EKONOMICKÁ UNIVERZITA V BRATISLAVE

FAKULTA HOSPODÁRSKEJ INFORMATIKY

Evidenčné číslo: 103004/I/2014/2482918026

Interpreter infixových aritmetických výrazov vytvorený v jazyku C++ pomocou syntaktického binárneho stromu

Diplomová práca

2014 **Róbert Lenčéš**

EKONOMICKÁ UNIVERZITA V BRATISLAVE

FAKULTA HOSPODÁRSKEJ INFORMATIKY

Interpreter infixových aritmetických výrazov vytvorený v jazyku C++ pomocou syntaktického binárneho stromu

Diplomová práca

Študijný program: Manažérske rozhodovanie a informačné technológie

Študijný odbor: 6258 Kvantitatívne metódy v ekonómii

Školiace pracovisko: Katedra aplikovanej informatiky

Vedúci záverečnej práce: Ing. Igor Košťál, PhD.

Bratislava 2014 Róbert Lenčéš

Čestné vyhlásenie

Čestne vyhlasujem, že záverečnú prácu som vypracoval samostatne a že som uviedol všetku použitú literatúru.

Močenok, 24. 4. 2014 .....

Róbert Lenčéš

Poďakovanie

Touto cestou by som chcel poďakovať Ing. Igorovi Košťálovi, PhD. za odborné rady a pripomienky, ktoré mi poskytol počas spracovávania diplomovej práce.

ABSTRAKT

LENČÉŠ, Róbert: *Interpreter infixových aritmetických výrazov vytvorený v jazyku C++ pomocou syntaktického binárneho stromu.* – Ekonomická univerzita v Bratislave. Fakulta hospodárskej informatiky; Katedra aplikovanej informatiky. – Vedúci záverečnej práce: Ing. Igor Košťál, PhD. – Bratislava: FHI EU, 2014, 63 s.

Cieľom záverečnej práce je implementovanie interpretera infixových aritmetických výrazov pomocou syntaktického binárneho stromu. Práca je rozdelená do troch kapitol a obsahuje 23 ilustrácií. Prvá kapitola pojednáva o pojme abstraktných dátových typov v informatike, ich dôležitých vlastnostiach a výhodách a o význame ich použitia. Ďalej rozoberá druhy najpoužívanejších abstraktných dátových typov pri programovaní súčasných aplikácií, pričom sa zameriava najmä na binárne stromy. Skúma ich dôležité charakteristiky, matematické vlastnosti, metódy prechodu uzlami stromu a spôsoby alokácie binárnych stromov v operačnej pamäti počítača. Prvá kapitola je zakončená rozborom aritmetických výrazov a rôznych notácií ich zápisu. Druhá kapitola obsahuje hlavný cieľ práce, postup spracovávania záverečnej práce a konkrétnejší popis jej jednotlivých podkapitol. Záverečná kapitola diplomovej práce sa zaoberá podrobnou charakteristikou jednotlivých krokov implementácie interpretera infixových aritmetických výrazov v jazyku C++. Na jednotlivých objektoch aplikácie rozoberá základné vlastnosti objektovo orientovaného programovania. Analyzuje základné formy zápisu aritmetických výrazov a rozoberá konverzný algoritmus na prevod výrazu z infixovej do postfixovej notácie. Posledná časť tretej kapitoly je venovaná popisu implementácie klienta, pomocou ktorého sú demonštrované výsledky diplomovej práce.

Kľúčové slová:

Abstraktný dátový typ, Lineárne zoznamy, Binárne stromy, Aritmetické výrazy, C++

ABSTRACT

LENČÉŠ, Róbert: *Interpreter of infix arithmetic expressions created in C++ language with the use of syntactic binary tree.* – University of Economics in Bratislava. Faculty of Economic Informatics; Department of Applied Informatics. – Thesis instructor: Ing. Igor Košťál, PhD. – Bratislava: FHI EU, 2014, 63 p.

Aim of the thesis is to implement interpreter of infix arithmetic expressions using the syntax binary tree. The thesis is divided into three chapters and contains 23 illustrations. The first chapter discusses the concept of abstract data types in computer science, their important characteristics and advantages and the importance of their use. It analyzes the most widely used kinds of abstract data types in the programming of present applications, focusing in particular on binary trees. It examines their important characteristics, mathematical properties, methods of transition nodes of the tree and methods for allocation of binary trees in the main memory of the computer. The first chapter is finished analysis of arithmetic expressions and their various notations. The second chapter contains the main objective, the procedure of processing of the thesis and a more specific description of the various subchapters. The final deals with the detailed characteristics of the various stages of the implementation interpreter infix arithmetic expressions in C++. There are analyzing basic features of object-oriented programming to individual application objects. Analyzes the basic forms of writing arithmetic expressions and discusses the conversion algorithm to convert expression from infix to postfix notation. The last part of the third chapter is devoted to a description of the implementation of the client, which is demonstrated through results of the thesis.

Key words:

Abstract data type, Linked lists, Binary trees, Arithmetic expressions, C++

Obsah

Zoznam ilustrácií a tabuliek **8**

Úvod **9**

1. Súčasný stav riešenej problematiky doma a v zahraničí **10**

1.1. Abstraktný dátový typ 10

1.2. Rozhranie ako prístupový bod k ADT 11

1.3. Lineárne zoznamy 12

1.3.1. Zásobník *13*

1.3.2. Fronty a obojstranné fronty *14*

1.3.3. Spôsoby alokácie lineárnych zoznamov *15*

1.4. Stromy 17

1.5. Binárne stromy 19

1.5.1. Prechod binárnym stromom *20*

1.5.2. Základné matematické vlastnosti binárnych stromov *21*

1.6. Počítačové spracovanie aritmetických výrazov 23

2. Cieľ práce, metodika práce a metódy skúmania **24**

3. Výsledky práce a diskusia **25**

3.1. Základná charakteristika aplikácie 25

3.2. Spracovanie aritmetického výrazu 27

3.3. Štruktúra pracovných objektov interpretera 29

3.4. Vstupný bod aplikácie - Listener 30

3.5. Lexikálny analyzátor – Scanner 32

3.6. Štruktúra dátových objektov binárneho stromu 40

3.7. Syntaktický analyzátor – Parser 47

3.7.1. Nerekurzívna stavba stromu *52*

3.8. Vyhodnocovač – Evaluator 54

3.9. Jednoduchý HTML klient ako konzument aplikácie 59

Záver **61**

Zoznam použitej literatúry **63**

Zoznam ilustrácií a tabuliek

Obrázok 1: Abstrakcia systému auta 10
Obrázok 2: Abstraktný dátový typ 11
Obrázok 3: Zásobník 13
Obrázok 4: Fronta 14
Obrázok 5: Obojstranná fronta 14
Obrázok 6: Sekvenčná alokácia lineárneho zoznamu 15
Obrázok 7: Spojová alokácia lineárneho zoznamu 15
Obrázok 8: Kruhový obojsmerný prepojený lineárny zoznam 16
Obrázok 9: Strom 17
Obrázok 10: Binárny strom 19
Obrázok 11: Trojito prepojený strom 19
Obrázok 12: Štruktúra pracovných objektov aplikácie (zdroj: [5]) 29
Obrázok 13: Trieda Listener 30
Obrázok 14: Trieda Scanner 32
Obrázok 15: Štruktúra Token 33
Obrázok 16: Diagram tried uzlov stromu 41
Obrázok 17: Syntaktický analyzátor - Parser 47
Obrázok 18: Vyhodnocovač - Evaluator 54
Obrázok 19: Objekt Leaf 56
Obrázok 20: Úvodná webová stránka klientskej aplikácie 59
Obrázok 21: Chybová správa 59
Obrázok 22: Vyhodnotený binárny strom 60
Obrázok 23: Zbaľovanie uzlov stromu 60

Úvod

Počítačové programy spočiatku pracovali iba s jednoduchými dátovými typmi, medzi ktorými sa postupne vytvárali zložitejšie štrukturálne vzťahy. Vznikali tak rôzne dátové štruktúry pre uchovávanie a prácu s rozsiahlymi dátami. V súčasnosti sú už v takmer každom vývojovom prostredí zabudované knižnice pre podporu objektových štruktúr. Medzi najčastejšie využívané informačné štruktúry dnes patria rôzne druhy abstraktných dátových typov (ADT), ako je napríklad zásobník, fronta, obojstranná fronta, stromy a iné.

V prvej časti diplomovej práce sa budeme zaoberať objasnením rôznych abstraktných dátových typov, princípom ich fungovania a významom použitia ADT pri vytváraní počítačových aplikácií. Ďalej rozoberieme druhy najpoužívanejších abstraktných dátových typov v súčasnosti. Podstatnú časť prvej časti práce venujeme dátovému typu strom a jeho najbežnejšej reprezentácii - binárnemu stromu. Preskúmame jednotlivé prvky binárneho stromu a vzťahy medzi nimi. Pozrieme sa na jednotlivé spôsoby alokácie binárneho stromu v operačnej pamäti počítača. Ďalej preskúmame rôzne typy prechodov cez uzly binárneho stromu a niektoré základné matematické vlastnosti binárnych stromov.

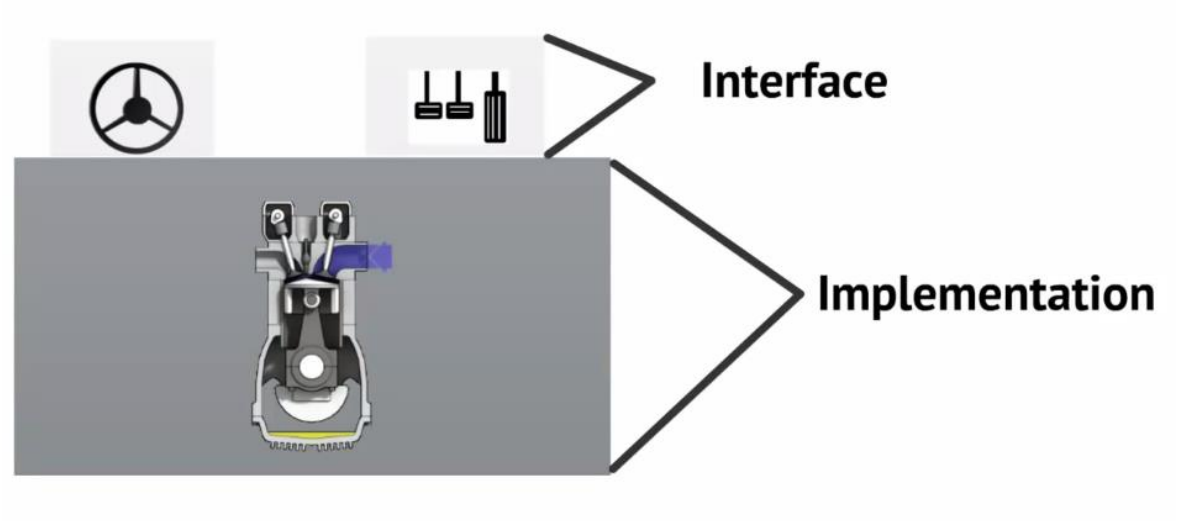
V druhej časti práce budeme na základe získaných vedomostí z teoretickej časti implementovať syntaktický binárny strom. Vytvorená aplikácia bude slúžiť ako interpreter infixových aritmetických výrazov. Aritmetickým výrazom sa v matematike rozumie výraz, ktorý sa skladá z ľubovoľného počtu číslic a operátorov medzi nimi. Povolenými operátormi sú sčítanie (+), odčítanie (-), násobenie (*) a delenie (/). Operátory sčítania a odčítania majú nižšiu prioritu ako operátory násobenia a delenia, priorita operácií však môže byť zvýšená pridaním zátvorky, pretože výraz v zátvorke má vždy najvyššiu prioritu. Ďalej sa budeme zaoberať spôsobom reprezentácie aritmetických výrazov a ich konverziou z infixovej notácie do postfixovej. Binárny strom budeme implementovať jazyku C++, ktorý je v súčasnosti veľmi obľúbeným objektovo orientovaným programovacím jazykom. Využijeme pritom princípy najnovšieho medzinárodného štandardu C++11, ktorý pridáva tomuto jazyku množstvo užitočných vlastností. Vytvorená aplikácia bude slúžiť ako server, ktorý bude prijímať požiadavky (aritmetické výrazy) od klientov a následne im bude posielať vyhodnotený binárny strom v textovom formáte JSON. V rámci práce vytvoríme aj jednoduchého HTML klienta, ktorý poslúži na zadanie a odoslanie aritmetického výrazu, zobrazenie výsledku a vykreslenie vyhodnoteného binárneho stromu.

1. Súčasný stav riešenej problematiky doma a v zahraničí

1.1. Abstraktný dátový typ

Abstraktné dátové typy (ADT) sú základnými elementmi pre vytváranie aplikácií v každom programovacom jazyku. Uľahčujú a najmä zefektívňujú prácu programátorov tým, že zovšeobecňujú konkrétne prvky systému. Vývojári sa potom už nemusia zaoberať podrobnosťami skúmaného systému, čo ich prácu podstatne urýchľuje a zefektívňuje.

Jednoduchým príkladom abstrakcie objektu reálneho sveta by mohlo byť auto. V súčasnosti každé auto disponuje okrem iného aj volantom a pedálovým ústrojenstvom, ktoré sú nevyhnutne potrebné k jeho zmysluplnému využitiu. Každý zo šoférov dobre vie, že volantom sa určuje smer pohybu auta a jednotlivými pedálmi sa riadi jeho rýchlosť. Väčšina z nich však nemá ani poňatia o tom, čo všetko sa skrýva za týmito „funkciami“ a v podstate ich to ani nemusí zaujímať.



Obrázok 1: Abstrakcia systému auta[1]

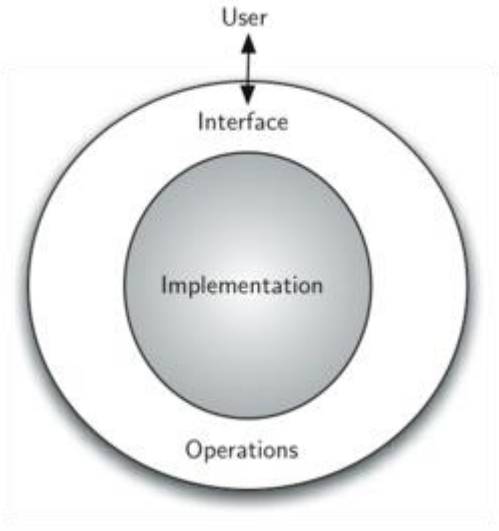
Detailné zostavenie auta – implementáciu, najlepšie pozná jeho výrobca. Toto zostavenie môže ľubovoľne meniť, napr. zavádzaním nových technológií, pričom na vodičské vedomosti používateľa auta to nebude mať žiadny vplyv, pretože rozhranie zostáva stále to isté. Volantom sa bude i naďalej udávať smer jazdy, brzdou sa bude spomaľovať a plynovým pedálom zrýchľovať aj po tom, ako sa „niečo“ v zhotovení auta zmenilo.

[1] Zdroj: http://www.youtube.com/watch?v=HcxqzYsiJ3k

1.2. Rozhranie ako prístupový bod k ADT

Najdôležitejšou vlastnosťou abstraktného dátového typu je skutočnosť, že všetky jeho prvky sú prístupné iba prostredníctvom poskytnutého rozhrania. Rozhranie je nepriepustné, to znamená, že používateľ cez neho nemôže vidieť implementáciu ADT. Povolené operácie na objekte ADT sú iba tie, ktoré poskytuje jeho rozhranie.

Dôvod tejto prísnej reštrikcie je veľmi jednoduchý. Ak by sme klientom (konzumentom ADT) sprístupnili implementáciu a potom by sme z nejakého dôvodu museli zmeniť reprezentáciu dát objektu, museli by sme upraviť všetky klientske aplikácie, ktoré daný objekt využívajú. Obmedzením prístupu k ADT iba prostredníctvom rozhrania zabezpečíme, že môžeme vnútornú implementáciu meniť podľa potreby bez vplyvu na klientske programy.



Obrázok 2: Abstraktný dátový typ[2]

Obrázok 2 znázorňuje interakciu používateľa s ADT. Používateľ komunikuje s abstraktným dátovým typom cez rozhranie, ktoré je tvorené vopred špecifikovanými operáciami. Vnútorná implementácia objektu pôsobí ako čierna skrinka. Používateľ disponuje informáciami o potrebných vstupných parametroch operácií a taktiež o návratových typoch jednotlivých funkcií. Samotná logika spracovania vstupných dát, ich ukladania a vyhodnotenia však zostáva pre používateľa ukrytá.

[2] Zdroj: http://interactivepython.org/runestone/static/pythonds/Introduction/introduction.html

1.3. Lineárne zoznamy

V súčasnosti najčastejšie využívané abstraktné dátové typy sú založené na lineárnom zozname. Lineárny zoznam je definovaný ako postupnosť n uzlov x_1, $x_{2.\ ---.}$ x_n, kde $n \geq 0$, x_1 reprezentuje prvý uzol zoznamu a x_n posledný uzol. Ďalej predpokladajme číslo k, pričom $1 < k <$ n. Potom platí, že prvok $x_{(k-1)}$ predchádza prvku x_k a prvok $x_{(k+1)}$ za ním nasleduje.

Nad definovaným lineárnym zoznamom môžeme vykonávať napríklad tieto operácie: [1]

- Získať prístup ku *k*-tému uzlu zoznamu a prečítať a/alebo zmeniť obsah jeho polí.
- Vložiť nový uzol tesne pred alebo za *k*-ty uzol.
- Odstrániť *k*-ty uzol.
- Zlúčiť dva alebo viac lineárnych zoznamov do jediného.
- Rozdeliť jeden lineárny zoznam do dvoch alebo viacerých zoznamov.
- Vytvoriť kópiu lineárneho zoznamu.
- Zistiť počet uzlov v zozname.
- Zoradiť uzly v zozname do vzostupného poradia podľa istých polí uzlov.
- Vyhľadať v zozname výskyt uzlu, ktorý v istom poli obsahuje určitú konkrétnu hodnotu.

