the implementation of this method.
- *Evaluator*, the application driver, which calls methods in *Expression*. You may use this to test your implementation. There are two sample test files etest1.txt and etest2.txt, appearing directly under the project folder.

You are also given the following class in package structures:

- *Stack*, to be (optionally) used in the evaluation process

Do **NOT** add any other classes. In particular, if you wish to use stacks in your evaluation implementation do **NOT** use your own stack class, **ONLY** use the one you are given. The reason is, we will be using this same *Stack* class when we test your implementation.

Notes on tokenizing the expression

You will need to separate out ("tokenize") the components of the expression in *makeVariableLists* and *evaluate*. Tokens include operands (variables and constants), operators ('+','-','*','/'), parentheses and square brackets. It may be helpful (but you are not required) to use *java.util.StringTokenizer* to tokenize

the expression. The *delims* field in the *Expression* class may be used in the tokenizing process.

The documentation of the StringTokenizer class says this:
StringTokenizer is a legacy class that is retained for compatibility reasons although its use is discouraged in new code. It is recommended that anyone seeking this functionality use the *split* method of *String* or the *java.util.regex* package instead.

For the purpose of this assignment, you may use *StringTokenizer* without issue. Alternatively, you may use the *split* method of the *String* class, or the Pattern and Matcher classes in the package *java.util.regex*.
Or, you may simply parse the expression by scanning it a character at a time.

Rules of implementation

- You may NOT modify any of the files except *Expression.java* in ANY way.
- You may NOT make ANY modifications to *Expression.java* EXCEPT:
  - Write in the bodies of the methods you are asked to implement,
  - Add private helper methods as needed (including the recursive *evaluate* method discussed below.)

Note that the *java.io.*, java.util.*,* and *java.util.regex.** import statements at the top of the file allow for using ANY class in *java.io, java.util,* and *java.util.regex* without additional specification or qualification.

Guidelines and recommendations for implementing *evaluate*

- Recursion (optional) for sub-expressions in parentheses

While recursion is optional for this assignment, using it to evaluate subexpressions will make it a LOT easier to write working code. (This is a great opportunity to learn how to use recursion in a realistic situation!!)
There are a couple of coding options if you want to use recursion:

- One option is to make the public evaluate method itself recursive.

So, for instance, if the main expression is

```
a-(b+A[B[2]])*d+3
01234567891111111 (these are the positions of the characters in the expression)
0123456
```

then, to recursively evaluate the subexpression in parentheses, you may call

the recursive `evaluate` method like this:

```
float res = evaluate(expr.substring(3, 11), vars, arrays);
```

- Another option is to write a separate private recursive *evaluate* method, with two indexes that mark the start and end of the subexpression in the main expression. Then, for the above example, you can call the recursive method like this:

```
float res = evaluate(expr, 3, 11, vars, arrays);
```

(The *expr* parameter is the original expression for every call.)
And, to start with, you may call the recursive *evaluate* method from the public *evaluate* method like this:

```
return evaluate(expr, 0, expr.length() - 1, vars, arrays);
```

which is the entire expression.
You will need to use this second option if you want to include other parameters in your recursive *evaluate*.
In either case, the auto grader will call the public *evaluate* method.

- Recursion (optional) for array index expressions (within '[and ']'), using the same approach as above.
- A stack may be used to store the values of operands as well as the results from evaluating subexpressions - see next point.
- Since * and / have precedence over + and -, it would help to store operators in another stack. (Think of how you would evaluate a+b*c with operands/ intermediate results on one stack and operators on the other.)
- When you implement the *evaluate* method, you may want to test as you go, implementing code for and testing simple expressions, then building up to more complex expressions. The following is an example sequence of the kinds of expressions you may want to build with:
  - 3
  - a
  - 3+
  - a+b
  - 3+4*5

- a+b*c
- Then introduce parentheses
- Then try nested parentheses
- Then introduce array subscripts, but no parentheses
- Then try nested subscripts, but no parentheses
- Then try using parentheses as well as array subscripts
- Then try mixing arrays within parentheses, parentheses within array subscripts, etc.

Correctness of expression and input files

- All input expressions will be correctly formatted
- All input files with values for variables and arrays will be correctly formatted, and will be guaranteed to have values for all variables in the expression that is being evaluated

So you don't need to do any checking for correctness of inputs in any of the methods.

Running the evaluator

You can test your implementation by running the *Evaluator* driver on various expressions and input variable values file.
When creating your own variable values files for testing, make sure they are directly under the project folder, alongside *etest1.txt* and *etest2.txt*. Since you are not going to turn in the *Evaluator.java* file, you may introduce debugging statements and other methods (such as printing out the variables or arrays array lists) as needed.

No variables

```
Enter the expression, or hit return to quit => 3
Enter variable values file name, or hit return if no variables =>
Value of expression = 3.0

Enter the expression, or hit return to quit => 3-4*5
Enter variable values file name, or hit return if no variables =>
Value of expression = -17.0

Enter the expression, or hit return to quit =>
```

Neither of the expressions above have variables, so just hit return when asked for the variable values file name.

