Univerzita Mateja Bela v Banskej Bystrici
Fakulta prírodných vied

DOKUMENTÁCIA

Zadanie 2: Riadkový kalkulátor

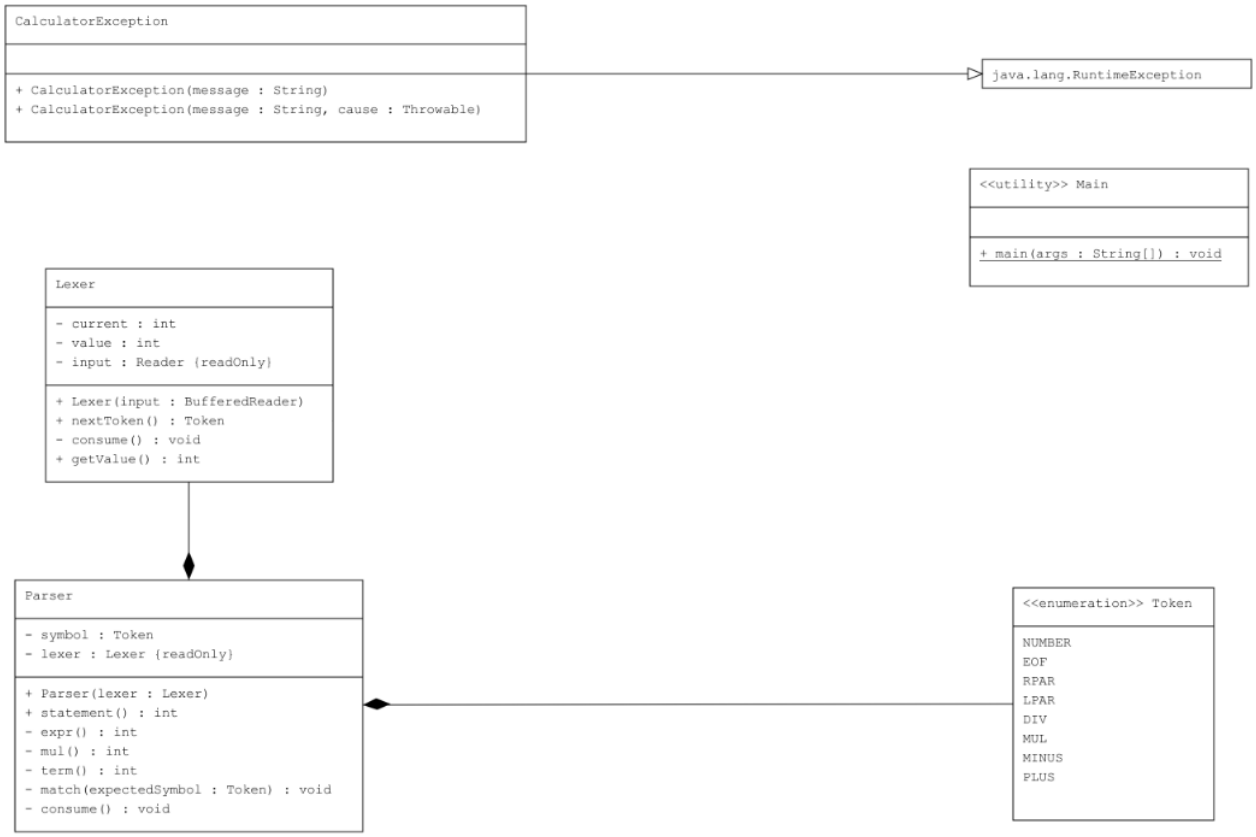
Meno a priezvisko: Jakub Uhrinčať
Študijný program: Aplikovaná informatika a tvorba softvéru
Predmet: Formálne jazyky a automaty
Akademický rok: 2024/2025
Semester: zimný

Úvod

Cieľom tohto projektu bolo vytvoriť program v jazyku Java (alebo alternatívne v C alebo Python), ktorý slúži na vyhodnocovanie matematických výrazov. Výrazy, ktoré program spracováva, sú zadávané vo forme reťazca. Program mal správne interpretovať operátory podľa ich priority a asociativity a následne vypočítať výsledok výrazu.

