NPRG041 – 2024/25 ZS – Cvičení MS – **Malý úkol C06**

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 vlastní hierarchii tříd výjimek. Konkrétně budeme uvažovat třídu `Exception` a od ní odvozené varianty `ParseException`, `MemoryException` a `EvaluationException`. Konstruktor nabídneme jediný, a to `Exception(const char* message)`. Jelikož budeme pracovat jen s fixními textovými hláškami blíže popisujícími nastalé chybové situace, předáme je pomocí céčkového řetězce. Z hlediska dalšího rozhraní už nabídneme jen metodu na získání této hlášky pomocí `const char* message() const`.

Pokud jde o konkrétní situace vyhazování výjimek a jejich textové hlášky, předpokládáme následující chování. Začněme výjimkami typu `ParseException`, které mají reagovat na nekorektní vstupní řetězce při jejich parsování.

- `Unknown token`: narazíme na nepovolený aneb neznámý token (např. `3+a`, `3+1a` nebo `3+a1`)
- `Overflow number`: nepodaří se nám korektně rozpoznat číselnou hodnotu kvůli jejímu přetečení
- `Missing operands`: nepodaří se nám sestavit uzel aktuální operace kvůli chybějícím operandům v zásobníku operandů (např. `1+`)
- `Unmatched closing parenthesis`: při zpracování uzavírací závorky chybí v zásobníku operátorů odpovídající otevírací závorka (např. `1+2)`)
- `Unmatched opening parenthesis`: při závěrečném čištění zásobníku operátorů narazíme na dosud neuzavřenou otevírací závorku (např. `(1+2`)
- `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`. Právě ta je vyvolána v situaci, kdy pokus o dynamickou alokaci pomocí operátoru `new` selže. Namísto ní už 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 (tzv. *memory leaks*).

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`. Současně se vyvarujeme opakovaného vyhodnocování pravého podstromu, nemusí totiž být triviální.

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"`. Skrze tento hlavičkový soubor tak opět musí být přímo či nepřímo dostupné vše potřebné.

Cílem tohoto úkolu je především ověřit schopnost implementace složitějšího algoritmu dle zadaného pseudokódu a korektní práce s dynamicky alokovanou pamětí ve spojitosti s objekty majícími netriviální životní cyklus a hlavně ošetřování chybových situací s cílem zabránit memory leakům. 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 dokonce i 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.

Poznamenejme opět, že naším cílem není vytvořit univerzální kód, pomocí kterého bychom byli schopni parsovat a zpracovávat výrazy libovolných typů. Všechna specifika aritmetických výrazů (operace, priority, asociativita atp.) tak do kódu opět můžeme vhodným způsobem napevno zadrátovat. Konkrétně pak není

rozumné parsovací algoritmus navrhnout jako cyklus nad obecnými tokeny, které bychom nejprve všechny rozpoznali, dočasně je uchovali v podobě instancí nějakých pomocných tříd a teprve pak je zpracovávali. Technické komplikace a nárůst objemu kódu by totiž nevyvážily přínosy. A stejně by takový postup nejspíše nebyl dostatečně obecný.

Zapomenout bychom zde naopak neměli na principy dekompozice aneb různé druhy tokenů nebo situací bychom měli zpracovávat pomocí vhodně navržených funkcí, díky nimž bychom měli dosáhnout přinejmenším oddělení high-level logiky celého parsovacího algoritmu od řešení jednotlivých low-level situací.

Pokud už nově díky parsování vstupních řetězců umíme vytvářet instance výrazů bezpečným způsobem, bylo by nejspíše vhodné znemožnit cílovým uživatelům nadále používat naše původní konstruktory výrazů a jednotlivých uzlů napřímo. Přestože takový postup je technicky relativně snadno proveditelný, všechny tyto původní konstruktory i tak ponecháme z důvodu testování veřejnými. A i nadále budeme předpokládat, že případné instance výrazů pomocí nich vytvořené budu korektní.

Jak už bylo řečeno, hlavním cílem úkolu je manuální řízení životního cyklu dynamicky alokovaných uzlů vnitřního stromu výrazu. Z tohoto důvodu je opět nutné využívat tradiční céčkové ukazatele na tyto uzly, nikoli unikátní chytré ukazatele `std::unique_ptr`.

Zatímco manuální dealokace už vytvořených uzlů a potažmo výrazů jako celků je poměrně přímočará a pokryli jsme ji návrhem vhodných destruktorů už v rámci minulého úkolu, zajištění korektní práce s dynamicky alokovanou pamětí v průběhu parsovacího algoritmu je řádově složitější výzvou. Zabránit vzniku memory leaků totiž musíme i v případě nejrůznějších krajních a chybových situací, které se mohou objevit v průběhu celého procesu, a v jejichž důsledku tedy tento proces ani neproběhne do úspěšného konce.

Konkrétně jde o všechny situace, v rámci kterých vyhazujeme už diskutované výjimky jakožto reakce na nekorektní vstupní řetězce nebo nedostatek paměti. Zejména u něj si musíme uvědomit, že k němu může dojít nejenom kvůli selhání námi přímo volaného operátoru `new` na nové uzly, ale také při používání standardních kontejnerů, které se bez dynamické alokace také neobejdou. Jinými slovy při používání zásobníku operátorů i zásobníku operandů, ale i při práci s řetězci typu `std::string`, a to i při jejich vytváření pomocí metody `substr`. Selhat může dokonce i vytváření prázdných instancí všech těchto kontejnerů.

Klíčovým aspektem tedy v první řadě je umět identifikovat všechna místa, kde k takovému selhání může dojít. Druhým klíčovým aspektem je postarat se o správný úklid aneb o manuální dealokaci všeho, co automaticky dealokováno nebude. Tím myslíme všechny instance uzlů, které už jsme zvládli vytvořit, ať už se stále ještě vyskytují v zásobníku operandů, nebo jsme je z něj už zvládli vyjmout. V případě jakéhokoli selhání tedy musíme být schopni z pohledu existujících uzlů provést jakýsi rollback.

Třetím klíčovým aspektem je pečlivé promyšlení míst, kde potřebné `try` / `catch` bloky budou umístěny. Cílem totiž je zabránit zbytečnému opakování kódu obsluhujícího naše kompenzační akce. Pomoci pak může přístup, kdy ne všechny problémy řešíme hned, ale necháváme je probublat až co nejvýše. V každém případě musíme zajistit, že parsovací metoda (v důsledku čehož i konstruktor třídy výrazu) opravdu vyřeší všechny problémy interně a navenek bude pouštět výhradně jen naše typy výjimek z hierarchie `Exception`.

Technicky budeme potřebovat formulovat `catch` bloky, kde budeme chtít odchytit libovolnou výjimku z této hierarchie. V takovém případě samozřejmě použijeme konstantní referenci, tedy `catch (const Exception& e) {...}`. Podstatné ale je, že takto odchycenou výjimku v případě potřeby opětovného vyhození nemůžeme vyhodit přes standardní kód `throw e;`, došlo by tím totiž k ořezání specializace. Namísto toho ji vyhodíme pomocí samotného `throw;`.