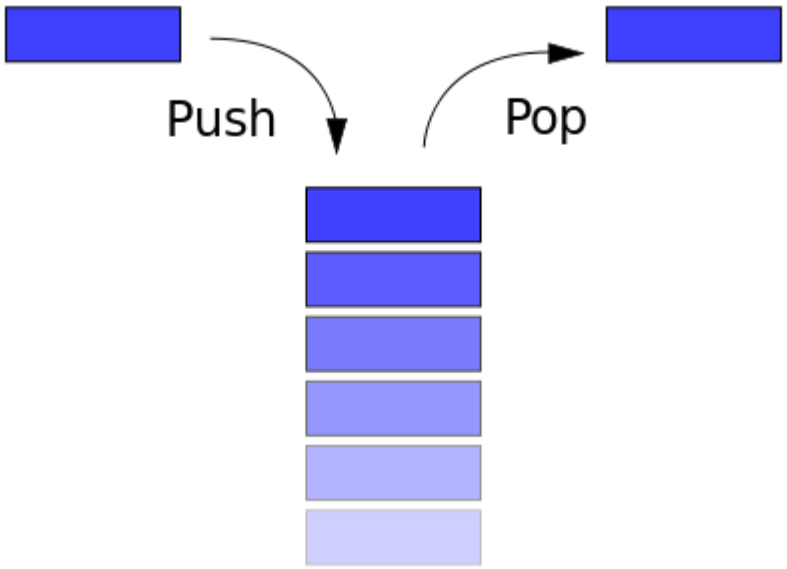
V nasledujúcich sekciách bližšie rozoberieme niektoré reprezentácie lineárneho zoznamu, pričom vychádzame zo skutočnosti, že konkrétny ADT nevyžaduje implementáciu všetkých vyššie uvedených operácií. Medzi najčastejšie využívané operácie patrí vloženie novej položky do zoznamu, odstránenie položky zo zoznamu a prístup ku konkrétnej položke. Podľa toho, akým spôsobom sú tieto operácie vykonávané, rozlišujeme: [1]

a) **Zásobník** (stack) – vkladanie, odstraňovanie a obvykle aj prístup prebieha iba na jednom konci zoznamu.
b) **Fronta** (queue) – vkladanie prebieha na jednom konci zoznamu, odstraňovanie a obvykle aj prístup prebieha na opačnom konci zoznamu.
c) **Obojstranná fronta** („double-ended-queue“ alebo deque) – vkladanie, odstraňovanie a obvykle aj prístup prebieha na oboch koncoch zoznamu.

1.3.1. Zásobník

Jedným z najznámejších abstraktných dátových typov v každom programovacom jazyku je zásobník (stack). Zásobník predstavuje kontajner, ktorý obsahuje niekoľko položiek. Jednotlivé položky môžu byť pridávané iba na vrchol zásobníka a odoberané iba z vrcholu zásobníka. V počítačových programoch je to väčšinou zásobník, ktorého metódy sa využívajú pri riadení prílivu a odlivu automatických premenných v operačnej pamäti. Zásobník je charakterizovaný pomocou operácií, ktoré je možné na ňom vykonať: [2]

- Môžeme vytvoriť nový zásobník.
- Môžeme pridať položku na vrchol zásobníka (operácia Push).
- Môžeme odobrať položku z vrcholu zásobníka (operácia Pop).
- Môžeme zistiť, či je zásobník plný.
- Môžeme zistiť, či je zásobník prázdny.



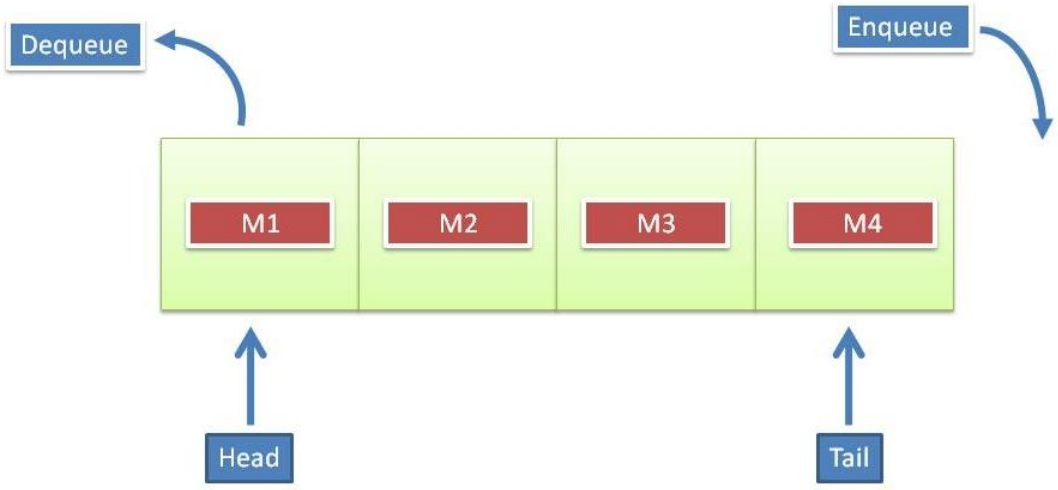
Obrázok 3: Zásobník[3]

Dno zásobníka tvorí posledná dostupná položka, ktorá môže byť odobratá iba v prípade, ak sa už v zásobníku žiadna iná položka nenachádza. Zásobník vo väčšine prípadov implementuje aj ďalšie užitočné operácie, ako je napríklad zistenie položky, ktorá sa nachádza na vrchole zoznamu (top).

[3] Zdroj: http://en.wikipedia.org/wiki/Stack_(abstract_data_type)

1.3.2. Fronty a obojstranné fronty

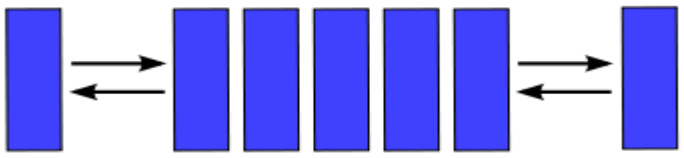
Fronta (queue), podobne ako zásobník, predstavuje kontajner, ktorý obsahuje zoradenú postupnosť prvkov. Nové položky sú taktiež pridávané na koniec fronty, odoberané sú však iným spôsobom – zo začiatku fronty. Fronta využíva metódu FIFO (First In First Out) pre narábanie s jej prvkami, naproti tomu zásobník je riadený metódou LIFO (Last In First Out) pri práci s jeho prvkami.



Obrázok 4: Fronta[4]

Rozhranie každej fronty musí implementovať dve základné operácie – vloženie novej položky (Enqueue) a výmaz najstaršej vloženej položky (Dequeue). Položky do fronty vstupujú na konci (tail) a vystupujú až v okamihu, keď sa dostanú do pozície čela (head).

Obojstranná fronta (double-ended-queue) je špeciálny typ fronty, ktorá má pravý a ľavý koniec. Položky môžu byť vkladané na ľubovoľnom konci fronty a odoberané môžu byť takisto z jej ľubovoľného konca.



Obrázok 5: Obojstranná fronta

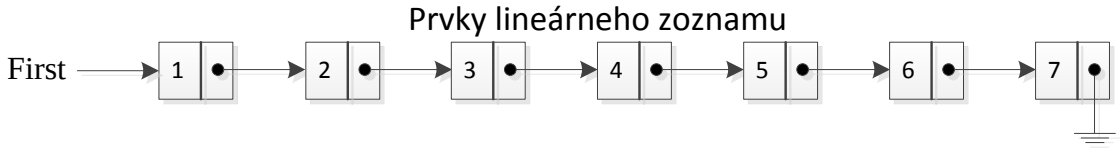
[4] Zdroj: http://riaanhanekom.com/blog/2009/08/04/ngenerics-overview-the-priority-queue/

1.3.3. Spôsoby alokácie lineárnych zoznamov

V tejto podkapitole sa budeme zaoberať dvoma základnými možnosťami alokácie lineárnych zoznamov v operačnej pamäti počítača. Prvým a najjednoduchším spôsobom uloženia lineárneho zoznamu je **sekvenčná alokácia.** Jedná sa o prirodzený spôsob ukladania položiek zoznamu do po sebe nasledujúcich pozícií operačnej pamäte jedna za druhou. Sekvenčná alokácia je implementovaná ako dynamicky vytvorené pole s vopred stanoveným maximálnym počtom jeho prvkov. Tento počet však nemusí byť konečný, pretože pole môžeme počas behu programu realokovať, teda zväčšovať, resp. zmenšovať jeho kapacitu.

Obrázok 6: Sekvenčná alokácia lineárneho zoznamu

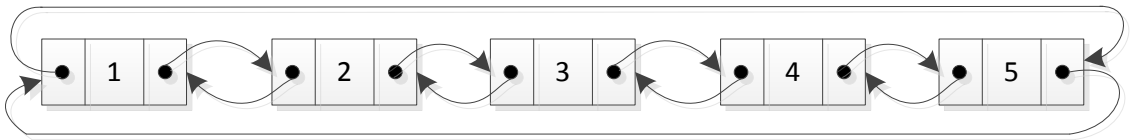
Nevýhodou sekvenčnej alokácie lineárneho zoznamu je už spomenutá nutnosť vopred určiť maximálny počet prvkov, ktoré budeme do kontajnera ukladať. Ak by sme alokovali pole s neprimerane vysokou kapacitou, spôsobili by sme zbytočné zabratie pamäte počítača, ktorú daná implementácia nikdy nevyužije. Na druhej strane, v prípade zvolenia príliš nízkej kapacity poľa sa dopúšťame rizika, že aplikácii sa nedostane dostatočné množstvo operačnej pamäte potrebnej na splnenie svojej úlohy. Ako sme už spomenuli, v prípade nedostatku pamäťového miesta môžeme veľkosť zoznamu rozšíriť, takáto operácia však celkom iste prinesie určité výkonnostné náklady. Flexibilnejším spôsobom ukladania dát lineárneho zoznamu v pamäti počítača je **spojová alokácia.** Spojová alokácia predstavuje schému, pri ktorej každý uzol obsahuje odkaz – ukazovateľ na ďalší prvok zoznamu. Z toho vyplýva, že jednotlivé prvky zoznamu už nemusia byť v operačnej pamäti uložené sekvenčne za sebou, ale môžu sa nachádzať na ľubovoľných miestach. K nasledujúcemu uzlu pristupujeme prostredníctvom ukazovateľa predchádzajúceho uzla.



Obrázok 7: Spojová alokácia lineárneho zoznamu

Na obrázku 7 vidíme spojovo alokovaný lineárny zoznam, ktorý je reprezentovaný ukazovateľom *First*. Tento ukazovateľ odkazuje na prvú položku zoznamu. Prístupom k tejto položke sa dostaneme k ďalšiemu ukazovateľu, ktorý ďalej odkazuje na nasledujúci prvok. Touto cestou by sme sa postupne dostali až k poslednému uzlu, ktorého ukazovateľ už neodkazuje na žiadny objekt. Nevýhodou spojového zoznamu je ťažší prístup ku konkrétnemu prvku zoznamu, ako to bolo pri sekvenčnom zozname, kde ku daným prvkom môžeme pristupovať jednoducho prostredníctvom indexov.

Špeciálnym typom spojovej alokácie sú **kruhové zoznamy**, pri ktorých posledný uzol zoznamu odkazuje späť na prvý uzol. Ak teda začneme prechádzať lineárny zoznam v ľubovoľnom bode, postupne sa dostaneme ku všetkým uzlom. Takéto zoznamy nemajú prvý ani posledný uzol. Ďalej môžeme definovať **obojsmerné prepojené zoznamy,** ktoré zabezpečujú vyššiu flexibilitu pri manipulácii s lineárnou postupnosťou prvkov tým, že každému uzlu priradíme až dva odkazy, pričom prvý odkaz smeruje na predchádzajúci prvok a druhý odkaz na nasledujúci prvok zoznamu. Ak navyše posledný prvok nasmerujme späť na prvý a prvý na posledný, dostaneme kruhový obojsmerný prepojený lineárny zoznam, ako to znázorňuje nasledujúci obrázok.



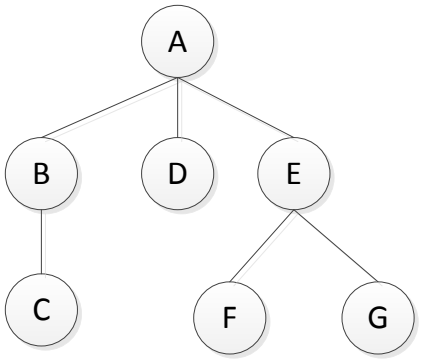
Obrázok 8: Kruhový obojsmerný prepojený lineárny zoznam

Jednotlivé typy alokácie pamäťového miesta počítača sa líšia svojimi výkonnostnými charakteristikami a náročnosťou implementácie. Niekedy môže byť ťažké rozhodnúť sa pre správny typ implementácie ADT. Avšak, ako sme už spomenuli, výhodou abstraktných dátových typov je skutočnosť, že môžeme meniť ich vnútornú implementáciu, teda môžeme nahradiť jeden typ alokácie druhým bez akéhokoľvek vplyvu na klientsky program.

1.4. Stromy

V predchádzajúcej časti sme sa venovali abstraktným dátovým typom založeným na lineárnom zozname. V počítačových algoritmoch sa však dosť často používajú aj nelineárne objektové štruktúry, z ktorých najznámejšie sú stromy. Strom predstavuje dátovú štruktúru tvorenú z uzlov - listov, medzi ktorými sú definované vzťahy vetvenia. Formálne môžeme strom definovať ako konečnú množinu T jedného alebo viacerých uzlov takých, že: [1]

a) existuje práve jeden špeciálne určený uzol nazývaný *koreň* stromu, root(T);
b) ostatné uzly (okrem samotného koreňa) sú rozdelené do m $\geq$ 0 disjunktných podmnožín T_1, ..., T_m, z ktorých každá je opäť stromom; stromy T_1, ..., T_m sa nazývajú *podstromy* daného koreňa.



Obrázok 9: Strom

Na obrázku 9 môžeme vidieť strom so siedmimi uzlami. Počet podstromov daného uzlu sa nazýva *stupeň uzla*. Koreňom stromu je uzol A so stupňom 3 a s podstromami {B,C}, {D} a {E,F,G}. Strom {E,F,G} má koreň {E} so stupňom 2. Koncovými uzlami stromu sú C, D, F a G so stupňom 0. Ďalej môžeme preskúmať úroveň každého uzla. Koreňový uzol A má úroveň 0, uzly B, D a E majú úroveň 1 a uzly C, F a G úroveň 2. Môžeme si všimnúť, že úroveň uzla je vzhľadom ku stromu zadaná rekurzívne. Výška stromu je definovaná maximálnou úrovňou uzlov stromu. Dĺžka cesty je daná súčtom úrovní všetkých uzlov stromu. Ďalej môžeme rozlišovať dĺžku vnútornej cesty a dĺžku vonkajšej cesty stromu.

Organizácia súborov v počítači je najbežnejším príkladom aplikácie stromovej štruktúry. Každý súbor je uložený v konkrétnom adresári. Koreňovým adresárom (uzlom) môže byť napríklad disk C[5]. Takýto uzol tiež nazývame *rodičom* všetkých svojich podstromov a zároveň ide o uzol, ktorý nemá žiadneho rodiča. Jednotlivé podstromy koreňového uzla sa zvyknú nazývať *súrodencami*, čo vyplýva z ich príbuzenského vzťahu medzi sebou navzájom. Zároveň sa nazývajú aj *potomkami* svojho rodiča. Každý uzol môže mať najviac jedného rodiča. Ďalej sa v literatúre často stretávame s pojmami *predchodca* a *nasledovník.* Na obrázku 9 môžeme uzly B a A označiť ako predchodcov uzla C a uzly E, F a G ako nasledovníkov uzla A. Ďalej sa v názvosloví stromových štruktúr stretávame s nasledujúcimi pojmami: [3]

- **vrchol –** jednoduchý objekt, uzol, ktorý môže mať meno a niesť ďalšie pridružené informácie,
- **hrana –** spojnica medzi dvoma vrcholmi,
- **cesta –** je určitý zoznam vrcholov, v ktorom sú jednotlivé susedné vrcholy prepojené hranami stromu.

Medzi koreňom a každým iným uzlom stromu existuje práve jedna cesta. Hrany stromu sú neorientované, čo znamená, že nie je určený ich smer. Uzly bez potomkov sa nazývajú *listy*, *koncové uzly* a tiež *vonkajšie uzly*. Naopak, uzly s najmenej jedným potomkom nazývame *nekoncové* alebo *vnútorné uzly.* Ďalej môžeme rozlíšiť:

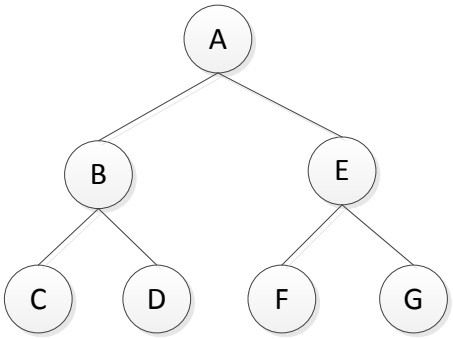
- **Usporiadaný strom –** je koreňový strom, kde záleží na vzájomnom poradí jeho podstromov. Ide o prirodzenú reprezentáciu stromu.
- **Neusporiadaný strom –** nezáleží na vzájomnom poradí podstromov.

Strom môžeme zakresliť rôznymi spôsobmi, v závislosti na umiestnení koreňa. Najčastejšie kreslíme strom s koreňom navrchu. Môžeme ho však umiestniť aj naspodok a potom jednotlivé uzly zakreslíme nad seba (ako prirodzený strom v prírode). Iným spôsobom zakreslenia stromu môže byť zľava – doprava, kde uzol stromu umiestnime naľavo a ďalšie uzly pripájame postupne ďalej doprava. Takýto strom, najmä ak má veľa listov, je vizuálne ľahšie čitateľný, pretože rovnakým spôsobom čítame napríklad aj text v knihách. Pri vykresľovaní stromu teda nezáleží na použitom spôsobe zápisu jeho uzlov, je ale veľmi dôležité správne zakresliť hrany a cesty týchto uzlov.

[5] V súborovej organizácii počítača sa vedľa disku C často vyskytujú aj iné disky. Takýto príklad znázorňuje dátovú štruktúru *les,* čo je množina nula alebo viac stromov.

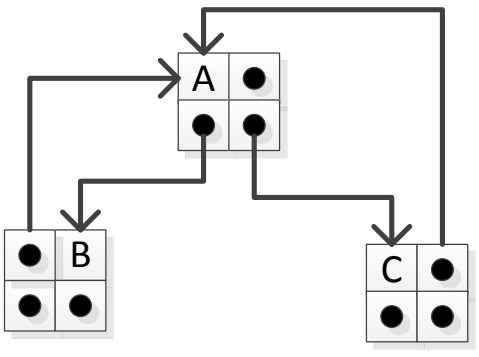
1.5. Binárne stromy

Medzi často využívané abstraktné dátové typy nepochybne patria aj binárne stromy. Binárny strom je definovaný ako konečná množina uzlov, ktorá je buď prázdna, alebo sa skladá z koreňa a prvkov dvoch disjunktných binárnych stromov, ktoré sa nazývajú ľavým a pravým podstromom daného koreňa [1]. Z uvedenej definície vyplýva, že každý uzol binárneho stromu má najviac dva podstromy.



Obrázok 10: Binárny strom

Pri implementácii binárneho stromu v počítačových programoch sa najčastejšie používa dátová štruktúra, resp. trieda s dvoma odkazmi – pre ľavý a pravý uzol. Môžeme si tu všimnúť podobnú reprezentáciu dát ako pri spojovej alokácii pamäte, avšak s tým rozdielom, že namiesto jedného ukazovateľa na nasledujúci prvok, pri binárnom strome máme ukazovatele dva. Takýto spôsob implementácie je efektívny najmä v programoch pristupujúcich k uzlom stromu od koreňa smerom dolu. Ak by sme potrebovali pristupovať k uzlom stromu aj opačným smerom, od potomka k rodičovi, bolo by nutné zaviesť tretí odkaz, ktorý by smeroval práve k rodičovi. Takýto strom potom nazývame *trojito prepojený strom*.



