Žilinská univerzita v Žiline

Fakulta riadenia a informatiky

Katedra informatiky

113/2016

KNIŽNICA PRE MANIPULÁCIU S LOGICKÝMI FUNKCIAMI

DIPLOMOVÁ PRÁCA

Študijný program:	Informačné systémy
Študijný odbor:	Spracovanie dát
Vedúci diplomovej práce:	Ing. Miroslav Kvaššay, PhD.

Patrik Rusnák

Žilina, 2017

Žilinská univerzita v Žiline

Fakulta riadenia a informatiky

Katedra informatiky

113/2016

KNIŽNICA PRE MANIPULÁCIU S LOGICKÝMI FUNKCIAMI

DIPLOMOVÁ PRÁCA

Študijný program:	Informačné systémy
Študijný odbor:	Spracovanie dát
Vedúci diplomovej práce:	Ing. Miroslav Kvaššay, PhD.

Patrik Rusnák

Žilina, 2017

Čestné vyhlásenie

Prehlasujem, že som zadanú diplomovú prácu vypracoval samostatne pod odborným vedením vedúceho diplomovej práce Ing. Miroslava Kvaššaya, PhD. a používal som výlučne zdroje uvedené v zozname použitej literatúry.

V Žiline, dňa 3.5.2017

.....

Patrik Rusnák

Poďakovanie

Veľmi rád by som sa touto cestou poďakoval svojmu vedúcemu diplomovej práce Ing. Miroslavovi Kvaššayovi, PhD. za jeho odborné vedenie, neoceniteľné rady a exaktné podnety, ktoré mi veľmi pomohli pri dokončovaní diplomovej práce.

Abstrakt

Patrik Rusnák: Knižnica pre manipuláciu s logickými funkciami
[Diplomová práca]
Žilinská Univerzita v Žiline, Fakulta riadenia a informatiky, Katedra informatiky.
Vedúci diplomovej práce: Ing. Miroslav Kvaššay, PhD.
Stupeň odbornej kvalifikácie: Inžinier v študijnom odbore Informačné systémy, zameranie Spracovanie dát
FRI ŽU v Žiline, 2017 (53s.)

Cieľom diplomovej práce je vytvoriť knižnicu pre efektívnu manipuláciu s booleovskými a viachodnotovými logickými funkciami. Práca je rozdelená na dve časti. V prvej časti sa nachádza teoretický základ booleovskej a viachodnotovej logiky. Predstavené sú tu pojmy ako booleovská algebra, viachodnotová logika, logická funkcia, normálne formy a logické derivácie. Druhá časť sa sústreďuje na predstavenie softvérovej knižnice vytvorenej v jazyku C++, pomocou ktorej je možné efektívne manipulovať s booleovskými a viachodnotovými logickými funkciami. Taktiež sú tu predstavené použité algoritmy pre parsovanie a pre spracovanie výsledku parsovania do viaccestného stromu. Prínos práce je vo využití softvérovej knižnice pri práci s logickými funkciami, taktiež v rozšíriteľnosti knižnice o ďalšie algebraické systémy.

Kľúčové slová:
Logická funkcia, logická derivácia, booleovská algebra, viachodnotová logika, údajové štruktúry

Abstract

Patrik Rusnák: Library for computations with logic functions
[Engineering thesis]
University of Žilina, Faculty of Management Science and Informatics, Department of informatics.
Tutor: Ing. Miroslav Kvaššay, PhD.
Qualification degree: Engineer in field Informatic systems, aim Data processing
FRI ŽU v Žiline, 2017 (53p.)

The aim of this thesis is to create a library for efficient manipulation of Boolean and multi-valued logic functions. This work is divided into two parts. The first part is the theoretical basis of the Boolean and many-valued logic. Concepts such as the Boolean algebra, many-valued logic, logic function, normal forms and logic derivatives are presented here. The second part focuses on presentation of the software library written in C++, which is used for efficient manipulation with the Boolean and many-valued logic functions. There are also presented algorithms used for the parsing and processing the result of parsing in form of multiway tree. The contribution of this work is in usage of the software library to work with logic functions and in opportunity to extend the library by other algebraic systems.

Keywords:
Logic function, logic derivative, Boolean algebra, multivalued logic, data structures

Obsah

Úvod

„Logika je síce neotrasiteľná, ale človeku, ktorý chce žiť, neodolá.“

Franz Kafka

Vo svete existuje množstvo javov, ktoré je možné ohodnotiť len pomocou dvoch hodnôt, či už pravda – nepravda, áno – nie, funkčný – pokazený alebo správny - nesprávny. Pre prácu s týmito hodnotami a ich reprezentáciu bola vyvinutá booleovská algebra, ktorá pracuje s hodnotami 0, resp. false a 1, resp. true a taktiež poskytuje operácie, ktoré umožňujú spracovať tieto hodnoty. S booleovskou algebrou sa dnes pracuje v takých odvetviach, akými sú napríklad elektronika, digitálna logika a v odvetviach matematiky, konkrétne výroková logika, teória hier či teória spoľahlivosti.

Avšak nie vždy sú iba dve hodnoty postačujúce. Príkladom môžu byť tvrdenia, o ktorých je ťažké s istotou povedať, či sú pravdivé alebo nie, napríklad „o týždeň bude pekne“ alebo „Vo vesmíre existuje planéta okrem zeme, ktorá je obývaná živými bytosťami“ prípadne „Mačka v boxe je mŕtva“ (v pokuse Schrödingera s mačkou). Taktiež je niekedy potrebné uvažovať nad tým, že systém nie je len funkčný alebo nefunkčný, ale jeho funkčnosť sa môže definovať vo viacerých stavoch, napríklad plne funkčný, funkčný, čiastočne funkčný a nefunkčný. Pre riešenie nielen týchto problémov vznikla viachodnotová logika, ktorá pracuje s viac ako dvomi hodnotami. Najznámejšie sú trojhodnotové logiky, v ktorých je tretia hodnota väčšinou definovaná ako „neznáma“, ale používané sú aj viachodnotové konečné logiky a viachodnotové nekonečné logiky, ako napríklad fuzzy logika či pravdepodobnostná logika.

Na to, aby bolo možné v booleovskej alebo viachodnotovej logike popísať javy alebo činnosti, je potrebné ich vyjadriť pomocou logickej funkcie, ktorá na základe hodnôt vstupných veličín vyjadrí výstup. Logickú funkciu je možné reprezentovať viacerými spôsobmi, pričom jedným z najpoužívanejších spôsobov je reprezentácia funkcie formou logického výrazu. Existujú špeciálne formy logického výrazu logickej funkcie nazývané normálne formy, ktoré sa využívajú hlavne v elektrotechnike, teórií spoľahlivosti či v teórii kódovania.

Pri logických funkciách je možné pomocou logických derivácií skúmať ich vlastnosti a ich závislosti na vstupných premenných. Avšak vykonávanie týchto symbolických, prípadne numerických výpočtov pre logické funkcie môže byť časovo

náročné a problematické. Preto je cieľom tejto práce vytvoriť softvérovú knižnicu, ktorá by umožňovala efektívnu prácu s logickými funkciami a v prípade potreby by ju bolo možné rozšíriť o ďalšie operácie, či algebraické systémy.

Práca sa skladá z dvoch častí. Prvá je teoretická a slúži na vysvetlenie pojmov ako booleovská algebra, booleovská funkcia, základné booleovské funkcie, viachodnotová logika, viachodnotová logická funkcia, operácie vo viachodnotovej logike, derivácia logických funkcií, jej priebeh či vysvetlenie niektorých typov derivácie. V druhej časti je predstavená knižnica v jazyku C++, ktorá má efektívne manipulovať s booleovskými a viachodnotovými logickými funkciami, pričom táto knižnica umožní vykonávať symbolické aj numerické výpočty, ako napríklad výpočet logických derivácií. To znamená, že tu budú predstavené dátové štruktúry použité pri vývoji knižnice, ich integrácia a použitie v knižnici a taktiež triedy a ich význam v knižnici.

1 Základné pojmy

Táto časť práce sa zameriava na vysvetlenie teoretických pojmov, ktoré je potrebné objasniť a osvojiť si pred predstavením samotnej softvérovej knižnice. Vysvetlené tu budú pojmy ako booleovská algebra, viachodnotová logika, jej reprezentácie, logická funkcia, či už viachodnotová alebo booleovská, spolu s jej spôsobmi reprezentácie, taktiež normálne formy logickej funkcie a logická derivácia logických funkcií .

1.1 Booleovská algebra

V dnešnej dobe je väčšina digitálnych počítačov založená na dvojhodnotovom, či binárnom systéme. Najznámejší a najpoužívanejší binárny systém je booleovská algebra. Nasledovný text vychádza hlavne z prác **[**1**], [**2**], [**3**], [**4**].**

V booleovskej algebre sú definované operácie spojenia a prieseku a taktiež tu je definovaná unárna operácia komplementu. Preto je booleovská algebra v abstraktnej algebre definovaná ako komplementárny a distributívny zväz, pričom tento typ algebraickej štruktúry obsahuje základné vlastnosti aj množinových a aj logických operácií. Pred tým, než bude zadefinovaná booleovská algebra, je potrebné zadefinovať určité základné pojmy.

Zväz $\mathbf{L}$ je tvorený čiastočne usporiadanou množinou $\mathbb{L}$ na ktorej je pre každé dva elementy x, y množiny $\mathbb{L}$ definované minimum (minimálna hodnota) a maximum (maximálna hodnota), teda platí:

$$(\forall x, y \in \mathbb{L})(\exists i, s \in \mathbb{L})\big((i = \min(x, y)) \wedge (s = \max(x, y))\big).$$

Operácia minimum sa tiež nazýva priesek a označuje sa znakom $\wedge$, pri operácii maximum sa používa označenie $\vee$ a pomenovanie spojenie. Zväz $\mathbf{L}$ je potom možné zapísať v tvare $(\mathbb{L},\wedge,\vee)$.

Distributívny zväz je taký zväz, v ktorom pre všetky x, y a z patriace do množiny $\mathbb{L}$ platia nasledovné vzťahy:

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \qquad x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$$

Ohraničený zväz $\mathbf{L} = (\mathbb{L},\wedge,\vee,0,1)$ je taký zväz $(\mathbb{L},\wedge,\vee)$, v ktorom pre konštanty $0, 1$ patriace do množiny $\mathbb{L}$ platia vzťahy:

$$\forall x \in \mathbb{L}, x \wedge 0 = 0 \text{ a zároveň } x \vee 0 = x$$
$$\forall x \in \mathbb{L}, x \wedge 1 = x \text{ a zároveň } x \vee 1 = 1$$

Konštanta 1 je nazývaná ako horné ohraničenie alebo maximum zväzu $\mathbf{L}$ a konštanta 0 je nazývaná ako dolné ohraničenie alebo minimum zväzu $\mathbf{L}$.

Komplementárny zväz je algebraická štruktúra $(\mathbb{L},\wedge,\vee,\neg,0,1)$, kde $(\mathbb{L},\wedge,\vee,0,1)$ je ohraničený zväz a pre každý element x patriaci do množiny $\mathbb{L}$ existuje komplementárny element $\neg x$ patriaci do množiny $\mathbb{L}$, pričom platí:

$$x \wedge \neg x = 0$$
$$x \vee \neg x = 1$$

Na základe všetkých predošlých definícií je možné definovať Booleovu algebru nasledovne:

Booleovská algebra je usporiadaná šestica $(\mathbb{A},\wedge,\vee,\neg,0,1)$, kde $\mathbb{A}$ je množina, na ktorej sú definované binárne operácie prieseku označené symbolom $\wedge$ a spojenia označené symbolom $\vee$, unárna operácia komplementu označená symbolom $\neg$ a dva elementy 0 a 1 (nazývané minimum a maximum, tiež označované aj symbolom $\bot$ resp. $\top$), taká že pre všetky elementy a, b a c množiny $\mathbb{A}$ platia nasledovné axiómy:

1. ***Asociatívnosť***

$$a \wedge (b \wedge c) = (a \wedge b) \wedge c \qquad a \vee (b \vee c) = (a \vee b) \vee c$$

2. ***Komutatívnosť***

$$a \wedge b = b \wedge a \qquad a \vee b = a \vee b$$

3. ***Zákon absorpcie***

$$a \wedge (a \vee b) = a \qquad a \vee (a \wedge b) = a$$

4. ***Neutrálny prvok***

$$a \wedge 1 = a \qquad a \vee 0 = a$$

5. ***Distributívnosť***

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c) \qquad a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$

6. Komplement

$$a \wedge \neg a = 0 \qquad a \vee \neg a = 1$$

Podľa posledných troch axióm (4-6), resp. zo zákona absorpcie je zrejmé, že a = b ∧ a práve vtedy, keď a ∨ b = b. Potom relácia ≤ definovaná ako $a \leq b$ je čiastočné usporiadanie s najmenším elementom 0 a najväčším elementom 1. Operácia spojenia $a \wedge b$ resp. operácia zjednotenia a ∨ b dvoch elementov sa zhoduje s hodnotou, akú má ich infinum, resp. ich suprémum s ohľadom na reláciu ≤.

Keď si pozorne všimneme zoznam dvojíc axióm, tak zistíme, že keď v jednej axióme zmeníme operáciu ∧ za ∨ a hodnotu 0 za 1, dostávame druhú axiómu. Tento princíp sa označuje ako princíp duality (angl. „*self-dual*“) **[**5**].**

Je zrejmé, že v Booleovej algebre sa používajú binárne operácie prieseku (nazývaná aj AND), spojenia (nazývaná aj OR) a unárna operácia komplementu (tiež známa aj ako NOT). Avšak v booleovskej algebre sa používajú aj iné operácie, ktoré je možné vyjadriť kombináciou troch základných operácií. Tieto operácie sú zobrazenia, ktoré transformujú 2 vstupné premenné $x, y \in \{0,1\}$ do tretej výstupnej premennej $z = f(x, y) \in \{0,1\}$, pričom existuje celkovo 16 rôznych možných zobrazení, pretože sú 2 možnosti hodnôt výstupnej premennej pre každú kombináciu vstupných premenných $f(0,0), f(0,1), f(1,0)$ a $f(1,1)$ **[**6**].** Tieto zobrazenia je možné vidieť na obrázku 1.1, kde sú uvedené názvy jednotlivých operácií, ich vyjadrenie vo forme hodnôt výstupnej premennej pre každú kombináciu vstupných premenných, ktorá sa nazýva pravdivostný vektor (bude podrobnejšie vysvetlený neskôr) a ich najčastejšie označovanie používané v odborných literatúrach. Keďže niektoré operácie sú využívané aj v softvérovej knižnici, budú teraz tieto booleovské operácie bližšie predstavené.

Operácia **NOT**, teda operácia negácie alebo komplementu je unárna operácia, ktorej výstupom je komplement k operandu, čo v booleovskej algebre znamená, že negáciou hodnoty 0 je 1 a negáciou hodnoty 1 je 0. Negáciu je však možné použiť aj na iné operácie (NAND, NOR, ...).

Operácia **AND** v booleovskej algebre je binárna operácia prieseku, ktorej výstupom je minimum z dvoch hodnôt operandov. To znamená, že táto operácia má na výstupe hodnotu 1 vtedy, keď obidva operandy majú hodnotu 1, inak je na výstupe hodnota 0.

Operácia **OR** v booleovskej algebre je binárna operácia spojenia, ktorej výstupom

je maximum z dvoch hodnôt operandov. Je zrejmé, že táto operácia má na výstupe hodnotu 0 práve vtedy, keď obidva operandy majú hodnotu 0, inak nadobúda operácia hodnotu 1.

Pravdivostný vektor	Označenie	Názov
$[0000]^T$	0	konštanta 0
$[0001]^T$	$xy,\ x \wedge y,\ x \,\&\, y$	AND, priesek, konjukcia
$[0010]^T$	$x \wedge \bar{y},\ x \not\supset y,\ [x > y],\ x \dot{-} y$	negovaná implikácia
$[0011]^T$	x	ľavá projekcia
$[0100]^T$	$\bar{x} \wedge y,\ x \not\subset y,\ [x < y],\ y \dot{-} x$	negovaná obrátená implikácia
$[0101]^T$	y	pravá projekcia
$[0110]^T$	$x \oplus y,\ x \not\equiv y,\ x \hat{} y$	XOR, neekvivalentnosť
$[0111]^T$	$x \vee y,\ x \mid y$	OR, spojenie, disjunkcia
$[1000]^T$	$\bar{x} \wedge \bar{y},\ \overline{x \vee y},\ x \,\overline{\vee}\, y,\ x \downarrow y$	NOR, negácia disjunkcie
$[1001]^T$	$x \equiv y,\ x \leftrightarrow y,\ x \Leftrightarrow y$	ekvivalencia
$[1010]^T$	$\bar{y},\ \neg y,\ !y,\ \sim y$	pravá negácia, pravý komplement
$[1011]^T$	$x \vee \bar{y},\ x \subset y,\ x \Leftarrow y,\ [x \geq y],\ x^y$	obrátená implikácia
$[1100]^T$	$\bar{x},\ \neg x,\ !x,\ \sim x$	ľavá negácia, ľavý komplement
$[1101]^T$	$\bar{x} \vee y,\ x \supset y,\ x \Rightarrow y,\ [x \leq y],\ y^x$	implikácia
$[1110]^T$	$\bar{x} \vee \bar{y},\ \overline{x \wedge y},\ x \,\overline{\wedge}\, y,\ x \mid y$	NAND, negácia konjukcie
$[1111]^T$	1	konštanta 1

Obr. 1.1 Všetky zobrazenia pre 2 vstupné premenné x,y

Operácia **NOR**, tiež známa aj ako Pierceova funkcia je v booleovskej algebre binárna operácia, ktorá je negáciou operácie **OR**. To znamená, že táto operácia má výstupnú hodnotu 1 práve vtedy, keď majú obidva operandy hodnotu 0. Inak má na výstupe hodnotu 1.

Operácia **NAND**, tiež známa aj ako Shefferova funkcia, je v booleovskej algebre binárna operácia, ktorá predstavuje negáciu operácie AND. Preto je zrejmé, že táto operácia dáva na výstupe hodnotu 0 len vtedy, keď hodnota obidvoch operandov je rovná 1, inak je na výstupe hodnota 1.

Operácia **ekvivalencie** je binárna operácia definovaná v booleovskej algebre, ktorá

porovnáva hodnotu dvoch operandov a pokiaľ sú ich hodnoty rovnaké, tak na výstupe má hodnotu 1, inak sa na výstup dostane hodnota 0.

Ako posledná bude predstavená operácia **XOR**, ktorá je binárnou operáciou definovanou v booleovskej algebre a predstavuje opak k operácii ekvivalencie. To znamená, že výstupom z tejto operácie je hodnota 1 práve vtedy, keď sa hodnoty jej operandov líšia, inak je výstupom hodnota 0.

Booleovská algebra nájde svoje využitie vo viacerých vedných odvetviach, ako napríklad pri logických obvodoch v elektrotechnike a počítačovom inžinierstve, pri práci s výrokmi vo výrokovej logike, pri procedúrach ako hlasovacie hry v teórii hier či pri hodnotení stavu systému a jeho komponentov v teórii spoľahlivosti **[**7**]**.

1.2 Viachodnotová logika

V predošlej časti bola priblížená booleovská algebra, ktorá pracuje s 2 hodnotami a sú na nej definované základné a odvodené operácie. Avšak nie vždy je táto algebra postačujúca na riešenie problémov, ktoré vznikajú či už v elektrotechnike pri viachodnotových logických elektrických obvodoch, v teórií spoľahlivosti, kde je potrebné rozlišovať viaceré úrovne práceschopnosti obvodu, resp. súčiastky obvodu a všade tam, kde je potrebné rozlišovať a pracovať s viac ako dvomi hodnotami. Preto vznikla viachodnotová logika, ktorá bude v tejto časti predstavená, pričom sa využívajú hlavne zdroje **[**7**]**, **[**8**]**, **[**9**]**, **[**10**]**.

Viachodnotová logika je teda zovšeobecnenie booleovskej logiky v tom zmysle, že úroveň pravdivosti nie je len dvojhodnotová, ale m-hodnotová, resp. nekonečná. Prvé náznaky, že tento typ logiky je potrebný uviedol už Aristoteles, ktorý okrem toho, že je pokladaný za „*otca logiky*“ tiež prišiel s paradoxom pravdivosti budúcich tvrdení. Vo svojom diele On Interpretation uviedol, že veta „*Morská bitka sa zajtra nebude konať,*“ sa nedá označiť jednoznačne za pravdivú, resp. nepravdivú, pokým nenastane zajtrajšok a bitka neprebehne, resp. prebehne [11]. Avšak Aristoteles nevytvoril systém pre viachodnotovú logiku, v ktorom by tento problém vyriešil. Myšlienka vytvorenia systému pre viachodnotovú logiku sa opäť objavila v dvadsiatom storočí. V roku 1920 prišiel poľský logik a filozof Jan Łukasiewicz so systémom trojhodnotovej logiky, kde používa tretiu hodnotu „*nevyjadriteľný*“, aby sa dokázal vyriešiť paradox pravdivosti budúcich tvrdení. Medzitým, americký matematik Emil L. Post v roku 1921 uvádza viachodnotovú

logiku, to znamená logiku pracujúcu s $m \geq 2$, kde m sú pravdivostné hodnoty. Po predstavení týchto logík nasledoval vývoj ďalších viachodnotových logík a s nimi spojených algebier, ktoré dokážu pracovať s tromi, štyrmi, či s m logickými hodnotami, prípadne existujú aj logiky s neobmedzeným počtom logických hodnôt.

Avšak skôr než budú definované algebry viachodnotových logických systémov, je vhodné si najskôr predstaviť operácie, ktoré sú používané v týchto algebrách. Taktiež je vhodné uviesť, že všetky operácie budú definované na množine $\mathbb{M} = \{0,1,\dots,m-1\}$, znak n predstavuje počet premenných a pre binárne operácie, teda operácie s dvoma operandmi, bude ďalej používané označenie dvojmiestne operácie, keďže označenie binárna viachodnotová operácia by bolo nevhodné či mätúce.

Operácia komplementu je unárna operácia, ktorá je definovaná nasledovne :

$$\bar{x} = (m-1) - x, \qquad x \in \mathbb{M}.$$

V trojhodnotovej logike je táto operácia definovaná ako $\bar{x} = 2 - x$ a pre hodnoty $0,1$ a 2 sú výstupom z tejto operácie hodnoty $2,1$ a 0.

