Univerzitet u Sarajevu
Elektrotehnički fakultet Sarajevo
Predmet: Razvoj programskih rješenja 2022/2023

Zadaća 1

Opis Zadataka

Potrebno je napraviti Java Maven konzolnu aplikaciju koja će se koristiti za parsiranje i izračunavanje artmetičkih izraza korištenjem Dijkstra algoritma za evaluaciju izraza opisanog u paragrafu ispod. Projekat treba biti postavljen na github repozitorij svakog od studenata. Prilikom predaje zadaće potreno je samo predati link na github repozitorij.

Za implementaciju je potrebno uraditi slijedeće:

- Potrebno je napraviti klasu ***ExpressionEvaluator*** koja će imati jednu javnu metodu ***evaluate*** koja prima parametar tipa ***String*** i vraća rezultat tipa ***Double.***
- Klasa ***ExpressionEvaluator*** treba da koristi dvije instance klase ***java.util.Stack*** kako bi implementirala Dijkstra algoritam koji je detaljno objašnjen u paragrafu ispod.
- Potrebno je napraviti klasu ***App*** koja ce imati ***main*** methodu koja parsira ulaz s konzole iz parametra ***args*** i vrši njegovu validaciju.
- Potrebno je napraviti klasu ***ExpressionEvaulatorTest*** koja ce imati minimalno 5 unit testova za provjeru ispravnosti rada algoritma.
- Projekat mora imati javadoc dokumentaciju na svim klasama i metodama. Takođe je portebno dodati ja maven javadoc plugin kako bi se mogla izgenerisati HTML dokumentacija.
- Ulazna klasa s main metodom kora biti deklarisana u maven jar pluginu kako bi se kompajlirani jar program mogao poreknuti iz komandne linije.
- U slučaju da uneseni izraz nije aritmetički validan program treba da baci izuzetak tipa ***RuntimeException***. Jedan od unit testova mora pokriti ovaj slučaj.
- Potrebno je uraditi samo implementaciju algoritma za izraze koji su omeđeni zagradama i koji nema podršku za prioritet operatora.
- Možete pretpostaviti da će svi izrazi biti razdvojeni spaceom radi lakšeg parsiranja ulaznog izraza. Izraz (1 + (5 * 20)) se smatra validnim dok se izraz (1 + (5* 20)) smatra nevalidnim. Ovo je jedna olakšica koja će vam pomoći da lakše kodirate zadaću.

Dijkstra’s Algorithm for expression evaluation

Arithmetic expression evaluation. As another example of stack usage, we consider a classic example that also demonstrates the utility of generics, involving computing the value of arithmetic expressions like this one:

(1 + ((2 + 3) * (4 * 5)))

If you multiply 4 by 5, add 3 to 2, multiply the result, and then add 1, you get the value 101. But how does the Java system do this calculation? Without going into the details of how the Java system is built, we can address the essential ideas by writing a Java program that can take a string as input (the expression) and produce the number represented by the expression as output. For simplicity, we begin with the following explicit recursive definition: an arithmetic expression is either a number or a left parenthesis followed by an arithmetic expression followed by an operator followed by another arithmetic expression followed by a right parenthesis. For simplicity, this definition is for *fully parenthesized arithmetic expressions*, which specify precisely which operators apply to which operands—youareabitmorefamiliarwithexpressionssuchas1 + 2 * 3, where we often rely on precedence rules instead of parentheses. The exact basic mechanisms that we consider can handle precedence rules,but we avoid that complication. For specificity, we support the familiar binary operators *, +, -, and /, as well as a square-root operator sqrt that takes just one argument. We could easily allow more operators and more kinds of operators to embrace a large class of familiar mathematical expressions, involving trigonometric, exponential, and logarithmic functions. Our focus is on understanding how to interpret the string of parentheses, operators, and numbers to enable performing the proper order of the low-level arithmetic operations that are available on any computer. Precisely how can we convert an arithmetic expression (a string of characters) to the value that it represents? A remarkably simple algorithm that was developed by E. W. Dijkstra in the 1960s uses two stacks (one for operands and one for operators) to do this job. An expression consists of parentheses, operators, and operands (numbers). Proceeding from left to right and taking these entities one at a time, we manipulate the stacks according to four possible cases, as follows:

- Push operands onto the operand stack.
- Push operators onto the operator stack.
- Ignore the left parentheses.
- On encountering a right parenthesis, pop an operator, pop the requisite number of operands, and push onto the operand stack the result of applying that operator to those operands.

After the final right parenthesis has been processed, there is one value on the stack, which is the value of the expression.

This method may seem mysterious at first, but it is easy to convince yourself that it computes the proper value: any time the algorithm encounters a subexpression consisting of two operands separated by an operator, all surrounded by parentheses, it leaves the result of performing that operation on those operands on the operand stack. The result is the same as if that value had appeared in the input instead of the subexpression, so we can think of replacing the subexpression with the value to get an expression that would yield the same result. We can apply this argument again and again until we get a single value. For example, the algorithm computes the same value for all of these expressions:

(1 + ((2 + 3) * (4 * 5)))

(1 + (5 * (4 * 5)))

(1 + (5 * 20))

(1 + 100)

101

The basic implementation of this algorithm within the ***main*** method is given below. This code is a simple example of an interpreter but the code is poorly structured and should be refactored in an object-oriented manner. Classes ***StdIn*** and ***StdOut*** are utility classes for managing IO for this implementation and they are not shown here.

Dijkstra's Two-Stack Algorithm for Expression Evaluation

```
public class Evaluate
{
   public static void main(String[] args)
   {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty())
      {  // Read token, push if operator.
         String s = StdIn.readString();
         if      (s.equals("("))               ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("-"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals("/"))    ops.push(s);
         else if (s.equals("sqrt")) ops.push(s);
         else if (s.equals(")"))
         {  // Pop, evaluate, and push result if token is ")".
            String op = ops.pop();
            double v = vals.pop();
            if      (op.equals("+"))    v = vals.pop() + v;
            else if (op.equals("-"))    v = vals.pop() - v;
            else if (op.equals("*"))    v = vals.pop() * v;
            else if (op.equals("/"))    v = vals.pop() / v;
            else if (op.equals("sqrt")) v = Math.sqrt(v);
            vals.push(v);
         }  // Token not operator or paren: push double value.
         else vals.push(Double.parseDouble(s));
      }
      StdOut.println(vals.pop());
   }
}
```