Obrázok 11: Trojito prepojený strom

Stromy môžeme do pamäte počítača uložiť aj inými spôsobmi, napríklad sekvenčnou alokáciou, kde jednotlivé uzly nasledujú bezprostredne za sebou. Výhody konkrétnej alternatívy závisia od typu operácií vkladania, odstraňovania a prechodom cez jednotlivé uzly stromu, potrebných pri manipulácii s danou stromovou štruktúrou.

1.5.1. Prechod binárnym stromom

Pri práci s binárnym stromom často potrebujeme preskúmať každý jeho uzol, prejsť uzol po uzle, až kým sa nedostaneme k poslednému prvku stromu. Pre lineárne zoznamy je takúto „návštevu“ každého prvku veľmi ľahko zrealizovať, pretože máme určený iba jeden smer cesty – dopredu, resp. aj späť pri obojsmerných frontách. Ako však z definície binárneho stromu vyplýva, tento obsahuje až dva ukazovatele (pri trojito prepojených stromoch dokonca až tri), a preto sa musíme rozhodnúť, ktorou cestou sa vybrať najskôr. Poznáme tri základné usporiadania, na základe ktorých môžeme uzly stromu navštíviť: [3]

- **Preorder**, kde navštívime uzol, potom navštívime jeho ľavý a pravý podstrom.
- **Inorder**, kde navštívime ľavý podstrom, potom navštívime uzol a nakoniec navštívime pravý podstrom.
- **Postorder**, kde navštívime ľavý a pravý podstrom a ako posledný navštívime uzol.

Pri prechode binárnym stromom každý uzol navštívime práve jedenkrát. Úplným prechodom stromu dostaneme lineárne usporiadanie jednotlivých uzlov, kde je jednoznačne určený *nasledujúci* uzol, ktorý v postupnosti nasleduje a *predchádzajúci* uzol danému uzlu. Názvy preorder, inorder a postorder vychádzajú z relatívnej pozície koreňa stromu vzhľadom k jeho potomkom. Aplikujme uvedené metódy napríklad na binárny strom z obrázka 10. Usporiadaním *preorder* dostaneme nasledujúci lineárny zoznam:

A – B – C – D – E – F – G

Využitím metódy *inorder,* kde koreň navštívime medzi návštevami jeho uzlov, dostaneme:

C – B – D – A – F – E – G

A konečne *postorder* metódou, kde koreň sa dostane na rad až po návšteve svojich uzlov:

C – D – B – F – G – E – A

1.5.2. Základné matematické vlastnosti binárnych stromov

Matematická teória binárnych stromov je nevyhnutne potrebná pre pochopenie výkonnostných charakteristík počítačových algoritmov. V nasledujúcom texte preto rozoberieme základné matematické vlastnosti binárnych stromov podľa [3].

a) **Binárny strom s *n* vnútornými uzlami má *n* + 1 vonkajších uzlov.**

   Môžeme to dokázať na základe metódy indukcie. Binárny strom bez vnútorných uzlov nemá vonkajší uzol, to znamená, že pre $n = 0$ vlastnosť platí. Každý binárny strom s vnútornými uzlami $n > 0$ má k vnútorných uzlov vo svojom ľavom podstrome a $n - k - 1$ vnútorných uzlov vo svojom pravom podstrome, kde $k \in (0, n - 1)$. Podľa induktívnej hypotézy má potom ľavý podstrom $k + 1$ vonkajších uzlov a pravý podstrom má $n - k$ vonkajších uzlov, z toho vyplýva, že celkový počet vonkajších uzlov je $n + 1$.

b) **Binárny strom s *n* vnútornými uzlami má 2*n* spojení: *n* – 1 spojení s vnútornými uzlami a *n* + 1 spojení s vonkajšími uzlami.**

   Každý uzol binárneho stromu okrem koreňa má jedinečného rodiča. S týmto rodičom ho spája hrana, to znamená, že existuje $n - 1$ spojení vnútorných uzlov. Podobne má každý z $n + 1$ vonkajších uzlov unikátne spojenie so svojim rodičom.

c) **Dĺžka vonkajšej cesty binárneho stromu s *n* vnútornými uzlami je o 2*n* dlhšia ako dĺžka vnútornej cesty.**

   Najskôr si zadefinujeme nové pojmy. **Dĺžka vonkajšej cesty** binárneho stromu je súčet úrovní všetkých vonkajších uzlov stromu. **Dĺžka vnútornej cesty** binárneho stromu je súčet úrovní všetkých vnútorných uzlov stromu.

   Vlastnosť, že dĺžka vonkajšej cesty je 2x dlhšia ako dĺžka vnútornej cesty si dokážeme na nasledujúcom príklade. Predpokladajme tvorbu binárneho stromu tak, že najskôr vytvoríme jediný vonkajší uzol. Následne n-krát zopakujeme naledujúci postup: vyberieme vonkajší uzol a nahradíme ho novým vnútorným uzlom, ktorý má dva vonkajšie uzly ako potomkov. Ak sme vybrali vonkajší uzol na úrovni k, potom sa dĺžka vnútornej cesty zvýši o k, ale dĺžka vonkajšej

cesty sa zvýši o $k + 2$. Táto situácia nastane z toho dôvodu, že sme odstránili jeden vonkajší uzol na úrovni k a pridali sme ďalšie dva na úrovni $k + 1$. Proces sa začína so stromom s nulovou dĺžkou vnútornej aj vonkajšej cesty. Po každých n krokoch sa dĺžka vonkajšej cesty zvýši vždy o 2 jednotky viac ako dĺžka vnútornej cesty.

d) Výška binárneho stromu s n vnútornými uzlami je najmenej log(n) a najviac $n - 1$.

Pripomeňme, že výška stromu je definovaná ako maximálna úroveň uzlov stromu. Označme výšku stromu ako h. Potom musí platiť nasledujúci vzťah:

$$2^{h-1} < n + 1 \leq 2^h,$$

pretože existuje $n + 1$ vonkajších bodov. Táto nerovnosť v sebe zahŕňa vlastnosť, že výška najlepšieho možného prípadu sa presne rovná log(n) zaokrúhlenému na najbližšie celé číslo. Najlepší prípad je typ vyváženého binárneho stromu, aký môžeme vidieť napríklad na obrázku 10. Opačným extrémom je degenerovaný strom s práve jedným listom s $n - 1$ spojeniami od koreňa k listu. Takýto binárny strom je označovaný ako najhorší možný prípad.

e) Dĺžka vnútornej cesty binárneho stromu s n vnútornými uzlami je najmenej n*log(n/4) a najviac n(n-1)/2.

Dĺžka vnútornej cesty pre najhorší prípad stromu (degenerovaný strom) je

$$0 + 1 + 2 + ... + (n - 1) = n(n - 1)/2$$

Definujme ďalej číslo *v_max*, ktoré je rovné číslu log(n) zaokrúhlenému na celé jednotky nadol. Najlepší prípad (vyvážený binárny strom) má potom (n + 1) vonkajších uzlov pri výške, ktorá nepresahuje číslo *v_max*. Ak násobením aplikujeme vlastnosť c), ktorá hovorí, že *dĺžka vonkajšej cesty binárneho stromu s n vnútornými uzlami je o 2n väčšia ako dĺžka vnútornej cesty,* dostávame hranicu:

$$(n + 1) * v\_max - 2n < n * \log(n/4)$$

1.6. Počítačové spracovanie aritmetických výrazov

Matematické výrazy najčastejšie zapisujeme v infixovom tvare. Jedná sa o prirodzený spôsob zápisu, ktorý je pre nás najľahšie pochopiteľný a ľahko spracovateľný. Infixová notácia znázorňuje aritmetický výraz takým spôsobom, že každý binárny operátor je umiestnený medzi dvoma operandmi, s ktorými bude vykonávať jemu príslušnú operáciu. Predpokladajme napríklad nasledujúci aritmetický výraz v infixovej notácii:

$$5 + 3 * (8 – 2) / 4 + 9$$

Poradie, v akom sa vykonáva spracovanie aritmetického výrazu je určené jednak prioritou operátorov, ale predovšetkým prostredníctvom zátvoriek. Každý výraz môže obsahovať ľubovoľné množstvo párov zátvoriek, ktoré sa môžu jednoducho vnárať do seba. Forma infixového zápisu aritmetického výrazu je ľahko pochopiteľná pre človeka. V počítačových algoritmoch sa však najčastejšie využíva infixová a postfixová notácia, pri ktorých je odstránená nutnosť používať zátvorky.

Postfixová notácia, tiež nazývaná aj reverzná poľská notácia [4] predstavuje taký spôsob zápisu, kde každý operátor nasleduje až po dvoch operandoch, ku ktorým sa vzťahuje. Pritom každý operand môže byť buď číselná hodnota alebo iný podvýraz, takisto zadaný v postfixovej notácii. Predchádzajúci aritmetický výraz v postfixovej notácii má teda tvar:

5 3 8 2 - * 4 / + 9 +

Prefixová notácia – je podobná postfixovej s tým rozdielom, že každý operátor je umiestnený pred operandami, ku ktorým sa vzťahuje. Podobne ako pri postfixovom tvare, operandom môže byť číslo alebo aj iný podvýraz, ktorý však musí byť tiež zadaný v prefixovom tvare. Ak by sme skonvertovali predchádzajúci aritmetický výraz podľa princípov prefixovej notácie, dostali by sme tvar:

+ + 5 / * 3 - 8 2 4 9

Prefixová a postfixová notácia sa bežnému človekovi zdajú byť na prvý pohľad dosť neprehľadé. Využitie nachádzajú najmä v počítačových programoch, ktoré manipulujú s aritmetickými výrazmi. Vyhodnotenie postfixových a prefixových výrazov je pre takéto programy menej zložité ako pri infixových výrazoch, pretože sa nemusia zaoberať zátvorkami, ale jednoducho čítajú a spracovávajú výraz z jednej strany do druhej.

2. Cieľ práce, metodika práce a metódy skúmania

Cieľom diplomovej práce je zosumarizovať základné poznatky z oblasti abstraktných dátových typov v počítačových algoritmoch. Práca analyzuje pojem a význam použitia ADT, ich druhy, stručný popis najčastejšie používaných ADT a základné vlastnosti stromov, so zameraním na binárne stromy. Na základe získaných vedomostí je následne hlavným zámerom diplomovej práce vytvoriť interpreter infixových aritmetických výrazov. Jeho implementácia bude uskutočnená v jazyku C++, vo vývojovom prostredí Microsoft Visual Studio 2013 a bude vybudovaná pomocou architektúry klient – server.

Tvorba diplomovej práce bola rozdelená do dvoch etáp. Prvá etapa je venovaná objasneniu pojmu abstraktný dátový typ a rozborom niektorých základných realizácií ADT. Vychádza najmä z poznatkov získaných štúdiom odborných publikácií (Knuth, 2008) a (Sedgewick, 2003), ktoré podrobne a výstižne preberajú problematiku počítačových algoritmov a preto boli veľmi cenným zdrojom informácií. Ďalej sa rozoberá problematika aritmetických výrazov a rôzne možnosti ich zápisu. Pre túto oblasť sme čerpali informácie z rôznych internetových portálov, najmä z internetovej encyklopédie Wikipédia. V prvej kapitole diplomovej práce sme využili metódu deskriptívnej analýzy, slúžiacej popisu samostatných charakteristík skúmaného systému a metódu matematickej indukcie.

Druhá etapa diplomovej práce je venovaná využitiu získaných teoretických poznatkov pri konkrétnej implementácii interpretera, využívajúceho dátovú štruktúru binárneho stromu. Interpreter infixových aritmetických výrazov bol vytvorený v neriadenom, natívnom jazyku C++ a slúži ako HTTP server, ktorý počúva požiadavky klienta v tvare internetovej požiadavky metódy POST. Veľmi cenným zdrojom informácií bola pre nás odborná kniha (Prata, 2007), ktorá do hĺbky rozoberá programovacie techniky jazyka C++. Hlavné myšlienky pri návrhu implementácie sme prebrali z rôznych internetových stránok a online dokumentov, a to najmä z veľmi podrobne spracovanej prednášky (Borovanský, online), venovanej stavbe vyhodnocovača aritmetických výrazov. Aby sme mohli vizuálne skontrolovať a zhodnotiť výsledky vyvinutej aplikácie, vytvorili sme jednoduchého HTML klienta, ktorý má za úlohu prijať aritmetický výraz od používateľa, odoslať tento výraz na server a vykresliť zodpovedajúci binárny strom, ktorý bude prijatý zo servera ako odpoveď. Na vykreslenie sme použili voľne prístupnú knižnicu D3.js, ktorú sme modifikovali nevyhnutnými úpravami, aby zodpovedala požiadavkám našej aplikácie.

3. Výsledky práce a diskusia

V nasledujúcej kapitole sa budeme venovať rozboru implementácie syntaktického binárneho stromu v jazyku C++. Tento objektovo orientovaný jazyk vyvinul Bjarne Stroustrup začiatkom 80. rokov minulého storočia v Bell Labs. V súčasnosti ešte stále patrí medzi populárne programovacie jazyky, a to najmä vďaka bohatej podpore vytvorených knižníc. Najnovšia verzia medzinárodného ISO štandardu pre jazyk C++ bola schválená 12. Augusta 2011 a nesie označenie C++11. Zavádza do jazyka viacero syntakticko-sémantických inovácií, medzi ktoré patrí napríklad automatická typová inferencia lokálnych entít (zabezpečená kľúčovým slovom **auto**), ďalej nový operátor **decltype,** vďaka ktorému môžeme zistiť dátový typ lokálnej premennej, inteligentné smerníky (smart pointers), ktorých životný cyklus je riadený automaticky (odpadá teda nutnosť manuálne uvoľňovať pamäť počítača), alebo lambda výrazy, ktoré prinášajú azda najvýznamnejšiu inováciu jazyka C++. Lambda výraz umožňuje definovať anonymnú funkciu, označovanú tiež ako ako λ-výraz. Ďalším prínosom štandardu C++11 je rozšírenie štandardnej knižnice (standard template library – STL) o nové kontajnery, ako je napr. **vector, list** alebo **map,** rozšírenie STL o množstvo nových algoritmov, či nové triedy **string** a **wstring.** Jazyk C++ sa však i naďalej vyvíja a ďalšie verzie medzinárodného štandardu nesú pracovné názvy C++14 a C++17. Nepochybne aj tieto prinesú množstvo zlepšení a jazyk C++ si tak udrží svoju dobrú pozíciu medzi modernými nástrojmi softvérových vývojárov aj v budúcnosti.

3.1. Základná charakteristika aplikácie

Interpreter infixových aritmetických výrazov sme vybudovali pomocou objektovo orientovanej paradigmy používanej v programovaní. OOP predstavuje metodiku vývoja softvéru, ktorá je založená na používaní dátových štruktúr, tiež nazývaných ako **objekty.** Každý objekt disponuje dátami a operáciami (metódami), ktorými vykonáva činnosti, na ktoré bol vytvorený. OOP zlepšuje čitateľnosť kódu, uľahčuje jeho písanie a zvyšuje jeho znovupoužiteľnosť. Ďalej sa v tejto práci stretneme s dedičnosťou, polymorfizmom, zapuzdrením a abstrakciou. Tieto vlastnosti sú charakteristickými črtami objektovo orientovaného programovania a preto sme sa rozhodli využiť výhody, ktoré prinášajú.

Na začiatok preskúmajme architektúru aplikácie. Ako sme už spomenuli vyššie, jedná sa o architektúru **klient-server,** kde vystupujú dva nezávislé systémy (systém na strane klienta a systém na strane servera), pričom ich vzájomná interakcia je zabezpečená komunikačným kanálom, najčastejšie počítačovou sieťou. Spôsobov komunikácie medzi klientom a serverom je viacero, v súčasnosti najjednoduchšou možnosťou je vzájomné posielanie správ. V okamihu, keď klient odošle správu, stráca nad ňou kontrolu. Správa sa preto javí ako nezávislá, samostatná jednotka, ktorá nesie informáciu o odosielateľovi, prijímateľovi a konkrétnej požiadavky. Sever prijíma požiadavku, spracuje ju a vyhodnotí. Na základe informácií o odosielateľovi posiela späť odpoveď, takisto vo formáte nezávislej správy. Ide o tzv. bezstavový spôsob komunikácie, kde subjekt po odoslaní správy neuchováva jej stav, nepamätá si dokonca ani to, že nejakú správu odoslal. Tento spôsob komunikácie je charakteristický najmä v oblasti servisne orientovanej architektúry (SOA) a taktiež v internetovej komunikácii založenej na REST službách. Pri takýchto službách sú prostredníctvom internetového protokolu HTTP posielané správy na konkrétne IP adresy, pričom každá správa je určená metódou, ktorá charakterizuje typ požiadavky:

- GET – získanie existujúceho zdroja.
- POST – vytvorenie nového zdroja.
- PUT – úprava existujúceho zdroja.
- DELETE – vymazanie existujúceho zdroja.

V našej implementácii sme sa rozhodli využiť výhody REST služieb, pričom sme na serverovej strane aplikácie zabudovali podporu iba pre správy typu POST. To znamená, že ostaté typy správ budú automaticky zamietnuté. Správy typu POST umožňujú odosielať vo svojom tele aj používateľom zadaný obsah, ktorý je server schopný spracovať. Pri metóde GET takáto možnosť nie je dostupná, pretože sa jedná o žiadosť konkrétneho, už existujúceho zdroja. Z toho vyplýva, že pri žiadosti o takýto zdroj vôbec nezáleží na údajoch v tele správy a preto ich server ani neberie do úvahy.

Ako sme už niekoľkokrát spomenuli, a ako to vyplýva aj z názvu diplomovej práce, interpreter sme vytvorili v jazyku C++. Ktorú stranu vo vzťahu klient/server však interpreter reprezentuje? Z povahy problému jednoznačne vyplýva, že sa jedná o stranu servera. Výhodou tejto architektúry je aj skutočnosť, že následne môže byť vytvorené nespočetné množstvo klientskych aplikácií, v ľubovoľnom programovacom jazyku, ktoré môžu využívať služby interpretera a spracovávať jeho výsledky vlastným spôsobom. Pre

ilustráciu výsledkov práce sme sa rozhodli vytvoriť aj jednoduchého klienta, ktorý zobrazí výsledok aritmetického výrazu a vykreslí vyhodnotený binárny strom. Využili sme pritom výhody jazykov HTML a JavaScriptu, ktoré nám umožňujú veľmi jednoduchým spôsobom vytvárať plnohodnotné webové aplikácie. JavaScript disponuje množstvom voľne dostupných knižníc, ktoré boli vytvorené na podporu budovania interaktívnych internetových aplikácií. Medzi ne nepochybne patrí knižnica jQuery. Menej známa, ale veľmi bohatá knižnica je taktiež D3.js, ktorá slúži na vykresľovanie grafov a nám poslúžila na zakreslenie výsledného binárneho stromu. Štruktúra klientskej aplikácie je veľmi jednoduchá. Jedná sa o internetovú stránku s jedným formulárom, ktorý obsahuje textové pole pre zadanie aritmetického výrazu a tlačidlo, ktorým sa tento výraz v podobe správy typu POST odošle na server. Odoslanie správy je zabezpečené využitím JavaScriptovej knižnice jQuery, konkrétne metódou `ajax`, ktorá správu odosiela asynchrónne. Po vyhodnotení výrazu bude pod formulárom aplikácie vykreslený binárny strom, pričom bude umožnená aj ďalšia manipulácia s ním, ako napríklad zmenšenie a zväčšenie stromu, alebo zabalenie jednotlivých uzlov a zobrazenie ich medzivýsledkov.