Úlohou bolo implementovať kalkulačku, ktorá prečíta vstupný reťazec, rozdelí ho na tokeny, následne analyzuje a vyhodnocuje matematický výraz podľa pravidiel precedencie operátorov. Dôležitým požiadavkom bolo, že program nemohol používať generátory jazykových procesorov ani interné funkcie ako eval, ktoré by za nás vykonali spracovanie výrazu. Namiesto toho bolo potrebné implementovať vlastnú lexikálnu a syntaktickú analýzu.

Na začiatku projektu bola poskytnutá kostra programu, ktorá obsahovala základné štruktúry a rozhranie pre implementáciu parsera, lexikálneho analyzátora (lexera) a vyhodnocovateľného výrazu. Tieto komponenty bolo potrebné vyplniť a prispôsobiť požiadavkám zadania. Pri zadaní a práci na programe sme sa inšpirovali dostupnou schémou pri zadaní, ktorá poukazuje na rozdelenie tried programu a je znázornená nižšie:



Opis programu

Program sa skladá z niekoľkých základných komponentov, pričom každý z nich je implementovaný ako samostatná trieda zodpovedná za konkrétnu časť spracovania výrazu.

Program je rozdelený do piatich hlavných tried:

- **CalculatorException** – trieda pre spracovanie a výpis chýb
- **Token** – trieda reprezentujúca jednotlivé tokeny, ktoré sa získavajú počas lexikálnej analýzy (číselné hodnoty, operátory, zátvorky).
- **Lexer** – trieda zodpovedná za rozdelenie vstupného reťazca na tokeny
- **Parser** – trieda, ktorá analyzuje tokeny a vyhodnodnocuje matematické výrazy.
- **Main** – hlavná trieda, ktorá riadi celý proces, zabezpečuje interakciu s používateľom a volanie metód z ostatných tried.

Týmto rozdelením sa dosiahla modularita a čitateľnosť programu, pričom každá trieda je zodpovedná za konkrétnu funkčnosť v celkovom procese vyhodnocovania matematických výrazov. Tento prístup uľahčuje správu kódu a umožňuje jednoduchú údržbu a rozširovanie programu v prípade potreby.

CalculatorException

Trieda CalculatorException slúži na spracovanie chýb, ktoré sa vyskytnú počas vyhodnocovania matematického výrazu. Je rozšírením triedy RuntimeException, čo znamená, že ide o chyby, ktoré môžu vzniknúť počas behu programu a sú spracovávané na úrovni výnimiek.

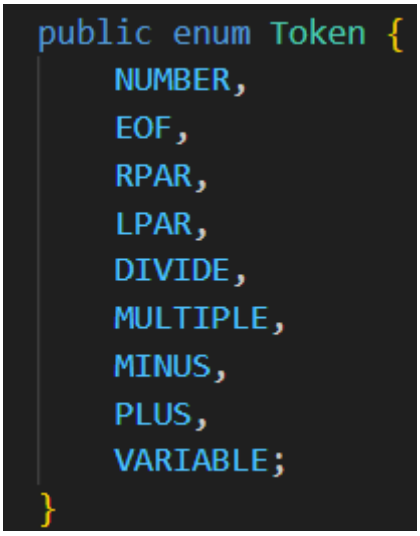
```java
public class CalculatorException extends RuntimeException {

    public CalculatorException(String message) {
        super(message);
    }

    public CalculatorException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Trieda poskytuje dva konštruktory:

- **CalculatorException(String message)** – umožňuje vytvoriť výnimku s chybovým hlásením.
- **CalculatorException(String message, Throwable cause)** – umožňuje pridať pôvodnú príčinu výnimky, čo je užitočné pri reťazení výnimiek a lepšom sledovaní príčin chýb.

Token

Trieda Token je enum, ktorý reprezentuje rôzne typy tokenov, ktoré sa môžu vyskytnúť počas analýzy a vyhodnocovania matematického výrazu. Tento enum je kľúčový pre lexer, ktorý rozdeľuje vstupný reťazec na jednotlivé tokeny. Každý token reprezentuje určitý typ symbolu, ako napríklad čísla, operátory alebo zátvorky.



Tento Token enum obsahuje nasledujúce hodnoty:

- **NUMBER**: číslo zo vstupného reťazca.
- **EOF**: koniec súboru alebo vstupu.
- **RPAR** , **LPAR**: uzatváracie a otváracie zátvorky.
- **DIVIDE**, **MULTIPLE**, **MINUS**, **PLUS**: operátory matematických operácií.
- **VARIABLE**: premenné, ktoré môžu byť nahradené hodnotami počas vyhodnocovania výrazu.

Lexer

Trieda Lexer je zodpovedná za rozdelenie vstupného reťazca na jednotlivé tokeny. Tento proces, nazývaný lexikálna analýza, je prvým krokom pri spracovaní matematického výrazu. Lexer prechádza vstupný reťazec znak po znaku, rozpoznáva jednotlivé symboly a prevádza ich na príslušné tokeny, ktoré následne použije parser na ďalšie spracovanie.

V našej triede Lexer pracujeme s týmito premennými:

- **current** –aktuálny znak zo vstupného reťazca Na začiatku je to prvý znak reťazca
- **value** – slúži na ukladanie hodnoty, ktorú lexer extrahuje z číselných reťazcov
- **input** – vstupný reťazec, ktorý obsahuje matematický výraz
- **position** –aktuálna pozícia v reťazci, ktorý sa spracováva. Na začiatku je nula
- **variableName** – uchováva názov premennej, ak lexer narazí na text reprezentujúci premennú.

Na začiatku sa inicializuje Lexer so vstupným reťazcom, ktorý obsahuje matematický výraz. Premenné input, current, a position sú nastavené, aby sa zabezpečilo, že lexer začne spracovávať vstup správne. Premenná current sa nastaví na prvý znak reťazca.