Dopredná cyklická operácia (angl. „*clockwise cycle operation*“) alebo tiež operácia r-tého cyklického komplementu je unárna operácia, ktorej definícia je nasledovná:

$$\hat{x}^r = x + r \,(\mathrm{mod}\, m), \;\; x \in \mathbb{M},$$

kde $r \in \mathbb{M}$ vyjadruje veľkosť posunu. Z tejto operácie je možné vyvodiť, že platia nasledovné vzťahy: $\hat{x}^0 = x \,(mod\, m), \hat{x}^0 = x + m(mod\, m) = x \,(mod\, m)$. Pre ukážku bude zvolená trojhodnotová logika a $r = 1$, čo znamená, že pre hodnoty $0,1$ a 2 budú výstupom hodnoty $1,2$ a 0.

Oknová operácia (angl. „*window literal operation*“) je unárna operácia, v ktorej sú definované hraničné hodnoty $a, b \in \mathbb{M}, a < b$ a platí

$${}_a x^b = \begin{cases} m-1 & \text{ak } a \leq x \leq b \\ 0 & \text{inak} \end{cases}, x \in \mathbb{M}.$$

V trojhodnotovej logike dostávame napríklad pre hodnoty $a = 1,\ b = 2$ a postupne pre vstupné hodnoty $0,1$ a 2 výstupné hodnoty $0,2$ a 2.

Operácia **MAX** je dvojmiestna operácia, ktorá je definovaná takto:

$$\mathrm{MAX}(x_1, x_2) = x_1 \vee x_2 = \begin{cases} x_1 & \text{ak } x_1 \geq x_2 \\ x_2 & \text{inak} \end{cases}, \qquad x_1, x_2 \in \mathbb{M}.$$

Táto operácia je pre $m = 2$ operáciou OR v booleovskej algebre. Túto operáciu môžeme definovať aj ako n-árnu a to nasledovne: $\mathrm{MAX}(x_1, x_2, \dots, x_n) = x_1 \vee x_2 \vee \dots \vee x_n$, $x_1, x_2, \dots, x_n \in \mathbb{M}$

Operácia **MIN** je dvojmiestna operácia, ktorá je definovaná takto:

$$\mathrm{MIN}(x_1, x_2) = x_1 \wedge x_2 = \begin{cases} x_1 & \text{ak } x_1 \leq x_2 \\ x_2 & \text{inak} \end{cases}, \qquad x_1, x_2 \in \mathbb{M}.$$

Táto operácia je pre $m = 2$ operáciou AND v booleovskej algebre. Túto operáciu môžeme definovať aj pre n premenných a to nasledovne: $\mathrm{MIN}(x_1, x_2, \dots, x_n) = x_1 \wedge x_2 \wedge \dots \wedge x_n$, $x_1, x_2, \dots, x_n \in \mathbb{M}$.

Ako posledné budú zadefinované dvojmiestne, resp. n-árne operácie **násobenia** a **sčítania modulo m** pre $x_1, x_2, \dots, x_n \in \mathbb{M}$ a to nasledovne:

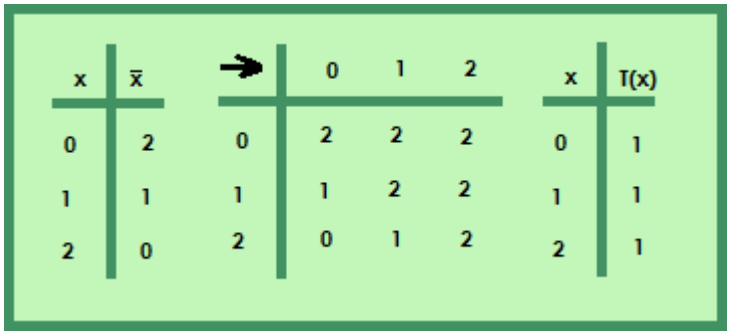
$$\mathrm{MOD-PROD}(x_1, x_2, \dots, x_n) = x_1 * x_2 * \dots * x_n \mod m,$$

$$\mathrm{MOD-SUM}(x_1, x_2, \dots, x_n) = x_1 + x_2 + \dots + x_n \mod m.$$

Po priblížení funkčnosti základných operácií vo viachodnotovej logike budú predstavené niektoré používané viachodnotové logiky a algebry. Väčšina z týchto logík je výsledkom logikov, ktorí chceli definovať logický systém poskytujúci viac vyjadrujúce a hodnotové nástroje ako tie, ktoré sú definované v tradičnom dvojhodnotovom logickom systéme.

Jeden z prvých publikovaných trojhodnotových logických systémov bol **Lukasiewiczov trojhodnotový logický systém**, ktorý bol predstavený v roku 1920. Tento systém vznikol hlavne z dôvodu vyriešenia paradoxu morskej bitky, pričom tento systém má okrem hodnôt pravda, nepravda aj hodnotu nevyjadriteľný (angl. „*indeterminate*“). Tento systém bol postavený na množine $\mathbb{M} = \{0 = \text{false}, 1 = \text{indeterminate}, 2 = \text{true}\}$, pričom v ňom boli definované len 2 základné operácie a to operácia komplementu a dvojmiestna operácia implikácie, avšak tento systém nevytvára funkcionálne kompletnú algebru, to znamená, že nie je možné len s pomocou týchto 2 operácií definovať n-parametrické zobrazenie. Ak však pridáme unárny operátor, známy ako „*T-funkcia*“, potom je možné sformulovať funkcionálne kompletnú algebru **[**9**]**, pričom definíciu

operátorov $\rightarrow, \mathrm{T}()$ spolu s operáciou komplementu je možné vidieť na obr. 1.2. Napriek tomu, že tento systém je postavený pre 3 hodnoty, je možné tento systém zovšeobecniť aj na viachodnotové systémy ba dokonca aj nekonečné viachodnotové systémy **[**9**]**.



Obr. 1.2 Základné operácie v Lukasiewiczovom logickom systéme

Aj keď viachodnotová logika Lukasiewicza bola publikovaná o rok skôr ako práca Emila Posta z roku 1921, **Postova algebra** je všeobecne citovaná ako prvý príklad viachodnotovej algebry, keďže jeho definícia viachodnotových logických algebier je funkcionálne kompletná. Postovu algebru budeme definovať nad lineárne usporiadanou množinou $\mathbb{M} = \{x_0, x_1, \dots, x_{m-1}\}$, $x_i < x_j$ ak $i < j$ s dvoma základnými operáciami, konkrétne operáciou prvého cyklického komplementu $x + 1\ (mod\ m)$ a dvojmiestnou operáciou MAX, t.j. $x_1 \vee x_2$. Je zrejmé, že pre $m = 2$ zodpovedá Postova logika klasickej logike so základnými operáciami negácie a disjunkcie, ktorá je taktiež funkcionálne kompletná [**8**].

Viachodnotová algebra predstavená pánmi Allenom a Givoneom v roku 1968 bola vyvinutá hlavne za účelom vybudovania matematických základov pre podporu viachodnotových logických obvodov. **Allenova a Givonova algebra** je definovaná ako usporiadaná šestica $(\mathbb{M}, \vee, \wedge, {}^{a}x^{b}, \mathbf{0}, \mathbf{1})$, kde $\mathbb{M}$ je konečná množina obsahujúca p elementov $\{0, 1, \dots, p-1\}$, dvojmiestna operácia $\vee$, resp. $\wedge$ je operácia MAX, resp. MIN, unárna operácia ${}^{a}x^{b}$ je oknová operácia, $\mathbf{0} = 0$ a $\mathbf{1} = p - 1$.

Existuje ešte množstvo ďalších používaných viachodnotových systémov, ako napríklad **Bochvarova logika**, čo je podobne ako Lukasiewiczova logika trojhodnotová logika, avšak s rozdielom schematickej implementácie tretej hodnoty, t.j. tu je význam tretej hodnoty definovaný ako nerozhodný (angl. „*undecidable*“) čo spôsobuje zmenu vo vyhodnocovaní základných operácií; **Kleenova Logika**, čo je opäť trojhodnotová logika, avšak tu je význam tretej hodnoty jednoducho neznámy (angl. „*unknown*“); **Bernsteinova**

algebra, čo je viachodnotová algebra a obsahuje základné operácie sčítania a násobenia modulo m, a mnoho ďalších.

Viachodnotová logika sa využíva vo viacerých odvetviach, ako napríklad v lingvistike pri modelovaní prirodzených jazykových fenoménov, v logike pri riešení problémov s viacerými možnými hodnotami pravdivosti, vo filozofii pri riešení starých paradoxov ako *Sorites* alebo *falakros* **[**11**]**, v elektrotechnike pri práci s obvodmi, ktoré majú viac napäťových úrovní **[**8**]** a v data miningu pri práci s atribútmi s viacerými hodnotami **[**9**]**.

1.3 Logická funkcia

Po zadefinovaní logických algebier je vhodné si predstaviť aj zobrazenia, ktoré sa dajú využiť pri popisovaní určitých javov, pri vyjadrovaní vzťahov medzi jednotlivými vstupnými parametrami javu alebo pri vyhodnocovaní výsledku javu na základe hodnôt jeho vstupných veličín. Tieto zobrazenia sa volajú logické funkcie a v tejto časti budú tieto logické funkcie zadefinované, pričom budú využívané hlavne zdroje **[**7**]**, **[**12**]**, **[**13**]**.

Prvý pojem, ktorý je potrebné zadefinovať je viachodnotová premenná. **Viachodnotová premenná** x je premenná, ktorá môže nadobúdať hodnoty z množiny $\mathbb{M} = \{0,1, \dots, m-1\}$. Pokiaľ je množina $\mathbb{M}$ dvojprvková, nazývame premennú x **booleovskou premennou**.

***M*-hodnotová logická funkcia** $f_m(x_1, x_2, \dots, x_n)$ n viachodnotových premenných $x_1, x_2, \dots, x_n$ je zobrazenie $f_m: \mathbb{M}^n \to \mathbb{M}$, kde množina $\mathbb{M} = \{0,1, \dots, m-1\}$ a n je prirodzené číslo. Pokiaľ je $m = 2$, potom sa 2-hodnotová logická funkcia nazýva booleovskou funkciou. Trojhodnotová logická funkcia je potom zobrazenie $f_3: \{0,1,2\}^n \to \{0,1,2\}$ a štvorhodnotová logická funkcia je zobrazenie $f_4: \{0,1,2,3\}^n \to \{0,1,2,3\}$. Väčšinou sa volí m také, aby bolo prvočíslo. Je možné na základe definície vyvodiť, že pre $n, m \in \mathbb{N}$ existuje m^{m^n} rôznych možných funkcií. Napríklad v trojhodnotovej logike existuje pre $n = 2$ $3^{3^2} = 729$ možných trojhodnotových logických funkcií 2 trojhodnotových premenných.

Logické funkcie možno reprezentovať viacerými spôsobmi. V tejto práci budú predstavené spôsoby reprezentácie formou pravdivostnej tabuľky, pravdivostného vektoru a logického výrazu, ktoré je možné vidieť na obrázku 1.3 pre dvojmiestnu operáciu MAX a pre binárnu operáciu OR.

Najjednoduchší spôsob reprezentácie logickej funkcie je formou **pravdivostnej tabuľky**. V každom riadku tejto tabuľky sú uvedené hodnoty výstupu z logickej funkcie a kombinácia vstupných premenných, ktorej spracovaním vznikne daný výstup. Pravdivostná tabuľka obsahuje všetky možné kombinácie vstupných premenných a k nim pridružených výstupov, čo znamená, že pre viachodnotové logické funkcie s veľkým m, prípadne veľkým počtom vstupných premenných nie sú veľmi používané.

Logická funkcia	Pravdivostná tabuľka	Pravdivostný stĺpcový vektor	Logický výraz
$f(x_1, x_2)$ $m = 3$	x_1 x_2 $f(x_1, x_2)$ 0 0 0 0 1 1 0 2 2 1 0 1 1 1 1 1 2 2 2 0 2 2 1 2 2 2 2	$\begin{bmatrix}0\\1\\2\\1\\1\\2\\2\\2\\2\end{bmatrix}$	$x_1 \vee x_2$
$f(x_1, x_2)$ $m = 2$	x_1 x_2 $f(x_1, x_2)$ 0 0 0 0 1 1 1 0 1 1 1 1	$\begin{bmatrix}0\\1\\1\\1\end{bmatrix}$	$x_1 \vee x_2$

Obr. 1.3 Rôzne reprezentácie logických funkcií

Pravdivostný vektor m-hodnotovej logickej funkcie f_m n viachodnotových premenných $x_1, x_2, \dots, x_n$ je definovaný nasledovne:

$$\mathbb{f}_m = [f_m(0), f_m(1), \dots, f_m(m^n - 1)]^T.$$

Jednotlivé hodnoty $i_1, i_2, \dots, i_n \in \{0,1, \dots, m^n - 1\}$ viachodnotových premenných $x_1, x_2, \dots, x_n$ predstavujú m-árnu reprezentáciu indexu i funkčnej hodnoty $f_m(i)$.

Logický výraz je forma zápisu logickej funkcie, ktorá pochádza z výrokovej

logiky. Táto forma zápisu umožňuje vyjadriť logickú funkciu algebraickou formou. Logický výraz budeme definovať nasledovne:

Pre zadaný konečný zoznam viachodnotových premenných $x_1, x_2, \dots, x_n$ je **logický výraz** $\phi(x_1, x_2, \dots, x_n)$ **pre viachodnotové premenné** $x_1, x_2, \dots, x_n$ taký výraz, že platí :

I. konštanty $0,1, \dots, m-1$ a viachodnotové premenné $x_1, x_2, \dots, x_n$ sú logické výrazy pre viachodnotové premenné $x_1, x_2, \dots, x_n$;

II. ak ϕ a ψ sú logické výrazy pre viachodnotové premenné $x_1, x_2, \dots, x_n$, potom pri použití operátov zadefinovanej viachodnotovej algebry, ktoré majú ako operandy ϕ a ψ, dostávame opäť logický výraz pre viachodnotové premenné $x_1, x_2, \dots, x_n$;

III. každý logický výraz pre viachodnotové premenné je vytvorený konečným počtom použití pravidiel I a II.

Pri booleovskej algebre sa používa pomenovanie booleovský výraz pre booleovské premenné $x_1, x_2, \dots, x_n$. Ďalej bude používané označenie logický výraz pre termín logický výraz pre viachodnotové premenné $x_1, x_2, \dots, x_n$. Pre ukážku budú teraz uvedené ďalšie príklady logických výrazov: v booleovskej algebre $\phi(x) = \bar{x}$, $\phi(x_1, x_2) = \overline{(x_1 \wedge x_2)}$, $\phi(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (x_1 \wedge x_3)$, v Allenovej a Givonovej algebre pre $m = 5$ $\psi(x_1, x_2, x_3) = {}^1x_1{}^3 \vee (x_2 \wedge x_3)$. Je vhodné tiež podotknúť, že jedna m-hodnotová logická funkcia $f_m(x_1, x_2, \dots, x_n)$ môže byť vyjadrená rôznymi logickými výrazmi, avšak každý výraz reprezentuje unikátnu m-hodnotovú logickú funkciu $f_m(x_1, x_2, \dots, x_n)$.

Využitie logických funkcií úzko súvisí s využitím viachodnotovej logiky, resp. booleovskej algebry, na ktorej je logická funkcia definovaná. Používa sa pri popise správania sa logického obvodu v elektrotechnike a počítačovom inžinierstve, pri vyjadrovaní výroku a vyhodnocovaní jeho pravdivosti vo výrokovej logike **[**8**]**, pri popisovaní správania sa systému pre potreby skúmania spoľahlivosti v teórií spoľahlivosti a v mnohých ďalších odvetviach **[**7**]**.

1.4 Normálne formy

Logickú funkciu je možné vyjadriť viacerými spôsobmi, pričom jedným z nich je formou logického výrazu. Existujú špeciálne logické výrazy, ktoré majú určitú definovanú formu. Tieto výrazy sú známe ako normálne formy a budú predstavené v tejto časti, pričom

budú využívané hlavne zdroje **[**7**], [**12**], [**13**], [**14**]**.

Najskôr budú predstavené normálne formy v booleovskej algebre, pričom každá booleovská funkcia sa dá vyjadriť jednou z týchto foriem. Pre ukážku sa v každej normálnej forme vyjadrí booleovská funkcia $q = [0,0,1,0,1,1,1,1]^{\mathrm{T}}$.

Jednou zo základných normálnych foriem je normálna disjunktívna forma (angl. „*Sum-of-products*“), ktorá pozostáva zo súčinových členov, ktoré sú definované nasledovne:

Súčinovým členom n booleovských premenných $x_1, x_2, \dots, x_n$ je logický výraz

$$S = \bigwedge_{i \in \mathbb{A}} x_i \wedge \bigwedge_{j \in \mathbb{B}} \bar{x}_j \,, \mathbb{A} \cap \mathbb{B} = \emptyset,$$

kde $\mathbb{A}, \mathbb{B}$ sú navzájom disjunktné množiny, pre ktoré platí $\mathbb{A} \cup \mathbb{B} \subseteq \{1,2, \dots, n\}$. Príkladom súčinových členov sú výrazy $1, x_1, \bar{x}_2, x_1 \wedge \bar{x}_2, \bar{x}_1 \wedge \bar{x}_2 \wedge x_3$.

Potom logický výraz $\phi(x_1, x_2, \dots, x_n)$, ktorý má tvar:

$$\phi(x_1, x_2, \dots, x_n) = \bigvee_{k=1}^{l} S_k = \bigvee_{k=1}^{l} \left(\bigwedge_{i \in \mathbb{A}_k} x_i \wedge \bigwedge_{j \in \mathbb{B}_k} \bar{x}_j \right),$$

je **normálna disjunktívna forma**, skrátene **NDF**, kde S_k je k-ty súčinový člen pre $k = 1,2, \dots, l$. Príkladom logických výrazov v NDF sú výrazy $\phi(x_1, x_2, x_3) = x_1 \vee x_2 \wedge \bar{x}_3, \phi(x_1, x_2) = x_1 \wedge x_2 \vee x_1 \wedge \bar{x}_2, \phi(x_1, x_2, x_3) = x_1 \wedge x_2 \vee x_1 \wedge \bar{x}_2 \vee x_1 \wedge x_2 \wedge x_3, \phi(x) = 1$.

Ďalšou formou je normálna konjunktívna forma (angl. „*Product-of-sums*“), ktorá pozostáva zo súčtových členov, ktoré sú definované takto:

Súčtovým členom n booleovských premenných $x_1, x_2, \dots, x_n$ je logický výraz

$$T = \bigvee_{i \in \mathbb{A}} x_i \vee \bigvee_{j \in \mathbb{B}} \bar{x}_j \,, \mathbb{A} \cap \mathbb{B} = \emptyset,$$

kde $\mathbb{A}, \mathbb{B}$ sú navzájom disjunktné množiny, pre ktoré platí $\mathbb{A} \cup \mathbb{B} \subseteq \{1,2, \dots, n\}$. Príkladom súčtových členov sú nasledovné výrazy: $0, x_1, \bar{x}_2, x_1 \vee x_2, x_1 \vee \bar{x}_2 \vee \bar{x}_3$.

Potom logický výraz $\phi(x_1, x_2, \dots, x_n)$, ktorý má tvar:

$$\phi(x_1, x_2, \dots, x_n) = \bigwedge_{k=1}^{l} T_k = \bigwedge_{k=1}^{l} \left(\bigvee_{i \in \mathbb{A}_k} x_i \vee \bigvee_{j \in \mathbb{B}_k} \bar{x}_j \right),$$

je **normálna konjunktívna forma**, skrátene **NKF**, kde S_k je k-ty súčtový člen pre $k = 1,2, \dots, l$. Príkladom NKF môžu byť logické výrazy $\phi(x_1, x_2, x_3) = (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_3), \phi(x_1, x_2) = (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2), \phi(x_1, x_2, x_3) = (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2 \vee x_3), \phi(x) = 0.$

Posledná normálna forma v booleovskej algebre, ktorá bude v tejto práci vysvetlená je **Reed-Mullerova expanzia**, ktorá sa podobá NDF ale s tým rozdielom, že operácia OR je tu nahradená operáciou XOR. To znamená, že logický výraz $\phi(x_1, x_2, \dots, x_n)$, ktorý má tvar:

$$\phi(x_1, x_2, \dots, x_n) = \oplus_{k=1}^{2^n} (c_k \wedge S_k), c_k \in \{0,1\}$$

je nazývaný **Reed-Mullerovou expanziou**. V Reed-Mullerovej expanzii je maximálne 2^n rôznych možností kombinácií vstupných hodnôt, pretože tu sa neberie do úvahy negácia, teda pre množinu $\mathbb{B}$ súčinového člena platí $\mathbb{B} = \emptyset$ (hovoríme o takzvanej kladnej Reed-Mullerovej expanzii, angl. „**positive-polarity Reed–Muller expansion**“). Príkladom Reed-Mullerovej expanzie sú logické výrazy $\phi(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_1 \wedge x_2 \oplus x_1 \wedge x_3 \oplus x_1 \wedge x_2 \wedge x_3$, $\phi(x_1, x_2) = x_1 \oplus x_1 \wedge x_2$.

Po predstavení normálnych foriem v booleovskej algebre nasledujú normálne formy vo viachodnotovej logike. Jednou z najznámejších je NDF, pričom operácie z booleovskej algebry sú nahradené operáciami z viachodnotovej logiky. Operácia OR je nahradená operáciou MAX, operácia AND je nahradená operáciou MIN a operácia NOT je nahradená literálnou operáciou, ktorá je definovaná nasledovne:

$$x^{\mathbb{S}} = \begin{cases} m-1 & x \in \mathbb{S} \\ 0 & \text{inak} \end{cases},$$

kde $\mathbb{S} \subseteq \{0,1, \dots, m-1\}$. Potom logický výraz $\phi(x_1, x_2, \dots, x_n)$, ktorý má tvar:

$$\phi(x_1, x_2, \dots, x_n) = \bigvee_{k=0}^{m-1} k \wedge S_k(x_1, x_2, \dots, x_n),$$

sa nazýva **NDF**, kde $S_k(x_1, x_2, \dots, x_n)$ je rozšírenie literálnej operácie viachodnotovej premennej na litelárnu operáciu logickej funkcie, ktorá je definovaná nasledovne:

$$S_k(x_1, x_2, \dots, x_n) = \begin{cases} m-1 & \text{ak } \phi(x_1, x_2, \dots, x_n) = k \\ 0 & \text{inak} \end{cases}, k \in \{0,1, \dots, m-1\}.$$

Príkladom je nasledovný výraz $\phi(x_1, x_2) = 1 \wedge x_1^1 \wedge x_2^{\{0,2\}} \vee 2 \wedge x_1^1$, ktorý reprezentuje ternárnu logickú funkciu $[0,1,0,2,2,2,0,1,0]^{\mathrm{T}}$.

