Tema 4 - Calculator de buzunar

Obiective

- familiarizarea cu structurarea aplicației prin aplicarea de *design patterns*
- folosirea design pattern-ului Visitor
- familiarizarea cu conceptele de parsare lexicală și semantică

Descriere și cerințe

Implementați un program care parsează și evaluează *expresii matematice*. Este obligatoriu să structurați codul folosind pattern-ul **Visitor**.

Sintaxa expresiilor

- Expresiile conțin doar numere de tip *Float*, operatori și paranteze, toate separate prin unul sau mai multe spații.
- Operațiile suportate sunt:
  - binare: `+, -, *, /, ^` (ridicarea la putere), în ordine, de la cel mai puțin prioritar:
    - `+, -`
    - `*, /`
    - ^ (atenție, spre deosebire de cei de mai sus, este doar asociativ dreapta: 2 ^ 3 ^ 4 = 2 ^ (3 ^ 4))
  - funcții unare: **`log`** (log in baza 10), **`sqrt`** (radical), **`sin`**, **`cos`**, **`-` (minus unar, pentru numere negative)**
    - au prioritate mai mare decat ceilalți operatori
    - minusul unar poate apărea doar la începutul expresiei, sau imediat după o paranteză deschisă (cu alte cuvinte, nu veți avea expresii de tipul 1 + - 2)
    - operatorul `-` unar aplicat unei funcții trebuie separat prin spațiu:
      - `- sqrt ( 2 )`, nu `-sqrt ( 2 )`
- Expresiile pot conține paranteze `(`, `)`, în orice combinații
- **Trebuie verificată corectitudinea *sintactică* a expresiilor**, iar dacă se întâlnește o eroare de sintaxă (operator invalid, număr neparsabil etc), se va arunca o excepție user-defined, numită`SyntacticException`, iar parsarea expresiei se va termina.

Exemple expresii:

- `1 + 2 - 3 * ( 4 + 5 ) / log ( 10 * sqrt ( 100 ) )`
- `- 2 * sin ( -5 + 2 ) + 4`

Rezultatul parsării unei expresii îl reprezintă rădăcina unui arbore, care va fi apoi parcurs în partea de evaluare. Mai jos observați un arbore de parsare pentru exemplul primei expresii.

```
   +
  / \
 1   -
    / \
   2   /
      / \
     *   log
    / \     \
   3   +     *
      / \   / \
     4   5 10  sqrt
                |
               100
```

Puteți folosi funcții standard Java de parsare a numerelor (`Float.ParseFloat(String)`) sau funcții standard de operare pe String-uri, care vă pot simplifica generarea arborelui. *Nu sunt permise decât funcționalități Java standard, precum cele menționate anterior sau framework-ul `Collections`*.

Evaluarea expresiilor

- Expresiile se evaluează în funcție de paranteze și de prioritatea operațiilor. Arborele de parsare pe care îl generați va fi punctul de plecare în calculul rezultatului.
- **Trebuie verificată corectitudinea *semantică* a expresiilor** o dată cu parcurgerea structurii, iar dacă se întâlnește o eroare, se va arunca o excepție user-defined, numită `EvaluatorException`, iar evaluarea expresiei se va termina. Va trebui să tratați erorile de calcul matematic:
  - împărțirea la 0
  - log(x) unde x <= 0
  - sqrt(x) unde x < 0

Implementare

Scheletul de cod al temei impune implementarea metodei `eval` din clasa *ExpressionParser*. Aceasta metodă primește ca argument expresia ce trebuie parsată și întoarce rezultatul acesteia.

Expresia este dată ca un șir de caractere, care trebuie despărțit în tokens (separate prin spații) pentru fiecare element al expresiei (număr, operator sau paranteză). După parsare, expresia va fi reprezentată ca un arbore binar ca în exemplul din secțiunea anterioară, și această structură de date va fi parcursă de către vizitatori. Un exemplu de algoritm pentru construcția arborelui găsiți în secțiunea Referințe.

Folosiți tipul *double* la parsare și la evaluarea rezultatelor intermediare, și faceți un cast înapoi la float la ultimul rezultat.