```
public Lexer(String input) {
    this.input = input;
    this.position = 0;
    if (input.length() > 0) {
        this.current = input.charAt(position);
    }
}
```

Hlavná metóda na získavanie ďalších tokenov zo vstupu je nextToken(). Táto metóda prechádza jednotlivé znaky reťazca, ignoruje biele znaky (pomocou metódy consume()) a postupne zisťuje, aký typ tokenu sa nachádza na aktuálnej pozícii. Ak sa jedná o číslo, lexer začne extrahovať hodnotu a vytvorí token typu NUMBER. Ak sa jedná o písmeno, lexer vytvorí token VARIABLE.

```
public Token nextToken() throws IOException {
    while (Character.isWhitespace(current)) {
        consume();
    }

    if (position >= input.length()) {
        return Token.EOF;
    }

    if (Character.isDigit(current)) {
        value = 0;
        while (Character.isDigit(current)) {
            value = value * 10 + (current - '0');
            consume();
        }
        return Token.NUMBER;
    }

    if (Character.isLetter(current)) {
        StringBuilder variable = new StringBuilder();
        while (Character.isLetter(current)) {
            variable.append((char) current);
            consume();
        }
        variableName = variable.toString();
        return Token.VARIABLE;
    }
```

Metóda rovnako pracuje aj s rozpoznávaním operátorov a zátvoriek, pokiaľ zaregistruje neznámy znak tak vyhodí výnimku, resp, vypíše na výstupe chybu a program ukončí. Kľúčovou metódou, ktorá slúži na prechod medzi znakmi v reťazci, je metóda consume(). Každým zavolaním sa posunie aktuálna pozícia o jeden znak dopredu a zároveň sa aktualizuje hodnota aktuálneho znaku. Ak niesme mimo rozsahu vstupu, current sa nastaví na aktuálny znak na pozícií. Ak prekročíme rozsah, current sa nastaví na -1, čo nám signalizuje koniec reťazca. Súčasťou lexera sú aj dve pomocné metódy getValue a getVariableName.

```
public void consume() {
    position++;
    if (position < input.length()) {
        current = input.charAt(position);
    } else {
        current = -1;
    }
}
```

Parser

Trieda Parser slúži na syntaktickú analýzu a vyhodnocovanie matematických výrazov. V tejto triede pracujeme s tokenmi, ktoré nám poskytuje lexer. Tento parser umožňuje analyzovať aritmetické výrazy obsahujúce operátory, čísla, zátvorky a premenné. V našom parseri pracujeme s premennými:

- **symbol**: aktuálny token, ktorý je spracovávaný parserom.
- **lexer**: inštancia triedy Lexer, ktorá poskytuje tokeny.
- **scanner**: používa sa na čítanie hodnôt pre premenné zo vstupu.

Po správnej inicializácií parsera sa postupne vykonáva pár hlavných metód triedy Parser. Každá z metód zabezpečuje spracovanie rôznych aritmetických operácií. Metóda statement() je vstupnou metódou pre náš parser, pretože volá metódu expr(). Metóda expr() spracúva celý aritmetický výraz. Prioritne má na starosti spracovanie súčtu a rozdielu, no začína analýzou term(), kde pracujeme s násobením a delením. Tento prístup nám zabezpečí hierarchiu aritmetických operácií a to prednosť napríklad násobenia pred sčítaním. Po tejto analýze v expr() pokračujeme kontrolou tokenov Plus či Minus. Pokiaľ sa vo výraze nachádzajú, vykoná sa príslušná operácia a pomocou consume() prejdeme na ďalší token.

