

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 odděluující 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 přesto detekovat a řešit budeme.

Vstupní řetězec budeme logicky vnímat jako posloupnost tokenů, kterými mohou být čísla, symboly operátorů nebo symboly otevíracích a zavíracích závorek. Pro zpracování těchto tokenů (aneb naparsování vstupního řetězce a vytvoření výsledného stromu výrazu) použijeme algoritmus shunting-yard popsáný v následujícím pseudokódu. Jeho základem je práce se dvěma zásobníky, jedním pro operátory (a také otevírací závorky) a druhým pro operandy (již vytvořené uzly stromu). Pro oba tyto zásobníky pak použijeme standardní kontejner `std::stack` z knihovny `<stack>`. Konkrétně v prvním případě `std::stack<char>` a ve druhém `std::stack<Node*>`.

```

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
14       ovšem n současně zleva asociativní do
15         odeber o ze zásobníku operátorů
16         odeber dva uzly r a l ze zásobníku operandů
17         vytvoř pro o na základě l a r nový vnitřní uzel a vlož jej do zásobníku operandů
18     přidej n na zásobník operátorů
19 while zásobník operátorů je neprázdný do
20   odeber o ze zásobníku operátorů
21   odeber dva uzly r a l ze zásobníku operandů
22   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 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.

Jediné předepsané rozhraní, které je nutné dodržet, spočívá v přidání následujícího konstruktora do třídy `Expression`. Jelikož vstupní řetězec nepotřebujeme uchovávat ani měnit, použijeme k jeho předání

typ `std::string_view` z knihovny `<string_view>`. Ten totiž umožňuje efektivní zapouzdření a unifikované zpracování céčkového řetězce i řetězce `std::string`, ve druhém případě navíc bez kopírování jeho obsahu.

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

Jelikož je celý parsovací algoritmus zjevně netriviální, nebylo by rozumné jej implementovat přímo uvnitř tohoto konstruktoru. Jádrem naší implementace tedy bude třída parseru, která celý proces parsování zapouzdří. Přestože její konkrétní podoba, rozhraní a dokonce ani název nijak předepsány nejsou, celý koncept řešení je nutné dodržet.

Nepochybně by se nabízelo implementovat všechny funkce této třídy jako statické, tedy bez záměru vytvářet jakékoli její instance (podobně jako u třídy počítadla ze třetího úkolu). S ohledem na požadovanou dekompozici alespoň na úrovni zpracování jednotlivých tokenů (přinejmenším kvůli oddělení high-level logiky celého parsovacího algoritmu od řešení jednotlivých low-level situací) takových funkcí ale bude poměrně hodně. Navíc bychom mezi nimi museli neustále předávat používané zásobníky, což by sice technicky jistě možné bylo, nebylo by to ale zrovna přívětivé.

Zvolíme tedy následující řešení. Uživatelům i tentokrát nabídneme veřejnou statickou funkci, např. v podobě `[[nodiscard]] static Node* parse(std::string_view input)`. Jejím úkolem bude realizovat celý proces parsování a vrátit získaný ukazatel na kořenový uzel. Uvnitř ní pomocí privátního konstruktoru vytvoříme instanci parseru, přičemž oba zásobníky schováme do jeho datových položek. Všechny ostatní pomocné metody pak tedy už budou moci být normální (myšleno nestatické), a zásobníky v nich tak budou přístupné přímo. Jelikož ale konstruktor bude privátní, uživatelé k němu nadále přístup mít nebudou.

Vysvětlíme ještě význam (dobrovolně) použitého atributu `[[nodiscard]]`. Jde o obecný mechanismus umožňující předávání dodatečných informací kompilátoru. Konkrétně tento atribut pak vynutí kontrolu, že návratová hodnota funkce není volajícím zcela ignorována. To samozřejmě nemůže automaticky zajistit korektnost programu, ale může snížit riziko případného opomenutí. Specificky v našem případě se to díky použití dynamické alokace doslova nabízí.

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 navrhujeme vlastní hierarchii tříd výjimek. Konkrétně začneme s базovou třídou `Exception` a od ní odvodíme varianty `ParseException`, `MemoryException` a `EvaluationException`. Konstruktor nabídneme jediný, a to `Exception(const char* message)`. Jelikož budeme pracovat jen s fixními textovými hláškami (definovanými jako konstantní výrazy) blíže popisujícími nastalé chybové situace, předáme je pomocí céčkového řetězce (a jejich vlastníky se nestáváme). 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. Začneme 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 (včetně ošetřování chybových situací kvůli zabránění *memory leaků*). To znamená, že je opět nutné využívat tradiční céčkové ukazatele na naše uzly, nikoli unikátní chytré ukazatele `std::unique_ptr`. 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.

Poznamenejme, že naším záměrem není vytvořit univerzální kód, pomocí kterého bychom byli schopni parsovat a zpracovávat výrazy libovolných forem. 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.

Zatímco manuální dealokace už vytvořených uzlů a potažmo výrazů jako celků je poměrně přímočará a pokryli jsme ji návrhem vhodných destruktůrů 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 nejrozumnějších krajních a chybových situacích, 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 řetězcem typu `std::string`, a to i při jejich vytváření (konstruktorem nebo i metodou `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.

Pro naplnění obou aspektů je potřeba pečlivé umístění potřebných `try / catch` bloků. Cílem totiž je zajistit nejenom korektnost, ale i zabránit zbytečnému opakování kódu obsluhujícího naše kompenzační akce. Pomoci pak může přístup, kdy ne všechny problémy řešíme hned, ale necháváme je záměrně probublávat výše až na vhodnější místa. V každém případě musíme zajistit, že parsovací metoda opravdu vyřeší všechny problémy interně a navenek bude pouštět výhradně jen naše typy výjimek z hierarchie `Exception`. To konkrétně znamená, že jiné výjimky (jako např. `std::bad_alloc`) se nesmí dostat už ani do konstruktůrů třídy `Expression`.

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

Na úplný závěr zdůrazněme, že v rámci testů v `ReCodExu` bohužel není možné otestovat opravdu všechna místa, kde k selhání diskutované dynamické alokace může dojít. Vše tedy před odevzdáním finální verze důkladně promyslete a velmi, opravdu velmi pečlivě zkontrolujte.