Variables, values loaded from file

```
Enter the expression, or hit return to quit => a
Enter variable values file name, or hit return if no variables => etest1.txt
Value of expression = 3.0

Enter the expression, or hit return to quit =>
```

Since the expression has a variable, *a*, the evaluator needs to be supplied with a file that has a value for it. Here's what *etest1.txt* looks like:

```
a 3
b 2
A 5 (2,3) (4,5)
B 3 (2,1)
d 56
```

Each line of the file begins with a variable name. For simple variables, the name is followed by the variable's integer value. For arrays, the name is followed by the array's length, which is followed by a series of (*index, integer value*) pairs.

Note: The index and integer value pairs must be written with no spaces around the index or integer value. So, for instance, (2, 3) or (2,3) or (2 ,3) are all incorrect. Make sure you adhere to this requirement when you create your own input files for testing.

If the value at a particular array index is not explicitly listed, it is set to *0* by default.
So, in the example above, A = [0,0,3,0,5] and B = [0,0,1]

Note that the variable values file can have values for any number of variables, so that it can be used as input for several expressions that contain one or more of the variables in the file.

Here are a couple more evaluations of expressions for which the variable values are loaded from *etest1.txt*:

```
Enter the expression, or hit return to quit => (a + A[a*2-b])
Enter variable values file name, or hit return if no variables => etest1.txt
Value of expression = 8.0
Enter the expression, or hit return to quit => a - (b+A[B[2]])*d + 3
Enter variable values file name, or hit return if no variables => etest1.txt
Value of expression = -106.0

Enter the expression, or hit return to quit =>
```

For a change of pace, here's *etest2.txt*, which has the following variables and values:

varx 6
vary 5
arrayA 10 (3,5) (8,12) (9,1)

And here are evaluations using this file:

Enter the expression, or hit return to quit => arrayA[arrayA[9]*(arrayA[3]+2)+1]-varx
Enter variable values file name, or hit return if no variables => etest2.txt
Value of expression = 6.0

Enter the expression, or hit return to quit =>

Submission

Submit your *Expression.java* source file.

Frequently Asked Questions

Q: Are array names all uppercase?
A: No. Arrays could have lower case letters in their names. You can tell if a variable is an array if it is followed by an opening square bracket. See, for example, the last example in the Expressions section, in which *varb* and *varz* are arrays: (varx + vary*varz[(vara+varb[(a+b)*33])])/55

Q: Can we delete spaces from the expression?
A: Sure.

Q: Will the expression contain negative numbers?
A: No. The expression will NOT have things like a*-3 or x+(-y). It will ONLY have the BINARY operators +, -, /, and *. In other words, each of these operators will need two values (operands) to work on. (The - in front of 3 in a*-3 is called a UNARY minus. UNARY operators will NOT appear in the input expression.) However, it is possible that in the process of evaluating the expression, you come across negative values, either because they appear in the input file, or because they are the result of evaluation. For instance, when evaluating a+b, a=6 and b=-9 as input values, and a result of -3 is a perfectly legitimate scenario.

Q: What if an array index evaluates to a non-integer such as 5/2?
A: Truncate it and use the resulting integer as the index.

Q: Could an array index evaluate to a negative integer?
A: No, you will not be given any input expression or values that would result in a negative integer value for an array index. In other words, you will not need to account for this situation in your code.

Q: What should I do on divide by zero?
A: You don't need to check for this situation.

Q: Could an array name be the same as a simple variable?
A: No. All variable names, for both simple variables and arrays, are unique.

Q: Should the expression "()" be reported as an error?
A: You don't have to do any error checking on the legality of the expression in the makeVariableLists or evaluate methods. When these methods are called, you may assume that the expression is correctly constructed. Which means you will not encounter an expression without at least one constant or variable, and all parens and brackets will be correctly formatted.

Q: Can I convert the expression to postfix, then evaluate the postfix expression?
A: NO!!! You have to work with the given traditional/infix form of the expression

Expression Evaluation Test Cases

makeVariableLists (20 pts)

Input Expression Points

```
 1 25 1
 2 abc 1
 3 xy[15] 1
 4 P[q[10]] 2
 5 m[n[X[5]]] 2
 6 B+B 2
 7 y+Y-z+R 2
 8 (a-(b+c)*d)/2 3
 9 b*(c/a)+z*a 3
10 (ab*3/defg) 3
```

evaluate (60 pts)

```
p 2
x 3
y 4
z 8
C 5 (1,1) (2,2) (3,3) (4,4)
D 6 (1,1) (3,3) (4,4) (5,5)
ARRY 3 (1,1) (2,2)
```

Input Expression Value Points

```
 1 255 255 2
 2 3+47 50 2
 3 13-4-12 -3 2
 4 150+14*15 360 2
 5 x 3 2
 6 p/y 0.5 2
 7 p-x+y 3 2
 8 p-x*z/p+y-4 -10 4
 9 (p) 2 2
10 p-(x-y) 3 4
11 p-((x-y)) 3 4
12 p-(x-y)*(2-z)+y 0 4
13 C[1] 1 2
14 C[x-2] 1 3
15 C[(z-y)] 4 4
16 C[3]+D[4] 7 4
17 C[D[0]] 0 4
18 z*(y-D[p+ARRY[1]-C[1]+x]) -8 6
19 C[D[ARRY[2]*2]]/2 2 5
```