3.2. Spracovanie aritmetického výrazu

V tejto podkapitole preskúmame najčastejšie používanú oblasť matematiky, ktorá sa zaoberá aritmetickými výrazmi. Aritmetickým výrazom rozumieme postupnosť operandov (číslic) a operátorov (znamienok), ktoré vykonávajú operáciu medzi svojimi operandmi. Platnými operáciami aritmetického výrazu sú sčítanie (+), odčítanie (-), násobenie (*) a delenie (/). Poradie vykonania operácií je určené ich prioritou. Operátory sčítania a odčítania majú nižšiu prioritu ako operátory násobenia a delenia. Aritmetický výraz však môže obsahovať aj zátvorky, ktoré slúžia na zvýšenie priority konkrétneho podvýrazu. Výraz v zátvorke má za každých okolností najvyššiu prioritu, pričom samostatné zátvorky sa môžu do seba vnárať. V takomto prípade bude najskôr vyhodnotený ten podvýraz, ktorý sa nachádza v najvnútornejšej zátvorke.

V prvej kapitole sme sa bližšie zoznámili s rôznymi notáciami aritmetických výrazov. Pripomeňme, že najčastejším a pre človeka najjednoduchšie čitateľným, je spôsob zápisu v tzv. **infixovej** notácii. Názov notácie vychádza z umiestenia operátora vzhľadom k jeho operandom. Pri infixovej notácii sa preto operátor nachádza medzi svojimi operandmi,

s ktorými vykonáva príslušnú operáciu. Spôsob infixového zápisu je však pre počítačové algoritmy ťažko spracovateľný. Vyplýva to najmä zo skutočnosti, že pri infixovej notácii sa používajú zátvorky. Naproti tomu existujú dve iné formy zápisu, využívajúce prefixovú a postfixovú notáciu, pri ktorých nutnosť používať zátvorky odpadá. Ako už zo samotných názvov vyplýva, v prefixovej notácii je operátor zapísaný pred svojimi operandmi a v postfixovej notácii až za svojimi operandmi. Prevod medzi jednotlivými notáciami je pomerne jednoduchý. Preskúmajme napríklad všeobecný algoritmus, konvertujúci aritmetický výraz z infixovej do postfixovej notácie. Predpokladajme, že výsledok (výstup) je tvorený reťazcom, resp. postupnosťou čísel a operátorov. Ďalej definujme pomocný zásobník, kde budeme odkladať operátory. Ako už vieme, zásobník je riadený metódou LIFO, to znamená, že posledný vložený operátor bude vybraný ako prvý. Infixový výraz čítame zľava doprava a postupujeme jednou z nasledujúcich možností:

a) Ak narazíme na číslo, umiestnime ho do výstupného reťazca a pokračujeme.
b) Ak narazíme na operátor a zásobník operátorov je prázdny, vložíme ho sem a pokračujeme ďalej. Ak zásobník nie je prázdny, postupne od vrchu vyberáme operátory s vyššou prioritou, ako je priorita aktuálneho operátora a umiestňujeme ich do výstupného reťazca. Ak narazíme na operátor s nižšou prioritou alebo na dno zásobníka, vložíme aktuálny operátor do zásobníka a pokračujeme.
c) Ak narazíme na ľavú zátvorku, tá putuje za každých okolností do zásobníka. V zásobníku má však medzi ostatnými operátormi najnižšiu prioritu.
d) Ak narazíme na pravú zátvorku, vyberáme operátory zo zásobníka a umiestňujeme ich do výstupného reťazca, až kým nenarazíme na ľavú zátvorku. Túto taktiež odstránime a pokračujeme ďalej.
e) Ak sme sa dostali na koniec aritmetického výrazu, postupne vyberáme zo zásobníka operátory a pripájame ich k výstupnému reťazcu, až kým nebude zásobník prázdny.

Výstupný reťazec bude následne obsahovať aritmetický výraz v postfixovej notácii. Ak napríklad aplikujeme uvedený algoritmus na infixový výraz 25 + 8 * (7 + 4), dostaneme výsledok v postfixovom tvare:

25 8 7 4 + * +

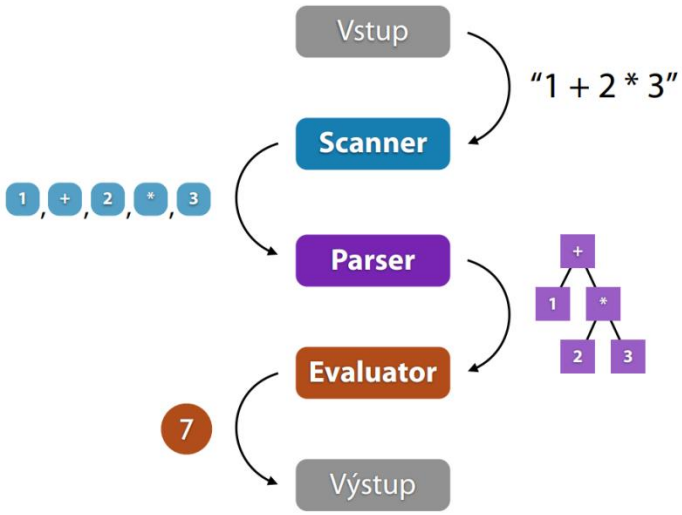
Výsledný tvar aritmetického výrazu je následne potrebné vyhodnotiť. K realizácii tejto požiadavky budeme taktiež potrebovať pomocný zásobník, avšak s tým rozdielom, že

nebudeme do neho vkladať operátory, ale operandy. Podobne ako pri konverznej metóde konvertujúcej infixový tvar výrazu na postfixový, prechádzame položkami aritmetického výrazu zľava doprava a pokračujeme jednou z nasledujúcich možností:

a) Ak narazíme na operand, putuje do zásobníka.
b) Ak narazíme na operátor, vyberieme z vrcholu zásobníka dva operandy, vykonáme operáciu príslušnú danému operátoru a jej výsledok opäť vložíme do zásobníka. Pri tomto kroku je ale potrebné si znova uvedomiť, že vkladanie a odoberanie hodnôt zásobníka sa riadi metódou LIFO, to znamená, že prvý vybraný operand musíme umiestniť na pravú stranu operácie a druhý vybraný operand na ľavú stranu. Ak by sme totiž vykonávali operáciu delenia a zamenili by sme delenca s deliteľom, dostali by sme neprávny výsledok.
c) Ak sme sa dostali na koniec postfixového reťazca, v zásobníku budeme mať jedinú hodnotu, ktorá reprezentuje vypočítaný výsledok aritmetického výrazu.

3.3. Štruktúra pracovných objektov interpretera

Nasledujúci obrázok zobrazuje postupnosť volania základných pracovných objektov aplikácie, ktoré zabezpečujú načítanie, spracovanie a vyhodnotenie aritmetického výrazu.



Obrázok 12: Štruktúra pracovných objektov aplikácie (zdroj: [5])

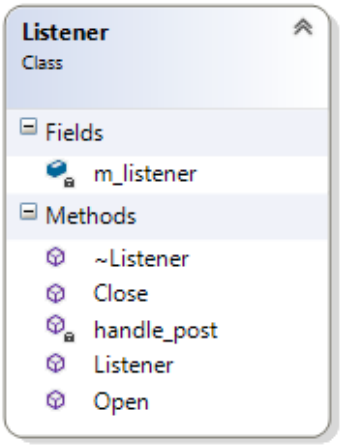
Môžeme tu vidieť tri základné objekty, ktoré sme v aplikácii tiež označili ako Workers, pretože sú hlavnými pracovníkmi spracovania aritmetického výrazu. Jedná sa o lexikárny

analyzátor – Scanner, ďalej syntaktický analyzátor – Parser a konečne vyhodnocovač – Evaluator. Ich spoločnou úlohou je načítať prijatý aritmetický výraz, rozdeliť ho na čo najmenšie položky (čísla, operátory a zátvorky), tieto následne spracovať do podoby syntaktického binárneho stromu a nakoniec tento strom vyhodnotiť, teda vypočítať aritmetický výraz a vrátiť výsledok klientovi. Pre tvorbu binárneho stromu sme do programu implementovali dve základné metódy, pričom výber konkrétnej z nich je podmienený veľkosťou prijatého aritmetického výrazu. Prvou metódou je rekurzívna stavba stromu, ktorá je typická pre binárne stromy kvôli ich prirodzenej rekurzívnej štruktúre. Druhým spôsobom je použitie nerekurzívnej metódy, s využitím pomocného zásobníka.

V nasledujúcich podkapitolách jednotlivo preskúmame každý pracovný objekt aplikácie, jeho základné vlastnosti a dôležité operácie. Začneme však tiež veľmi dôležitým objektom, ktorý má za úlohu *„počúvať“* na konkrétnom porte IP adresy požiadavky klientov, teda prijímať správy v podobe aritmetických výrazov a odpovedať im tiež správou, ktorá bude obsahovať vyhodnotený binárny strom v textovom formáte štruktúry JSON.

3.4. Vstupný bod aplikácie - Listener

Na nasledujúcom obrázku môžeme vidieť štruktúru triedy `Listener`, ktorej inštancia počúva požiadavky klientov a keď dostane správu, zabezpečí jej adekvátne spracovanie.



Obrázok 13: Trieda Listener

Môžeme tu vidieť jednu premennú **m_listener**, ktorá je objektom triedy **http_listener**. Jedná sa o triedu vytvorenú pre spracovávanie HTTP požiadaviek na špecifikovanej IP adrese. Stretávame sa tu s veľmi dôležitou vlastnosťou objektovo orientovaného programovania, ktorou je **kompozícia** objektov. Kompozícia objektov predstavuje vnáranie objektov do nadradeného objektu a je jadrom komponentového programovania [6]. Hovoríme, že využívame vlastnosti objektu **m_listener** prostredníctvom inštančného objektu triedy **Listener**. To znamená, akoby sme zabalili objekt **m_listener** a vytvorili sme tak triedu, ktorá pracuje na vyššej úrovni abstrakcie ako samotná trieda **http_listener**.

Medzi metódami triedy môžeme vidieť konštruktor, deštruktor, ďalej metódu **Open**, určenú pre spustenie počúvania a metódu **Close**, ktorá počúvanie ukončí. Najdôležitejšou je však metóda **handle_post**, ktorej vykonávanie sa spustí ihneď, ako server dostane požiadavku prostredníctvom správy typu POST. Najskôr bolo však nevyhnutné zadefinovať podporu pre správy typu POST a nastaviť, aby boli presmerované práve tejto metóde . Zabezpečili sme to v konštruktore triedy **Listener**, metódou **support** objektu **m_listener** s potrebnými parametrami. Ostatné typy správ (napr. GET, PUT, DELETE) interpreter nebude podporovať a preto sme ich v konštruktore nezaregistrovali. Takéto správy budú teda automaticky zamietnuté. Nasledujúci fragment kódu uvádza konštruktor triedy **Listener**.

```
Listener::Listener(const http::uri& url) : m_listener(http_listener(url))
{
	m_listener.support(
		methods::POST,
		std::tr1::bind(&Listener::handle_post,
		this,
		std::tr1::placeholders::_1));
}
```

Metóda **handle_post** prijíma HTTP požiadavku od klienta (správu), z ktorej extrahuje aritmetický výraz a uloží ho do lokálnej premennej typu **wstring**.

```
wstring expression = request.extract_string().get();
```

Po vyhodnotení výrazu vracia odpoveď klientovi, ktorú pretransformuje do formátu JSON.

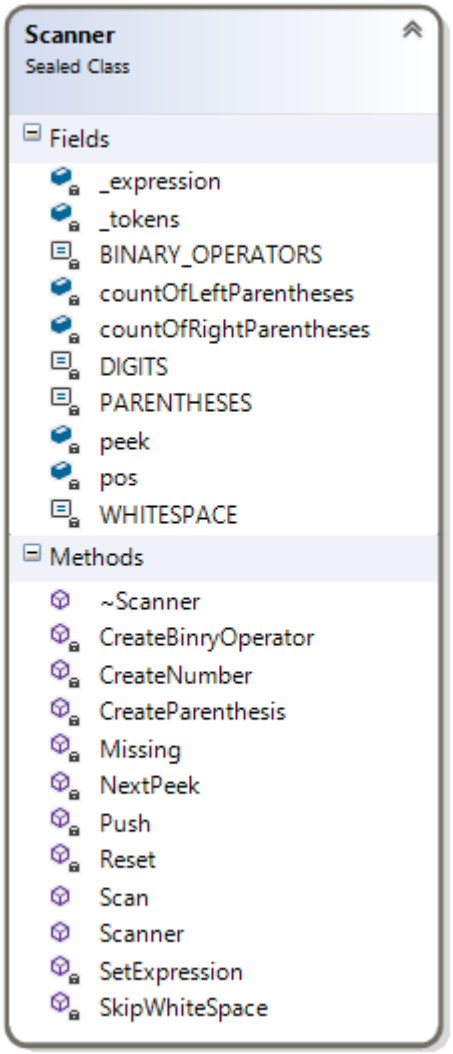
```
string result = e->Evaluate(&expression);
request.reply(status_codes::OK, web::json::value::parse(istringstream(result)));
```

Počas spracovávania môže nastať aj chyba, napr. ak je prijatý aritmetický výraz neplatný.

```
request.reply(status_codes::NotAcceptable,
web::json::value::parse(istringstream(error)));
```

3.5. Lexikálny analyzátor – Scanner

Po prijatí klientskej požiadavky a extrahovaní aritmetického výrazu začína jeho spracovanie. Ako sme mohli vidieť, výraz je uložený vo formáte **`wstring`**. Wstring je typ objektu, ktorý uchováva viacbajtové textové znaky. Takýto formát aritmetického výrazu je však pre naše účely nepoužiteľný. Počítačový program totiž nevie vyhodnotiť výraz zapísaný v textovom reťazci. Potrebuje vedieť, ktoré znaky sú čísla, ktoré sú operátory alebo zátvorky. Úlohou skenera je preto skenovať textový reťazec znak za znakom a vytvárať objekty, ktoré následne odošle na vyhodnotenie. Skener má teda za úlohu rozdeliť textový reťazec na čo najmenšie časti, ktoré sú už v konečnej podobe nedeliteľné. Môže to byť napríklad operátor +, ľavá zátvorka, číslo 25 alebo záporné číslo -4.



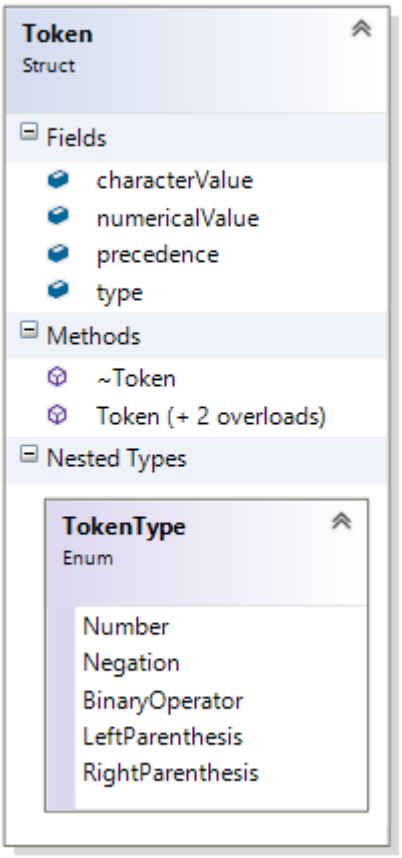
Obrázok 14: Trieda Scanner

Skener, ako prvý objekt, ktorý spracováva prijatý aritmetický výraz, vykonáva zároveň aj prvú kontrolu správnosti zápisu. Ak totiž narazí na neplatný znak, ktorý nie je povoleným v aritmetickom výraze, okamžite ukončí svoju činnosť chybovou správou. Taktiež počas skenovania výrazu počíta ľavé a pravé zátvorky. Ak niektorá zátvorka chýba, klient je okamžite informovaný, na ktorý typ zátvorky zrejme pozabudol.

Preskúmajme teraz bližšie zapečatenú triedu **`Scanner`**, z ktorej už nemôžu dediť iné objekty. Trieda obsahuje niekoľko súkromných členov, medzi ktorými sú niektoré napísané samými veľkými písmenami. Takúto konvenciu sme použili pre reťazcové konštanty. Uchovávajú hodnoty potrebné pri skenovaní textového reťazca. Napríklad **`BINARY_OPERATORS`** je konštanta typu **`string`** a má hodnotu „+-/*“. Ak skener nájde v skenovanom reťazci niektorý z týchto znakov, okamžite indikuje, že musí vytvoriť nový objekt typu binárneho operátora. Obdobná situácia nastane, ak sa práve skenovaný znak (v implementácii označovaný ako **`peek`**) nachádza v reťazcovej konštante **`PARENTHESES`**.

V takomto prípade vytvorí ľavú alebo pravú zátvorku. Pre uchovávanie všetkých povolených objektov aritmetického výrazu sme vytvorili štruktúru **Token**.

Ako môžeme vidieť na obrázku 15, každý token aritmetického výrazu je určený svojím typom. Token môže byť inicializovaný iba takým typom, aký je povolený v aritmetickej operácii. Preto je definovaný ako enumerátor, čo znamená, že môže nadobúdať iba hodnotu z vopred vymenovaných typov – číslo (**Number**), záporné číslo (**Negation**), binárny operátor (**BinaryOperator**), ľavá zátvorka (**LeftParenthesis**) alebo pravá zátvorka (**RightParenthesis**). Ďalej si môžeme všimnúť dve súkromné premenné pre uchovanie hodnoty tokenu. V prípade operátora alebo zátvorky bude v konštruktore objektu nastavená premenná **characterValue** typu **char**. V prípade čísla alebo záporného čísla bude v konštruktore nastavená premenná **numericalValue** typu **double**. Konštruktor štruktúry je teda preťažený. Pri vytváraní nového tokenu je zavolaná tá alternatíva, ktorá je adekvátna typu vytváraného objektu. Dôležitým atribútom každého tokenu je tiež jeho priorita. Táto sa uchováva v premennej **precedence** typu **int** a je taktiež nastavená v konštruktore. Budeme ju potrebovať neskôr, pri vyhodnocovaní výrazu.