Ďalšou známou formou je **polynomiálna forma**, tiež známa aj ako **Reed-Mullerova expanzia** logickej funkcie. Pri viachodnotovej logike sa na rozdiel od booleovskej algebry používajú celočíselné operácie súčtu, násobenia, umocnenia a zvyšku po celočíselnom delení. To znamená, že polynomiálna forma viachodnotovej funkcie f_m n viachodnotových premenných $x_1, x_2, \dots, x_n$ je definovaná nasledovne:

$$\phi(x_1, x_2, \dots, x_n) = f^{(0)} + \sum_{s=1}^{m^n-1} f^{(s)} * x_1^{s_1} * x_2^{s_2} * \dots * x_n^{s_n} =$$

$$f^{(0)} + f^{(1)} * x_n + f^{(2)} * x_n^2 + \cdots + f^{(m^n-1)} * x_1^{m-1} * x_2^{m-1} * \dots * x_n^{m-1} \pmod m,$$

kde $f^{(s)} \in \{0,1, \dots, m-1\}$ sú koeficienty polynomiálnej formy a exponenty $s_1, s_2, \dots, s_n$ predstavujú polynomiálny rozvoj indexu s. Príkladom polynomiálnej formy pre $m = 3$ je výraz $\phi(x_1, x_2) = x_2 + 2 * x_1 + x_1 * x_2 + 2 * x_1^2 + x_1^2 * x_2 \bmod 3$, ktorý reprezentuje ternárnu logickú funkciu $[0,1,2,1,1,1,0,1,2]^{\mathrm{T}}$.

Normálne formy logických funkcií sa používajú vo viacerých odvetviach, napríklad špeciálna NDF, zvaná Hornova klauzula (podmienka) sa používa vo výrokovej logike a taktiež v logickom programovaní **[**15**]**. V oblasti logických obvodov nájde nesporné využitie NDF a NKF, napríklad pri získavaní normálnej formy z Karnaughovej mapy funkcie **[**7**]**, taktiež NDF nájde nesporné využitie v teórii spoľahlivosti. Reed-Mullerova expanzia nájde svoje využitie v oblasti kódovania a v elektrotechnike **[**8**]**.

1.5 Logické derivácie

Logické funkcie slúžia na popis správania sa systému, ktorý dostáva určité vstupy a na základe nich dostávame určitý výstup. V určitých odvetviach, ako napríklad elektrotechnika či teória spoľahlivosti, je však potrebné vedieť určiť závislosť zmeny výstupu z funkcie na základe zmeny vstupu jedného či skupiny vstupov. Na tento účel sa používa matematický nástroj, ktorý sa volá logická derivácia. V tejto časti bude priblížený

pojem logickej derivácie, pričom sa bude využívať zdroj **[7].**

Najskôr bude priblížená booleovská derivácia (angl. „*Boolean derivative*“), ktorá je nasledovná:

Booleovská derivácia booleovskej funkcie n booleovských premenných $f_2(x_1, x_2, \dots, x_n)$ podľa booleovskej premennej $x_i, i \in \{1,2, \dots, n\}$ má tvar:

$$\frac{\partial f_2}{\partial x_i} = \underbrace{f_2(x_1, \dots, x_i, \dots, x_n)}_{\text{základná funkcia}} \oplus \underbrace{f_2(x_1, \dots, \bar{x}_i, \dots, x_n)}_{\text{funkcia s negovaným } x_i}$$
$$= f_2(x_1, \dots, 0, \dots, x_n) \oplus f_2(x_1, \dots, 1, \dots, x_n)$$
$$= f_2(x_1, \dots, 1, \dots, x_n) \oplus f_2(x_1, \dots, 0, \dots, x_n)$$

Je možné si všimnúť, že booleovská derivácia vznikne použitím operácie XOR medzi základnou funkciou a funkciou s negovanou premennou x_i a nie je závislá od hodnoty premennej x_i. Pre lepšie pochopenie bude teraz uvedený príklad pre deriváciu funkcie $f_2(x_1, x_2) = x_1 \wedge x_2$ podľa premennej x_1 :

$$\frac{\partial f_2}{\partial x_1} = (x_1 \wedge x_2) \oplus (\bar{x}_1 \wedge x_2) = (0 \wedge x_2) \oplus (1 \wedge x_2) = x_2.$$

Okrem booleovskej derivácie booleovskej premennej budú predstavené ešte dve ďalšie formy booleovskej derivácie, ktoré pracujú so zmenou viacerých premenných súčasne a postupne.

Booleovská derivácia booleovskej funkcie $f_2(x_1, x_2, \dots, x_n)$ **podľa vektoru** k booleovských premenných $\mathbf{x}_k = [x_{i_1}, x_{i_2}, \dots, x_{i_k}]$, kde $i_1, \dots, i_k \in \{1, \dots, n\}$ má nasledovný tvar:

$$\frac{\partial f_2}{\partial \mathbf{x}_k} = f_2(x_1, .., x_{i_1}, \dots, x_{i_k}, \dots, x_n) \oplus f_2(x_1, .., \bar{x}_{i_1}, \dots, \bar{x}_{i_k}, \dots, x_n).$$

Viacnásobná booleovská derivácia booleovskej funkcie $f_2(x_1, x_2, \dots, x_n)$ podľa k booleovských premenných $x_{i_1}, x_{i_2}, \dots, x_{i_k}$ má tvar:

$$\frac{\partial f_2}{\partial x_{i_1} \partial x_{i_2} \dots \partial x_{i_k}} = \underbrace{\frac{\partial}{\partial x_{i_1}}\left(\frac{\partial}{\partial x_{i_2}}\left(\dots \frac{\partial f_2}{\partial x_{i_k}}\right)\dots\right) = \frac{\partial}{\partial x_{i_k}}\left(\frac{\partial}{\partial x_{i_{k-1}}}\left(\dots \frac{\partial f_2}{\partial x_{i_1}}\right)\dots\right)}_{\text{oboma smermi}}.$$

Tieto booleovské derivácie indikujú zmenu booleovskej funkcie pri zmene booleovskej premennej, resp. premenných, avšak nehovoria nič o smere zmeny booleovskej premennej, resp. premenných a funkcie. Na zistenie smeru zmeny sa používa smerová booleovská derivácia, pričom podľa smeru je definovaná priama a opačná smerová derivácia.

Priama smerová derivácia booleovskej funkcie $f_2(x_1, x_2, \dots, x_n)$ podľa booleovskej premennej $x_i, i \in \{1,2, \dots, n\}$ určuje rovnaký smer zmeny f_2 a x_i, teda platí:

$$\frac{\partial f_2}{\partial_+ x_i} = \frac{\partial f_2(0 \to 1)}{\partial x_i(0 \to 1)} = \frac{\partial f_2(1 \to 0)}{\partial x_i(1 \to 0)} = (\bar{x}_i \oplus f_2)\frac{\partial f_2}{\partial x_i}$$
$$= \overline{f_2(x_1, \dots,0, \dots, x_n)} \wedge f_2(x_1, \dots,1, \dots, x_n).$$

Opačná smerová derivácia booleovskej funkcie $f_2(x_1, x_2, \dots, x_n)$ podľa booleovskej premennej $x_i, i \in \{1,2, \dots, n\}$ určuje opačný smer zmeny f_2 a x_i, čo znamená, že platí:

$$\frac{\partial f_2}{\partial_- x_i} = \frac{\partial f_2(0 \to 1)}{\partial x_i(1 \to 0)} = \frac{\partial f_2(1 \to 0)}{\partial x_i(0 \to 1)} = (x_i \oplus f_2)\frac{\partial f_2}{\partial x_i}$$
$$= \overline{f_2(x_1, \dots,1, \dots, x_n)} \wedge f_2(x_1, \dots,0, \dots, x_n).$$

Je vhodné poukázať na fakt, že existuje vzťah medzi booleovskou deriváciou a smerovými deriváciami. Keď sa operácia XOR vyjadrí len pomocou operácií OR,AND a NOT, výsledkom bude priama a opačná smerová booleovská derivácia spojená operáciou OR.

Podobne ako pri booleovskej derivácie booleovskej premennej, aj pri smerovej derivácii budú predstavené ešte dve ďalšie formy, ktoré pracujú so zmenou viacerých premenných súčasne a postupne.

Smerová derivácia booleovskej funkcie $f_2(x_1, x_2, \dots, x_n)$ **podľa vektoru** k booleovských premenných $\mathbf{x}_k = [x_{i_1}, x_{i_2}, \dots, x_{i_k}]$, kde $i_1, \dots, i_k \in \{1, \dots, n\}$ ktoré menia svoju hodnotu z $[y_{i_1}, y_{i_2}, \dots, y_{i_k}]$ na $[\bar{y}_{i_1}, \bar{y}_{i_2}, \dots, \bar{y}_{i_k}]$, kde $y_{i_j} \in \mathbb{B}$, $j \in \{1, \dots, k\}$ má nasledovný tvar:

$$\frac{\partial f_2}{\partial \mathbf{x}_k} = f_2(x_1, .., x_{i_1}, \dots, x_{i_k}, \dots, x_n) \oplus f_2(x_1, .., \bar{x}_{i_1}, \dots, \bar{x}_{i_k}, \dots, x_n)$$

$$= \underbrace{\overline{f_2(x_1, \dots, y_{i_1}, \dots, y_{i_k}, \dots, x_n)} \wedge f_2(x_1, \dots, \bar{y}_{i_1}, \dots, \bar{y}_{i_k}, \dots, x_n)}_{\text{pri } f_2(0\to 1)} \vee$$

$$\underbrace{\overline{f_2(x_1, \dots, \bar{y}_{i_1}, \dots, \bar{y}_{i_k}, \dots, x_n)} \wedge f_2(x_1, \dots, y_{i_1}, \dots, y_{i_k}, \dots, x_n)}_{\text{pri } f_2(1\to 0)}.$$

Viacnásobná smerová booleovská derivácia booleovskej funkcie $f_2(x_1, x_2, \dots, x_n)$ podľa k booleovských premenných $x_{i_1}, x_{i_2}, \dots, x_{i_k}$ ktoré menia svoju hodnotu z $y_{i_1}, y_{i_2}, \dots, y_{i_k}$ na $\bar{y}_{i_1}, \bar{y}_{i_2}, \dots, \bar{y}_{i_k}$, kde $y_{i_j} \in \mathbb{B},\ j \in \{1, \dots, k\}$ má tvar:

$$\frac{\partial f_2}{\partial_+ x_{i_1} \partial_+ x_{i_2} \dots \partial_+ x_{i_k}} = \frac{\partial}{\partial_+ x_{i_1}}\left(\frac{\partial}{\partial_+ x_{i_2}}\left(\dots \frac{\partial f_2}{\partial_+ x_{i_k}}\right)\dots\right) = \frac{\partial}{\partial_+ x_{i_k}}\left(\frac{\partial}{\partial_+ x_{i_{k-1}}}\left(\dots \frac{\partial f_2}{\partial_+ x_{i_1}}\right)\dots\right),$$

$$\frac{\partial f_2}{\partial_- x_{i_1} \partial_- x_{i_2} \dots \partial_- x_{i_k}} = \frac{\partial}{\partial_- x_{i_1}}\left(\frac{\partial}{\partial_- x_{i_2}}\left(\dots \frac{\partial f_2}{\partial_- x_{i_k}}\right)\dots\right) = \frac{\partial}{\partial_- x_{i_k}}\left(\frac{\partial}{\partial_- x_{i_{k-1}}}\left(\dots \frac{\partial f_2}{\partial_- x_{i_1}}\right)\dots\right).$$

Podobne ako pri booleovskej derivácii, aj pri logickej derivácií sa zisťujú dynamické vlastnosti viachodnotovej logickej funkcie. Avšak, na rozdiel od booleovskej derivácie existujú rôzne typy logických derivácií. V tejto práci bude vysvetlená smerová logická derivácia (angl. „*Direct Partial Logic Derivative*“), ktorá je definovaná nasledovným spôsobom:

Smerová logická derivácia logickej funkcie $f_m(x_1, x_2, \dots, x_n)$ podľa viachodnotovej premennej $x_i, i \in \{1,2, \dots, n\}$ má tvar:

$$\frac{\partial f_m(j \to h)}{\partial x_i(s \to r)} = \{f_m(x_1, \dots, s, \dots, x_n) \leftrightarrow j\} \wedge \{f_m(x_1, \dots, r, \dots, x_n) \leftrightarrow h\},$$

$$kde\ s, r, j, h \in \{0,1, \dots, m-1\}, s \neq r, j \neq h,$$

ktorá vyjadruje, či zmena viachodnotovej premennej z hodnoty s na hodnotu r zmení výstup z logickej funkcie j na k. Výsledkom tejto derivácie je preto buď 0 alebo $m-1$, pretože operácia $\wedge$ predstavuje operáciu MIN a operácia $\leftrightarrow$ predstavuje operáciu ekvivalencie vo viachodnotovej logike, ktorú je možné definovať nasledovným spôsobom:

$$x \leftrightarrow y = \begin{cases} m-1 & \text{ak } x = y \\ 0 & \text{inak} \end{cases}, x, y \in \mathbb{M}.$$

Podobne ako pri smerovej booleovskej derivácií aj smerovú logickú deriváciu je možné definovať pre zmenu viacerých viachodnotových premenných súčasne, resp. postupne.

Smerová logická derivácia logickej funkcie $f_m(x_1, x_2, \dots, x_n)$ podľa vektoru viachodnotových premenných $\mathbf{x}_k = [x_{i_1}, x_{i_2}, \dots, x_{i_k}]$, kde $i_1, \dots, i_k \in \{1, \dots, n\}$ ktoré menia svoju hodnotu z $\mathbf{y}_k = [y_{i_1}, y_{i_2}, \dots, y_{i_k}]$ na $\mathbf{z}_k = [z_{i_1}, z_{i_2}, \dots, z_{i_k}]$, kde $y_{i_j}, z_{i_j} \in \mathbb{M}$, $j \in \{1, \dots, k\}$, pričom sa má výstup z logickej funkcie zmeniť z j na h, kde $j, h \in \mathbb{M}$ je definovaná nasledovne:

$$\frac{\partial f_m(j \to h)}{\partial \mathbf{x}_k(\mathbf{y}_k \to \mathbf{z}_k)} = \{f(x_1, \dots, y_{i_1}, \dots, y_{i_k}, \dots, x_n) \leftrightarrow j\} \wedge \{f(x_1, \dots, z_{i_1}, \dots, z_{i_k}, \dots, x_n) \leftrightarrow h\}.$$

Viacnásobná smerová logická derivácia logickej funkcie $f_m(x_1, x_2, \dots, x_n)$ podľa k viachodnotových premenných $x_{i_1}, x_{i_2}, \dots, x_{i_k}$, kde $i_1, \dots, i_k \in \{1, \dots, n\}$, ktoré menia svoju hodnotu z $y_{i_1}, y_{i_2}, \dots, y_{i_k}$ na $z_{i_1}, z_{i_2}, \dots, z_{i_k}$, pričom $y_{i_j}, z_{i_j} \in \mathbb{M}$, $j \in \{1, \dots, k\}$ pri zmene výstupu z logickej funkcie z j na h, kde $j, h \in \mathbb{M}$ sa definuje nasledovne:

$$\frac{\partial f_m(j \to h)}{\partial x_{i_1} \partial x_{i_2} \dots \partial x_{i_k}} = \frac{\partial}{\partial x_{i_1}}\left(\frac{\partial}{\partial x_{i_2}}\left(\dots \frac{\partial f_m(j \to h)}{\partial x_{i_k}}\right)\dots\right) = \frac{\partial}{\partial x_{i_k}}\left(\frac{\partial}{\partial x_{i_{k-1}}}\left(\dots \frac{\partial f_m(j \to h)}{\partial x_{i_1}}\right)\dots\right).$$

Využitie logických, či booleovských derivácií je hlavne v oblasti elektrotechniky pri analýze vlastností logických obvodov, keďže hlavne na tieto účely boli logické derivácie vyvinuté **[**8**]**. Avšak je možné sa stretnúť s logickými deriváciami aj v iných odvetviach, ako v oblasti diskrétnych udalostných systémov, kde podporuje ich modelovanie, testovanie a syntézu **[**16**]** alebo v teórií spoľahlivosti, kde je potrebné určiť, či porucha jedného komponentu systému môže spôsobiť výpadok celého systému **[**8**]**. Booleovské derivácie sa dokonca využívajú na zistenie hraníc binárneho obrázka **[**17**]**.

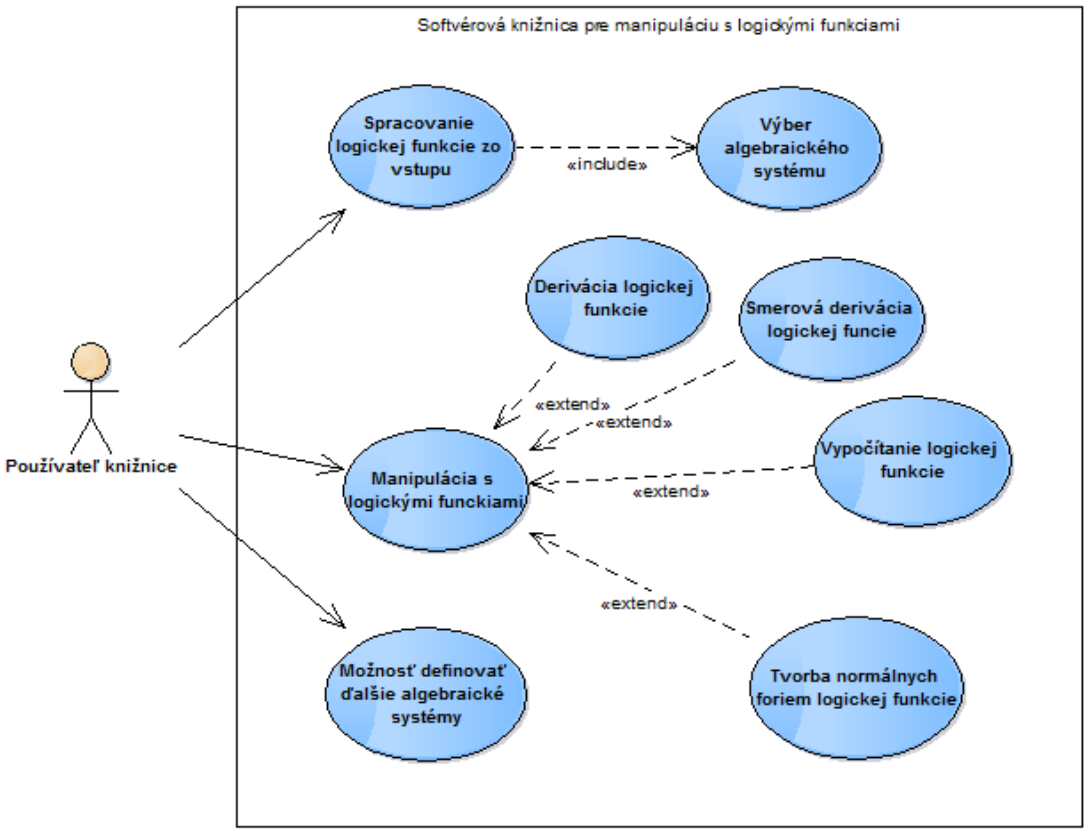
2 Knižnica pre manipuláciu s logickými funkciami

Význam logických funkcií, ich rôznych foriem a derivácií je dôležitý vo viacerých vedných oblastiach. Avšak vyjadriť normálnu formu, či určiť derivácie logických funkcií môže byť časovo a výpočtovo náročný proces. Preto je vhodné, aby sa tieto úkony dali vykonávať rýchlo a efektívne prostredníctvom softvérovej knižnice, ktorej vytvorenie je cieľom tejto práce.

V tejto časti práce bude predstavená softvérová knižnica pre manipuláciu s logickými funkciami, jej implementácia, použité dátové štruktúry či algoritmy. Knižnica je napísaná v programovacom jazyku C++, pretože tento jazyk je výkonný, časom preverený a poskytuje všetky potrebné prostriedky pre vývoj objektovo orientovanej softvérovej knižnice [18].

2.1 Funkčné požiadavky a analýza existujúcich riešení

Od vyvíjanej softvérovej knižnice sa očakáva, že bude umožňovať prácu s logickými funkciami, či už booleovskými alebo viachodnotovými.



Obr. 2.1 Funkčné požiadavky na Softvérovú knižnicu

To znamená, že je potrebné určitým spôsobom spracovať zadaný vstup v zadanom logickom algebraickom systéme do takej formy, aby bolo možné vykonávať numerické výpočty a základné booleovské derivácie a logické smerové derivácie. Taktiež je potrebné umožniť transformáciu booleovskej funkcie do NDF, resp. NKF. Pri vývoji je potrebné myslieť na možnosť dodefinovania vlastných algebraických systémov či vlastných operácií.

Na základe týchto požiadaviek sa najskôr uskutočnil prieskum už existujúcich riešení a ich analýza. Analýza bola zameraná na softvérové knižnice v C++, ktoré by mohli poskytovať nástroje na realizáciu všetkých požiadaviek, pričom ich zoznam spolu s výhodami a nevýhodami je možné vidieť na obr. 2.2. Je možné si všimnúť, že žiadna z analyzovaných softvérových knižníc nepokryla všetky požiadavky a preto ich použitie by nebolo vhodné. Na základe týchto zistení sa potvrdila potreba vytvorenia vlastnej softvérovej knižnice pre prácu s logickými funkciami.

