NPRG041 – 2023/24 ZS – Cvičení MS – **Malý úkol C07**

Aritmetické výrazy II

V tomto domácím úkolu se opět budeme věnovat tématu aritmetických výrazů a práce s nimi. Konkrétně převezmeme veškerý zdrojový kód z minulého úkolu a aktuálně nabízenou funkcionalitu rozšíříme o možnost konstrukce našich výrazů ze vstupních řetězců v infixové notaci.

Podoba vstupu přesně odpovídá předpokladům, které jsme uvedli už minule. Pracujeme tedy s operacemi sčítání, odčítání, násobení a dělení. První dvě uvedené mají nižší precedenci než zbývající dvě, všechny pak jsou zleva asociativní. Kromě čísel (přirozená čísla nebo nula bez unárních znamének) se ve výrazu mohou vyskytovat i pomocné závorky, ovšem jen kulaté. Jednotlivá čísla, operátory i závorky se vyskytují bezprostředně za sebou, v řetězci se žádné oddělující mezery ani jiné bílé znaky nevyskytují.

Příkladem vstupního řetězce může být např. `10*2+3*((1+14)-18)-10`. Obecně můžeme předpokládat, že vstupní řetězce jsou validní (syntakticky dobře formované). Jak ale popíšeme později, určité nekorektní situace detekovat budeme.

Jádrem implementace nejspíše bude jakási třída parseru, který své rozhraní nabídne prostřednictvím statických funkcí. Jejich konkrétní podoba však není nijak předepsána. Aneb jediné, co je potřeba přesně dodržet, je přidání následujícího konstruktoru do našich současných výrazů `Expression`:

- `Expression(const std::string& input)`: vytvoří novou instanci aritmetického výrazu na základě parsování předaného vstupního řetězce v infixové notaci

Vstupní řetězec budeme logicky vnímat jako posloupnost tokenů, kterými mohou být čísla, symboly operátorů nebo symboly jednotlivých závorek. Pro zpracování těchto tokenů (aneb naparsování vstupního řetězce) pak použijeme algoritmus shunting-yard.

---

1 **foreach** token t ve vstupním infixovém výrazu **do**
2 **if** t je číslo **then**
3 vytvoř pro t nový listový uzel a vlož jej do zásobníku operandů
4 **else if** t je otevírací závorka `(` **then**
5 dej `(` na zásobník operátorů
6 **else if** t je zavírací závorka `)` **then**
7 **while** na vrcholu zásobníku operátorů je nějaký operátor o **do**
8 odeber o ze zásobníku operátorů
9 odeber dva uzly r a l ze zásobníku operandů
10 vytvoř pro o na základě l a r nový vnitřní uzel a vlož jej do zásobníku operandů
11 odeber `(` ze zásobníku operátorů
12 **else** t je operátor n
13 **while** na zásobníku operátorů je operátor o s precedencí vyšší než n nebo také stejnou, je-li ovšem n současně zleva asociativní **do**
14 odeber o ze zásobníku operátorů
15 odeber dva uzly r a l ze zásobníku operandů
16 vytvoř pro o na základě l a r nový vnitřní uzel a vlož jej do zásobníku operandů
17 přidej n na zásobník operátorů
18 **while** zásobník operátorů je neprázdný **do**
19 odeber o ze zásobníku operátorů
20 odeber dva uzly r a l ze zásobníku operandů
21 vytvoř pro o na základě l a r nový vnitřní uzel a vlož jej do zásobníku operandů

Pokud algoritmus proběhne úspěšně až do úplného konce, zůstane v používaném zásobníku operandů právě jeden uzel. Ten bude tvořit kořenový uzel celého našeho výrazu, a tedy jej stačí vzít a zapouzdřit do instance výrazu, který máme za úkol vytvořit.

Dodejme, že celý algoritmus má takovou vlastnost, že dokáže úspěšně zpracovat i určité vstupní výrazy, které ve skutečnosti validní nejsou. Jelikož navíc nejsme schopni všechny takové situace přímočaře detekovat, ani se o to pokoušet nebudeme. Na druhou stranu, jak už bylo naznačeno, v určitých situacích zjevných chyb ke správnému závěru dojít můžeme. A proto takové situace podchytíme a ošetříme.