Obrázok 15: Štruktúra Token

Najdôležitejšou úlohou skenera je skenovať každý jeden znak aritmetického výrazu. Táto úloha je vykonávaná metódou **Scan** ktorá prijíma jeden formálny parameter. Týmto parametrom je smerník na textový reťazec typu **wstring**. Návratovým typom metódy **Scan** je smerník na abstraktný dátový typ fronty (**queue**), ktorý uchováva smerníky na objekty typu **Token**. Smerníky sú užitočným nástrojom a to najmä pri odovzdávaní väčších objektov. Ak by sme totiž vrátili pole tokenov hodnotou, program by musel inicializovať nové pole a skopírovať doň každú jednu položku zoznamu. Naproti tomu, ak návratovým typom bude smerník, pracujeme stále s tým istým zoznamom. Nasledujúci fragment kódu zobrazuje deklaráciu metódy **Scan** z hlavičkového súboru **Scanner.h**:

```
queue<Token *>* Scan(wstring * Expression);
```

Definícia metódy **Scan** je umiestená v súbore **Scanner.cpp** a má nasledujúci tvar:

```
queue<Token *> * Scanner::Scan(wstring * Expression)
{
	Reset();
	SetExpression(Expression);
	NextPeek();
	while (true)
	{
		SkipWhiteSpace();
		if (DIGITS.find(peek) != string::npos)
		{
			CreateNumber(false);
		}
		else if (BINARY_OPERATORS.find(peek) != string::npos)
		{
			bool isNegation = false;

			if (peek == '-')
			{
				if (_tokens->empty())
				{
					isNegation = true;
				}
				else if (_tokens->back()->characterValue == '(')
				{
					isNegation = true;
				}
			}

			if (isNegation)
			{
				CreateNumber(true);
			}
			else
			{
				CreateBinaryOperator();
			}
		}
		else if (PARENTHESES.find(peek) != string::npos)
		{
			CreateParenthesis();
		}
		else if (peek == NULL)
		{
			break;
		}
		else
		{
			string error = "Neplatný znak '";
			error.push_back(peek);
			error += "' na pozícii " + to_string(pos);

			throw invalid_argument(error);
		}
	}

	if (countOfLeftParentheses < countOfRightParentheses) Missing('(');
	else if (countOfRightParentheses < countOfLeftParentheses) Missing(')');
	return _tokens;
}
```

Preskúmajme teraz jednotlivé položky kódu metódy **Scan**. Jej prvou úlohou je nastavenie všetkých premenných skenera na východiskové hodnoty. Splnenie tejto úlohy zabezpečí metóda **Reset**, ktorá vynuluje počítadlo aktuálnej skenovanej pozície (reprezentované súkromnou premennou **pos**) a vyprázdni výslednú frontu smerníkov naskenovaných tokenov s korektným uvoľnením operačnej pamäte počítača. Jedná sa o deštrukciu objektov vytvorených dynamicky počas skenovania výrazu. Funkcia **Reset** teda pripraví skener pre ďalšiu iteráciu skenovania aritmetického výrazu. Potom je zavolaná jednoduchá funkcia **SetExpression**, ktorá uloží smerník na prijatý aritmetický výraz do svojej lokálnej smerníkovej premennej **_expression**. Túto premennú budú neskôr využívať ostatné súkromné metódy skenera. Následne je zavolaná funkcia **NextPeek**, ktorá má za úlohu priradiť znakovej premennej **peek** nasledujúci znak aritmetického výrazu. Ako sme už uviedli, úlohou skenera je načítať každý jeden znak výrazu a zabezpečiť jeho adekvátne spracovanie. Pri prvom zavolaní metódy **NextPeek** je preto premenná **peek** inicializovaná hodnotou prvého znaku reťazca. Po jej ďalšom zavolaní bude hodnota **peek** prepísaná hodnotou znaku, ktorý nasleduje za práve spracovaným znakom. Nasledujúci fragment kódu metódy **NextPeek** zobrazuje priradenie konkrétneho znaku aritmetického výrazu do premennej **peek**. Tento znak je určený aktuálne nastavenou pozíciou v skenovanom reťazci. Po získaní skenovaného znaku bude pozičná premenná **pos** inkrementovaná o jednotku.

```
peek = char((*_expression)[pos++]);
```

Predpokladajme vstupný reťazec 25+2. Po prvom zavolaní funkcie **NextPeek** bude premenná **peek** inicializovaná hodnotou 2, nasledujúcim volaním jej bude priradená hodnota 5, ďalej hodnota + a nakoniec 2. V momente, keď sa skener dostane na koniec reťazca, premenná **peek** bude nastavená prázdnou hodnotou **NULL** a skenovanie sa ukončí.

```
if (pos == _expression->length())
{
	peek = NULL;
}
```

Pri písaní aritmetického výrazu človek zvyčajne zadáva aj medzery, aby bol napísaný výraz ľahšie čitateľný a prehľadnejší. Pre počítačový program však takéto biele znaky v reťazci nemajú žiaden význam a preto budú metódou **SkipWhiteSpace** ignorované.

```
void Scanner::SkipWhiteSpace()
{
	while (WHITESPACE.find(peek) != string::npos) NextPeek();
}
```

Konštantná premenná **WHITESPACE** je definovaná ako inštancia triedy **string**. V rozhraní tejto triedy môžeme nájsť metódu **find**, ktorá hľadá prijatý argument v textovom reťazci svojho objektu. Jej návratovým typom je číslo, ktoré udáva index prvého výskytu hľadaného podreťazca. Ak sa však žiadna zhoda hľadaného výrazu nenájde, návratovou hodnotou bude statická konštanta triedy **string** s názvom **npos** a hodnotou -1. V našom prípade teda definujeme cyklus, ktorý prebieha až dovtedy, kým práve skenovaný znak nebude mať zmysluplnú hodnotu, tzn. nebude sa nachádzať medzi prázdnymi znakmi.

Funkciu hľadania výskytu aktuálne naskenovaného znaku využívame aj v tele hlavnej funkcie **Scan**. Ako sme si mohli všimnúť, vo vnútri cyklu **while** sa nachádza rozhodovacia štruktúra, ktorá sa na základe nájdenia znaku **peek** v konkrétnom konštantnom textovom reťazci rozhodne, ktorou vetvou má pokračovať. Ak je napríklad práve skenovaný znak číslom, bude zavolaná funkcia **CreateNumber**. Podobná situácia nastane, keď sa dostane na rad zátvorka. V takom prípade bude aktivovaná funkcia **CreateParenthesis**.

Zložitejšia situácia nastane v prípade, ak vstupným znakom bude operátor. Ako vieme, v matematike poznáme aj záporné čísla a preto musíme pri znaku '-' určiť, či používateľ mal na mysli operáciu odčítania dvoch čísel alebo chcel iba napísať záporné číslo. Túto kontrolu sme implementovali veľmi jednoduchým spôsobom. Záporné čísla sa totiž v aritmetickom výraze môžu nachádzať iba na dvoch miestach. Prvým miestom je začiatok výrazu a druhým je miesto, ktoré nasleduje bezprostredne za ľavou zátvorkou. Ak sa teda práve skenovaný znak '-' nenachádza na takomto mieste, jedná sa o operátor a preto exekúcia programu bude pokračovať funkciou **CreateBinaryOperator**. V opačnom prípade bude zavolaná už spomenutá funkcia **CreateNumber**, tentoraz však s parametrom **true**.

Parameter funkcie **CreateNumber** je typu **bool** a určuje, či sa jedná o záporné číslo alebo nie. Na jeho základe sa pri vytváraní nového číselného tokenu vyberie ten správny typ.

```
void Scanner::CreateNumber(bool IsNegation)
{
	...

	if (IsNegation)
	{
		Push(new Token(Negation, atof(temp.c_str())));
	}
	else
	{
		Push(new Token(Number, atof(temp.c_str())));
	}
}
```

Uvedená funkcia na vytvorenie čísla však musí obsahovať aj ďalšiu logiku pre spracovanie nasledujúceho znaku v aritmetickom reťazci, aby nenastala taká situácia, že sa vytvorí iba časť čísla. Matematické číslo môže mať totiž aj viac cifier, dokonca sa môže jednať aj o desatinné číslo. Takéto číslo je považované za nedeliteľný token, nemá zmysel napríklad rozdeľovať číslicu 25 na čísla 2 a 5. Funkcia **`CreateNumber`** musí preto počítať s tým, že ďalším skenovaným znakom môže byť číslica, ktorá je ešte súčasťou vytváraného tokenu. Aby sme získali a vytvorili správny token, deklarovali sme dočasnú premennú **`temp`** typu **`string`**, ku ktorej postupne pridávame nasledujúci znak výrazu, pokiaľ patrí k danému číslu. Konštantná premenná **`DIGITS`** obsahuje postupnosť číslic vrátane desatinnej bodky.

```
while (DIGITS.find(peek) != string::npos)
{
      temp += peek;
      NextPeek();
      SkipWhiteSpace();
}
```

V okamihu, keď už bude číslo kompletné, program opúšťa cyklus a pokračuje vytvorením nového tokenu, ktorého odkaz následne vloží do fronty naskenovaných tokenov. Konverziu dočasnej textovej premennej **`temp`** na číslo typu **`double`** zabezpečí funkcia **`atof`**.

```
Push(new Token(Number, atof(temp.c_str())));
```

Preskúmajme teraz funkciu **`CreateBinaryOperator`**, ktorá slúži na vytvorenie tokenu typu **`BinaryOperator`**. Táto metóda má veľmi jednoduchú štruktúru a obsahuje iba príkaz na vytvorenie samotného tokenu, jeho súčasné vloženie do výsledkovej fronty a príkaz pre posunutie skenera na nasledujúci znak aritmetického výrazu.

```
void Scanner::CreateBinaryOperator()
{
      Push(new Token(BinaryOperator, peek));

      NextPeek();
}
```

Podobný algoritmus sme využili aj pri implementácii metódy **`CreateParenthesis`**, ktorá slúži na vytváranie tokenov pre zátvorky. Pribudla pri ňom iba rozhodovacia štruktúra pre určenie správneho typu zátvorky (ľavá alebo pravá) a taktiež príkaz na inkrementovanie počítadla zátvoriek príslušného typu. Počítadlo využijeme neskôr pri kontrole počtu ľavých a pravých zátvoriek.

```
void Scanner::CreateParenthesis()
{
	if (peek == '(')
	{
		countOfLeftParentheses++;
		Push(new Token(LeftParenthesis, peek));
	}
	else
	{
		countOfRightParentheses++;
		Push(new Token(RightParenthesis, peek));
	}

	NextPeek();
}
```

Pri uložení smerníka novovytvoreného tokenu do výsledkovej fronty využívame metódu **Push**, preto v nasledujúcej časti objasníme jej funkcionalitu. Táto metóda je rozšírením klasickej metódy **push** poskytnutej prostredníctvom rozhrania ADT fronty. Konkrétne ju rozširuje o overovaciu funkciu, ktorá je ďalšou formou kontroly správnosti zápisu vstupného výrazu. Podľa matematických pravidiel, každý aritmetický výraz môže začínať číslom, záporným číslom alebo ľavou zátvorkou. Túto požiadavku overuje prvá podmieňovacia vetva metódy **Push**, ktorá je vykonávaná v tom prípade, ak je fronta prázdna, to znamená, že sa nachádzame na začiatočnom tokene aritmetického výrazu.

```
void Scanner::Push(Token* token)
{
	bool CanPush = false;
	if (_tokens->empty())
	{
		if (token->type == Number ||
		    token->type == Negation ||
		    token->type == LeftParenthesis)
		       CanPush = true;
	}
	else
	{
		auto lastTokenType = _tokens->back()->type;
		switch (token->type)
		{
		    ...
		}
	}

	if (CanPush)
	{
		_tokens->push(token);
	}
	else
	{
		throw invalid_argument("Neplatný výraz");
	}
}
```

Ak fronta nie je prázdna, to znamená, že skener už spracoval niektoré tokeny výrazu a nachádza sa na nejakej inej pozícii ako je začiatočná, je potrebné overiť, či sa práve skenovaný znak môže nachádzať na svojej pozícii vzhľadom na typ predchádzajúceho tokenu. Typ tokenu na začiatku fronty si teda načítame do premennej **`lastTokenType`**, ktorá je deklarovaná typovým identifikátorom auto. Toto kľúčové slovo zabezpečí jej implicitnú typovú inferenciu podľa hodnoty jej inicializačného výrazu. V našom prípade teda prekladač automaticky nahradí kľúčové slovo auto za typ enumerátoru **`TokenType`**. Po úspešnom načítaní typu posledného naskenovaného tokenu využívame rozhodovaciu štruktúru **`switch`** s vetvami podľa jednotlivých typov. Preskúmajme napríklad vetvu spustenú v prípade typu záporného čísla. Ako vieme, záporné číslo môže byť umiestnené iba na začiatku výrazu alebo v bezprostrednej pozícii za ľavou zátvorkou. Keďže sa exekúcia programu dostala až k tomuto bodu, s istotou vieme, že sa nenachádzame na začiatku výrazu. Zostáva nám teda preskúmať už iba splnenie druhej podmienky. Ak táto bude splnená, nastavíme logickú premennú **`CanPush`** hodnotou **`true`**.

```
case Negation:
     if (lastTokenType == LeftParenthesis) CanPush = true;
     break;
```

Pokiaľ podmienka splnená nebude, premenná **`CanPush`** zostane nastavená inicializačnou hodnotou **`false`** a následne bude klientovi odoslaná chybová správa neplatného výrazu. Podobným princípom sme implementovali aj ostatné vetvy rozhodovacieho príkazu **`switch`**. Prejdime si ešte raz jednotlivé typy tokenov a preskúmajme, za akým typom môžu nasledovať, ak nezohľadňujeme už preskúmanú možnosť prázdnej fronty:

- NEGATION (záporné číslo)
  - Predchádzajúci token musí byť typu ľavej zátvorky.
- NUMBER (číslo)
  - Predchádzajúci token môže byť typu ľavej zátvorky alebo operátora.
- BINARYOPERATOR (operátor)
  - Prechádzajúci token môže byť číslo, záporné číslo alebo pravá zátvorka.
- LEFTPARENTHESIS (ľavá zátvorka)
  - Predchádzajúci token môže byť operátor alebo ľavá zátvorka.
- RIGHTPARENTHESIS (pravá zátvorka)
  - Predchádzajúci token môže byť číslo, záporné číslo alebo pravá zátvorka.

V okamihu, keď skener preskúma každý znak aritmetického výrazu, hodnota aktuálnej skenovanej hodnoty **peek** bude nastavená na hodnotu **NULL** a funkcia skenovania končí.

Nasleduje skontrolovanie počítadiel pravých a ľavých zátvoriek a v prípade ich nesúladného počtu je volaná metóda **Missing,** ktorá prijíma jeden formálny parameter typu **char**. Táto metóda má za úlohu zostaviť správu, ktorá požiada klienta o doplnenie chýbajúceho znaku, určeného práve vstupným parametrom.

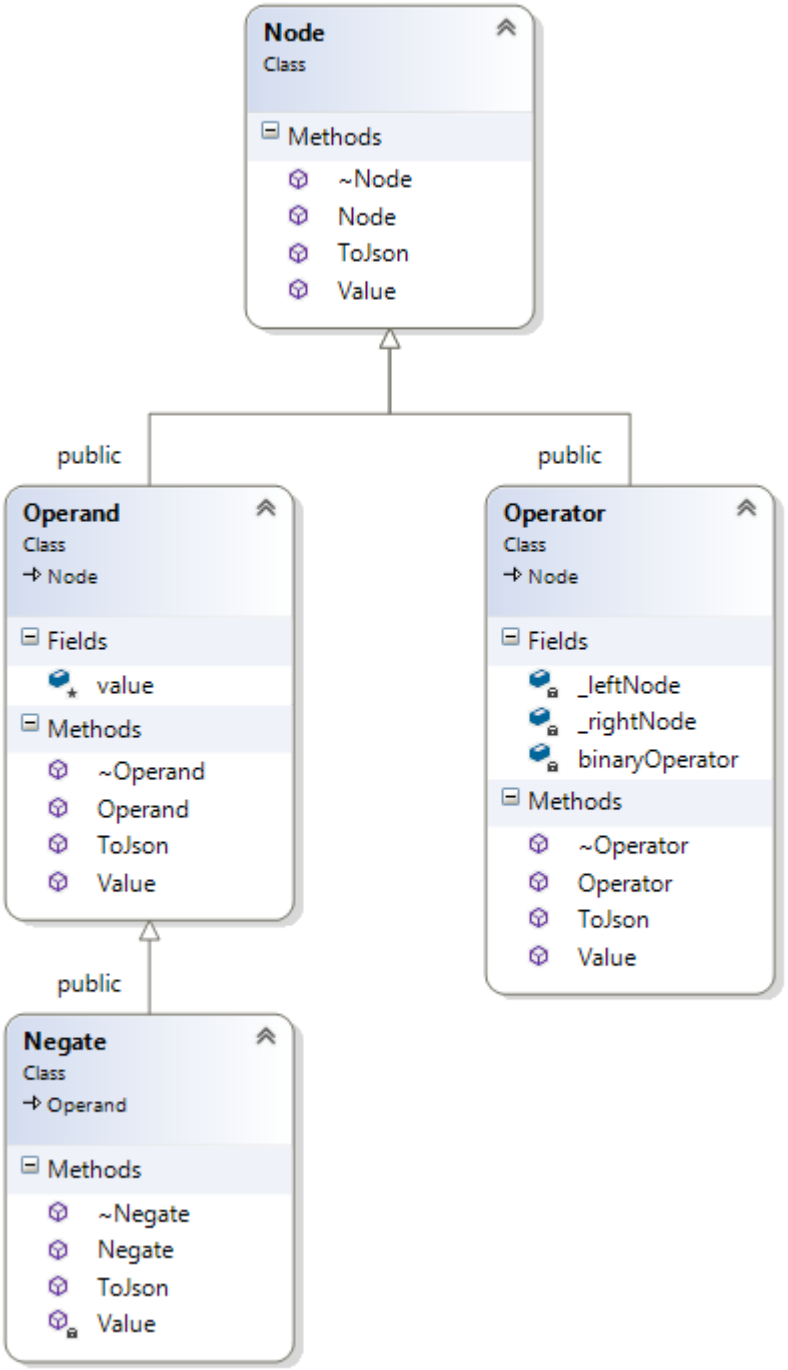
```
if (countOfLeftParentheses < countOfRightParentheses) Missing('(');
else if (countOfRightParentheses < countOfLeftParentheses) Missing(')');
```

Funkcia skenovania môže byť však okrem korektného ukončenia, keď je úspešne naskenovaný každý jeden token výrazu, ukončená aj chybovým spôsobom. Tento nastane v prípade, keď skener prečíta znak, ktorý nie je povoleným v aritmetickom výraze. V takomto prípade skener ukončí svoju činnosť a klientovi je okamžite odoslaná správa o nesprávnom zápise aritmetického výrazu.

3.6. Štruktúra dátových objektov binárneho stromu

V tejto podkapitole sa bližšie oboznámime s jednotlivými objektmi, reprezentujúcimi uzly binárneho stromu. Vo všeobecnosti môže byť pre každý definovaný typ tokenu vytvorený samostatný objekt. Objektová reprezentácia každého typu tokenu v konečnom dôsledku prinesie jednoduchšiu použiteľnosť a pružnejšiu škálovateľnosť vytvorenej aplikácie.

Ako sme už uviedli vyššie, vytvorená aplikácia vyhodnocuje aritmetický výraz a buduje binárny strom dvoma rozdielnymi spôsobmi, pričom výber konkrétnej z nich je určený veľkosťou vstupného výrazu, konkrétne počtom naskenovaných tokenov. Prvá metóda, typická pre binárne stromy, má rekurzívnu povahu a využíva objekty, ktoré sú akoby kópiami jednotlivých typov enumerátora **TokenType.** Druhá metóda, jednoduchšia na implementáciu, využíva iba jeden základný objekt a niekoľko pomocných zabudovaných ADT pre ukladanie medzivýsledkov počas vyhodnocovania binárneho stromu. Preskúmajme najskôr objekty použité pri rekurzívnej metóde, na ktorých ilustrujeme niektoré ďalšie dôležité vlastnosti OOP. Nasledujúca schéma zobrazuje diagram tried týchto objektov spolu so vzťahmi dedičnosti, ktoré sú definované medzi nimi.