Názov knižnice a odkaz	Výhody knižnice	Nevýhody knižnice
Lepton Mathematical Expression Parser https://simtk.org/home/lepton	Umožňuje spracovať matematické výrazy a aj ich optimalizovať, podporuje aj náročnejšie operácie ako aj definovanie vlastných funkcií.	Pracuje s množinou reálnych čísel,ale neexistuje možnosť definovania vlastnej množiny, taktiež môžu byť problémy pri vyjadrovaní NF.
C++ Mathematic Expression Library http://www.partow.net/programming/exprtk/	Umožňuje spracovať matematické výrazy a okrem základných operácií a funkcií je tu podpora viacerých dátových typov a sústav	Problém so symbolickými výpočtami a ťažko rozšíriteľné o špeciálne množiny hodnôt
An extensible math expression parser with plug-ins http://www.codeproject.com/Articles/7335/An-extensible-math-expression-parser-with-plug-in	Umožňuje pracovať so základnými matematickými operáciami a funkciami, tiež je tu možnosť definovať nové operácie a funkcie	Nezameriava sa na náročnejšie matematické funkcie, taktiež môžu byť problémy s licenciou
Armadillo http://arma.sourceforge.net/	Umožnuje pracovať s preddefinoavnými operáciami a funkciam, taktiež ponúka syntax podobnú s MatLabom	Problém pri definovaní vlastných operácií a symbolických výpočtov

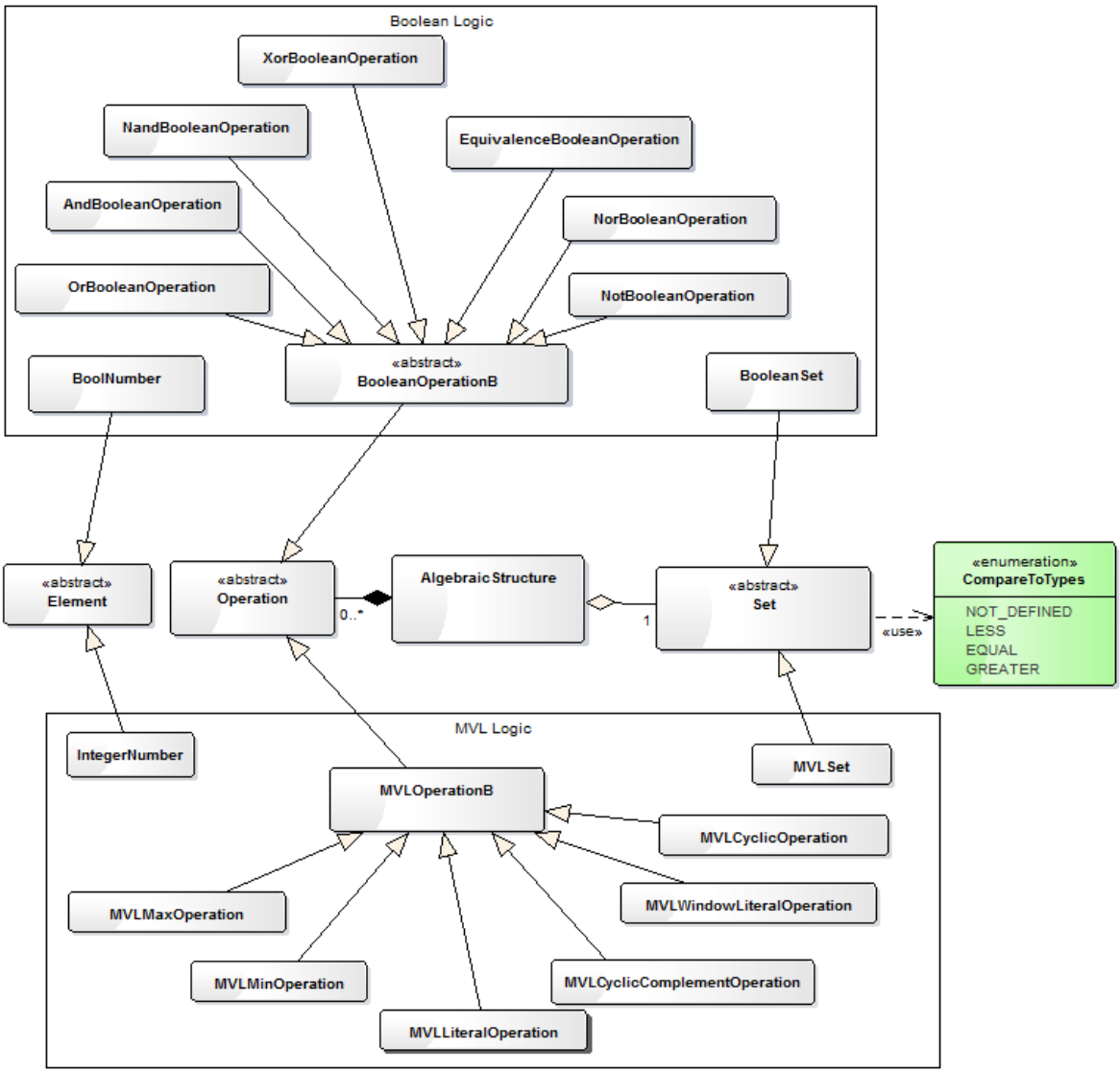
Obr. 2.2 Analýza existujúcich riešení

2.2 Návrh a implementácia softvérovej knižnice

Pri návrhu a vývoji softvérovej knižnice sa prihliadalo na splnenie všetkých funkčných požiadaviek a na minimalizáciu časovej a pamäťovej náročnosti použitých algoritmov. Softvérová knižnica je preto rozdelená na dva menné priestory a to algebra a parsing_and_processing. V mennom priestore algebra sa nachádzajú triedy pre vytvorenie algebraických systémov a prácu s nimi a v mennom priestore parsing_and_processing sa nachádzajú triedy pre spracovanie funkcie z textového vstupu, jej následné uloženie do formy stromu a prácu s ňou.

2.2.1 Menný priestor algebra

V tomto mennom priestore sa nachádzajú triedy, ktoré umožňujú vytvárať algebraické systémy, taktiež tu sú definované triedy reprezentujúce booleovskú algebru a viachodnotovú logiku. Všetky vytvorené triedy a vzájomné vzťahy medzi nimi je možné vidieť na obr. 2.3, pričom je možné si všimnúť, že všetky triedy týkajúce sa výlučne booleovskej algebry či viachodnotovej logiky sú ohraničené príslušnými hranicami.



Obr. 2.3 Vzťahy medzi triedami v mennom priestore algebra

Po predstavení vzťahov medzi triedami nasleduje popis a význam jednotlivých tried a ich dôležitých metód.

Element

Abstraktná trieda **Element** slúži ako predloha pre triedy reprezentujúce hodnoty v algebraickom systéme. Aby určitá trieda mohla byť potomkom triedy Element, je potrebné v nej implementovať všetky virtuálne metódy triedy Element, ku ktorým patrí **clone** slúžiaca na získanie hlbokej kópie elementu, **doOperation**, ktorá vykonáva všetky podporované štandardné operácie definované v enume **TypeOfOperation** s ďalším daným elementom, pričom vráti výsledok operácie a **toString**, ktorá vráti textovú reprezentáciu elementu. Pre potreby výpisu informácie o elemente štandardným výstupom je tu preťažený operátor <<.

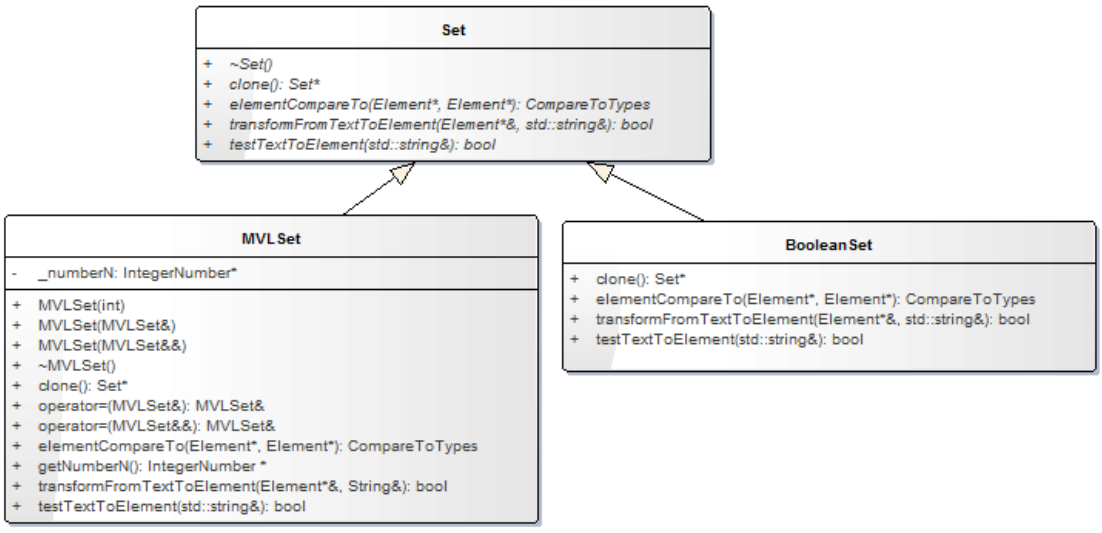
Obr. 2.4 Diagram tried pre triedy Element, BoolNumber a IntegerNumber

BoolNumber

Jednou z realizácií abstraktnej triedy Element je trieda **BoolNumber**, ktorá reprezentuje hodnotu booleovskej algebry, t.j. môže nadobúdať len hodnoty true a false, resp. 0 a 1. Pre vytvorenie objektu tejto triedy sa tu nachádzajú viaceré konštruktory, ktoré vytvoria nový objekt triedy BoolNumber zo zadaného textového vstupu, z hodnoty typu bool, či z iného objektu triedy BoolNumber, pričom sa tento objekt skopíruje použitím kopírovacieho konštruktora, alebo sa jeho hodnota presunie použitím move konštruktora. Taktiež sa tu nachádza deštruktor a implementované sú aj operátory priradenia. Samozrejmosťou sú realizácie všetkých virtuálnych metód predka pre ich použitie s booleovskými hodnotami, getter a setter na hodnotu elementu či preťaženie všetkých operátorov, ktoré sa používajú pri práci s booleovskými hodnotami.

IntegerNumber

Ďalšou realizáciou abstraktnej triedy Element je trieda **IntegerNumber**, ktorá sa používa na reprezentáciu celých čísel. Na vytvorenie celého čísla ponúka trieda IntegerNumber viacero konštruktorov, pričom sa na vytváranie nového objektu používa jeho textová reprezentácia, hodnota typu int, prípadne iný objekt triedy IntegerNumber, ktorého hodnota sa do nového objektu nakopíruje alebo presunie. Samozrejmosťou je deštruktor, operátor priradenia na skopírovanie, resp. presunutie hodnoty, implementácia všetkých virtuálnych metód pre potreby práce s celočíselnými hodnotami a preťaženie operátorov pre prácu s celými číslami.



Obr. 2.5 Diagram tried pre triedy Set, BooleanSet a MVLSet

Set

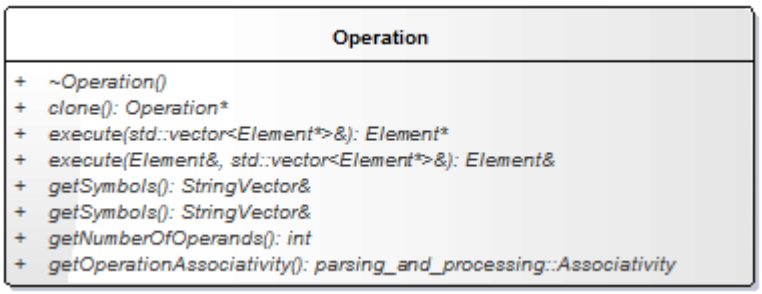
Keďže je algebra definovaná na určitej množine, je potrebné mať abstraktnú triedu, ktorá by ju reprezentovala. Preto je v mennom priestore algebra abstraktná trieda **Set**. Táto trieda obsahuje virtuálne metódy ako **clone**, ktorá podobne ako pri abstraktnej triede Element slúži na vytvorenie hlbokej kópie množiny, **elementCompareTo** na porovnanie dvoch elementov množiny, **transformFromTextToElement**, ktorej úlohou je transformovať platnú textovú reprezentáciu elementu na objekt typu Element a **testTextToElement**, ktorá potvrdzuje platnosť textovej reprezentácie elementu množiny.

BooleanSet

Jedná sa o implementáciu abstraktnej triedy Set, ktorá reprezentuje Booleovu množinu. V tejto triede sú implementované všetky virtuálne metódy predka a taktiež sa tu používajú štandardné konštruktory, deštruktor a operátory priradenia.

MVLSet

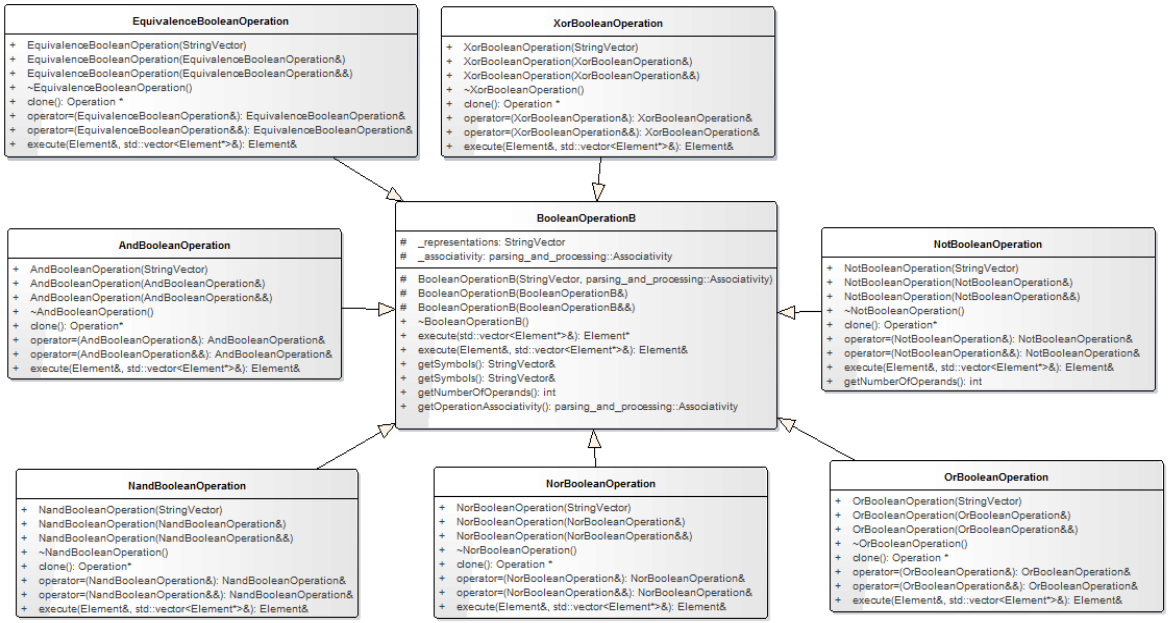
Je to implementácia abstraktnej triedy Set, ktorá predstavuje množinu hodnôt viachodnotovej logiky. Na to, aby sme vytvorili objekt tejto triedy je potrebné zadať celočíselné ohradenie množiny, prípadne iný objekt, z ktorého sa hraničná hodnota skopíruje, resp. presunie. Taktiež je tu definovaný deštruktor a operátor priradenia pre kopírovanie, resp. presun hraničnej hodnoty a sú tu implementované všetky virtuálne metódy predka.



Obr. 2.6 Diagram triedy Operation

Operation

Aby bolo možné definovať vlastné operácie v algebre, je potrebné mať abstraktnú triedu **Operation**. V tejto triede sú definované virtuálne metódy ako **clone**, ktorá sa používa na získanie hlbokej kópie operácie, **getSymbols** na získanie použitých reprezentácií pre operáciu, **getNumberOfOperands** na získanie počtu operandov, s ktorými operácia pracuje, **getOperationAssociativity** na zistenie asociatívnosti operácie a metódy **execute**, ktoré pre zadané vstupné elementy vyjadria výstup, pričom sa výstup buď uloží do už vytvoreného elementu alebo sa vytvorí a vráti nový element, ktorý predstavuje výslednú hodnotu.



Obr. 2.7 Diagram tried booleovských operácií

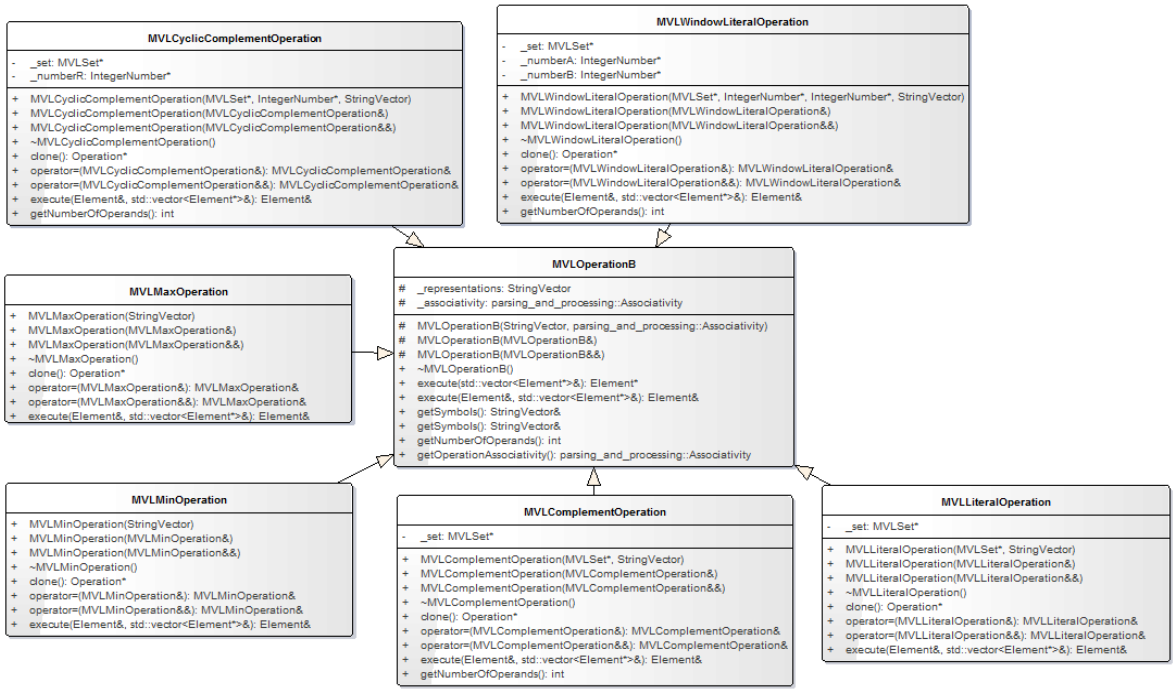
BooleanOperationB

Abstraktná trieda **BooleanOperationB** je potomkom triedy Operation a predkom každej triedy, ktorá reprezentuje booleovskú operáciu. Obsahuje textové reprezentácie a asociativitu operácie, pričom sú tieto atribúty prístupné v potomkoch. Taktiež sú v potomkoch prístupné konštruktory, ktoré buď inicializujú atribúty zo zadaných hodnôt alebo hodnoty nakopírujú, resp. presunú z už vytvorenej booleovskej operácie. Z predka sú tu implementované metódy **getSymbols**, **getNumberOfOperands**, v ktorej sa vracia štandardne číslo 2, keďže väčšina booleovských operácií má 2 operandy, **getOperationAssociativity** a **execute**, pričom je potrebné v potomkoch tejto triedy

prekryť metódu **execute**, ktorá vykoná svoju činnosť a výsledok uloží už do vytvoreného elementu.

Triedy reprezentujúce binárne operácie

Triedy **OrBooleanOperation**, **NorBooleanOperation**, **NotBooleanOperation**, **XorBooleanOperation**, atď., reprezentujúce operácie v booleovskej algebre sú potomkami abstraktnej triedy BooleanOperationB. Tieto triedy majú definovaný konštruktor, v ktorom stačí zadať textové reprezentácie operácie a zavolá sa konštruktor predka, pričom asociativita závisí od typu operácie a netreba ju zadávať pri vytváraní. Taktiež sa v každej triede nachádza príslušný kopírovací konštruktor a move konštruktor, deštruktor a operátory priradenia. Z abstraktnej triedy Operation sú implementované metódy **clone** a **execute**, ktorá zo vstupných elementov na základe typu operácie vyjadrí výstup, ktorý uloží do výstupného elementu. V triede **NotBooleanOperation** je prekrytý getter **getNumberOfOperands**, keďže sa jedná o operáciu s 1 operandom.



Obr. 2.8 Diagram tried viachodnotových operácií

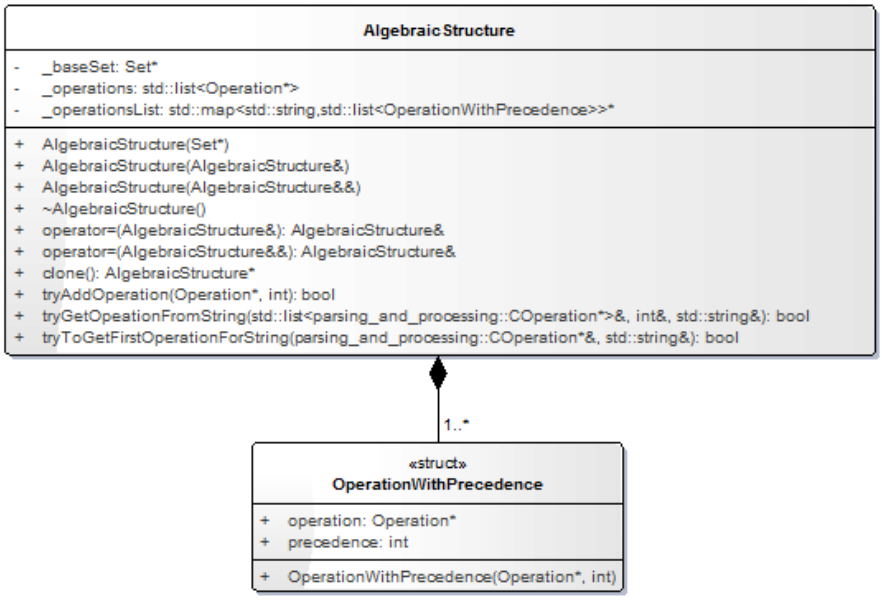
MVLOperationB

Podobne ako pri booleovskej algebre, aj vo viachodnotovej logike existuje predok každej definovanej viachodnotovej operácie, ktorý je potomkom triedy Operation. Jedná sa

o abstraktnú triedu **MVLOperationB**, ktorá obsahuje atribúty pre uloženie textových reprezentácií a asociativity operácie prístupné v potomkoch. Taktiež sú v potomkoch prístupné konštruktory, ktoré buď inicializujú atribúty zo zadaných hodnôt alebo hodnoty nakopírujú, resp. presunú z už vytvorenej booleovskej operácie. Z abstraktnej triedy Operation sú tu implementované gettery ako **getSymbols**, **getNumberOfOperands**, v ktorej sa vracia štandardne číslo 2, **getOperationAssociativity** a metódy **execute**, pričom je potrebné v potomkoch prekryť tú metódu **execute**, v ktorej sa vykoná operácia a výsledok z nej sa uloží už do existujúceho elementu.

Triedy reprezentujúce viachodnotové operácie

Sú to triedy **MVLLiteralOperation**, **MVLMinOperation**, **MVLMaxOperation**, **MVLComplementOperation,** atď., ktoré sú potomkami abstraktnej triedy MVLOperationB. A reprezentujú niektorú viachodnotovú operáciu. V každej triede je definovaný kopírovací konštruktor, move konštruktor a parametrický konštruktor, ktorý vyžaduje zadať určité hodnoty atribútov, pričom v každom je potrebné zadať textové reprezentácie operácie, pri niektorých operáciách je potrebné zadať aj množinu, či ďalšie elementy. Samozrejmosťou týchto tried je definovaný deštruktor, operátory priradenia a metóda **clone**. Tiež je v každej triede prekrytá metóda **execute**, ktorá z hodnôt vstupných elementov vloží do výstupného elementu výsledok príslušnej operácie.



