Assignment 2

COMP 250 Fall 2017

posted:	Sunday, Oct. 8, 2017
due:	Sunday, Oct. 22, 2017 at 23:59

The Teaching Assistants handling this assignment are Tabish Syed (email) and Jack Guo (email). Their office hours will be posted on mycourses Announcements.

The same General Instructions apply as in Assignment 1.

Introduction

In your usage of Java language so far you have come across various programming constructs such as variables, operators, control flow statements etc. Similar constructs are present in many other programming languages. In this assignment, you will deal specifically with expressions and their evaluation. You will parse expressions and evaluate them using a stack.

An **expression** is a programming language construct which evaluates to a single value. It is composed of different elements such as constants, variables, operators, method calls etc, put together according to the *syntax* of the language. For example (2 + 3) in Java evaluates to a single value 5 and is composed of operator + and constants 2 and 3. A similar expression in a different programming language may follow different syntax. For example in Lisp the expression is written as (+ 2 3) .

Assignment

Your task in this assignment is to evaluate arithmetic expressions and calculate their values. The statements are written in a programming language whose syntax is described below. You will use Java to write a parser and an evaluator for expressions written in this language. For simplicity you will only deal with input expressions that are composed of positive integers and the operators that are listed below.

+	addition	++	increment
–	subtraction	– –	decrement
*	multiplication	[ ]	absolute value
/	integer division		

You may assume that values of expressions are of type `int` in Java and are within the range specified by that type. Since the subtraction operator is included in the list below, the value of an expression can be either positive or negative.

Increment or decrement operations on constants are not allowed in Java. But our language does allow it. In particular, our language has the pre-increment operation which increments values of

an integers or an expression, and similarly it has the pre-decrement operation which decrements values. For example, the value of "(++5)" is "6" while expression "(– –9)" evaluates to 8.

Note that operators +, –, *, / are binary operators, i.e, they operate on two elements and produce a value, whereas the operators ++, – – and [ ] are unary operators, i.e, they operate on a single element and produce a value. Consider an expression "(++(3 + 2))" in our language. The binary operator "+" operates on *two* `int` values 2 and 3 and produces 5. The unary operator "++" then operates on the result 5, a *single* value, and produces 6 which is the value of the expression. Similarly "((++(5 - 2)) + 3)" evaluates to 7.

The absolute value operator is denoted by [ ], instead of the | | which is generally used in arithmetic. We use the square brackets because they have a left and right versions which makes parsing easier. An expression within the [ ] operator evaluates to its absolute value. Note that no such operator exists in Java. See Table 1 for more examples of expressions.

Formal language

Your task in this assignment is to write code to parse and evaluate expressions. You can assume that your code will only be tested on valid expressions. But how can we define what a valid expression is? As you will learn if you take COMP 330 and later a compilers course such as COMP 520, there are formal methods for defining languages. Indeed most programming languages are defined using these formal methods.

The syntactic rules for expressions and operators in our expression language can be defined as follows. A *digit* is defined:

$$digit = 0 \mid 1 \mid 2 \mid ... \; 9$$

The vertical bar | does not appear in expressions. Rather it is used here to mean *or*, that is, a digit can be any one of the characters '0', '1',.. '9'. Using the definition of a digit, we can then define a *positive integer* in a *recursive* manner as follows,

$$positiveInteger \;=\; digit \mid digit \;\; positiveInteger$$

so a *positive integer* is a *digit* or a digit followed by a *positive integer*. Notice that in this definition we allow positive integers to start with digit '0', but you can redefine a *positive integer* to exclude this case if we wish. We can also define operators

$$IncDecOperator = ++ \;\mid\; --$$
$$binaryOperator = + \mid - \mid * \mid /$$

In a similar manner we define an *expression* as follows:

$$
\begin{aligned}
\textit{expression} \;=\; & (\ \textit{positiveInteger}\ \textit{binaryOperator}\ \textit{positiveInteger}\ \ ) \mid \\
& (\ \textit{expression}\ \textit{binaryOperator}\ \textit{positiveInteger}\ \ ) \mid \\
& (\ \textit{positiveInteger}\ \textit{binaryOperator}\ \textit{expression}\ ) \mid \\
& (\ \textit{expression}\ \textit{binaryOperator}\ \textit{expression}\ ) \mid \\
& (\ \textit{IncDecOperator}\ \ \textit{positiveInteger}\ ) \mid \\
& (\ \textit{IncDecOperator}\ \ \textit{expression}\ ) \mid \\
& [\ \textit{expression}\ ] \mid \\
& [\ \textit{positiveInteger}\ ]
\end{aligned}
$$

An expression is therefore one of the following:

- a positive integer or an expression followed by a binary operator followed by a positive integer or an expression, enclosed within round brackets
- an *increment or decrement operator* followed by a positive integer or by an expression, enclosed within round brackets.
- an expression or a positive integer, enclosed within square brackets

Note that a valid expression is always contained within brackets. Thus "(3+1)" is a valid expression but "3+1" is not. This requirement on the brackets ensures that there is a unique order for evaluating the (sub)expressions in the case that an expression has multiple operators. We will return to this issue later in the course when we discuss expression trees.

In your solution, *you may assume that each expression has been parenthesized with proper nesting*. Therefore, you don't need to worry about operator precedence issues nor checking for balanced parenthesis. Table 1 below lists some valid expressions and their respective values

Expression	Evaluated Value
(2 / 3)	0
((– –(2 + 3)) – 1)	3
(++((7 - 2) * 3))	16
[((7 + 2) * (3 – 4))]	9
((7 - 2) * (++(4 - 3)))	10

Table 1: Example expressions with corresponding values

Parsing

Your first task is to parse in input expression string into meaningful units called *tokens*. Parsing will take an expression as a string and return an array list of the tokens in the expression, such that each token is string. We use an array list because it is more efficient than a linked list in certain situations. Thus, the expression "(27 + 572)" should be parsed into the list of five tokens: "(", "27", "+", "572", ")". Note that white spaces in an expression string must be ignored when parsing.

We can then use this list of tokens in the `eval()` method to calculate the value of the expression. It is important that for any expressions containing the "++" and "– –" operators be stored as a single token rather than as two. Thus, "(++25)" should be parsed as "(", "++", "25", ")" rather than "(", "+", "+", "25", ")". Also note that "25" must be not be parsed as "2", "5".

The parsing is done in the Expression constructor. *You will need to fill in the missing code where it is indicated.*

You should use the Java `StringBuilder` class to build each token as you parse the expression string. The `append()` and `delete()` methods of the `StringBuilder` class will be helpful in tokenizing the string.

Evaluation

The evaluation is done in the `eval` method. Again, your task is to fill in the missing code where it is indicated. The easiest approach is to use two stacks, one to store values (named `valueStack` in the starter code) and another (named `operatorStack`) to store tokens such as operators or brackets.

We emphasize that you may assume that all expressions are valid; that is, they are syntactically correct and will evaluate to a single value. HINT: Considering that all expressions are syntactically correct, there is no need to store the opening bracket of an expression in the stacks.

Testing

We provide an additional tester class to test your expression class along with sample testing files (`sample_in.txt`, `sample_value.txt`, `sample_exp.txt`) which test for simple expressions only. *Your code will be evaluated on a much more extensive and challenging set of test cases, and we encourage you to post complicated test cases on the discussion board.*

Grading

Your grade will be based on the number of test cases passed. No points for code-style or readability, though you should comment your code for your own benefit.

a) Implement parser in the `Expression` constructor **(40 points)**

b) Implement support for operators +, –, *, / **(20 points)**

c) Implement support for operators – – and ++ **(20 points)**

d) Implement support for operator [ ] **(20 points)**

Other Specifications

All your code should be written in the `Expression.java` file. Submit a file `A2.zip` which contains only the file `Expression.java` .

Only add code where it is indicated.

Do not add import statements.

You can use whatever package statement you want. The grading software will remove it.

Good luck and have fun!