Obrázok 16: Diagram tried uzlov stromu

Prvá trieda, ktorá je v aplikácii najdôležitejšia a predstavuje každý jeden uzol binárneho stromu, nesie názov **`Node`**. Ako si môžeme všimnúť, ide o rodičovskú triedu a každý ďalší objekt v diagrame dedí jej vlastnosti. Dedičnosť je základným atribútom objektovo orientovaného programovania a má za cieľ zvýšiť znovupoužiteľnosť kódu. Trieda **`Node`** je zároveň implementovaná ako abstraktná základná trieda (AZT) [2], ktorej úlohou je najmä

popisovať rozhranie triedam, ktoré z nej dedia. Výhodou abstraktnej základnej triedy je skutočnosť, že nemusí definovať metódy, ktoré deklarovala vo svojom rozhraní. Samotná definícia týchto funkcií je totiž často požadovaná práve od svojich potomkov. AZT však môže poskytnúť aj svoju vlastnú definíciu metódy. Táto bude potom použitá v prípade, ak ju potomok neprepíše vlastnou implementáciou. Ako už vyplýva z názvu AZT, jej úlohou je abstrahovať spoločné vlastnosti viacerých tried a vytvoriť tak spoločné rozhranie, ktorým sa jednotlivé podtriedy musia riadiť. V implementácii môžeme potom vytvoriť pole smerníkov na základnú triedu, do ktorého bude povolené ukladať rôzne objekty, ktoré sú v príbuzenskom vzťahu k tejto triede. V prípade exekúcie konkrétnej metódy abstraktnej základnej triedy, poskytnutej prostredníctvom rozhrania, bude vykonaná práve tá alternatíva, ktorá prináleží príslušnému objektu.

V jazyku C++ sa neimplementované funkcie abstraktnej základnej triedy nazývajú tiež ***čisto virtuálne funkcie***. Rozdiel oproti klasickým funkciám je ten, že čisto virtuálne funkcie majú na konci svojej deklarácie výraz **= 0.** Ak deklarácia triedy obsahuje aspoň jednu čisto virtuálnu funkciu, potom nie je možné túto triedu inštanciovať. Znamená to, že nie je možné z takejto triedy vytvoriť konkrétny objekt, slúži iba ako základná trieda. Aby však trieda bola ozajstnou AZT, musí obsahovať aspoň jednu čisto virtuálnu funkciu. Abstraktná trieda **`Node`** disponuje dvoma takýmito funkciami. Prvá slúži na získanie hodnoty uzla a druhá na získanie textovej reprezentácie podstromu uzla vo formáte JSON.

```
class Node
{
public:
      Node();
      virtual ~Node();

      virtual double Value() = 0;
      virtual string ToJson() = 0;
};
```

Ako môžeme vidieť, deklarácia triedy je veľmi jednoduchá. Obsahuje iba konštruktor, deštruktor a dve už spomenuté virtuálne funkcie. Bližšie si môžeme všimnúť deštruktor, ktorý je deklarovaný kľúčovým slovom **`virtual`**. Toto kľúčové slovo zabezpečí, aby sa po deštrukcii objektu zdedenej triedy zavolal aj deštruktor základnej triedy a vykonal tak dodatočnú finalizáciu zničenia objektu, napríklad uvoľnenie operačnej pamäte, uzatvorenie dátových spojení a pod. Použitie nevirtuálneho deštruktora môže mať za následok únik pamäte alebo iné neznáme chyby, ktoré sa prejavia až neskôr a môžu zapríčiniť nežiaduce problémy na serveri aplikácie.

Ďalšou triedou na diagrame tried, ktorá dedí zo základnej abstraktnej triedy, je trieda **Operand**. Operandom sa v aritmetickom výraze nazýva každé číslo, s ktorým bude vykonávaná požadovaná operácia. Aj v našej implementácii je **Operand** triedou, ktorej inštačné objekty budú reprezentovať jednotlivé číselné tokeny aritmetického výrazu.

```
class Operand : public Node
{
public:
	Operand(double Value);
	~Operand();

	double Value();
	string ToJson();

protected:
	double value;
};
```

V deklarácii triedy **Operand** môžeme vidieť verejný konštruktor, ktorý má za úlohu inicializovať chránenú premennú **value** typu **double**. Táto premenná slúži na uchovanie číselnej hodnoty objektu. Ďalej tu vidíme deštruktor a dve základné metódy, ktorých implementácia je požadovaná rodičovskou abstraktnou triedou **Node**. Definícia funkcie **Value** je veľmi jednoduchá a vracia práve hodnotu svojej chránenej premennej **value**. Trochu zložitejšia je definícia funkcie **ToJson**, ktorej úlohou je vrátiť reťazcovú reprezentáciu svojho podstromu. Ako však vieme, operand už v binárnom strome nemá žiadne vetvy a preto jednoducho vráti iba hodnotu svojho uzlu v textovom formáte JSON.

```
string Operand::ToJson()
{
	return "{\"name\": \"" + to_string(Value()) + "\"}";
}
```

Trieda **Negate** slúži pre vytváranie objektov záporných čísel aritmetického výrazu. Záporné číslo je však tiež iba číslom a preto sme sa rozhodli túto triedu implementovať ako špeciálny typ triedy **Operand**. To znamená, že bude dediť všetky jej vlastnosti a predefinuje iba dve základné metódy **Value** a **ToJson** pre získanie zápornej hodnoty premennej **value**.

```
double Negate::Value()
{
	return -value;
}


string Negate::ToJson()
{
	return "{\"name\": \"" + to_string(Value()) + "\"}";
}
```

Poslednou a azda najzložitejšou triedou reprezentujúcou uzol stromu je trieda **Operator.** Inštanciami tejto triedy sú objekty, ktoré uchovávajú posledný a najdôležitejší typ tokenu aritmetického výrazu – operátor. Podobne, ako ostatné, aj trieda **Operator** povinne implementuje dve základné čisté virtuálne metódy svojej rodičovskej triedy. Obsahuje však aj dva nové členy, ktorými sú smerníkové premenné na objekty typu **Node.** Pripomeňme, že takýto smerník môže ukazovať na akúkoľvek inštanciu triedy, ktorá je v príbuzenskom vzťahu k deklarovanej triede. Môže teda odkazovať aj na jej potomkov do ktoréhokoľvek pokolenia.

```
class Operator : public Node
{
public:
      Operator(char BinaryOperator, Node* Left, Node* Right);
      ~Operator();
      double Value();
      string ToJson();

private:
      char binaryOperator;
      Node* _leftNode;
      Node* _rightNode;
};
```

V implementácii triedy prvá smerníková premenná **_leftNode** odkazuje na ľavý podstrom, ktorým môže byť číslo, záporné číslo alebo znova operátor. Pravá smerníková premenná **_rightNode** bude uchovávať pravý podstrom aktuálneho uzla, ktorým môže byť taktiež buď číslo, záporné číslo alebo operátor. Oba tieto smerníky musia byť v každej inštancii triedy **Operator** inicializované, čo je zabezpečené v konštruktore triedy. Ako vieme, každý operátor musí mať ľavý aj pravý operand a preto tieto premenné nesmú zostať nevyplnené.

```
Operator::Operator(char BinaryOperator, Node* Left, Node* Right)
{
      binaryOperator = BinaryOperator;

      _leftNode     = Left;
      _rightNode    = Right;
}
```

Konštruktor triedy zároveň inicializuje aj znakovú premennú **binaryOperator,** ktorá špecifikuje typ operácie a môže nadobúdať hodnoty určujúce aritmetickú operáciu, teda scítanie (+), odčítanie (-), násobenie (*) a delenie (/). Veľmi dôležitým je aj deštruktor triedy **Operator**, ktorý obsahuje príkazy na deštrukciu objektov pravého a ľavého podstromu. Keď sa životnosť inštančného objektu triedy **Operator** končí, nie je už ďalej

potrebné uchovávať objekty uložené v jeho poduzloch a preto je nevyhnutné odstrániť ich z operačnej pamäte počítača a uvoľniť tak výpočtové prostriedky iným procesom.

```
Operator::~Operator()
{
	delete(this->_leftNode);
	delete(this->_rightNode);
}
```

Objekty triedy **`Operator`** reprezentujúce uzol binárnej operácie nad svojim ľavým a pravým podstromom majú definovanú funkciu pre získanie hodnoty uzla nasledovným spôsobom.

```
double Operator::Value()
{
	switch (this->binaryOperator)
	{
	case '+':
		return _leftNode->Value() + _rightNode->Value();
	case '-':
		return _leftNode->Value() - _rightNode->Value();
	case '*':
		return _leftNode->Value() * _rightNode->Value();
	case '/':
		auto divisor = _rightNode->Value();

		if (divisor == 0)
		{
			throw invalid_argument("Delenie nulou!");
		}

		return _leftNode->Value() / divisor;
	}
}
```

Výsledok operácie ja závislý na typu operátora uloženého v znakovej premennej **`binaryOperator`**. Podľa definovaného typu je potom v rozhodovacej štruktúre **`switch`** vykonávaná k nemu príslušná vetva. V každej vetve je potrebné najskôr jednotlivo vyhodnotiť ľavý a pravý podstrom a následne na získaných hodnotách previesť adekvátnu binárnu operáciu. Podľa asociačných pravidiel operátorov [7] bude pre každú operáciu najskôr vyhodnotený ľavý podstrom a to volaním metódy **`Value`** objektu **`_leftNode`**. Následne bude rovnakým spôsobom vyhodnotený aj pravý podstrom. Je potrebné si uvedomiť, že každý takýto podstrom môže byť jednoduchým číslom, záporným číslom alebo znovu binárnym operátorom. V prípade poslednej spomenutej možnosti vyhodnotenie podstromu prebieha v rekurzívnom režime, to znamená, že exekúcia programu smeruje stále viac do hĺbky stromu, až kým nenarazí na vonkajší uzol, ktorý už nemá ďalších potomkov. Ak program narazí na takýto uzol v hĺbke *h*, jeho číselná hodnota bude vrátená volajúcemu uzlu v hĺbke *h-1*, ktorý medzi ňou a hodnotou z druhého

podtromu vykoná príslušnú operáciu a výsledok znova vráti volajúcemu uzlu v hĺbke *h-2*. Takýmto spôsobom sa exekúcia vyhodnocovacej metódy dostane opäť ku koreňu stromu, ktorého konečná hodnota bude indikovať výsledok celého aritmetického výrazu.

Vetva delenia rozhodovacieho príkazu **`switch`** metódy **`Value`** definovanej na objekte triedy **`Operator`** zároveň vykonáva ďalšiu kontrolu správnosti aritmetického výrazu. Ako isto vieme, delenie nulou v matematike nemá zmysel. Ak sa preto v pozícii deliteľa vyskytne číslo 0, klient je okamžite informovaný o tejto chybe.

Druhou dôležitou funkciou objektu triedy **`Operator`**, vyžadovanou rozhraním abstraktnej základnej triedy, je metóda **`ToJson`**, ktorá zabezpečí konverziu aktuálneho uzla a všetkých jeho podstromov do textového reťazca vo formáte JSON.

```
string Operator::ToJson()
{
	string result = "{\"name\": \"";
	result.push_back(binaryOperator);
	result += "\",";
	result += "\"value\": \"" + to_string(Value()) + "\",";

	// append children
	result += "\"children\": [";
	result += _leftNode->ToJson();
	result.push_back(',');
	result += _rightNode->ToJson();
	result += "]}";

	return result;
}
```

Štruktúra správy vyhodnoteného binárneho stromu vo formáte JSON je nasledovná:

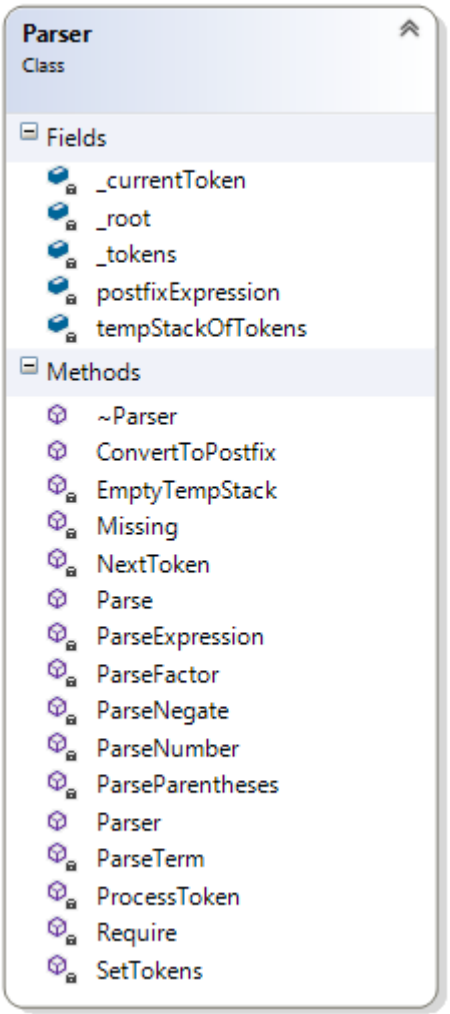
- Každý uzol stromu je definovaný svojím menom. Meno uzla je dané atribútom **`name`** a pri číselných objektoch je reprezentované hodnotou daného uzla. Pri operačných uzloch meno udáva typ aritmetickej operácie, teda +,-,* alebo /.
- Operačný uzol stromu obsahuje aj atribút **`value`**, ktorý udáva hodnotu daného uzla. Takto sú klientskej aplikácii poskytované aj medzivýsledky každej jednotlivej binárnej operácie.
- Každý operačný uzol stromu je rozšírený aj o uzly svojich potomkov. Potomkovia sú definovaní atribútom **`children`** a predstavujú pole dvoch objektov, ktoré reprezentujú pravý a ľavý podstrom daného uzla.

Návratovou hodnotou metódy **`ToJson`** spustenej na koreňovom uzle stromu bude vyhodnotený binárny strom vo formáte JSON, ktorý môže byť ľahko spracovaný klientom.

3.7. Syntaktický analyzátor – Parser

Po preskúmaní základných objektov reprezentujúcich jednotlivé uzly binárneho stromu sa teraz oboznámime s postupom jeho budovania. Vytváranie binárneho stromu zabezpečuje ďalší pracovný objekt aplikácie – `Parser`. Jeho úlohou je na základe prijatého zoznamu tokenov vytvoriť vzájomne prepojený zoznam uzlov stromu.

V prvej kapitole diplomovej práce sme sa oboznámili s dvoma základnými spôsobmi alokácie lineárnych zoznamov. Jednalo sa o sekvenčnú alokáciu, kde jednotlivé položky zoznamu sú uložené bezprostredne za sebou a spojovú alokáciu, pri ktorej každá jedna položka lineárneho zoznamu obsahuje odkaz na nasledujúcu položku. Spojovú alokáciu pamäte sme využili aj pri prvom spôsobe tvorby binárneho stromu, avšak s tým rozdielom, že namiesto jedného odkazu, každý uzol stromu odkazuje až na dve nasledujúce položky, totiž ľavý a pravý podstrom. Túto skutočnosť sme mohli vidieť pri analýze objektu `Operator`, kde premenná `_leftNode` odkazuje na jednu položku zoznamu uzlov stromu a premenná `_rightNode` odkazuje na ďalšiu položku tohto zoznamu.



Obrázok 17: Syntaktický analyzátor - Parser

Preskúmajme teraz jednotlivé atribúty a metódy samotnej triedy `Parser`. Podobne ako skener, aj táto trieda disponuje premennou `_tokens`. Jedná sa taktiež o smerníkovú premennú, ktorá odkazuje na naskenovanú frontu tokenov, ktorých alokovanie zabezpečil skener. Úlohou parsera je postupne pretransformovať každý jeden naskenovaný token na objekt príslušného typu uzla stromu. Poradie spracovávaných tokenov je pritom veľmi dôležité a musí sa striktne riadiť metódou FIFO, to znamená, že parser si vyberá tokeny v takom poradí, v akom boli vložené do zoznamu. Z tohto dôvodu sme zoznam tokenov implementovali ako jednosmernú frontu, ktorá je na tento účel presne stavaná. Parser teda postupne vyberá položku za položkou a priraďuje

ju súkromnej smerníkovej premennej **_currentToken**, ktorá odkazuje na práve spracovávaný token výrazu. Premenná **_root** je deklarovaná ako ukazovateľ na objekt typu abstraktnej triedy **Node** a predstavuje koreňový uzol binárneho stromu. Tento uzol je základným objektom stromu a prostredníctvom neho sa vieme dostať ku všetkým ostatným uzlom, pretože obsahuje ukazovatele na objekty svojho pravého a ľavého podstromu.

