Programming Assignment 2

Expression Evaluation

Posted Fri, Sep 30

Due Fri, Oct 21, 11:00 PM (WARNING!! NO GRACE PERIOD).

Extended deadline (with ONE time free extension pass): Mon, Oct 24, 11:00 PM (NO GRACE PERIOD)

Worth 65 points (6.5% of course grade)

In this assignment you will implement a program to evaluate an arithmetic expression **USING RECURSION AND STACKS**.

Read the Course Policies page, **Programming Assignments** section for important information about lateness, program not compiling, etc.

You will work **individually** on this assignment. Read the DCS Academic Integrity Policy for Programming Assignments - you are responsible for this. In particular, note that **"All Violations of the Academic Integrity Policy will be reported by the instructor to the appropriate Dean".**

You get ONE extension pass for the semester, no questions asked. There will be a total of 5 assignments this semester, and you may use this one time pass for any assignment. **A separate Sakai assignment will be opened for extensions AFTER the deadline for the regular submission has passed.**

IMPORTANT - READ THE FOLLOWING CAREFULLY!!!

- Assignments emailed to the instructor or TAs will be ignored--they will NOT be accepted for grading. We will only grade submissions in Sakai.
- If your program does not compile, you will not get any credit.

Most compilation errors occur for two reasons:

1. You are programming outside Eclipse, and you delete the "package" statement at the top of the file. If you do this, you are changing the program structure, and it will not compile when we test it.
2. You make some last minute changes, and submit without compiling.

To avoid these issues, (a) START EARLY, and give yourself plenty of time to work through the assignment, and (b) Submit a version well before the deadline so there is at least something in Sakai for us to grade. And you can keep submitting later versions (up to 10) - we will accept the LATEST version.

Expressions

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
- Scalar (simple, non-array) variables with integer values
- Arrays of integers, indexed with a constant or a subexpression
- Addition, subtraction, multiplication, and division operators
- 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 scalars or arrays) will be made up of one or more letters only (case sensitive, so Xyz is different from xyz)
- 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

Download the attached expression_project.zip file to your computer. DO NOT unzip it. Instead, follow the instructions on the Eclipse page under the section "Importing a Zipped Project into Eclipse" to get the entire project into your Eclipse workspace.

You will see a project called Expression Evaluation with the following classes in package apps:

- ScalarSymbol
  This class represents a simple variable with a single value. Your implementation will create a ScalarSymbol object for every simple variable in the expression. You don't have to implement anything in this class, so **do not make any changes to it.**

- ArraySymbol
  This class represents an array of integer values. Your implementation will create an ArraySymbol object for every array variable in the expression. You don't have to implement anything in this class, so **do not make any changes to it.**

- Expression
  This class represents the expression as a whole, and consists all the following steps of the evaluation process:

  1. **15 pts**: buildSymbols - This method populates the two instance fields, scalars and arrays, with all simple (scalar) variables, and all array variables, respectively, 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.

  2. loadSymbolValues - This method reads values for all scalars and arrays from a file, into the ScalarSymbol and ArraySymbol objects stored in the scalars and arrays array lists. This method is already implemented, **do not make any changes**.

3. **50 pts**: evaluate - This method evaluates the expression.
   You will fill in the implementation of this method. You MUST use RECURSION to evaluate parenthesized subexpressions and array index expressions. You can write a separate private recursive method and call it from this public method.

Two other methods, printScalars and printArrays are implemented for your convenience, and may be used to debug the loadSymbolValues method.

- Evaluator, the application driver, which calls methods in Expression

You are also given the following class in package structures:

- Stack, to be used in the evaluation process

Lastly, two test files are included, etest1.txt and etest2.txt, appearing directly under the project folder.

Do not add any other classes. In particular, don't use your own stack class, make sure you use the one you are given.

You will need to separate out ("tokenize") the components of the expression in buildSymbols and evaluate. Tokens include operands, operators, parentheses and square brackets. It may be helpful (but you are not required) to use java.util.StringTokenizer to tokenize the expression. See the loadSymbolValues method in Expression for an example of using StringTokenizer to extract variable names. The delims field in the Expression class may be used in the tokenizing process.

Observe the following rules while working on Expression.java:

- You may NOT add any import statements to the file.
  Note that the java.io.* and java.util.* import statements at the top of the file allow for importing ANY class in java.io or java.util packages without additional specification.
- You may NOT add any fields to the Expression class.
- You may NOT modify the headers of any of the given methods.
- You may NOT delete any methods.
- You MAY add helper methods if needed, as long as you make them private.

Rules and guidelines for Implementing evaluate

- An expression may contain sub-expressions within parentheses - you MUST use RECURSION to evaluate sub-expressions.
- Recursion MUST also be used to evaluate array subscripts as well, since a subscript is an expression.
- 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.)
- 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+4
  - 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.

We will grade your program by running it through several test cases.

You may assume that all input symbol values files will be correctly formatted, and every file will be guaranteed to have values for all symbols in the expression that is being evaluated. You don't have to check the validity of the symbol values files in your code.

NOTE:

When we test your evaluate method, we will use OUR implementation of the buildSymbols method. This is for your benefit, so that in the event that your buildSymbols does not work correctly, your evaluate method will not be adversely affected.

Running the evaluator

No variables

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

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

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

Neither of the expressions above have variables, so just hit return to skip the symbol loading part.

Variables, values loaded from file

```
Enter the expression, or hit return to quit => a
Enter symbol values file name, or hit return if no symbols => 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 scalar variables, the name is followed by the variable's integer value. For array variables, the name is followed by the array's length, which is followed by a series of (index, integer value) pairs. If the value at a particular array index is not explicitly listed, it is set to 0 by default.

So, A = [0,0,3,0,5] and B = [0,0,1]

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

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

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

Enter the expression, or hit return to quit => a - (b+A[B[2]])*d + 3
Enter symbol values file name, or hit return if no symbols => 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 symbols 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 symbol values file name, or hit return if no symbols => etest2.txt
Value of expression = 6.0

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

Submission

Submit your **Expression.java** file ONLY.

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 "Expression" section:

(varx + vary*varz[(vara+varb[(a+b)*33])])/55

Q: Can we delete spaces from the expression?

A: Sure.

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: 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 buildSymbols or evalute 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. Which means you won't need to deal with "()".

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.