```
public int expr() {
    int result = term();

    while (symbol == Token.PLUS || symbol == Token.MINUS) {
        Token op = symbol;
        consume();
        if (op == Token.PLUS) {
            result += term();
        } else if (op == Token.MINUS) {
            result -= term();
        }
    }
    return result;
}
```

Metóda term(), pracuje na podobnom princípe. Ako bolo spomenuté, jediný rozdiel je v práci s násobením a delením. Takže ak sa nájde token Multiple alebo Divide, vykoná sa opäť príslušná operácia a zavolá sa metóda consume(). Primárny rozdiel je v prvotnej analýze na začiatku metódy, kde voláme metódu nul().

Metóda nul() slúži na spracovanie znakov a tým pádom nám dokáže rozoznať či je daný znak číslo, premenná alebo výraz v zátvorkách. Na začiatku sa nastaví result na nulovú hodnotu a začína sa s analýzou. Pokiaľ je symbol identfikovaný ako číslo, zavcolá sa pomocná funkcia getValue() na získanie hodnoty a pokračuje sa na ďalší znak. Pokiaľ je znak vyhodnotený ako premenná, názov sa uloží pomocou metódy gatVariableName().

Pri tomto kroku pracujeme s pomocnou metódou getVariableValue(), ktorá zabezpečí pri vyhodnotení znaku ako premennej ďalši vstup používateľa pre získanie hodnoty konkrétnej premennej. V prípade identikovania znaku ľavej zátvorky sa zabezpečí čítanie znakov až po znak pravej zátvorky a celý reťazec je vyhodnotený ako expr().

```
public int nul() {
    int result = 0;

    if (symbol == Token.NUMBER) {
        result = lexer.getValue();
        consume();
    } else if (symbol == Token.VARIABLE) {
        String variableName = lexer.getVariableName();
        result = getVariableValue(variableName);
        consume();
    } else if (symbol == Token.LPAR) {
        consume();
        result = expr();
        match(Token.RPAR);
    } else {
        throw new CalculatorException("Neplatný token: " + symbol);
    }

    return result;
}
```

Naše pomocné metódy v triede parser sú spomínaná metóda getVariableValue(), pre získanie hondoty premennej, resp. potrebu vstupu používateľa. Ďalšou pomocnou metódou je match(), pomocou ktorej kontrolujeme očakávaný token a metóda consume(), ktorú poznáme už pri lexeri a ktorá slúži na prechod na ďalší token pri analýze.

Main

Trieda **Main** je vstupným bodom celej aplikácie. Jej úlohou je zabezpečiť interakciu s používateľom, načítať matematický výraz, spracovať ho a vypísať výsledok.

Na začiatku očakávame vstupnný výraz, s ktorým následne budeme pracovať a analyzovať ho. Pre správne fungovanie potrebujeme samozrejme inštancie tried Lexer a Parser. Tým zabezpečíme lexikálnu analýzu a spracovanie výrazu s vyhodnotením tokenov, operátorov. To nám pomôže pri následnom počítaní výrazu a následnom vyhodnotení pomocou premennej result kde sa uloží hodnota parser.statement(). Súčasťou triedy main je aj výnimka čítania chybného vstupu či chybné vyhodnotenie výrazu.