Preskúmajme teraz definíciu hlavnej funkcie parsera, ktorej návratovým typom je smerník na koreňový uzol binárneho stromu.

```
Node* Parser::Parse(queue<Token *> * Tokens)
{
	SetTokens(Tokens);

	_root = ParseExpression();

	return _root;
}
```

Hodnotu koreňového uzla si funkcia získava volaním súkromnej metódy **ParseExpression**, ktorá má za úlohu spracovať celý aritmetický výraz v podobe zoznamu tokenov. Ako však môžeme vidieť v nasledujúcej definícii metódy **ParseExpression**, spracovanie je rozdelené na drobnejšie časti aritmetického výrazu, ktorých parsovanie zabezpečuje súkromná metóda **ParseTerm**. Najskôr bude teda spracovaná prvá menšia časť a v prípade, že nasledujúcim tokenom bude operátor sčítania alebo odčítania, spracuje sa aj pravá strana operácie a výsledkom bude uzol typu operátora s ukazovateľmi na svoje operandy.

```
Node* Parser::ParseExpression()
{
	auto _left = ParseTerm();
	while (true)
	{
		if (_currentToken == nullptr || _currentToken->type != BinaryOperator)
		{
			return _left;
		}

		auto op = _currentToken->characterValue;
		if (op == '+' || op == '-')
		{
			NextToken();
			Node* _right = ParseTerm();
			_left = new Operator(op, _left, _right);
		}
		else
		{
			return _left;
		}
	}
}
```

Preskúmajme však ďalej definíciu funkcie **`ParseTerm`**. Táto taktiež rozkladá vyriešenie problému na menšie časti tým, že volá pomocnú funkciu **`ParseFactor`**, a v prípade, že nasledujúcim tokenom bude operátor násobenia alebo delenia, vyžiada si aj vyhodnotenie pravého operandu. V takomto prípade bude výsledným uzlom znova operátor, tentokrát však s vyššou prioritou. V opačnom prípade bude navrátený uzol prvého spracovaného faktora, teda jednoduchý objekt reprezentujúci číslo, záporné číslo alebo zátvorku.

```
Node* Parser::ParseTerm()
{
	auto _left = ParseFactor();

	while (true)
	{
		if (_currentToken == nullptr || _currentToken->type != BinaryOperator)
		{
			return _left;
		}

		auto op = _currentToken->characterValue;

		if (op == '*' || op == '/')
		{
			NextToken();
			Node* _right = ParseFactor();
			_left = new Operator(op, _left, _right);
		}
		else
		{
			return _left;
		}
	}
}
```

Ako si môžeme všimnúť, pri alokovaní nového operátora nie je ihneď navrátená hodnota tohto uzla volajúcej funkcii. Exekúcia programu pokračuje v cykle **`while`**, ktorý bude ukončený až v prípade, ak nasledujúci token nebude operátorom násobenia alebo delenia. Touto funkcionalitou zabezpečíme rekurzívnu stavbu stromu, ktorá preskúma každý token zoznamu a ku každému operačnému uzlu stromu vytvorí príslušný ľavý aj pravý podstrom.

```
Node* Parser::ParseFactor()
{
	switch (_currentToken->type)
	{
	case Number:
		return ParseNumber();
	case Negation:
		return ParseNegate();
	case LeftParenthesis:
		return ParseParentheses();
	}
}
```

Metóda **ParseFactor** má veľmi jednoduchú štruktúru a jej úlohou je na základe typu aktuálne spracovávaného tokenu vytvoriť k nemu príslušný objektu uzla stromu. V prípade čísla bude teda dynamicky vytvorený nový objekt triedy **Operand,** resp. **Negate**, ak sa jedná o záporné číslo. V prípade ľavej zátvorky bude situácia mierne zložitejšia. V takejto situácii je totiž potrebné samostatne vyhodnotiť celý podvýraz v zátvorke, keďže zátvorky majú v aritmetickom výraze najvyššiu prioritu. Vnútorný strom reprezentujúci výraz v zátvorke získame opätovným zavolaním funkcie **ParseExpression.**

```
Node* Parser::ParseParentheses()
{
	NextToken();

	auto _innerExpression = ParseExpression();
	Require(')');

	return _innerExpression;
}
```

Funkcia **ParseFactor** zároveň zabezpečuje spracovanie viacerých vnorených zátvoriek, pričom každý jeden takýto podvýraz bude samostatným podstromom, ktorý bude nakoniec pripojený k príslušnej vetve práve spracovávaného uzla binárneho stromu. Po vyhodnotení vnútorného výrazu je požadovaný token typu uzatváracej zátvorky. Túto požiadavku zabezpečuje metóda **Require**, ktorá v argumente prijíma práve jeden znak. V prípade, ak znaková premenná nasledujúceho tokenu nemá hodnotu tohto znaku, klientskej aplikácii bude vrátená chybová správa žiadajúca o opravu zadaného aritmetického výrazu.

```
void Parser::Require(char c)
{
	if (_currentToken == nullptr || _currentToken->characterValue != c)
	{
		Missing(c);
	}
}
```

Metóda **Missing** má rovnakú štruktúru ako jej rovnomenný náprotivok v triede **Scanner.** Jej úlohou je vyhodiť výnimku typu neplatného argumentu s chybovou správou. Táto výnimka bude následne zachytená v hraničnom objekte **Listener** a správa odoslaná používateľovi.

```
void Parser::Missing(char c)
{
	string error = "Chýba znak '";
	error.push_back(c);
	error.push_back('\'');

	throw invalid_argument(error);
}
```

Po úspešnom spracovaní všetkých tokenov a vytvorení binárneho stromu bude metódou **Parse** navrátený smerník ku koreňovému uzlu **_root**. Každý jeden uzol binárneho stromu je uložený na konkrétnom mieste operačnej pamäte. Po odoslaní textovej reprezentácie stromu klientovi je preto potrebné tieto pamäťové miesta uvoľniť. Keďže vlastníkom dynamicky alokovaných uzlov stromu nie je žiadny objekt aplikácie, môžeme k nim pristupovať iba prostredníkom takého objektu, ktorý disponuje odkazom na prvý uzol stromu. Takýto smerník uchováva práve objekt triedy **Parser** a preto sme sa rozhodli zveriť úlohu dealokovania binárneho stromu práve deštruktoru tejto triedy.

```
Parser::~Parser()
{
	if (_root != nullptr)
	{
		delete _root;
	}
}
```

Na prvý pohľad sa môže zdať, že uvedená definícia deštruktora vymaže z pamäte iba koreňový uzol stromu. Pravda je však taká, že príkaz **delete** vykonaný na koreňovom uzle spustí reťazovú reakciu, pri ktorej budú postupne vymazané všetky jeho listy. Táto reťazová reakcia je zabezpečená deštruktorom triedy **Operator**, ktorý opäť volá príkaz **delete** na oboch svojich listoch. Takýmto spôsobom bude teda z operačnej pamäte počítača skutočne vymazaný každý jeden uzol vytvoreného binárneho stromu.

```
Operator::~Operator()
{
	delete(this->_leftNode);
	delete(this->_rightNode);
}
```

Rekurzívnou metódou prebieha aj vyhodnotenie stromu, spustené na koreňovom uzle. Prostredníctvom smerníkov na nasledujúce uzly program navštívi a vyhodnotí každý jeden list binárneho stromu. Výsledky jednotlivých uzlov sú postupne vyhodnocované, až kým sa exekúcia programu nevráti späť ku koreňovému uzlu, ktorého výsledkom bude hodnota celého stromu. V implementácii aplikácie sme však funkciu **Value** pre vyhodnotenie uzla vložili do metódy **ToJson**, ktorá vypočítanú hodnotu uzla stromu obalí textom a vytvorí tak požadovanú štruktúru výsledku. Textové výsledky každého uzla stromu sú rekurzívne spájané a návratovou hodnotou koreňového uzla bude preto štruktúra celého binárneho stromu vo formáte JSON. Pripomeňme, že binárny strom sme preskúmali metódou **preorder,** kde sme najskôr navštívili uzol, potom jeho ľavý a nakoniec pravý podstrom.

3.7.1. Nerekurzívna stavba stromu

V predchádzajúcej časti sme preskúmali stavbu stromu rekurzívnym spôsobom s využitím spojovej alokácie jednotlivých uzlov. Binárny strom však môžeme zostaviť aj nerekurzívnym spôsobom, pri ktorom sú jednotlivé uzly uložené v zozname za sebou a preto nie je nutné uchovávať smerník na nasledujúci uzol. Aby sme však takto zapísaný výraz mohli jednoducho vyhodnotiť, je potrebné prekonvertovať ho do postfixovej notácie, kde operátor nasleduje až za svojimi operandmi. Na tento účel sme vytvorili konverznú metódu **`ConvertToPostfix`**, ktorá zabezpečí prevod infixovej reprezentácie výrazu na postfixovú reprezentáciu. Metóda prijíma zoznam naskenovaných tokenov a jej návratovým typom je zoznam tých istých tokenov, avšak už s prehodeným poradím.

```
queue<Token *> Parser::ConvertToPostfix(queue<Token *> * Tokens)
{
	SetTokens(Tokens);

	while (_currentToken != nullptr)
	{
		ProcessToken(new Token(_currentToken));
		NextToken();
	}

	EmptyTempStack();
	return postfixExpression;
}
```

Konverzná metóda má jednoduchú štruktúru, ktorá postupne spracováva každý jeden token prijatého zoznamu metódou **`ProcessToken`**. Spracovanie tokenu je riadené jeho typom a využíva pomocnú frontu **`tempStackOfTokens`** na dočasné odkladanie tokenov a výslednú frontu tokenov s názvom **`postfixExpression`**, ktorá je zároveň návratovou hodnotou konverznej metódy. S algoritmom prevodu infixovej notácie do postfixovej sme sa už zoznámili v podkapitole 3.2. a preto teraz uvedieme jednotlivé vetvy metódy **`ProcessToken`**, implementujúce tento algoritmus. Podľa bodu a) konverznej metódy, ak sa dostane na rad číslo, resp. záporné číslo, toto bude umiestené do výstupného postfixového reťazca a pokračuje spracovanie nasledujúceho tokenu. Keďže v našej implementácii je postfixový reťazec reprezentovaný frontou, jednoducho na jej koniec vložíme odkaz na číselný token.

```
if (Token->type == Number || Token->type == Negation)
{
	postfixExpression.push(Token);
	return;
}
```

Bod b) už využíva aj pomocnú frontu operátorov, ktorých pohyb je riadený podľa ich priority. Každý token disponuje súkromnou premennou **precedence**, ktorá určuje jeho prioritu a je nastavená v konštruktore triedy. V prípade číselného tokenu je táto premenná inicializovaná hodnotou 0, v prípade niektorej zátvorky hodnotou 1, operátory sčítania a odčítania budú mať prioritu 2 a operátory násobenia a delenia prioritu 3. Ak práve spracovávaný token má vyššiu prioritu ako token na vrchole zásobníka, alebo zásobník je prázdny, jednoducho bude vložený do tohto zásobníka. Ak má ale tento token prioritu nižšiu, postupne vyberáme z vrcholu zásobníka tokeny s vyššou prioritou a umiestňujeme ich do výstupnej fronty, až kým nenarazíme na token s nižšou prioritou, alebo na dno zásobníka. Následne práve spracovávaný token vložíme do zásobníka.

```
if (tempStackOfTokens.empty() || Token->type == LeftParenthesis)
{
	tempStackOfTokens.push(Token);
	return;
}

auto _tokenOnTheTop = tempStackOfTokens.top();

if (_tokenOnTheTop->precedence < Token->precedence)
{
	tempStackOfTokens.push(Token);
	return;
}
else
{
	while (_tokenOnTheTop->precedence >= Token->precedence)
	{
		postfixExpression.push(_tokenOnTheTop);
		tempStackOfTokens.pop();

		if (tempStackOfTokens.empty())
		{
			break;
		}
		else
		{
			_tokenOnTheTop = tempStackOfTokens.top();
		}
	}

	tempStackOfTokens.push(Token);
	return;
}
```

Podľa bodu c) implementovaného konverzného algoritmu, ak narazíme na ľavú zátvorku, táto putuje za každých okolností do pomocného zásobníka, ako to môžeme vidieť na predchádzajúcom fragmente kódu. Token ľavej zátvorky má však v tomto zásobníku najnižšiu prioritu, ktorá bola v konštruktore nastavená na hodnotu 1.

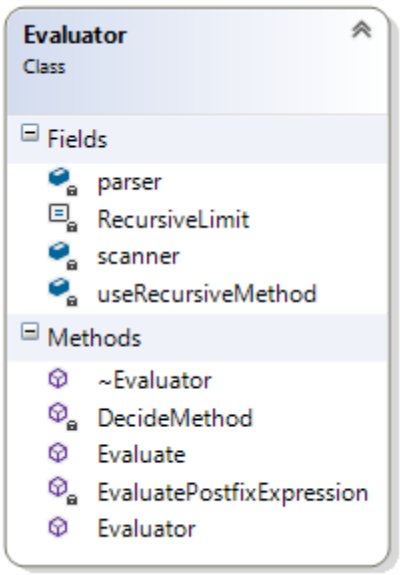
Bod d) rieši problém pravej zátvorky. V takomto prípade premiestňujeme tokeny z dočasného zásobníka do výstupnej fronty dovtedy, kým sa táto zátvorka nespáruje s príslušnou ľavou zátvorkou. Pripomeňme, že zátvorky už v postfixovom zápise nie sú potrebné a z tohto dôvodu budú vymazané z operačnej pamäte počítača príkazom **`delete`**.

```
if (Token->type == RightParenthesis)
{
      while (_tokenOnTheTop->type != LeftParenthesis)
      {
            postfixExpression.push(_tokenOnTheTop);
            tempStackOfTokens.pop();
            _tokenOnTheTop = tempStackOfTokens.top();
      }

      delete _tokenOnTheTop;
      delete Token;
      tempStackOfTokens.pop();
      return;
}
```

Výsledná fronta smerníkov na naskenované tokeny aritmetického výrazu, ktorých poradie bolo metódou **`ConvertToPostfix`** prehodené podľa princípov postixovej notácie, bude vrátená poslednému pracovnému objektu aplikácie, ktorý zabezpečí jej vyhodnotenie.

3.8. Vyhodnocovač – Evaluator

V predchádzajúcich podkapitolách sme sa oboznámili s dvoma dôležitými pracovnými objektmi aplikácie. Najskôr to bol skener, ktorý ako prvý spracúva prijatý aritmetický výraz a pre jednotlivé jeho položky vytvára tokeny príslušného typu. Následne je tento zoznam tokenov odovzdaný objektu triedy **`Parser`**, a to prostredníctvom metódy, od ktorej sa požaduje ich ďalšie spracovanie. Ako sme už uviedli, interpreter implementuje dve základné metódy na vyhodnotenie aritmetického výrazu, pričom výber konkrétnej alternatívy záleží na konečnom počte jeho položiek. Na diagrame triedy **`Evaluator`** môžeme vidieť, že obsahuje súkromné objekty **`parser`** a **`scanner`**. Stretávame sa tu opäť s kompozíciou objektov, čo je veľmi dôležitá vlastnosť objektovo



Obrázok 18: Vyhodnocovač - Evaluator

orientovaného programovania. Vyhodnocovač prijíma požiadavku v tvare aritmetického výrazu a už sám sa postará o vytvorenie objektov skenera a parsera, ktorých funkcionalitu interne využíva. Dodajme, že vytvorenie týchto objektov prebieha v konštruktore triedy **`Evaluator`** a ich dealokovanie v deštruktore tejto triedy.

Nasledujúci fragment kódu zobrazuje deklaráciu vyhodnocovača z hlavičkového súboru **`Evaluator.h`**. Trieda obsahuje súkromnú konštantnú premennú **`RecursiveLimit`**, ktorá slúži na definovanie maximálneho počtu tokenov aritmetického výrazu pre použite rekurzívnej metódy. Na základe tejto konštanty sa vyhodnocovač rozhodne, či využije rekurzívnu metódu alebo nie a toto rozhodnutie si uloží do súkromnej logickej premennej **`useRecursiveMethod`**, ktorú využije neskôr pri rozhodovacej podmienke.

```
class Evaluator
{
public:
	Evaluator();
	~Evaluator();
	string Evaluate(wstring * Expression);
private:
	const int RecursiveLimit = 20;
	bool useRecursiveMethod;
	Scanner * scanner;
	Parser * parser;

	void DecideMethod(queue<Token *> * Tokens);
	string EvaluatePostfixExpression(queue<Token *> Tokens);
};
```

Rozhodovacia metóda má jednoduchú štruktúru založenú na porovnaní veľkosti fronty s konštantou určujúcou maximálny počet tokenov pre použitie rekurzívnej metódy. Ak bude teda počet naskenovaných tokenov väčší ako 20, na vyhodnotenie výrazu sa použije nerekurzívna metóda, v opačnom prípade vyhodnotenie stromu prebehne rekurzívne.

```
void Evaluator::DecideMethod(queue<Token *> * Tokens)
{
	useRecursiveMethod = Tokens->size() <= RecursiveLimit;
}
```

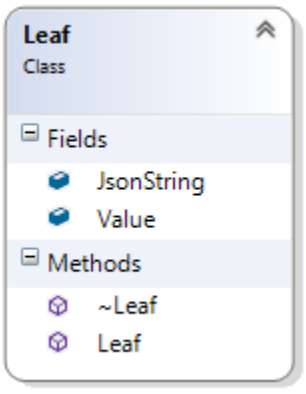
Dynamické vytvorenie objektu triedy **`Evaluator`** je zabezpečené po prijatí správy od klienta, zachytenej v metóde **`handle_post`** objektu **`Listener`**.

```
void Listener::handle_post(http_request request)
{
	wstring expression = request.extract_string().get();
	Evaluator * e = new Evaluator();
	...
}
```

Následne bude vyhodnocovač požiadaný o vyhodnotenie výrazu. Funkcia vyhodnotenia najskôr požiada skener o frontu tokenov, ktoré následne poskytne príslušnej metóde objektu parsera na spracovanie. Výber konkrétnej metódy je závislý na už spomenutej rozhodovacej premennej **`useRecursiveMethod`**, inicializovanej metódou **`DecideMethod`**.

```
string Evaluator::Evaluate(wstring * Expression)
{
	auto tokens = scanner->Scan(Expression);

	DecideMethod(tokens);

	if (useRecursiveMethod)
	{
		cout << "Prebieha vyhodnocovanie rekurzívnou metódou ..." << endl;
		auto root = parser->Parse(tokens);
		return root->ToJson();
	}
	else
	{
		cout << " Prebieha vyhodnocovanie nerekurzívnou metódou ..." << endl;
		auto nodes = parser->ConvertToPostfix(tokens);
		return EvaluatePostfixExpression(nodes);
	}
}
```

Ak exekúcia programu pokračuje rekurzívnou stavbou stromu, výsledkom bude jediný uzol binárneho stromu, ktorým je koreňový uzol. Textová reprezentácia vyhodnoteného stromu vo formáte JSON bude potom získaná volaním metódy **`ToJson`** tohto uzla. V prípade pužitia nerekurzívnej stavby stromu je parser požiadaný o konvertovanie zoznamu tokenov uloženého v infixovom poradí na zoznam v poradí postfixovom. Tento výsledok bude následne odovzdaný metóde **`EvaluatePostfixExpression`**, ktorej návratovou hodnotou bude taký istý textový formát vyhodnoteného binárneho stromu, ako pri použití rekurzívnej metódy, avšak vyskladaný už iným – nerekurzívnym spôsobom. Na tento účel sme do aplikácie implementovali nový objekt reprezentujúci list stromu – **`Leaf`**. Štruktúra tohto objektu je veľmi jednoduchá a okrem konštruktora a deštruktora obsahuje dve verejné premenné uchovávajúce hodnotu daného uzla. Prvá premenná nesie názov **`Value`** a podobne ako pri objekte triedy **`Node`**, aj tu slúži pre uloženie číselnej hodnoty príslušného uzla. V prípade operátora je týmto uzlom samostatný podstrom a preto je táto premenná inicializovaná vypočítanou hodnotou celého podstromu. Premenná **`JsonString`** objektu **`Leaf`** zapuzdruje vypočítanú hodnotu daného listu do textového formátu JSON.



Obrázok 19: Objekt Leaf

Po preskúmaní triedy **`Leaf`** môžeme analyzovať samotnú metódu vyhodnotenia postixového výrazu. Jej základnou úlohou je spracovať každý jeden token v zozname, ktorého smerník prijala prostredníctvom formálneho argumentu. Pripomeňme, že v tomto zozname sa už nenachádzajú zátvorky, pretože v postfixovej notácii je priorita operátorov už určená ich poradím.

```
string Evaluator::EvaluatePostfixExpression(queue<Token *> Tokens)
{
	stack<Leaf> stackOfLeafs;

	while (!Tokens.empty())
	{
		auto _tokenOnTheFront = Tokens.front();

		...

		delete _tokenOnTheFront;
		Tokens.pop();
	}

	return stackOfLeafs.top().JsonString;
}
```

Vyhodnocovacia metóda spracováva každý jeden token v poradí, ktoré je typické pre frontu, teda prvý vložený token bude vybraný ako prvý. Prechádzame teda zoznam tokenov v postfixovej notácii zľava doprava. Po úspešnom spracovaní tokenu bude vykonané jeho dealokovanie z pamäte počítača a následne odstránenie z fronty. Ešte pred samotnou deštrukciou je však z každého tokenu vytvorený nový objekt triedy **`Leaf`** a tento je uložený v lokálnom zásobníku listov stromu **`stackOfLeafs`**. Návratovou hodnotou metódy **`EvaluatePostfixExpression`** bude hodnota premennej **`JsonString`** toho uzla, ktorý sa nachádza na vrchole zásobníka, teda koreňového uzla, ktorý je zároveň aj jediným uzlom v zásobníku po exekúcii vyhodnocovacieho algoritmu.