Obr. 2.9 Diagram triedy AlgebraicStructure

AlgebraicStructure

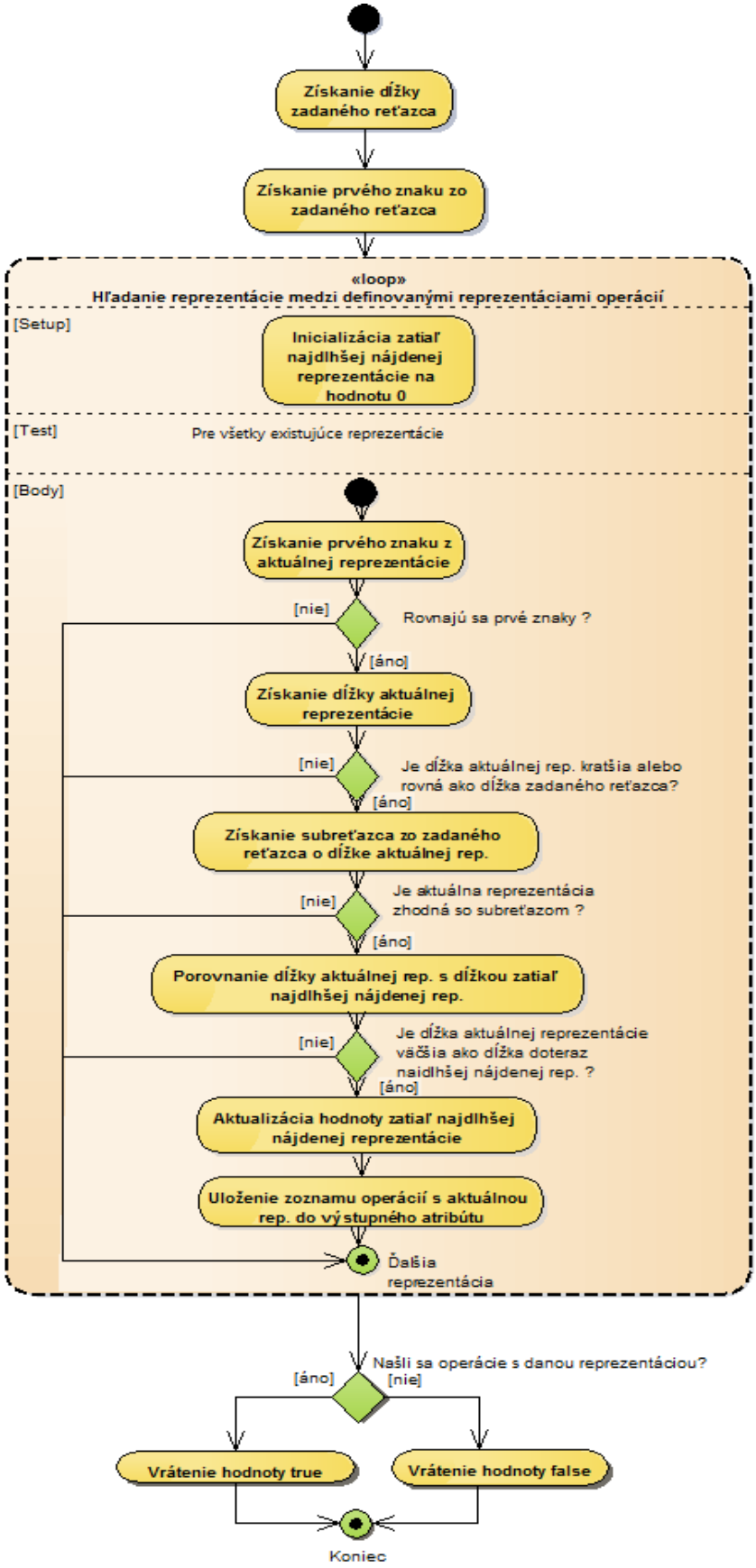
Poslednou triedou, ktorá bude v mennom priestore algebra predstavená, je **AlgebraicStructure**. Táto trieda reprezentuje algebraickú štruktúru, resp. algebru v tom zmysle, že je tu atribút prestavujúci množinu, nad ktorou je algebra postavená a taktiež atribút predstavujúci zoznam operácií, ktoré sú v danej algebre definované. Taktiež je tu aj atribút, ktorý uchováva operácie spolu s ich prioritami podľa textovej reprezentácie operácií.

Pre vytvorenie objektu tejto triedy je potrebné použiť konštruktor, pričom je potrebné zadať smerník na objekt triedy Set alebo iný objekt tejto triedy, z ktorého sa hodnoty atribútov buď skopírujú alebo presunú. Pre potreby uvoľnenia alokovanej pamäte objektu tejto triedy je definovaný deštruktor a taktiež tu sú prekryté operátory priradenia. Na vytvorenie kópie objektu tejto triedy je možné využiť metódu **clone** a na pridanie novej operácie do algebraickej štruktúry je vhodné použiť metódu **tryAddOperation**. Zaujímavou je metóda **tryGetOpeationFromString**, ktorá sa snaží na základe zadaného textu a začiatočnej pozície v tomto texte nájsť najvýstižnejšiu textovú reprezentáciu operácií, pričom ak sa nájdenie podarí, tak sa tieto operácie uložia do výstupného parametra. Jej fungovanie je zobrazené na obr. 2.10, kde je možné si všimnúť, že sa jedná o hľadanie najlepšieho vhodného kandidáta na textovú reprezentáciu operácie, čo znamená nájdenie najdlhšej textovej reprezentácie uložených operácií v algebraickom systéme. Nakoniec sa tu nachádza metóda **tryToGetFirstOperationForString**, ktorá vráti prvú nájdenú operáciu s danou textovou reprezentáciou.

2.2.2 Menný priestor parsing_and_processing

V tomto mennom priestore sú vytvorené triedy, ktoré slúžia pri parsovaní textového vstupu, pri vytváraní stromu spracovaním výsledku parsovania a pri práci s logickou funkciou. Všetky vytvorené triedy v tomto mennom priestore a vzájomné vzťahy medzi nimi je možné vidieť na obr. 2.11.

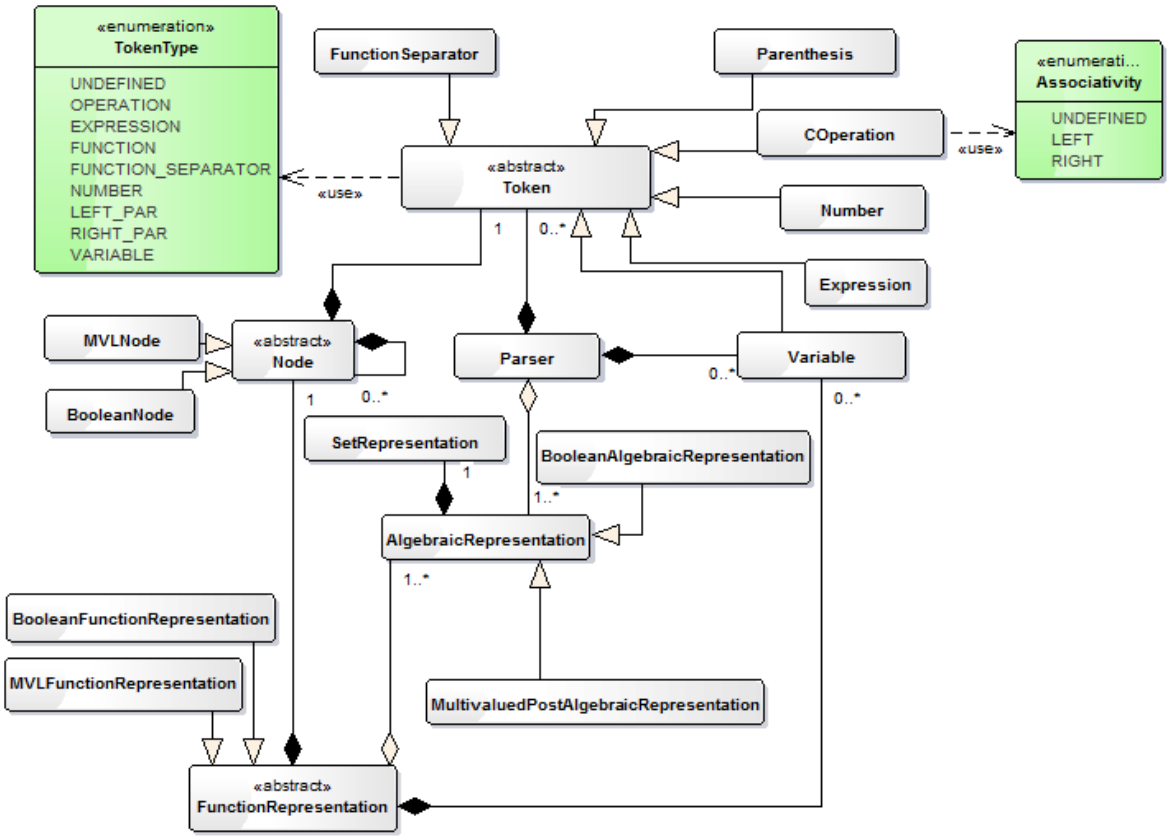
Avšak skôr než budú detailnejšie predstavené jednotlivé triedy tohto menného priestoru spolu s detailnejším zobrazením diagramov tried, je potrebné si priblížiť princíp parsovania vstupného textu a taktiež princíp vybudovania stromu z výsledku parsovania vstupného textu.



Obr. 2.10 Activity diagram pre metódu tryGetOpeationFromString

Princíp parsovania

Pri parsovaní je potrebné zvoliť taký algoritmus, ktorý dokáže pracovať nielen s logickými funkciami zapísanými pomocou logického výrazu, ale aj inými matematickými funkciami. Taktiež je vyžadované, aby bol algoritmus efektívny. Na základe týchto požiadaviek sa pre parsovanie využíva implementácia tzv. shunting-yard algoritmu, ktorý predstavil Edsger Dijkstra **[**19**]**.



Obr. 2.11 Vzťahy medzi triedami v mennom priestore parsing_and_processing

Hlavný princíp shunting-yard algoritmu je v tom, že transformuje matematický, resp. logický výraz z textového vstupu (napr. x1 + x2) do formy obrátenej poľskej notácie (angl. „Reverse polish notation“), skrátene OPN (napr. x2 x1 +) **[**20**]**. Pre získanie OPN je potrebné evidovať 2 zásobníky. Prvý zásobník, ktorý bude nazvaný ako výstupný, bude uchovávať výstupnú množinu tokenov a druhý zásobník nazvaný pomocný zásobník bude uchovávať operácie, zátvorky a funkčné separátory pre potreby zachovania priority operátorov. Pojem token je v tomto algoritme chápaný ako textová reprezentácia s určitou významovou hodnotou **[**21**]**. To znamená, že napríklad token „x1“ je chápaný ako premenná, token „1“ je chápaný ako číslo či token „+“ je chápaný ako operácia OR.

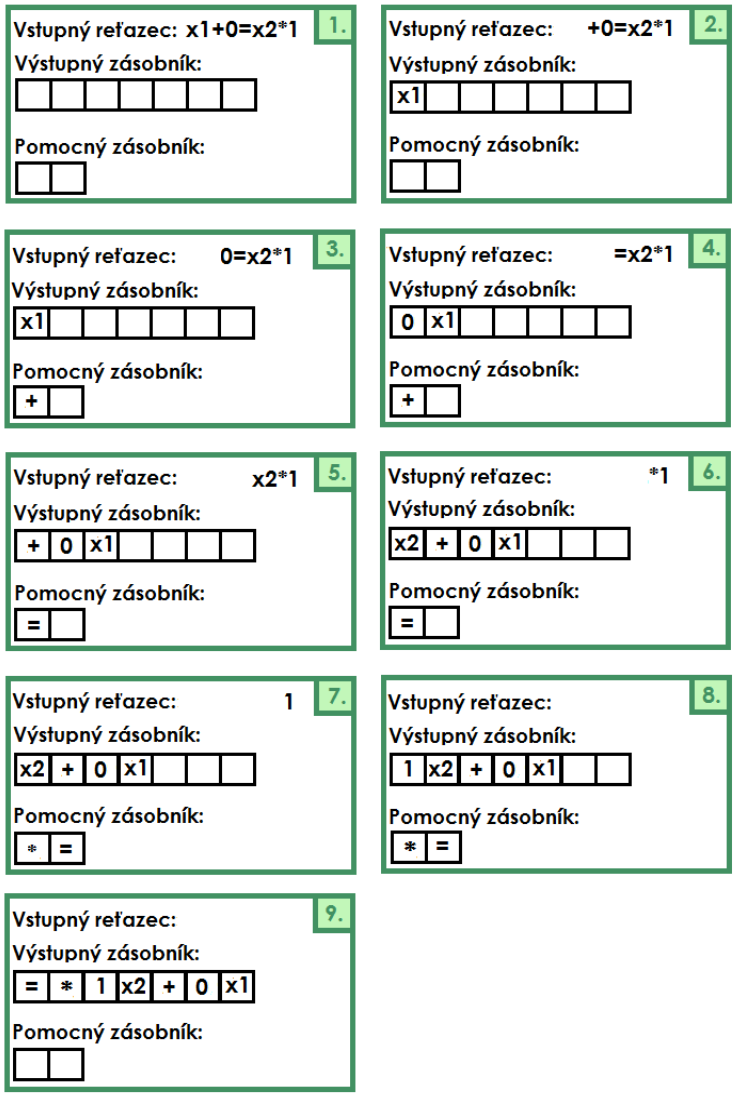
Algoritmus teda pracuje tak, že postupne rozpoznáva tokeny v textovom vstupe

a vloží ich do jedného z dvoch zásobníkov podľa nasledovných pravidiel:

- Ak je token číslo alebo premenná, tak vlož tento token do výstupného zásobníka.
- Ak je token operáciou, tak ho vlož do pomocného zásobníka. Avšak predtým, než sa tak stane, skontroluj prioritu operácie na vrchu pomocného zásobníka. Ak sa tam nachádza operácia s vyššou alebo rovnakou prioritou a vkladajúca operácia je zľava asociatívna alebo vkladajúca operácia je sprava asociatívna a na vrchu pomocného zásobníka je operácia s vyššou prioritou, potom vyber operáciu z vrchu pomocného zásobníka a vlož ju do výstupného zásobníka. Opakuj tento postup, pokým nie je pomocný zásobník prázdny alebo podmienka pre odobratie operácie z vrchu pomocného zásobníka nie je splnená.
- Ak je token funkciou, tak ho vlož do výstupného zásobníka a do pomocného zásobníka vlož token reprezentujúci začiatok funkcie. Ak je však token funkčným separátorom, potom vyberaj postupne tokeny z pomocného zásobníka a vkladaj ich do výstupného zásobníka pokým nenarazíš na začiatok funkcie. Ak sa začiatok funkcie nenájde, potom prehlás funkciu a tým pádom aj textový vstup za neplatný.
- Ak je token ľavou zátvorkou, tak ho vlož do pomocného zásobníka. Avšak ak je token pravou zátvorkou, potom vyberaj tokeny z pomocného zásobníka a vkladaj ich do výstupného zásobníka pokým nie je na vrchu pomocného zásobníka ľavá zátvorka. Potom ešte vyber ľavú zátvorku z pomocného zásobníka. Pokiaľ nenarazíš na ľavú zátvorku, potom zátvorky v textovom vstupe sú zadané nesprávne a prehlás textový vstup za neplatný.

Pokiaľ sa už spracujú všetky tokeny z textového vstupu, postupne sa presunú tokeny z pomocného zásobníka do výstupného zásobníka. Ešte je potrebné dodať, že pri spracovávaní tokena sa taktiež testuje, či daný token môže mať pred sebou daný typ tokenu prípadne či textový vstup môže začať, resp. končiť tokenom daného typu.

Na obr. 2.12 je možné vidieť postupné spracovanie vstupného reťazca x1+0=x2*1, ktorý predstavuje booleovskú funkciu v ktorej operácia * je operáciou AND s prioritou 2, operácia + je operácia OR s prioritou 3, operácia = je operáciou ekvivalencie s prioritou 4, pričom každá operácia je zľava asociatívna.



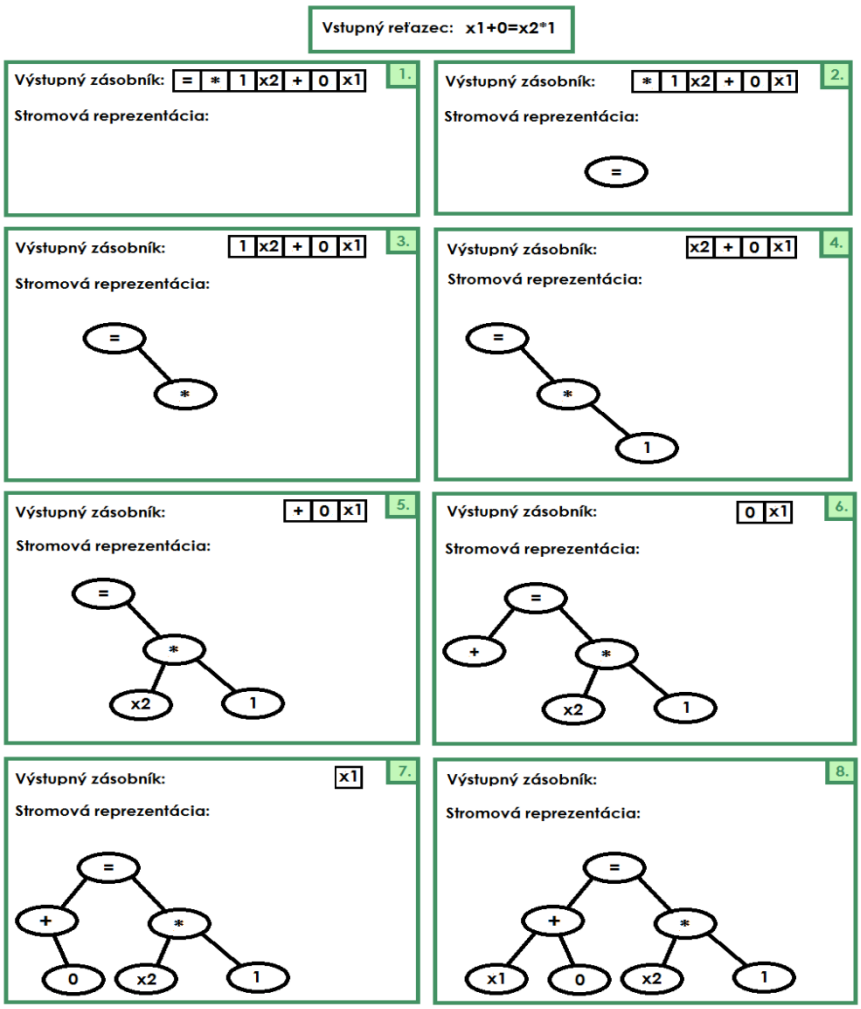
Obr. 2.12 Spracovanie textového vstupu shunting-yard algritmom

Princíp budovania viaccestného stromu reprezentujúceho funkciu

Po úspešnom vykonaní parsovania sa textový vstup transformuje na OPN uloženú vo výstupnom zásobníku. Avšak takáto reprezentácia nie je najvhodnejšia pre vykonávanie určitých špecifických operácií (napr. transformácia na NDF či smerová derivácia). Preto je potrebné transformovať výstupný zásobník do formy viaccestného stromu, pričom sa využije usporiadanie tokenov vo výstupnom zásobníku.

Budovanie viaccestného stromu prebieha pomocou algoritmu, ktorého vykonávanie je znázornené na obr. 2.13. Algoritmus začína tým, že najskôr sa zistí, či je výstupný zásobník prázdny. Ak je prázdny, tak algoritmus končí. Avšak pokiaľ zásobník nie je prázdny, tak sa vykoná nasledovný postup:

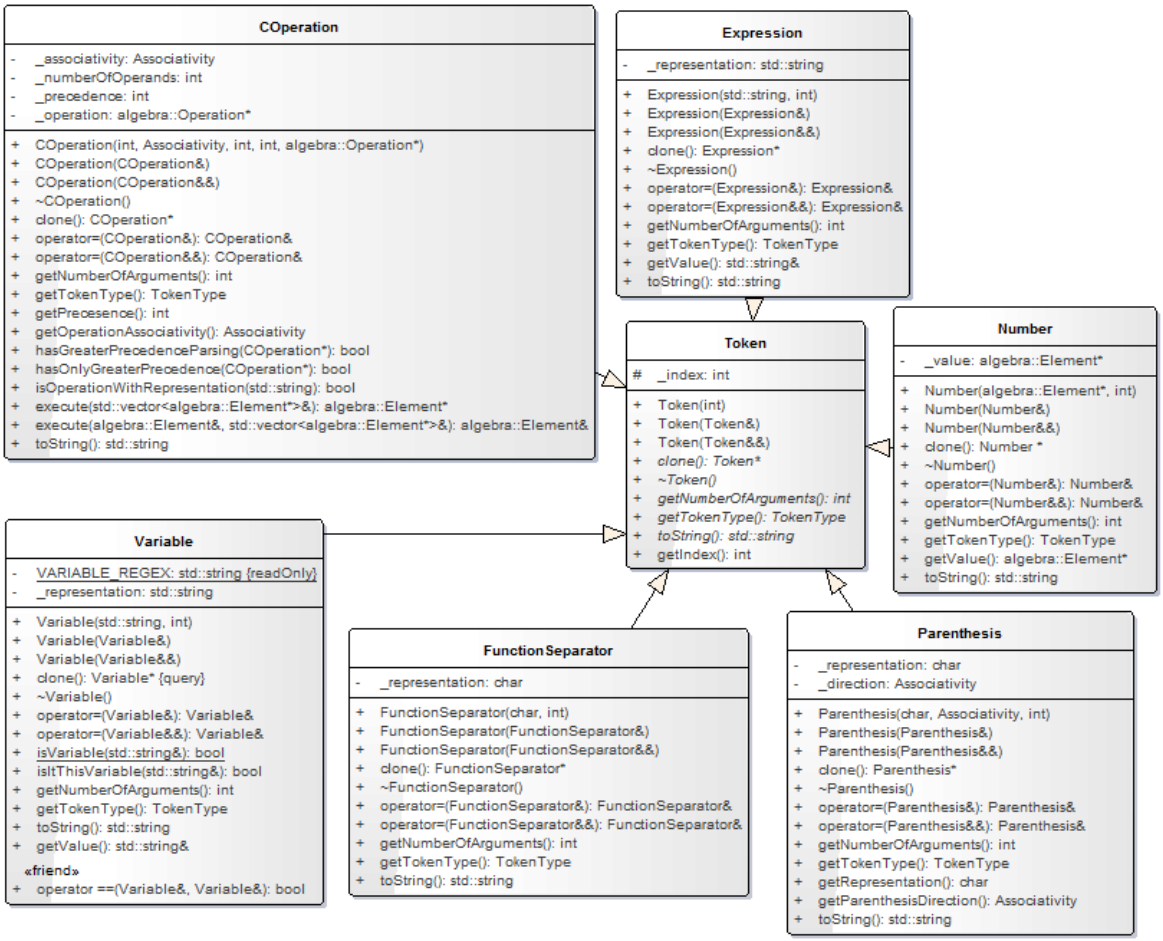
1. Vytvor vrchol a nastav tento vrchol ako koreň.
2. Vykonávaj tieto úkony rekurzívne, pokiaľ nevybuduješ viaccestný strom:
   2.1. Vyber token z výstupného zásobníka a vlož ho do vrcholu. Pokiaľ sa žiadny token v zásobníku nenachádza, potom prehlás strom za neplatný a skonči.
   2.2. Ak token nemá žiadne argumenty, potom koniec.
   2.3. Ak má token n argumentov, potom opakuj nasledovné kroky n krát:
      2.3.1. Vytvor nový vrchol.
      2.3.2. Tento vrchol nastav ako syna vrcholu, ktorý obsahuje token z kroku 2.3.
      2.3.3. Spusť rekurzívne algoritmus od kroku 2.1. nad synom.