```java
public class Main {
    Run | Debug
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new InputStreamReader(System.in))) {

            System.out.println(x:"Zadajte matematický výraz:");

            String input = reader.readLine();

            Lexer lexer = new Lexer(input);
            Parser parser = new Parser(lexer);

            int result = parser.statement();
            System.out.println("Výsledok: " + result);

        } catch (IOException e) {
            System.out.println(x:"Chyba pri čítaní vstupu.");
        } catch (CalculatorException e) {
            System.out.println("Chyba: " + e.getMessage());
        }
    }
}
```

Testovanie programu

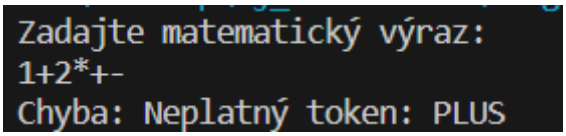
Testovanie programu je neoddeliteľnou súčasťou vývoja kvalitného softvéru. Pomáha zabezpečiť, že program správne vykonáva všetky požadované funkcie, odhalí možné chyby a poskytuje používateľovi správne výsledky. V prípade našej kalkulačky sa testovanie zameriava na rôzne typy vstupov, ktoré sú schopné pokryť široké spektrum scénarov a zabezpečiť, že program sa správne vysporiada s bežnými aj okrajovými prípadmi.

Prvý príklad testovania ukazuje správne vyhodnotenie jednoduchého výrazu, kde sa operátory rešpektujú v správnom poradí podľa prednosti: vstup 1+2*3 by mal dať výstup 7, pretože násobenie má prednosť pred sčítaním. Ďalší testovací príklad, (1+2)*3, ukazuje, že zátvorky správne určujú prioritu operácií, čím sa dosiahne správny výsledok 9. Takéto testy overujú, že kalkulačka správne implementuje aritmetické pravidlá a prednosť operátorov.

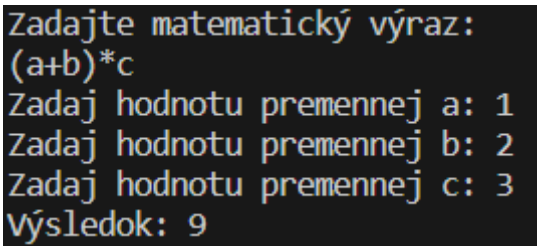
```
Zadajte matematický výraz:
1+2*3
Výsledok: 7
```

```
Zadajte matematický výraz:
(1+2)*3
Výsledok: 9
```

V ďalšom teste sa zadáva neplatný výraz 1+2*-+, ktorý je syntaxovou chybou. Očakávaný výstup je chyba, pretože výraz nie je platný podľa pravidiel syntaxe aritmetických operácií. Tento prípad testuje schopnosť programu zvládnuť nesprávne zadané výrazy a správne informovať používateľa o chybe. Testovanie na takýchto neplatných vstupoch je rovnako dôležité ako testovanie na správnych vstupoch, pretože pomáha zabezpečiť stabilitu a predvídateľnosť správania programu.



Posledný príklad testovania ukazuje použitie premenných v matematických výrazoch. Pri tomto vstupe sa používateľovi zobrazí výzva na zadanie hodnôt premenných. Pre príklad sme zvolili tri premenné a,b,c. Pre jednoduchú ukážku sme zvolili rovnaký výraz, ktorý bol prezentovaný vyššie. Tento prípad testuje schopnosť programu vykonávať viacero operácií v rámci jedného výrazu pri zohľadnení správnej priority operátorov.



Záver

V závere môžeme povedať, že testovanie nášho programu potvrdilo jeho správnu funkčnosť a schopnosť správne vykonávať všetky požadované operácie. Všetky testy, vrátane základných aritmetických operácií, zátvoriek, chýb v syntaxe a práce s premennými, prebehli úspešne a program sa zachoval stabilne vo všetkých scenároch. Zároveň sme sa uistili, že kalkulačka správne reaguje na platné aj neplatné vstupy, čím sme zabezpečili jej robustnosť a spoľahlivosť.

Splnili sme všetky požiadavky zadania a vytvorili program, ktorý nielenže poskytuje správne výsledky, ale tiež efektívne spracováva chyby a interaguje s používateľom. Tento projekt nám umožnil získať cenné skúsenosti v oblasti vývoja softvéru, prácu s lexikálnym analýzom, parserom a testovaním, čo sú kľúčové oblasti v tvorbe kvalitného kódu. Výsledkom je funkčný a spoľahlivý kalkulátor, ktorý je schopný zvládnuť rôzne matematické výrazy podľa špecifikovaných pravidiel.