Preskúmajme ďalej spracovanie jednotlivých typov tokenov podľa vyhodnocovacieho algoritmu definovaného v podkapitole 3.2. Bod a) uvedeného algoritmu hovorí, že ak narazíme na operand, putuje do zásobníka. Pre každý číselný token vytvoríme teda novú inštanciu listu s príslušnými hodnotami a vložíme ho na vrchol zásobníka listov.

```
if (_tokenOnTheFront->type == Number)
{
	stackOfLeafs.push(Leaf(
		_tokenOnTheFront->numericalValue,
		"{\"name\": \"" + to_string(_tokenOnTheFront->numericalValue) + "\"}"
	));
}
```

Obdobná situácia nastane aj v prípade, keď sa dostane na rad číslo so zápornou hodnotou. Jediným rozdielom bude vynásobenie hodnoty nového listu číslom -1.

```
else if (_tokenOnTheFront->type == Negation)
{
     stackOfLeafs.push(Leaf(
         _tokenOnTheFront->numericalValue * -1,
         "{\"name\": \"" + to_string(_tokenOnTheFront->numericalValue * -1) + "\"}"
     ));
}
```

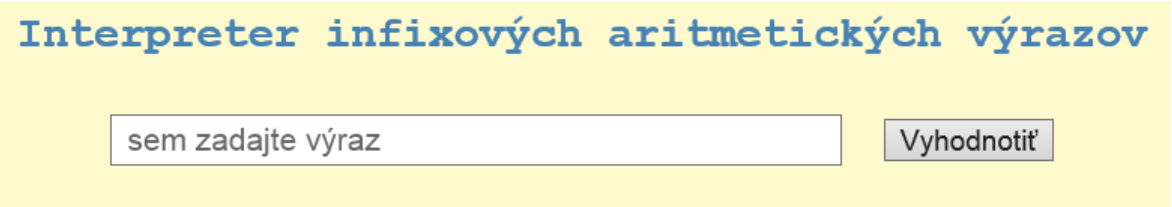
Zložitejšie spracovanie prebieha pre operačné tokeny, pre ktoré je potrebné zo zásobníka vybrať dva operandy, prehodiť ich poradie a následne vykonať s nimi príslušnú operáciu. Pri operácii delenia je nutné zabezpečiť kontrolu, aby hodnota deliteľa nebola nulová.

```
else if (_tokenOnTheFront->type == BinaryOperator)
{
      auto rightLeaf = stackOfLeafs.top();
      stackOfLeafs.pop();
      auto leftLeaf = stackOfLeafs.top();
      stackOfLeafs.pop();

      double value = 0;

      switch (_tokenOnTheFront->characterValue)
      {
      case '+':
            value = leftLeaf.Value + rightLeaf.Value;
            break;
      case '-':
            value = leftLeaf.Value - rightLeaf.Value;
            break;
      case '*':
            value = leftLeaf.Value * rightLeaf.Value;
            break;
      case '/':
            if (rightLeaf.Value == 0) throw invalid_argument("Delenie nulou!");
            value = leftLeaf.Value / rightLeaf.Value;
            break;
      }

      string jsonString = "{\"name\": \"";
      jsonString.push_back(_tokenOnTheFront->characterValue);
      jsonString += "\",";
      jsonString += "\"value\": \"" + to_string(value) + "\",";

      // append children
      jsonString += "\"children\": [";
      jsonString += leftLeaf.JsonString;
      jsonString.push_back(',');
      jsonString += rightLeaf.JsonString;
      jsonString += "]}";

      stackOfLeafs.push(Leaf(value, jsonString));
}
```

Ako môžeme vidieť v predchádzajúcej vetve metódy `EvaluatePostfixExpression`, k textovej premennej `jsonString` operátora budú pripojené aj textové hodnoty jej ľavého aj pravého podstromu. To znamená, že textová reprezentácia `jsonString` koreňového uzla bude skutočne obsahovať celý vyhodnotený binárny strom v požadovanej štruktúre JSON.

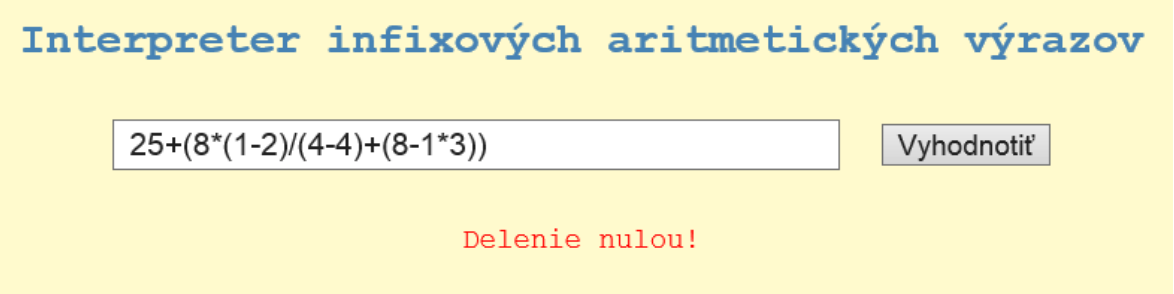
3.9. Jednoduchý HTML klient ako konzument aplikácie

Vytvorený interpreter slúži ako server, ktorý prijíma od klientov správy s aritmetickým výrazom a odosiela späť vyhodnotený binárny strom tohto výrazu v textovom formáte JSON. Aby sme mohli demonštrovať možné spracovanie výsledného stromu, vytvorili sme jednoduchého klienta aplikácie, ktorý odosiela a prijíma správy prostredníctvom hypertextového prenosového protokolu HTTP. Využili sme pritom výhody jazykov HTML a JavaScriptu, ktoré v dnešnej dobe poskytujú veľmi bohatú platformu pre vývoj platformovo nezávislých interaktívnych aplikácií. Úvodná webová stránka klientskej aplikácie obsahuje jeden formulár, ktorý je tvorený vstupným textovým poľom pre aritmetický výraz a tlačidlom pre odoslanie zadaného výrazu.



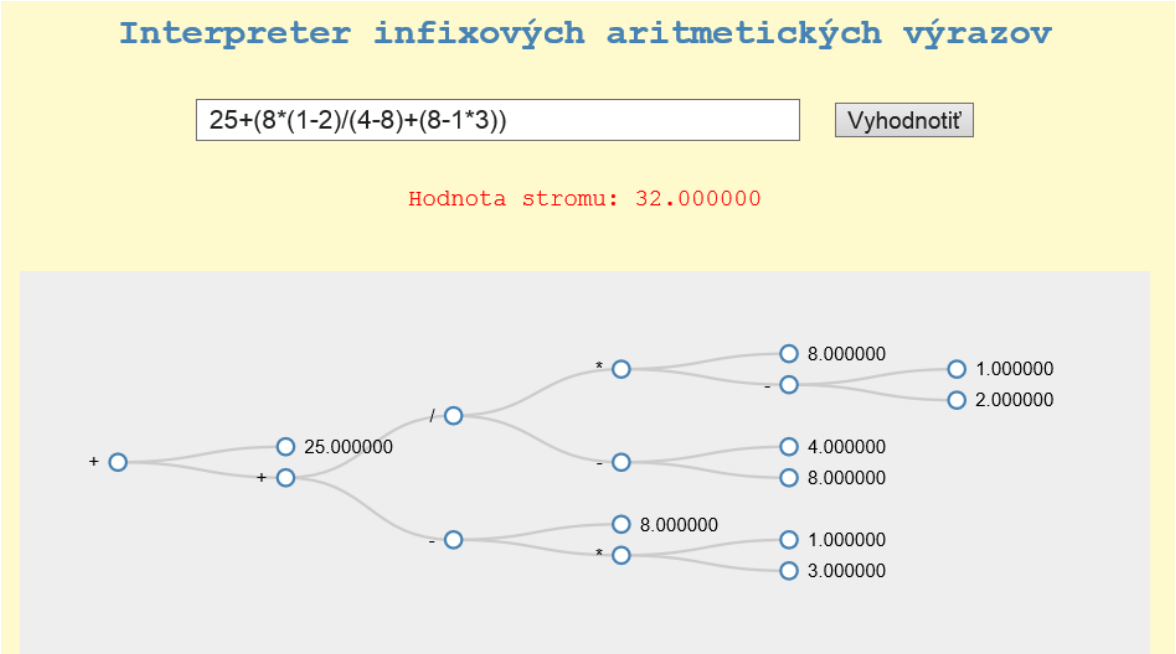
Obrázok 20: Úvodná webová stránka klientskej aplikácie

V prípade odoslania neplatného výrazu bude používateľ informovaný chybovou správou.



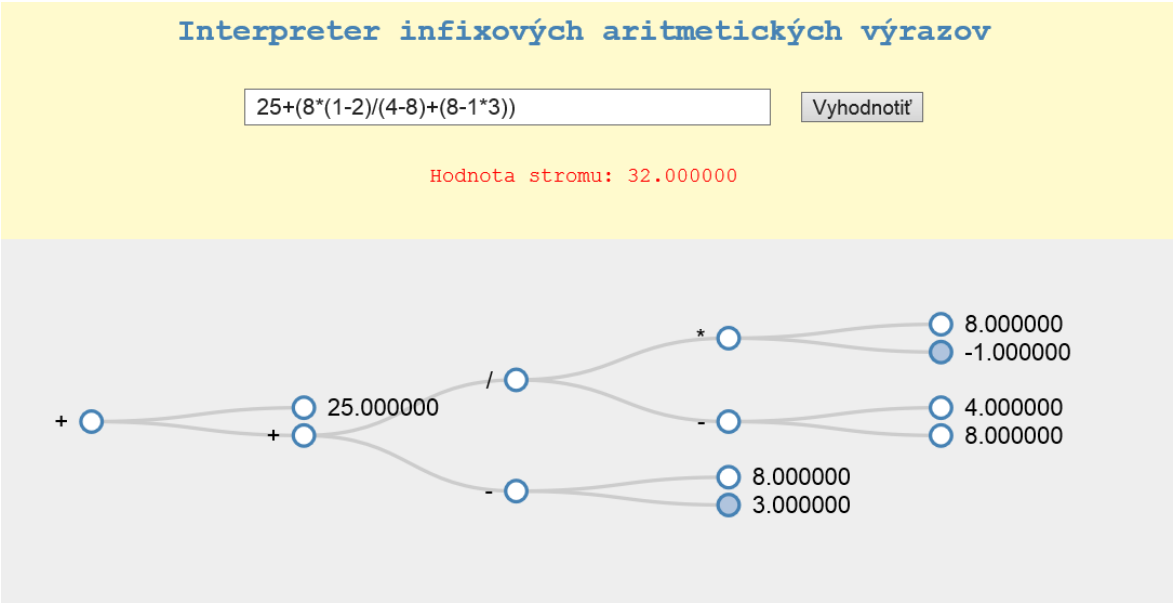
Obrázok 21: Chybová správa

Ak zadaný výraz bude mať správnu formu, klientska aplikácia príjme od servera vyhodnotený binárny strom, o ktorého vykreslenie sa postará JavaScriptová knižnica D3.js.



Obrázok 22: Vyhodnotený binárny strom

Po kliknutí na jednotlivé uzly stromu ich môžeme zabaliť a zobraziť tak medzivýsledky každého podstromu. Táto funkcionalita je možná vďaka tomu, že každý uzol stromu bol na serveri vyhodnotený zvlášť a jeho hodnota bola vložená do JSON formátu stromu.



Obrázok 23: Zbaľovanie uzlov stromu

Záver

Abstraktné dátové typy sú veľmi významným nástrojom v oblasti softvérového inžinierstva. Svoje využitie nachádzajú v rôznych implementáciách, od základných druhov algoritmov až po robustné počítačové aplikácie. Vďaka abstraktným dátovým typom môžeme využívať raz napísaný počítačový kód na viac účelov. V objektovo orientovanom programovaní je to veľmi dôležitá vlastnosť a nazýva sa tiež ako znovupoužiteľnosť kódu. Jej hlavným prínosom je niekoľkonásobné zefektívnenie a uľahčenie práce programátorov. To je dosiahnuté tým, že odbremeňuje vývojárov od nutnosti vytvárať taký objekt, ktorý už napísal iný programátor. Prostredníctvom rozhrania k takémuto abstraktnému dátovému typu je veľmi jednoduché využívať jeho funkcionalitu. Koncept abstraktných dátových typov umožňuje vytvárať rozsiahle systémy od nižších až po vyššie stupne abstrakcie. Jeden abstraktný dátový typ môže zapuzdrovať iný a tak využívať jeho výhody. ADT umožňujú vytvárať stále výkonnejšie abstraktné mechanizmy, ktoré zabezpečia efektívnejšie využitie počítačových systémov pre riešenie úloh.

V diplomovej práci sme sa zamerali na objasnenie pojmu abstraktných dátových typov, analyzovali sme význam ich použitia a uviedli sme najpoužívanejšie druhy ADT. Preskúmali sme dva základné spôsoby alokácie lineárnych zoznamov v operačnej pamäti počítača – sekvenčnú a spojovú alokáciu, pričom sme analyzovali výhody aj nevýhody každej z nich v konkrétnych situáciách. Ďalej sme sa zamerali na oblasť stromov v počítačových algoritmoch a rozobrali sme základné vlastnosti stromových štruktúr. Podrobnejšie sme sa zoznámili s binárnymi stromami, s možnosťami prechodu ich uzlami a so základnými matematickými vlastnosťami binárnych stromov. Prvú kapitolu sme ukončili analýzou infixového, prefixového a postfixového zápisu aritmetických výrazov a rozborom konverzného algoritmu na prevod výrazu z infixovej do postfixovej notácie.

V druhej kapitole diplomovej práce, ktorá mala za úlohu aplikovať teoretické vedomosti do konkrétnej implementácie, sme sa oboznámili s postupom práce tvorby interpretera infixových aritmetických výrazov. Aplikácia bola vybudovaná pomocou architektúry klient – server, ktorá prináša nespornú výhodu platformovej nezávislosti klientskej aplikácie, využívajúcej služby servera. Server sme vytvorili z natívnej konzolovej C++ aplikácie. Jazyk C++ poskytuje veľmi bohatú platformu na budovanie natívnych aplikácií. Aplikácia teda slúži ako server, ktorý počúva na vopred špecifikovanej IP adrese.

V okamihu, keď server príjme požiadavku od klienta, vykoná sa jej spracovanie a odoslanie odpovede späť používateľovi. Klientom pritom môže byť hocijaký program, schopný odosielať internetové správy metódou POST. Takýto program môže byť implementovaný v rôznych jazykoch, ako napr. C, C++, C#, Visual Basic, Java, HTML, PHP, JavaScript a množstvo iných.

Ďalej sme rozoberali jednotlivé pracovné aj dátové objekty serverovej časti aplikácie a samotnú funkcionalitu týchto objektov. Definovali sme základnú štruktúru spracovania aritmetického výrazu, kde je najskôr tento výraz skenovaný a rozložený na samostatné tokeny. Tieto sú následne parsované do podoby syntaktického binárneho stromu a tento je nakoniec vyhodnotený. Využili sme vlastnosti objektovo orientovaného programovania, ako je dedičnosť a zapuzdrenie objektov, alebo abstraktné základné triedy. Využitie týchto základných vlastností sme demonštrovali na objektoch reprezentujúcich jednotlivé typy uzlov binárneho stromu. Pre samostatnú stavbu stromu a jeho vyhodnotenie sme implementovali dve základné alternatívy. Prvú s rekurzívnou metódou, využívajúcou spojovú alokáciu jednotlivých uzlov stromu, a druhú s nerekurzívnou metódou, využívajúcou sekvenčnú alokáciu uzlov. Výber konkrétnej metódy sme podmienili veľkosťou prijatého aritmetického výrazu.

V poslednej časti práce sme objasnili postup tvorby jednoduchého klienta, ktorý má za úlohu sprístupniť aplikačnú logiku interpretera konečnému používateľovi. Vytvorili sme jednoduchý HTML formulár s textovým poľom pre aritmetický výraz a potvrdzujúcim tlačidlom. Funkcionalitu klientskej časti aplikácie sme rozšírili prostredníctvom JavaScriptu. Využili sme pritom výhody webového prehliadača Internet Explorer verzie 11, ktorý umožňuje asychrónne odosielať správy na lokálnu IP adresu počítača. Na tento účel sme využili metódu `ajax` z knižnice jQuery. Ďalšou nápomocnou JavaScriptovou knižnicou nám bola D3.js (Data-Driven Documents), ktorej miernou úpravou sme dosiahli vykreslenie vyhodnoteného binárneho stromu na obrazovku používateľa.

Interpreter infixových aritmetických výrazov by bolo možné do budúcnosti rozšíriť aj o ďalšie matematické operácie, ako sú napríklad goniometrické funkcie, logaritmické funkcie, funkcie mocnín a odmocnín, percenta a faktoriálu. Ďalším prínosom by mohlo byť zabudovanie podpory pre rôzne konštantné premenné, zadávané znakovými hodnotami, ako je napr. premenná pre Eulerovo číslo (e) alebo premenná pre Ludolfovo číslo (π).

Zoznam použitej literatúry

[1] D. E. Knuth, Umění programování, Brno: Computer Press, a.s., 2008.

[2] S. Prata, Mistrovství v C++, Brno: Computer Press, a.s., 2007.

[3] R. Sedgewick, Algoritmy v C, Praha: SoftPress, s.r.o., 2003.

[4] Wikipedia, „Postfixová notace," [Online]. Available: http://cs.wikipedia.org/wiki/Postfixov%C3%A1_notace. [Cit. 15 04 2014].

[5] P. Borovanský, „Kalkulačka (vyhodnocovač matematických výrazov)," [Online]. Available: http://dai.fmph.uniba.sk/courses/PARA/Prednasky/yon_lecture_calc_full.pdf. [Cit. 15 04 2014].

[6] J. Hanák, C++: Akademický výučbový kurz, Bratislava: Vydavateľstvo EKONÓM, 2010.

[7] J. Hanák, Programování v jazyce C, Kralice na Hané: Computer Media s.r.o., 2010.

[8] Wikipedia, „Prefixová notace," [Online]. Available: http://cs.wikipedia.org/wiki/Prefixov%C3%A1_notace. [Cit. 15 04 2014].

[9] M. Balík, „Abstraktní datový typ," [Online]. Available: https://edux.fit.cvut.cz/oppa/BI-PA1/prednasky/pa1-13-adt.pdf. [Cit. 15 04 2014].

[10] I. Krivyakov, „MathConverter - How to Do Math in XAML," [Online]. Available: http://www.codeproject.com/Articles/239251/MathConverter-How-to-Do-Math-in-XAML. [Cit. 15 04 2014].

[11] E. Roberts, „Expression Trees," [Online]. Available: http://cs.stanford.edu/people/eroberts/courses/cs106b/chapters/14-expression-trees.pdf. [Cit. 15 04 2014].

[12] Premshee Pillai, „Infix to Postfix Conversion," [Online]. Available: http://scriptasylum.com/tutorials/infix_postfix/algorithms/infix-postfix/index.htm. [Cit. 15 04 2014].