Obr. 2.13 Spracovanie výstupného zásobníka na viaccestný strom

Po vykonaní algoritmu sa buď vytvorí viaccestný strom, alebo sa prehlási vybudovaný strom za neplatný, pokiaľ je počet tokenov menší, resp. väčší ako požadovaný počet pre vybudovanie viaccestného stromu. Následne je možné nad vytvoreným stromom vykonávať rôzne potrebné symbolické či numerické úkony.

Pre vykonávanie parsovania zo zadaného textového vstupu a vytvorenia viaccestného stromu z výstupného zásobníka sú v softvérovej knižnici vytvorené triedy v mennom priestore parsing_and_processing, ktoré budú teraz priblížené.



Obr. 2.14 Diagram tried pre Token a jeho potomkov

.

Token

Abstraktná trieda **Token** predstavuje token, teda časť vstupného textu z určitou hodnotou. Uchováva sa tu číselná hodnota indexu začiatku tokenu vo vstupnom texte, pričom je táto hodnota prístupná aj v potomkoch a je možné zistiť jej hodnototu pomocou getteru **getIndex**. Taktiež sú v potomkoch prístupné konštruktory, pričom je potrebné zadať index, prípadne iný objekt typu Token, z ktorého sa hodnoty nakopírujú, resp.

presunú. Každý potomok tejto triedy bude musieť implementovať metódy ako **clone**, slúžiaca na získanie kópie tokenu, **getNumberOfArguments** pre získanie číselnej informácie o počte argumentov, ktoré token potrebuje, **getTokenType** na zistenie typu tokena a **toString** pre získanie textovej informácie o tokene.

Potomkovia triedy Token

Každá trieda, ktorá je potomkom triedy token, reprezentuje určitý špeciálny typ tokena ako číslo, operácia, premenná, atď., pričom implementuje všetky virtuálne metódy predka. V každej triede sa nachádza parametrický konštruktor, ktorý okrem indexu potrebuje ešte ďalšie hodnoty, kopírovací a move konštruktor, deštruktor, operátory priradenia, či implementácia všetkých virtuálnych metód.

Trieda **Expression** reprezentuje výraz (napr. $5 * x + 8$), čo znamená, že obsahuje atribút na uloženie textovej reprezentácie výrazu a taktiež getter na túto hodnotu. Pri vytvorení je preto potrebné zadať okrem indexu aj textovú reprezentáciu daného výrazu.

Trieda **Number** predstavuje číselnú hodnotu zo vstupného textu. Preto sa tu nachádza atribút typu Element, ktorý uchováva túto hodnotu a jeho hodnotu je možné získať zavolaním príslušného gettera. Pri vytváraní objektu tejto triedy je preto potrebné okrem indexu zadať aj pointer na objekt triedy Element predstavujúci danú číselnú hodnotu.

Trieda **Parenthesis** sa používa na reprezentáciu zátvoriek. Preto je potrebné evidovať znak reprezentujúci zátvorku a taktiež informáciu o tom, či sa jedná o pravú alebo o ľavú zátvorku. Tieto hodnoty atribútov je možné zistiť príslušnými gettermi. Taktiež je zrejmé, že pri vytváraní objektu tejto triedy je potrebné okrem indexu zadať aj znak reprezentujúci zátvorku a typ zátvorky.

Trieda **FunctionSeparator** reprezentuje oddeľovač parametrov funkcie. Nachádza sa tu atribút na uchovávanie znakovej reprezentácie oddeľovača. To znamená, že do konštruktora je potrebné zadať aj túto hodnotu okrem indexu oddeľovača vo vstupnom texte.

Trieda **Variable** je používaná na reprezentáciu premennej, pričom je v tejto triede definovaný statický atribút **VARIABLE_REGEX**, ktorý predstavuje regulárny výraz pre reprezentáciu premennej. Pokiaľ je potrebné zistiť, či daný reťazec môže byť premennou, stačí použiť statickú metódu **isVariable**, ktorá otestuje, či zadaný reťazec začína malým alebo veľkým písmenom latinskej abecedy, prípadne znakmi ‚$‘ a ‚_‘ a či ostatné znaky

sú malé alebo veľké písmena latinskej abecedy, čísla, prípadne znak ‚_‘. Nachádza sa tu taktiež atribút na uchovávanie reprezentácie premennej, pričom jeho hodnotu je potrené zadať pri vytváraní objektu tejto triedy a na zistenie jeho hodnoty je tu definovaný príslušný getter. Pre zistenie, či zadaný reťazec je reprezentáciou príslušnej premennej je možné použiť metódu **isItThisVariable**, prípadne ak chceme zistiť, či 2 objekty tejto triedy sú rovnaké (prestavujú tú istú premennú) stačí použiť operátor ==.

Posledným potomkom triedy Token je trieda **COperation**, ktorá reprezentuje operáciu vo vstupnom texte. Nachádzajú sa tu atribúty reprezentujúce asociativitu operácie, počet operanov, prioritu operácie a smerník na príslušnú operáciu. Pre vytvorenie objektu tejto triedy je preto potrebné zadať všetky hodnoty atribútov aj s indexom. Gettery tu sú definované len pre asociativitu, počet operandov a prioritu. Pre potrebu porovnávania operácií podľa shunting-yard algoritmu je tu metóda **hasGreaterPrecedenceParsing** a pre jednoduché porovnávanie priorít operácií je tu metóda **hasOnlyGreaterPrecedence**, ktorá skúma, či daná operácia má menšiu prioritu ako parametrom zadaná operácia. Taktiež sú tu metódy **execute**, ktoré volajú príslušné metódy atribútu reprezentujúceho operáciu, či metóda **isOperationWithRepresentation**, ktorou je možné zistiť, či daná operácia je reprezentovaná zadaným textovým výrazom.

SetRepresentation

Trieda **SetRepresentation** predstavuje nadstavbu nad abstraktnou triedou Set v tom zmysle, že poskytuje nástroje na zistenie, či určitý element patrí do danej množiny. To znamená, že sa tu nachádza atribút obsahujúci smerník na objekt triedy Set a taktiež atribút obsahujúci regulárny výraz pre reprezentáciu čísel z danej množiny. Na vytvorenie objektu tejto triedy je potrebné použiť jeden z definovaných konštruktorov, či už move konštruktor, kopírovací konštruktor alebo parametrický konštruktor, pri ktorom je potrebné zadať regulárny výraz a smerník na množinu. Samozrejmosťou sú deštruktor a operátory priradenia. Taktiež sa tu nachádza metódy ako **clone**, ktorá vráti smerník na hlbokú kópiu daného objektu, getter na regulárny výraz, **isWellFormedNumber** na zistenie, či zadaný reťazec spĺňa podmienky správne formulovaného čísla množiny, **testNumberFromString** na zistenie, či je zadaný reťazec platnou reprezentáciou čísla z množiny, čo znamená, že má správnu formuláciu a taktiež element reprezentovaný týmto reťazcom patrí do príslušnej množiny, či metóda **tryGetNumberFromString**, ktorá pracuje podobne ako metóda **testNumberFromString** avšak s tým rozdielom, že táto metóda uloží aj element

patriaci do príslušnej množiny do výstupného parametra.

Obr 2.15 Diagram tried SetRepresentation, AlgebraicRepresentation a potomkov

AlgebraicRepresentation

Táto trieda predstavuje základ pre všetky algebraické reprezentácie, čo je vlastne súhrn objektov z tried, potrebných pre správnu funkcionalitu algebraického systému. To znamená, že táto trieda uchováva smerník na objekty tried AlgebraicStructure a SetRepresentation, ktoré sú prístupné v potomkoch a taktiež sa tu uchováva textová informácia o algebraickej reprezentácií prístupná aj v potomkoch tejto triedy. Pre vytvorenie objektu tejto triedy je možné použiť kopírovací konštruktor, move konštruktor alebo parametrický konštruktor, ktorému je potrebné zadať smerníky na objekty triedy SetRepresentation a AlgebraicStructure. Nachádza sa tu aj deštruktor, operátory priradenia,

metóda **clone** na získanie hlbokej kópie objektu, gettery pre všetky atribúty, metóda **equals** na zistenie, či sa zadaná algebraická reprezentácia zhodná s príslušnou algebraickou reprezentáciou a metóda **tryToGetFirstOperationWithRepresentation**, ktorá sa pokúsi nájsť prvú operáciu, ktorej reprezentácia je zhodná so zadanou textovou reprezentáciou.

Potomkovia triedy AlgebraicRepresentation

Každý z potomkov triedy AlgebraicRepresentartion predstavuje konkrétny algebraický systém, preto sa v potomkoch nachádza len jeden bezparametrický konštruktor, ktorý slúži na to, aby vytvoril objekt, v ktorom zadefinuje všetky potrebné hodnoty atribútov predka.

Trieda **BooleanAlgebraicRepresentation** reprezentuje booleovskú logiku, čo znamená, že pri vytváraní objektu tejto triedy sa vytvorí booleovská množina, vytvoria sa všetky definované booleovské operácie v mennom priestore algebra a následne sa ešte vytvorí objekt triedy SetRepresentation, ktorý obmedzí množinu elementov na elementy, ktoré sú reprezentované len booleovskými hodnotami, čo znamená hodnotami 0 a 1.

Trieda **MultivaluedPostAlgebraicRepresentation** predstavuje viachodnotovú Postovu algebru, čo znamená, že sa pri vytváraní objektu tejto triedy vytvorí viachodnotová množina so zadaným m, vytvoria sa základné operácie definované v Postovej algebre a taktiež aj ďalšie používané viachodnotové operácie definované v mennom priestore algebra. Na záver sa ešte vytvorí objekt triedy SetRepresentation, ktorý obmedzí množinu elementov na elementy, ktoré patria do množiny $\{0,1,\ldots,m-1\}$.

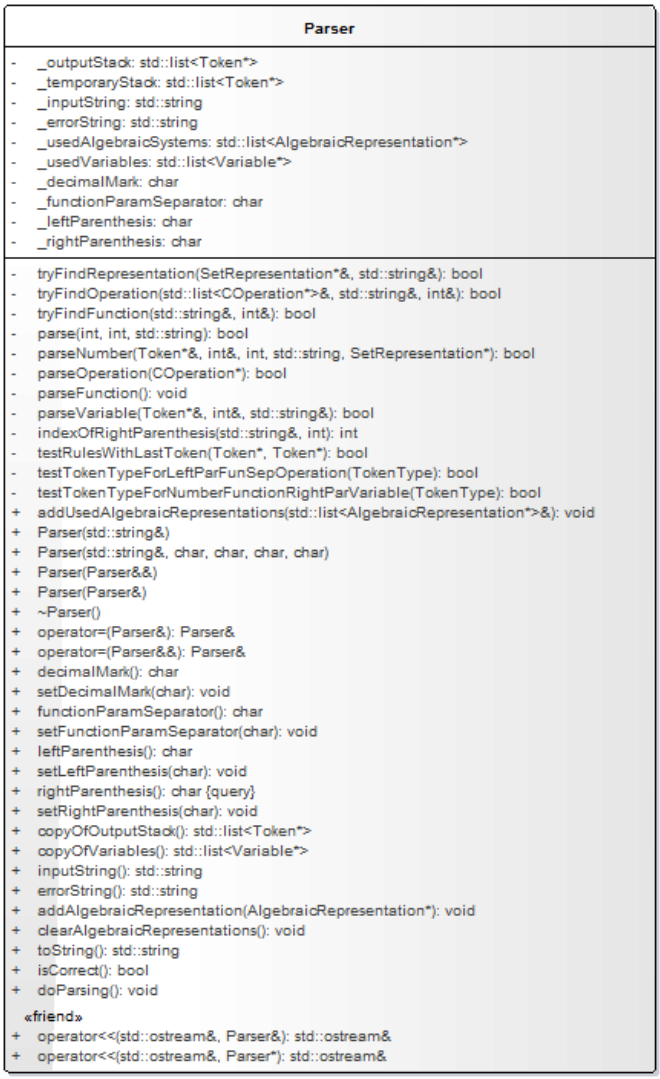
Parser

Jednou z hlavných tried tohto menného priestoru je trieda **Parser**, ktorej hlavnou úlohou je realizácia parsovania podľa shunting-yard algoritmu. Na to, aby to bolo možné, je potrebné v tejto triede uchovávať hodnotu vstupného reťazca. Taktiež sa tu uchovávajú použité algebraické systémy formou zoznamu objektov typu AlgebraicRepresentation, zoznam použitých premenných vo vstupnom reťazci, pomocný zásobník a výstupný zásobník prostredníctvom zoznamu objektov typu Token či znakové reprezentácie dôležitých významových prvkov ako pravá a ľavá zátvorka, desatinná čiarka a funkčný separátor. Nachádza sa tu aj atribút pre potreby ukladania chybových hlásení vo forme reťazca.

Pokiaľ je potrebné vytvoriť objekt tejto triedy, je možné pre tento účel použiť

kopírovací konštruktor, move konštruktor, prípadne jeden z definovaných parametrických konštruktorov. Prvý na vytvorenie nového objektu tejto triedy potrebuje len vstupný reťazec, pričom sa hodnoty dôležitých významových prvkov nastavia na štandardné hodnoty, čo znamená že desatinná čiarka je reprezentovaná čiarkou, funkčný separátor je reprezentovaný dvojbodkou a zátvorky sú reprezentované znakmi ‚(' a ‚)'. V druhom je potrebné zadať okrem vstupného reťazca aj reprezentácie jednotlivých dôležitých významových prvkov. Pre zničenie objektu tejto triedy je tu definovaný deštruktor, taktiež sú tu definované operátory priradenia gettery a settery na všetky dôležité významové prvky a gettery na vstupný reťazec a chybové hlásenie.

Jednou z hlavných metód tejto triedy je metóda **doParsing**, ktorou je možné spustiť parsovanie vstupného reťazca zavolaním privátnej metódy **parse**, ktorá implementuje shunting-yard algoritmus, čo znamená, že postupne zo zadaného reťazca od zadanej



Obr. 2.16 Diagram triedy Parser

pozície prechádza všetky znaky a určuje ich význam, potom vytvorí príslušný typ tokenu, ktorý podľa pravidiel shunting-yard algoritmu vloží do správneho algoritmu. Pokiaľ parsovanie neprebehlo v poriadku, vyhodí sa tu výnimka, ktorá obsahuje informáciu o chybe. Avšak skôr než je možné spustiť parsovanie, vykoná sa kontrola správnosti nastavenia dôležitých významových prvkov zavolaním privátnej metódy **isCorrect** a pokiaľ niektoré dôležité významové prvky sú reprezentované rovnakými znakmi, tak sa taktiež vyhodí výnimka a parsovanie sa nevykoná.

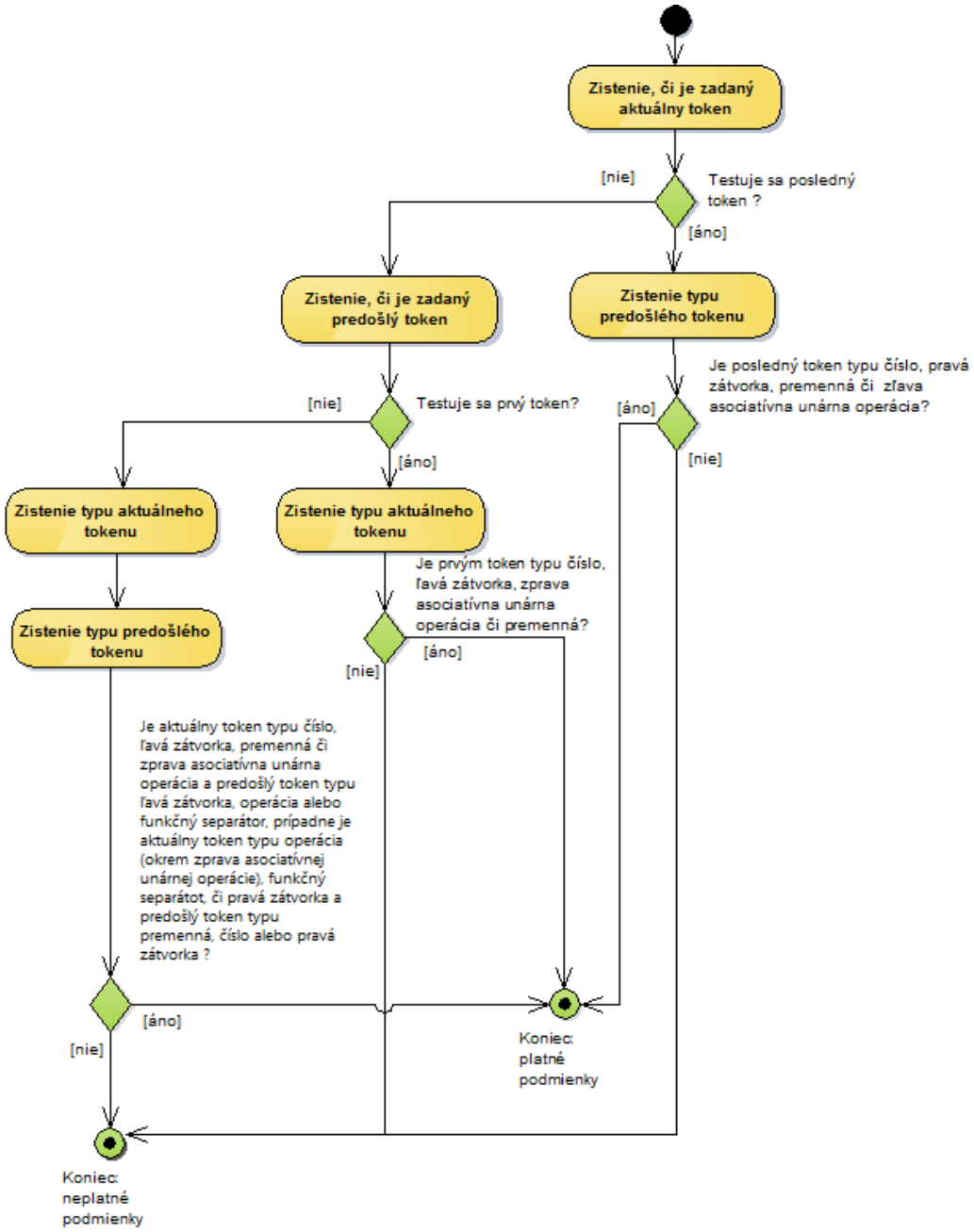
Na to, aby bolo možné rozpoznať zo znaku vstupného reťazca či sa jedná o číselný znak sa používa pri parsovaní privátna metóda **tryFindRepresentation**, ktorá nad každou algebraickou reprezentáciou skontroluje, či zadaný znak by mohol byť platným číslom. Pokiaľ to môže byť číslo určitej algebraickej reprezentácie, do výstupného parametra sa vloží smerník na objekt triedy SetRepresentation z tejto algebraickej reprezentácie. Inak sa upozorní volajúca metóda, že zadaný znak nemôže byť číslom. Keď sa potvrdí, že zadaný znak by mal byť číslom, zavolá sa metóda **parseNumber**, ktorá postupne z danej pozície vo vstupnom reťazci odoberá znaky, až pokiaľ sú dané znaky platnou číselnou reprezentáciou. Po získaní textovej číselnej reprezentácie sa z nej získa číselný token, čiže objekt triedy Number.

Privátna metóda **indexOfRightParenthesis** taktiež nájde svoje využitie pri parsovaní. Jej hlavnou úlohou je zistiť, či sú zátvorky správne formulované, čo znamená, že zisťuje v zadanom reťazci od zadanej pozície existenciu pravej zátvorky pre každú ľavú zátvorku.

Pre rozpoznanie reprezentácie operácie zo vstupného reťazca sa používa privátna metóda **tryFindOperation**, ktorá pre každú algebraickú reprezentáciu skontroluje, či sa v jej algebraickej štruktúre nenachádza operácia s danou textovou reprezentáciou získanou zo vstupného reťazca. Pokiaľ sa žiadna taká operácia nenájde, tak sa táto skutočnosť oznámi volanej triede. Avšak ak sa taká operácia nájde, táto operácia je potom podľa pravidiel shunting-yard algoritmu vložená do správneho zásobníka zavolaním privátnej metódy **parseOperation**.

Na záver, pokiaľ pri parsovaní sa aktuálny znak zo vstupného reťazca nevyhodnotí žiadnymi testami ako znak reprezentujúci číslo, dôležité významové prvky či operáciu, tak sa spustí privátna metóda **parseVariable**. Táto metóda postupne od zadanej pozície vstupného reťazca spracováva jednotlivé znaky a testuje, či sú tieto znaky platnými znakmi pre premennú a nie sú to znaky reprezentujúce dôležité významové prvky, operáciu či sa jedná o biele znaky (jedná sa o medzeru, tabulátor, line feed, a pod.). Pokiaľ sa narazí na

znak, ktorý by nevyhovoval zadaným podmienkam, tak sa zoberie reprezentácia premennej od zadanej pozície až po nevyhovujúci znak. Následne sa z tejto reprezentácie vytvorí token reprezentujúci premennú a vloží sa do výstupného zásobníka a do zoznamu používaných premenných, pokiaľ sa tam ešte nenachádza.

