Expressions

Due: Friday, April 12 @ 11:59pm

Updates and Pitfalls

-

Objective

Gain more experience with functions as parameters and the stack data structure while exploring how computers evaluate the expressions we write

Description

Reading:

- https://en.wikipedia.org/wiki/Shunting-yard_algorithm

You will be writing code that will evaluate infix expressions (ex 15 - (8 + 9 / 3) evaluates to 4). To accomplish this task you address both order of operations and parentheses to arrive at the correct value. One way to do this is to take advantage of the Shunting-Yard algorithm which can convert an infix expression into a postfix expression. This postfix expression can then be evaluated, or the Shunting-Yard algorithm can be modified to evaluate the expression as it's being parsed.

Project Structure

1. Create a new project in IntelliJ
2. In the src folder create a package named expressions
3. In the expressions package create a new Scala object named Expressions
4. In the src folder create a package named tests

Testing Objectives (30 points)

You may complete the primary and testing objectives in any order. Similar to the Genetic Algorithm homework, the primary objective is a generalization of the 2 testing objectives. It is recommended that you complete both testing objectives first to gain an understanding of the Shunting Yard algorithm and rewrite your code generically for the primary objective, but you do have the option of starting with the primary objective then calling the primary objective method to complete the testing objective(s).

Testing Objective 1

In the expressions.Expressions object write a method named evaluateArithmetic that takes an expression as a String and returns its evaluation as a Double. Your method must account for parentheses and the following operators:

- "^" exponentiation
- "*" multiplication
- "/" division
- "+" addition
- "-" subtraction

Use the standard PEMDAS order of operations (^ first, then * and /, then + and -, and parentheses override any ordering). You may assume that exponentiation is left associative for this assignment (eg. 2^3^4 would evaluate to 4096), though it's a right associative operator.

Example: "10 - (8/12.0*6)/2-1" evaluates to 7.0

You should ignore spaces in your method.

Example: "2.5+2.5" and " 2.5 + 2.5 " both evaluate to 5.0

You may assume all inputs will be valid expressions (Do not test with invalid expressions).

Testing: In the tests package write a test suite named TestArithmetic that tests evaluateArithmetic. Do not test with consecutive exponentiation (ex: 2^3^4). This would require us to handle the difference between left and right associativity which is beyond the purpose of this assignment. Instead, you may assume ^ is left associative and don't write test cases that expect it to be right associative.

Testing Objective 2

In the expressions.Expressions object write a method named evaluateBoolean that takes an expression as a String and return its evaluation as a Boolean. Your method must account for parentheses and the following operators:

- "&&" and
- "||" or
- "xor" exclusive or
- "->" implication
- "<>" if and only if

For the order of operations use && first, then || and xor, then -> and <>. Assume all operations are left associative.

The operands will all be the strings “true” or “false”

Examples:
“(true -> false) <> (false || false)” evaluates to true
“true || false && false -> false xor true && false” evaluates to false

You may assume all inputs will be valid expressions (Do not test with invalid expressions).

Note: To control the difficulty of the assignment we are only using binary operators and excluding unary operators such as ! (the not operator).

Testing: In the tests package write a test suite named TestBoolean that tests evaluateBoolean.

Primary Objective (35/20 points)

Implement a generic expression evaluator.

In the expressions.Expressions object write a method named evaluate that can evaluate expressions of any type, operators, and order of operations. This method will:

- Take a type parameter A
    - This will be the type of the expression (ex. Double for testing objective 1 and Boolean for testing objective 2)
- As the first parameter, takes the expression to be evaluated as a String
- As the second parameter, takes a function that takes a String and returns an A
    - This function will be used convert the operators in the expression into the appropriate type (ex. input.toDouble for testing objective 1 and input.toBoolean for testing objective 2)
- As the third parameter, takes a Map[String, (A, A) => A] representing the operators that can appear in the expression
    - The keys in this Map are the operators as Strings as they will appear in the expression (ex. “*”, “-”, “&&”)
    - You may assume that no two operators are substrings of each other
    - You may assume that the expression will not contain an operator as a substring unless it represents the operator itself (eg. you can use use find/replace on the operator strings)
- As a fourth parameter, takes a List[List[String]] containing the order of operations
    - This is a list of lists of operators. Any operator appearing in an earlier list that another operator will have higher precedence (evaluated first). Operators in the same list will have equal precedence (ex. For arithmetic this would be List(List(“^”), List(“*”, “/”), List(“+”, “-”))
- Returns an object of type T which is the evaluation of the expression

For clarification, testing objective methods that use the generic evaluate method are provided below.

```
def evaluateArithmetic(expression: String): Double = {
 val pow = (a: Double, b: Double) => Math.pow(a, b)
 val mul = (a: Double, b: Double) => a * b
 val div = (a: Double, b: Double) => a / b
 val add = (a: Double, b: Double) => a + b
 val sub = (a: Double, b: Double) => a - b

 val operatorTable: Map[String, (Double, Double) => Double] = Map(
   "^" -> pow,
   "*" -> mul,
   "/" -> div,
   "+" -> add,
   "-" -> sub
 )

 val order = List(List("^"), List("*", "/"), List("+", "-"))

 evaluate(expression, (s: String) => s.toDouble, operatorTable, order)
}


def evaluateBoolean(expression: String): Boolean = {
 val and = (a: Boolean, b: Boolean) => a && b
 val or = (a: Boolean, b: Boolean) => a || b
 val xor = (a: Boolean, b: Boolean) => (a || b) && !(a && b)
 val implies = (a: Boolean, b: Boolean) => !(a && !b)
 val iff = (a: Boolean, b: Boolean) => (a && b) || (!a && !b)

 val operatorTable: Map[String, (Boolean, Boolean) => Boolean] = Map(
   "&&" -> and,
   "||" -> or,
   "xor" -> xor,
   "->" -> implies,
   "<>" -> iff
 )

 val order = List(List("&&"), List("||", "xor"), List("->", "<>"))

 evaluate(expression, (s: String) => s.toBoolean, operatorTable, order)
}
```

Hint: It can be helpful to “tokenize” the expression to separate the operators and operands. One way to do this is to replace all operators and parentheses with the operator/parenthesis surrounded by a delimiter (You can use “_” as this is not used in any testing). Then split the expression on this delimiter and ignore any splits that are the empty string.

Example: “(3+4)*3” becomes “_(_3_+_4_)__*_3” after calling expression.replace(op, “_” + op + “_”) for each operator/parentheses and you can split on “_” to get all the tokens.

Bonus Objective (25 points)

The bonus objective has a separate submission in AutoLab and is treated as a separate assignment. The 50 point cap does not apply to the bonus objective making a total of 75 points available for this homework.

Write a method to evaluate scripts.

In the expressions.Expressions object write a method named runScript that takes a multi-line script as a file and returns the evaluation of the last line of the script.

Features of the scripts:

- The scripts can contain variables (You'll have to track the value of each variable)
- Variables can be reassigned
- Variables do not have keywords when declared (Python-like syntax)
- Each line except the last will contain an assignment with a variable name on the left of the = and an expression on the right side
- The last line in the script will only contain an expression. Return the evaluation of this expression. If you read a line that does not contain an "=" you may assume it is the last line in the script
- The scripts will not contain blank lines

Example script for arithmetic:

```
x = 3*3^2
y = x
y = y + 2
9^(y-x)
```

Evaluates to 81

The runScript method has the same signature (parameters and return value) as evaluate from the primary object except instead of taking an expression as the first parameter it takes the filename of a script that is to be executed.