Nejenom za tímto účelem navrhneme hierarchii tříd výjimek. Konkrétně předpokládáme třídu `Exception` a od ní odvozené varianty `ParsingException`, `MemoryException` a `EvaluationException`. Konstruktory nabídneme dva: `Exception(const std::string& message)` a `Exception(std::string&& message)`, kde v obou případech očekáváme textovou hlášku blíže popisující nastalou chybovou situaci. Z hlediska dalšího rozhraní už nabídneme jen metodu na získání této hlášky pomocí `const std::string& what() const`.

Pokud jde o konkrétní situace vyhazování výjimek a jejich textové hlášky, předpokládáme následující chování. Začněme výjimkou `ParsingException`, která má reagovat na nekorektní vstupní řetězce při jejich parsování.

- `Unknown token`: narazíme na nepovolený aneb neznámý token
- `Malformed number token`: nepodaří se nám korektně rozpoznat číselnou hodnotu, a to včetně jejího přetečení (uvažujeme typ `int`)
- `Missing operands`: nepodaří se nám sestavit uzel aktuální operace kvůli chybějícím operandům v zásobníku operandů
- `Unmatched closing parenthesis`: při zpracování uzavírací závorky nenajdeme v zásobníku operátorů odpovídající otevírací závorku
- `Unmatched opening parenthesis`: při závěrečném čištění zásobníku operátorů narazíme na dosud neuzavřenou otevírací kulatou závorku
- `Unused operands`: na samotném konci algoritmu zásobník operandů obsahuje více než jeden uzel
- `Empty expression`: nebo ve stejné situaci naopak žádný

Pamatovat ale musíme i na situace, kdy nebudeme mít dostatek paměti pro provedení dynamické alokace jednotlivých uzlů stromu. Takové případy detekujeme odchycením standardní výjimky `std::bad_alloc`. Namísto ní pak jen vyhodíme naši výjimku `MemoryException` s textovou hláškou `Unavailable memory`.

Ať už parsování selže z jakéhokoli výše uvedeného důvodu, nesmíme zapomenout na korektní dealokaci všech již úspěšně vytvořených uzlů. Jinak bychom nebyli schopni zajistit atomicitu a konzistenci celého algoritmu z hlediska správného využívání paměti. Jinými slovy by mohlo nekontrolovatelně a nenávratně docházet ke ztrátě paměti.

Nakonec ještě ošetříme situaci, kdy bychom se při vyhodnocování již zkonstruovaných výrazů pokoušeli dělit nulou. V takovém případě vyhodíme výjimku `EvaluationException` s hláškou `Division by zero`.

I tentokráte odevzdejte všechny vytvořené zdrojové soubory (`*.cpp` a `*.h`) kromě hlavního souboru `Main.cpp` s funkcí `main`. Ten předdefinovaný pak bude obsahovat direktivu `#include "Expression.h"`.

Cílem tohoto úkolu je především ověřit schopnost implementace složitějšího algoritmu dle zadaného pseudokódu. Pokud jde o technické prostředky, je potřeba demonstrovat použití kontejneru zásobníku, a to v podobě polymorfního kontejneru umožňujícího vkládat instance různých typů uzlů. Opět dodržte obvyklé požadavky na úkoly kladené. Z hlediska správné dekompozice nezapomeňte vyčlenit mechanismus na porovnávání precedencí operátorů, ať už jej realizujete jakkoli.

Jak již bylo naznačeno, samotnou implementaci shunting-yard algoritmu není vhodné umístit přímo do daného konstruktoru výrazu `Expression` aneb vyčleníme ji do samostatné třídy. Vzhledem k rozsahu si pak jistě zaslouží svůj vlastní modul aneb nezapomínejme členit náš kód do vhodných souborů. Dodejme také, že veškeré pomocné proměnné jako zásobník operátorů nebo operandů mají jen dočasný charakter. Nedává tedy smysl je uchovávat i po skončení procesu parsování, a tedy není vhodné je řešit jako datové položky třídy parseru.