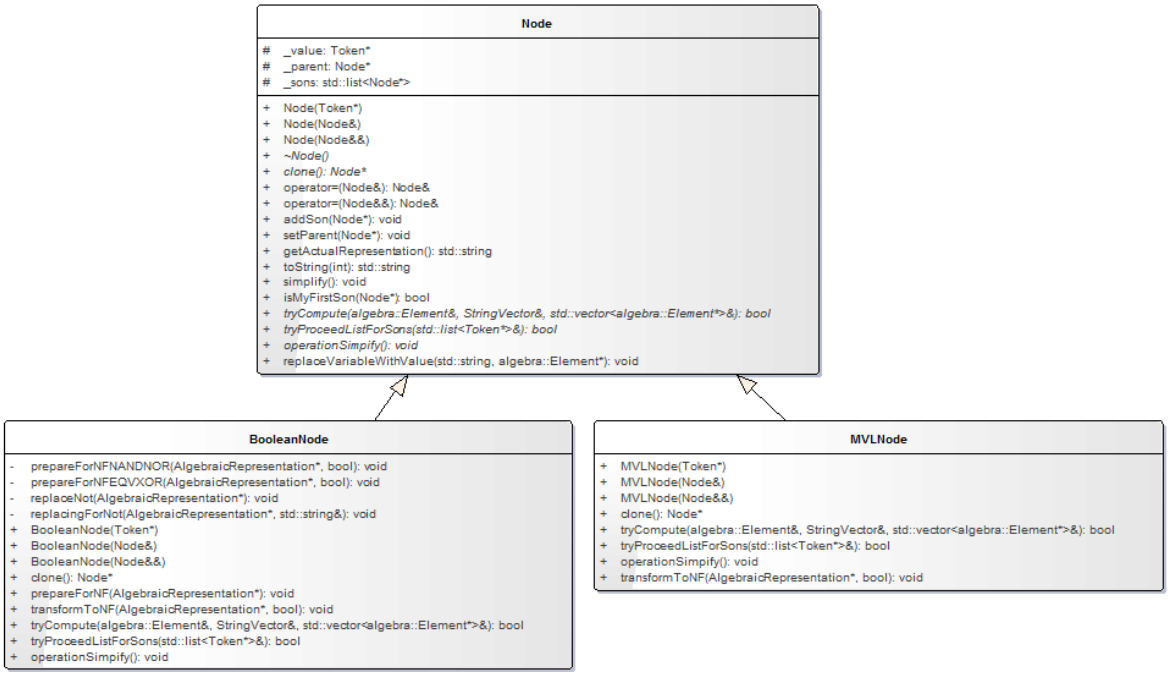


Obr. 2.17 Activity diagram pre metódu testRulesWithLastToken

Pri parsovaní taktiež prebieha kontrola platnosti typu aktuálneho tokenu voči typu predošlého tokenu, či kontrola typu prvého a posledného tokenu pomocou volania privátnej metódy **testRulesWithLastToken**, ktorej princíp a jednotlivé pravidlá je možné

vidieť na obr. 2.17, pričom na testovanie toho, či je zadaný typ tokenu ľavá zátvorka, funkčný separátor alebo operácia sa používa privátna metóda **testTokenTypeForLeftParFunSepOperation** a na testovanie toho, či je token typu číslo, premenná alebo pravá zátvorka sa používa privátna metóda **testTokenTypeForNumberFunctionRightParVariable**.

Ako posledné budú priblížené metódy ako **copyOfOutputStack** na získanie hlbokej kópie výstupného zásobníka, **copyOfVariables** na získanie hlbokej kópie použitých premenných v rozparsovanom vstupnom reťazci, **addUsedAlgebraic Representations** pre nakopírovanie použitých algebraických reprezentácií pri parsovaní do výstupného parametra, **addAlgebraicRepresentation** na vloženie smerníka na algebraickú reprezentáciu do zoznamu použitých algebraických reprezentácií a **clearAlgebraicRepresentations** na vyprázdenie zoznamu použitých algebraických reprezentácií.



Obr. 2.18 Diagram tried Node, BooleanNode a MVLNode

Node

Abstraktná trieda **Node** predstavuje vrchol viaccestného stromu, ktorý sa vybuduje zo zadaného výstupného zásobníka podľa predstaveného algoritmu. V každom vrchole sa preto uchováva obsah reprezentovaný smerníkom na objekt triedy Token, smerník na otca, ktorý je objektom tejto triedy a zoznam synov, teda smerníkov na objekty tejto triedy.

Definované sú tu konštruktory, konkrétne kopírovací konštruktor, move konštruktor a parametrický konštruktor očakávajúci smerník na objekt triedy Token, ktoré sú prístupné v potomkoch tejto triedy či operátory priradenia.

K virtuálnym metódam, ktoré je potrebné implementovať v potomkoch patria metódy **clone** pre získanie hlbokej kópie, deštruktor, **tryCompute** používaná pre získanie číselnej hodnoty z aktuálneho vrcholu, ktorá sa uloží do výstupného parametra, **tryProceedListForSons** používaná pri budovaní stromu z výstupného zásobníka a **operationSimpify** na zjednodušenie aktuálneho vrchola, pokiaľ má v sebe uložený token typu operácia.

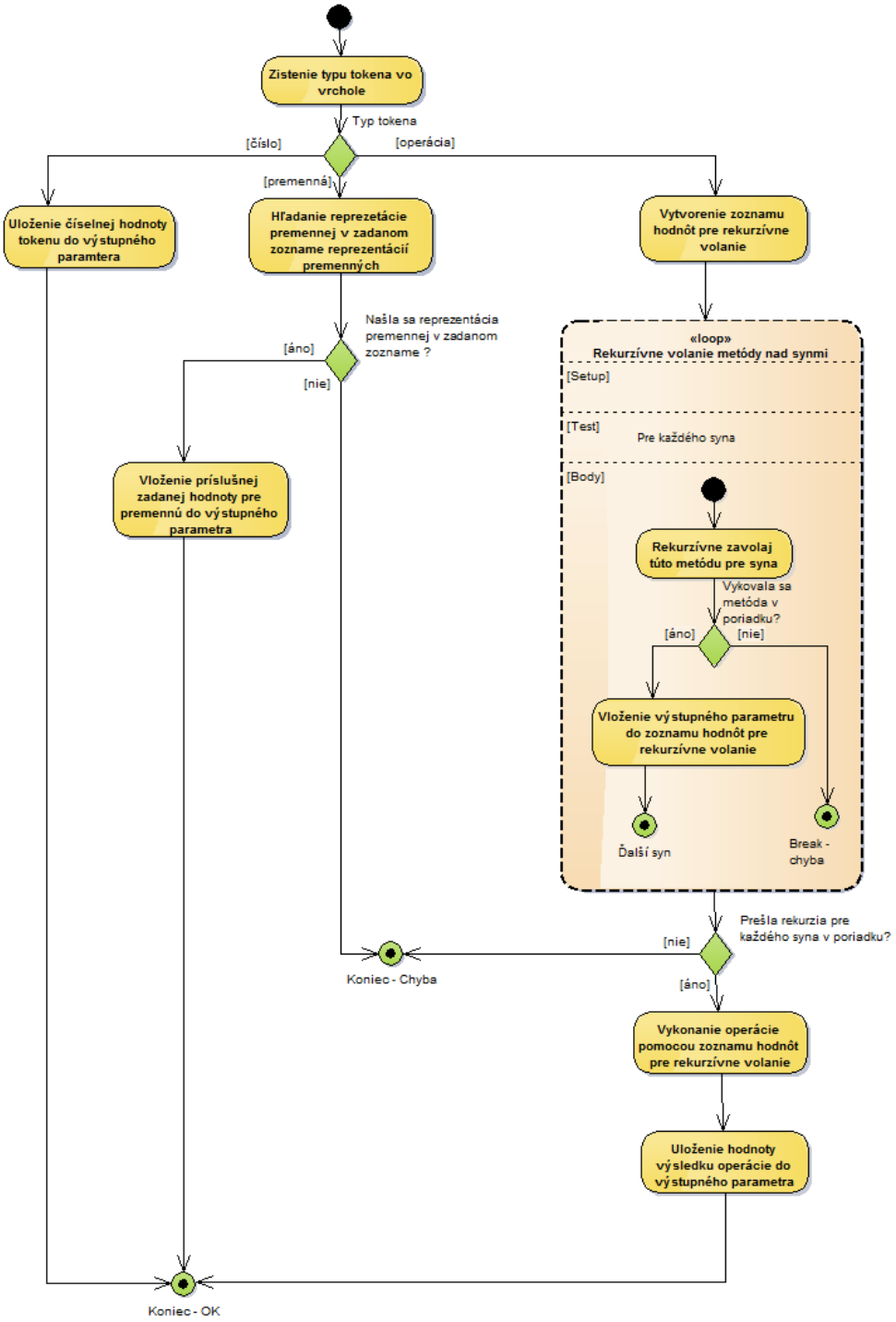
K metódam, ktoré potenciálne upravujú hodnoty atribútov patria metódy ako **addSon**, ktorá pridá zadaný vrchol aktuálnemu vrcholu ako syna a zadanému vrcholu sa nastaví aktuálny vrchol ako otec pomocou metódy **setParent**, **isMyFirstSon** na zistenie, či zadaný vrchol je prvým synom aktuálneho vrcholu, **replaceVariableWithValue**, ktorá sa najskôr zavolá nad všetkými synmi aktuálneho vrcholu a potom sa nahradí token typu premenná, ktorý je reprezentovaný zadaným reťazcom za token typu číslo, ktorý obsahuje zadaný element, či **simplify** na zjednodušenie vrcholu, čo znamená, že najskôr sa spustí táto metóda nad synmi aktuálneho vrcholu a potom pokiaľ je v aktuálnom vrchole uložený token typu operácia sa zavolá metóda operationSimplify.

Taktiež sa tu nachádzajú dve metódy metódy na získanie textovej reprezentácie. Prvou je metóda **toString**, ktorá vytvorí textovú reprezentáciu vrchola v strome, pričom sa najskôr do výstupného reťazca vloží zadaný počet medzier, následne reprezentácia tokenu, ktorý je uložený v aktuálnom vrchole a potom sa táto metóda zavolá nad každým synom aktuálneho vrchola, pričom sa ich výstup z operácie vloží do výstupnej reprezentácie vrcholu. Druhou je metóda **getActualRepresentation**, ktorá slúži na získanie výrazu reprezentujúceho aktuálny vrchol a jeho synov, pričom je kladený dôraz na to, aby boli vložené zátvorky, pokiaľ je v aktuálnom vrchole uložená operácia s vyššou prioritou a v niektorom zo synov je uložená operácia s nižšou prioritou.

BooleanNode

Trieda **BooleanNode** je potomkom abstraktnej triedy Node a predstavuej vrchol v strome reprezentujúci booleovskú funkciu. Táto trieda má definované 3 konštruktory, ktoré v sebe volajú príslušné konštruktory predka.

Všetky abstraktné metódy z predka tu sú definované z ohľadom na booleovskú algebru. Metóda **tryCompute**, ktorej princíp je možné vidieť na obr. 2.19 pracuje tak, že vyčísluje hodnotu booleovskej funkcie na základe zadaných reprezentácií booleovských



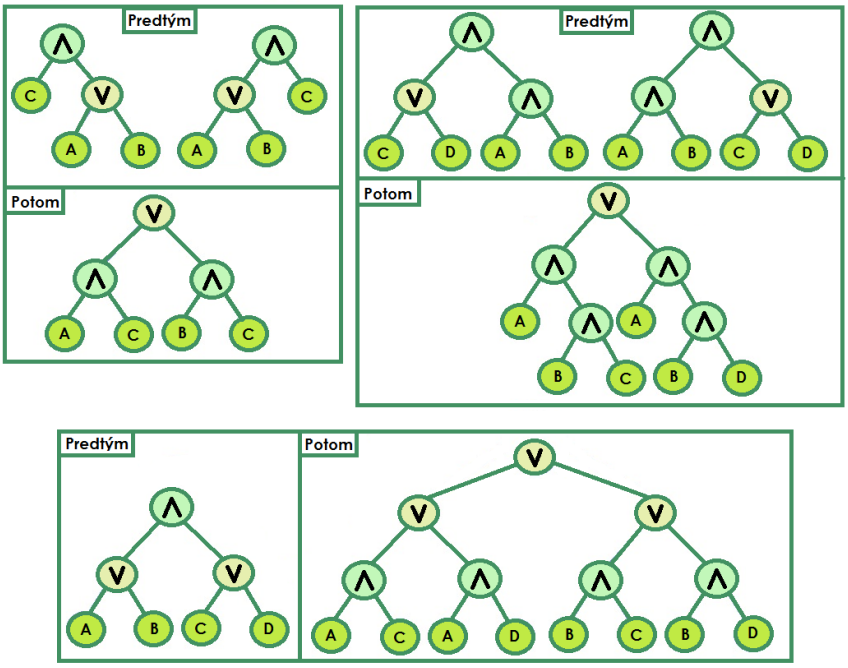
Obr. 2.19 Activity diagram pre metódu tryCompute

premenných a ich príslušných hodnôt, pričom výsledok tejto operácie sa uloží do

výstupného parametru, pokiaľ výpočet prebehne v poriadku. Metóda **clone** vytvorí kópiu aktuálneho vrcholu, pričom sa hodnota smerníka na otca nastaví na nullptr. Implementácia metódy **tryProceedListForSons** pracuje tak, že sa najskôr zistí z aktuálneho tokena, koľko argumentov očakáva, teda koľko synov je potrebné vytvoriť. Pokiaľ sa toľko tokenov v zadanom zásobníku nenachádza, metóda končí a oznámi neúspech volajúcej metóde. Potom sa postupne vytvára príslušný počet synov, pričom po vytvorení syna za zo zadaného zásobníka odstráni token použitý na vytvorenie syna a následne sa nad synom zavolá rekurzívne táto metóda. Poslednými implementovanými metódami sú deštruktor a **operationSimpify**, ktorá skontroluje typy tokenov uložených v synoch. Pokiaľ sú vo všetkých synoch uložené len tokeny typu číslo, prebehne výpočet výslednej hodnoty operácie na základne hodnôt tokenov v synoch a následne sa odstránia všetci synovia a hodnota v aktuálnom vrchole a miesto nej sa do aktuálneho vrcholu vloží token typu číslo obsahujúci výslednú hodnotu operácie.

Okrem implementácie virtuálnych metód sa tu nachádzajú metódy, ktoré sa využívajú pri transformácií funkcie na normálny tvar, či už na NDF alebo na NKF. Jednou z nich je privátna metóda **replaceNot**, ktorá sa používa na transformáciu vrcholu obsahujúceho operáciu NOT, pričom syn tohto vrcholu obsahuje booleovskú operáciu. Transformácia prebieha podľa pravidiel booleovskej algebry, napr. NOT(x1 EQV x2) sa transformuje na x1 XOR x2, NOT(NOT x1) sa transformuje na x1 alebo NOT(x1 AND x2) sa transformuje na x1 NAND x2. Pri transformácií vrcholu sa využíva privátna metóda **replacingForNot**, ktorá transformuje aktuálny vrchol tak, že operáciu NOT nahradí zadanou operáciou a ako synov aktuálneho vrcholu nastaví synov syna aktuálneho vrcholu. Následne sa táto metóda zavolá nad všetkými synmi daného vrcholu. Po transformácií sa zabezpečí, že všetky vrcholy obsahujúce operáciu NOT sa budú nachádzať len nad vrcholmi obsahujúcimi token typu premenná alebo číslo. Ďalšou metódou je **prepareForNF**, ktorá sa používa na to, aby sa v aktuálnom vrchole, ktorý uchováva token typu operácia nachádzali len operácie AND, OR a NOT. To znamená, že všetky ostatné booleovské operácie je potrebné transformovať na formu obsahujúcu len tieto operácie, napr. x1 NOR x2 sa transformuje na NOTx1 AND NOT x2, x1 NAND x2 sa transformuje na NOTx1 OR NOT x2, prípadne x1 EQV x2 sa transformuje na NOTx1 AND NOT x2 OR x1 AND x2. Tieto transformácie vykonávajú privátne metódy **prepareForNFNANDNOR** pre transformáciu vrcholov obsahujúcich operácie NAND a NOR a **prepareForNFEQVXOR** pre transformovanie vrcholov obsahujúcich operácie EQV a XOR. Následne sa nad všetkými synmi aktuálneho vrcholu zavolá táto metóda.

Poslednou metódou pre transformáciu na normálnu formu je metóda **transformToNF**, ktorá podľa zadanej logickej hodnoty transformuje vrchol tak, aby zodpovedal NDF, resp. NKF. Najskôr sa táto metóda zavolá nad všetkými synmi aktuálneho vrcholu. Potom sa testuje, či sa v aktuálnom vrchole nachádza operácia AND, resp. OR. Pokiaľ sa daná operácia nachádza v aktuálnom vrchole, zistia sa typy tokenov v synoch a pokiaľ sa pod operáciou AND nachádza operácia OR, resp. pokiaľ sa pod operáciou OR nachádza operácia AND, vykoná sa transformácia. Všetky neželané prípady a ich transformácie pre NDF je možné vidieť na obr. 2.20, pričom pre NKF je potrebné zameniť AND za OR a OR za AND.



Obr. 2.20 Neželané usporiadanie operácií AND a OR pri NDF a ich transformácia

MVLNode

Trieda **MVLNode** je potomkom abstraktnej triedy Node a predstavuje vrchol stromu reprezentujúceho viachodnotovú funkciu. Pre vytvorenie objektu tejto triedy sú k dispozícii 3 konštruktory, ktoré v sebe volajú príslušné konštruktory predka.

Z predka sú tu implementované všetky virtuálne metódy. Jedno z nich je metóda **clone**, ktorá vytvorí hlbokú kópiu vrcholu, pričom vytvorená kópia nemá definovaného predka. Ďalej sú to metódy **tryCompute**, **tryProceedListForSons** a **operationSimpify**, ktorých vykonávanie je podobné ako pri triede BooleanNode avšak s tým rozdielom, že miesto booleovských operácií, hodnôt a vrcholov sa tu používajú viachodnotové operácie, hodnoty a vrcholy. Taktiež aj metóda **transformToNF** pracuje na podobnom princípe ako

v triede BooleanNode, avšak tu sa miesto operácií AND a OR používajú viachodnotové operácie MIN a MAX.

Obr. 2.21 Diagram tried pre triedu FunctionRepresentation a jej potomkov

FunctionRepresentation

Táto abstraktná trieda je používaná na reprezentáciu zadanej funkcie, pričom sa tu implementuje algoritmus na vytvorenie viaccestného stromu a všetky základné operácie pre prácu funkčnou reprezentáciou. Na to, aby táto trieda mohla uchovávať funkciu sa tu nachádzajú atribúty pre uloženie textovej reprezentácie funkcie, smerníka na koreň stromu, ktorý je objektom triedy Node, zoznamu použitých algebraických reprezentácií, či zoznamu použitých premenných. Tieto atribúty sú prístupné v potomkoch tejto triedy a taktiež sú prístupné v potomkoch aj konštruktory, konkrétne kopírovací konštruktor, move konštruktor, implicitný konštruktor bez parametrov a 2 parametrické konštruktory,

pričom jednému konštruktoru je potrebné zadať objekt triedy Parser a druhému je potrebné zadať koreň, zoznam použitých algebraických reprezentácií a premenných.

V potomkoch sú prístupné taktiež aj metódy ako **copyVariablesFromFunction**, ktorá sa používa na kopírovanie zoznamu premenných zo zadanej funkcie, **removeVariable** na odstránenie premennej so zadanou reprezentáciou zo zoznamu premenných, **isVariableUsed** na potvrdenie, či sa vo funkcií nachádza premenná so zadanou reprezentáciou, **tryToGetOperationWithRepresentation** na získanie operácie so zadanou reprezentáciou, pokiaľ taká operácia existuje a tryToJoinAlgebraicSystems, ktorá do výstupného atribútu vloží všetky aktuálne používané algebraické reprezentácie a taktiež reprezentácie zadaných funkčných reprezentácií, pokiaľ sa už v zozname nenachádzajú.

Avšak je potrebné v potomkoch tejto triedy implementovať virtuálne metódy, konkrétne **tryProcessParserOutputStack**, ktorá reprezentuje algoritmus budovania viaccestného stromu, **clone** na získanie hlbokej kópie aktuálneho objektu, **tryCompute** na vypočítanie výstupnej hodnoty funkcie pre zadané hodnoty premenných, **tryGetElementForFunctionFromString** pre získanie číselnej hodnoty pre zadaný reťazec a **getFunctionType**, ktorá vráti typ funkcie.

Taktiež sa tu nachádzajú metódy, ktoré sú už implementované a je ich možné používať. Príkladom sú operátory priradenia a <<, metódy **getActualRepresentation**, ktorá nad koreňom zavolá metódu s rovnaký názvom, pokiaľ je koreň definovaný, **tryReplaceVariableWithValue** na nahradenie premennej zadanou číselnou hodnotou, pokiaľ sa vo funkcií nachádza premenná so zadanou reprezentáciou, **tryReplaceVariablesWithValues** ktorá nahradzuje premenné s zadanými reprezentáciami príslušnými zadanými číselnými hodnotami, pričom sa viacnásobne volá metóda tryReplaceVariableWithValue, **simplify,** ktorá sa snaží o zjednodušenie reprezentácie funkcie, **getAlgebraicSystems** pre získanie zoznamu algebraických reprezentácií, **getCopyOfUsedVariables** na získanie kópie zoznamu použitých premenných a **toString** používaná pre získanie textovej informácie o funkcií, pričom sa vypíše zadaný vstupný reťazec a testová reprezentácia viaccestného stromu.

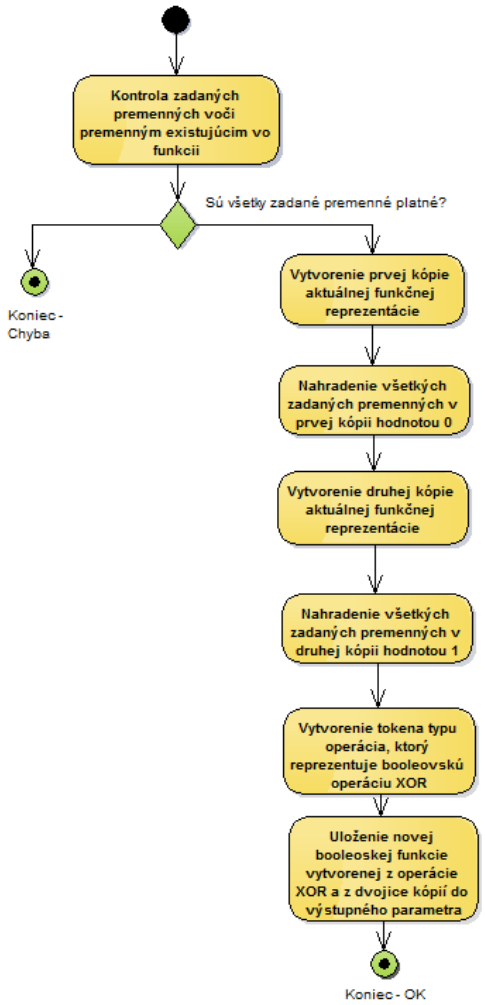
BooleanFunctionRepresentation

Trieda **BooleanFunctionRepresentation** je potomkom triedy Function Representation, ktorá reprezentuje booleovskú funkciu. Pre vytvorenie objektu tejto triedy je možné použiť jeden z 5 konštruktorov, pričom 4 z nich volajú príslušný konštruktor

predka a zvyšný parametrický konštruktor potrebuje pre vytvorenie nového objektu tejto triedy zoznam booleovských funkčných reprezentácií a objekt typu Token, ktorý bude použitý pri vytvorení koreňa a následne sa tomuto koreňu nastavia synovia, ktorými sú kópie koreňov všetkých booleovských funkčných reprezentácií zo zadaného zoznamu.

Z predka sú tu implementované všetky virtuálne metódy ako **clone**, **tryCompute**, pričom sa najskôr vykoná kontrola existencie všetkých zadaných reprezentácií premenných v booleovskej funkcií a následne sa zavolá nad koreňom metóda s rovnakým názvom, **tryGetElementForFunctionFromString**, ktorá prechádza všetkými definovanými algebraickými reprezentáciami pokým sa jej buď podarí nájsť hľadanú operáciu, alebo sa v žiadnej algebraickej reprezentácií nenachádza operácia so zadanou textovou reprezentáciou, **getFunctionType**, ktorá vráti informáciu o tom, že sa jedná o booleovskú funkciu a **tryProcessParserOutputStack**, ktorá najskôr skontroluje, či je zadaný zásobník prázdny. Pokiaľ nie, vytvorí koreň typu BooleanNode, vloží doňho prvý token zo zadaného zásobníka a spustí nad koreňom metódu tryProceedListForSons.

K metódam, ktoré sú tu definované patria metódy **prepareForNF**, ktorá nad koreňom zavolá metódu s rovnakým názvom, **transformToNF**, ktorá v sebe najskôr zavolá privátnu metódu prepareForNF a následne nad koreňom zavolá metódu s rovnakým menom, **tryDoSimpleDerivation**, ktorá reprezentuje booleovskú deriváciu, prípadne booleovskú deriváciu podľa vektora booleovských premenných a ktorej princíp je zobrazený na obr. 2.22, **tryDoMultipleSimpleDerivations**, ktorá reprezentuje viacnásobnú booleovskú deriváciu a pracuje tak, že viacnásobne zavolá metódu vykonávajúcu booleovskú deriváciu, **tryDoDirectPartialDerivation**, ktorá reprezentuje booleovskú smerovú deriváciu, resp. booleovskú smerovú deriváciu podľa vektora booleovských premenných a na rozdiel od booleovskej derivácie je tu potrebné použiť zadané hodnoty pre premenné a funkciu, taktiež ich znegovať, aby sa získali zmenené hodnoty premenných a funkcie v danom smere a miesto operácie XOR použiť operácie AND a NOT, či metóda **tryDoMultipleDirectPartialDerivations**, ktorá reprezentuje viacnásobnú booleovskú smerovú deriváciu a na vykonávanie svojej činnosti používa viacnásobne metódu tryDoDirectPartialDerivation.



Obr. 2.22 Activity diagram pre metódu tryDoSimpleDerivation

MVLFunctionRepresentation

Trieda **MVLFunctionRepresentation** je potomkom triedy FunctionRepresentation a reprezentuje viachodnotovú logickú funkciu. Na vytvorenie objektu tejto triedy je k dispozícií 5 konštruktorov, pričom takmer všetky v sebe volajú príslušný konštruktor predka. Jedinou výnimkou je parametrický konštruktor, ktorý potrebuje na vytvorenie objekt triedy Token a zoznam viachodnotových logických funkcií. Tento konštruktor potom vytvorí koreň, ktorý v sebe obsahuje zadaný token a následne mu nastaví nových synov, ktorými sú kópie koreňov logických funkcií zo zadaného zoznamu.

Keďže sa jedná o potomka triedy FunctionRepresentation, sú tu implementované všetky jeho abstraktné metódy ako **clone**, **tryGetElementForFunctionFromString**, **tryProcessParserOutputStack**, **getFunctionType** a **tryCompute**,ktoré pracujú podobne ako v triede BooleanFunctionRepresentation len s tým rozdielom, že sa miesto booleovských vrcholov, operácií a funkcií používajú ich viachodnotové ekvivalenty.

Taktiež sa tu nachádzajú ďalšie metódy, ktoré vykonávajú činnosti špecifické pre viachodnotovú funkciu. Metóda **transformToNF** slúži na transformáciu funkcie do tvaru SOP, resp. POS, pričom metóda pracuje tak, že sa nad koreňom zavolá metódu s rovnakým názvom. Ďalšou metódou je metóda **tryDoDirectPartialDerivation**, ktorá slúži na získanie smerovej derivácie viachodnotovej funkcie, pričom je potrebné zadať hodnotu funkcie pred zmenou hodnoty a po zmene hodnoty, reprezentácie platných premenných a ich hodnoty pred zmenou hodnoty a po zmene hodnoty. Následne sa vytvorí funkcia reprezentujúca smerovú deriváciu, pričom sa použijú kópie aktuálnej funkcie, v ktorých sa nahradia príslušné premenné hodnotami pre a po zmene hodnoty a operácie MIN a LIT. Poslednou metódou je **tryDoMultipleDirectPartialDerivations**, ktorá reprezentuje viacnásobnú smerovú deriváciu viachodnotovej funkcie a na vykonávanie svojej činnosti viacnásobne volá metódu tryDoDirectPartialDerivation.

Záver

„Celkový úspech je len nahromadením stoviek, ak nie tisícov pokusov a snáh, ktoré nikto nikdy neocení“

Brian Tracy

Booleovská a viachodnotová logika nájde svoje využitie v mnohých odvetviach. Vďaka týmto logikám je možné pracovať s výrokmi či určovať ich pravdivosť. V elektrotechnike je možné popisovať funkcionalitu obvodu logickou funkciou, robiť optimalizácie logickej funkcie, ktoré znamenajú aj optimalizovaný logický obvod, či skúmať vlastnosti logického obvodu a jeho častí. Taktiež v teórií hier sa používa booleovská a viachodnotová logika pri popisovaní jednoduchých hier a pri práci s týmito hrami alebo v teórií spoľahlivosti pri vyjadrovaní štrukturálnej funkcie systému, pri zisťovaní závislosti systému od určitého komponentu či pri zisťovaní ovplyvnenia hodnoty výstupu zo štrukturálnej funkcie systému pri zadanej zmene hodnoty komponentu systému. Preto je vhodné mať softvérový nástroj pre efektívnu prácu a manipuláciu s logickými funkciami v týchto logikách, ktorý by bol v prípade potreby rozšíriteľný o ďalšie operácie, prípadne algebraické systémy.

Softvérová knižnica, ktorá je popísaná v tejto práci, je vyvinutá presne pre tento účel. Je napísaná v jazyku C++ (štandard C++ 11), pretože tento jazyk je výkonný, časom preverený a poskytuje všetky potrebné prostriedky pre vývoj objektovo orientovanej softvérovej knižnice. Všetky údajové štruktúry používané v softvérovej knižnici boli vybrané na základe zabezpečenia najnižších zložitostí pri vykonávaní používaných operácií s nimi. V softvérovej knižnici sa využívajú nové prostriedky štandardu C++ 11 (move konštruktor, move operátor =, nullptr, ...), ktoré optimalizujú kód z pamäťovej či výpočtovej náročnosti. Taktiež bola knižnica otestovaná, aby sa v nej nenachádzali úniky pamäti (angl. „*memory leak*“).

Napriek tomu, že knižnica spĺňa všetky zadané požiadavky, jej vývoj tým nekončí. Ďalší smer vývoja softvérovej knižnice je v rozširovaní už existujúcich algebraických systémov a pridaní ďalších základných algebraických systémov ako celé čísla, desatinné čísla a pod. Taktiež je plánované, aby sa pri parsovaní dali používať aj funkcie, ktoré sa doprogramujú do softvérovej knižnice alebo sa použijú funkcie zadefinované pri behu programu používajúceho softvérovú knižnicu.

Bibliografia

[1] Louis Comtet, *Advanced Combinatorics: The Art of Finite and Infinite Expansions*. Dordrecht and Boston: D. Reidel Publishing Co., 1974.

[2] J. Michael Dunn and Gary M. Hardegree, *Algebraic Methods in Philosophical Logic*.: Oxford University Press US, 2001.

[3] Steven Skiena and Sriram Pemmaraju, *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*.: Cambridge University Press, 1990.

[4] Hilary A. Priestley and Brian A. Davey, *Introduction to Lattices and Order*.: Cambridge University Press, 2002.

[5] Rouben L. Goodsterin, *Boolean Algebra*.: Courier Dover Publications, 2012.

[6] Donald E. Knuth, *The Art of Computer Programming, Volume 4 Fascicle 0, Introduction to Combinatorial Algorithms and Boolean Functions*.: Addison-Wesley Professional, 2008.

[7] Yves Crama and Peter L. Hammer, *BOOLEAN FUNCTIONS - Theory, Algorithms, and Applications*.: Cambridge University Press, 2011.

[8] Svetlana N. Yanushkevich, D. Michael Miller, Vlad P. Shmerko, and Radomir S. Stanković, *Decision Diagram Techniques for Micro- and Nanoelectronic Design*.: Taylor & Francis Group, LCC, 2006.

[9] Siegfried Gottwald. (2015, Mar.) "Many-Valued Logic", The Stanford Encyclopedia of Philosophy. [Online]. https://plato.stanford.edu/archives/spr2015/entries/logic-manyvalued/

[10] D. Michael Miller and Mitchell A. Thornton, *Multiple Valued Logic: Concepts and Representations*.: Morgan & Claypool Publishers, 2008.

[11] Blackwell Companions to Philosophy, *A Companion to Philosophical Logic*, Dale I. Jacquette, Ed.: Blackwell Publishers Ltd, 2002.

[12] Dorothea Frede, *Oxford Studies in Ancient Philosophy*.: Oxford University Press, 1985.

[13] Peter Kaprálik. (2017, Mar.) Fakulta elektrotechniky a informatiky. [Online]. http://aladin.elf.stuba.sk/~kapralik/dsboolfunkcie.pdf

[14] Alexis De Vos, *Reversible Computing: Fundamentals, Quantum Computing, and Applications*.: John Wiley & Sons, 2011.

[15] Alfred Horn, "On sentences which are true of direct unions of algebras," *Journal of Symbolic Logic*, vol. I, no. 16, pp. 14–21, 1951.

[16] Proceedings of a Joint Workshop held in Prague, August 1992, *Discrete Event Systems: Modeling and Control*, Sivano Balemi, Petr Kozák, and Rein Smedinga, Eds.: Birkhäuser, 2012.

[17] Sos A. Agaian, Karen A. Panetta, Shahan C. Nercessian, and Ethan E. Danahy, "Boolean derivatives with application to edge detection for imaging systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. II, no. 40, pp. 371-382, Apríl 2010.

[18] Allain Alex, *Jumping into C++*.: Cprogramming.com, 2013.

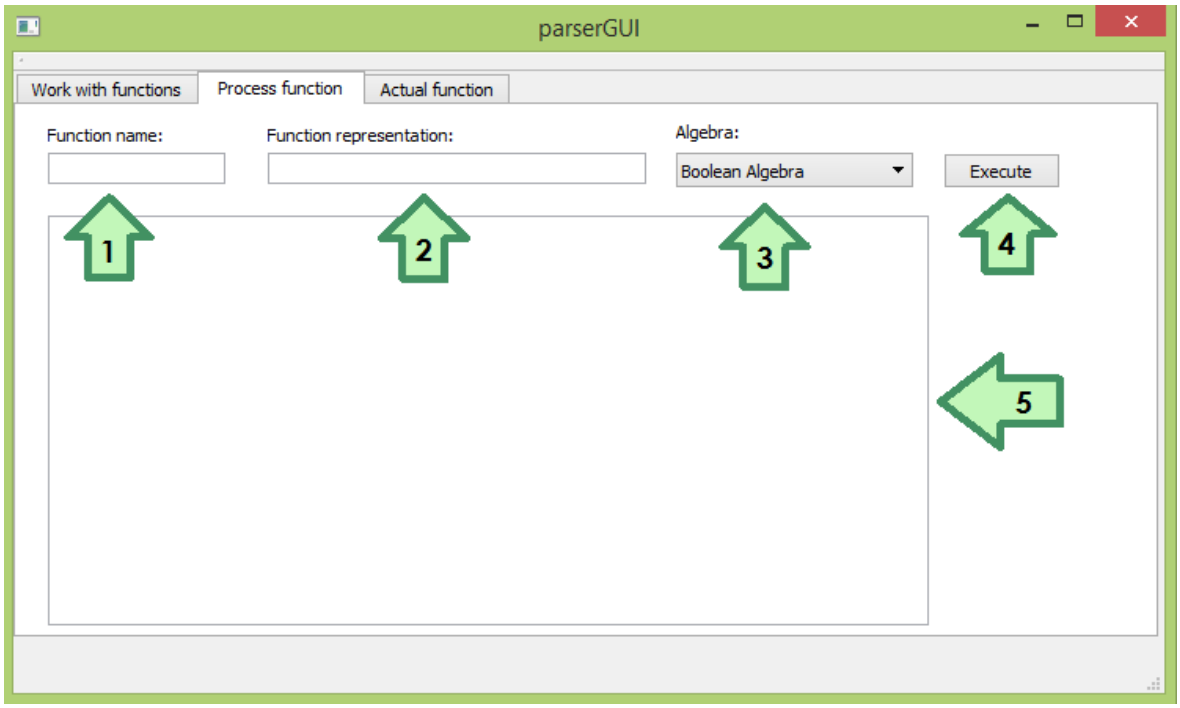
[19] Theodore Norvell. (2017, Apríl) Parsing Expressions by Recursive Descent. [Online]. http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm

[20] Bruno R. Preiss, *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*.: Wiley India Pvt. Limited, 2008.

[21] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman, *Compilers: Principles, Techniques, & Tools, Second Edition*.: Pearson Education, Inc., 2007.

Prílohy

A – Používateľská príručka pre testovacie GUI knižnice – karta Process function



1) Textové pole pre zadanie názvu funkcie, ktorý je potrebné zadať, napr. fun.

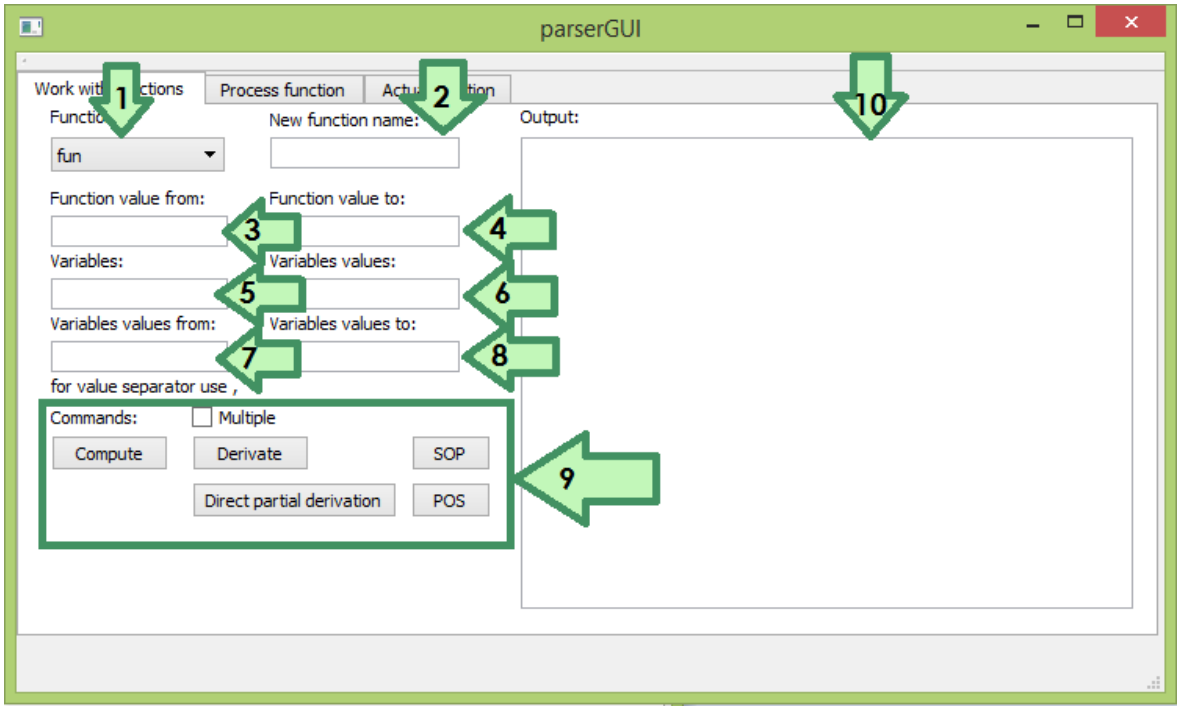
2) Textové pole, do ktorého je potrebné zadať textovú reprezentáciu funkcie, napr. x1 AND x2.

3) Combo box (rozbaľovateľný zoznam objektov) pre výber algebry, v ktorej je zadávaná funkcia definovaná.

4) Tlačidlo pre spustenie procesu spracovania zadaných vstupov do formy viaccestného stromu.

5) Textová plocha, na ktorú sa vypisuje výstup z parsovania a spracovania zadaných vstupov.

B – Používateľská príručka pre testovacie GUI knižnice – karta Work with functions



1) Combo box (rozbaľovateľný zoznam objektov) pre výber funkcie, s ktorou sa bude pracovať.

2) Textový pole pre zadanie mena výstupnej funkcie, pričom ak sa zadá meno výstupnej funkcie, tak sa táto funkcia uloží do zoznamu funkcií, inak sa výsledky operácií ako Derivate, Direct partial derivation, SOP a POS len vypíšu.

3) Textové pole pre zadanie začiatočnej funkčnej hodnoty pre potreby smerových derivácií.

4) Textové pole pre zadanie zmenenej funkčnej hodnoty pre potreby smerových derivácií, pričom túto hodnotu je potrebné zadať len pri viachodnotových smerových deriváciách.

5) Textové pole pre zadanie reprezentácií premenných, pre ktoré sa budú vykonávať operácie Compute, Derivate a Direct partial derivation.

6) Textové pole pre zadanie hodnôt premenných, ktoré sa použijú pri

vykonávaní operácie Compute.

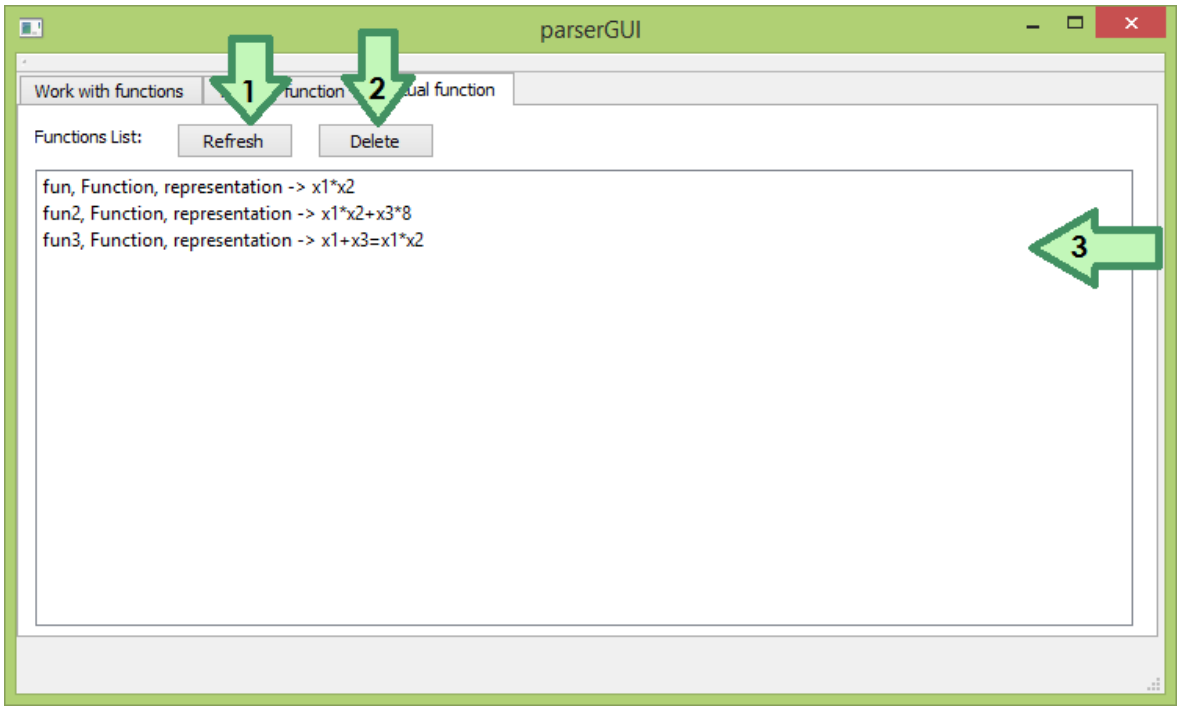
7) Textové pole pre zadanie začiatočných hodnôt premenných, pre ktoré sa bude vykonávať operácia Direct partial derivation.

8) Textové pole pre zadanie zmenenej hodnôt premenných, pričom tieto hodnoty je potrebné zadať len pri viachodnotových smerových deriváciách.

9) Oblasť, v ktorej sa spúšťajú jednotlivé operácie. Je tu možnosť vypočítať hodnotu funkcie pre zadané hodnoty všetkých premenných pomocou tlačidla Compute, možnosť vykonávať rôzne druhy derivácie booleovských funkcií pomocou tlačidla Derivate, pričom pre vykonanie viacnásobnej booleovskej derivácie je potrebné zaškrtnúť políčko Multiple, možnosť vykonávať rôzne druhy smerovej derivácie logických funkcií pomocou tlačidla Direct partial derivation, pričom pre vykonanie viacnásobnej smerovej derivácie je potrebné zaškrtnúť políčko Multiple a možnosť získať SOP, resp. POS formu booleovskej funkcie pomocou tlačidla SOP, resp. POS. Netreba zabudnúť na to, že pokiaľ chceme niektorú z transformácie funkcie uložiť, je potrebné zadať jej nové meno.

10) Textová plocha, na ktorú sa vypisuje výstup z vykonania zvolenej operácie.

C – Používateľská príručka pre testovacie GUI knižnice – karta Actual function



1) Tlačidlo pre obnovenie zoznamu aktuálne evidovaných logických funkcií.

2) Tlačidlo pre vymazanie vybranej logickej funkcie zo zoznamu aktuálne evidovaných logických funkcií, pričom sa po stlačení ešte zobrazí dialógové okno, v ktorom je potrebné potvrdiť vymazanie vybranej funkcie

3) Zoznam aktuálne evidovaných logických funkcií, v ktorom sa zobrazuje názov funkcie a jej aktuálna reprezentácia.

D – Priložené CD, na ktorom sa nachádza program a jeho zdrojové kódy