Abstract

Title:	Matrix typesetting in Adobe InDesign
Author:	Miroslav Král
Author's e-mail	miika@students.zcu.cz
Department:	Department of Computer Science and Engineering
Supervisor:	Ing. Petr Lobaz
Supervisor's e-mail:	lobaz@kiv.zcu.cz

Abstract:
This work is focused on matrix typesetting of Mr. Jeřábek's plugin for Adobe InDesign. There are described Software Development Kit of InDesign, launching of plugin and the main goal of this work - branch out the plugin into matrix tipesetting.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 16.5.2008 Miroslav Král

Obsah

Seznam obrázků

1 Úvod

Prioritím cílem mé bakalářské práce bylo rozšířit plugin pro program Adobe InDesign CS ing. Lukáše Jeřábka o sazbu matic a z toho plynoucí sazbu soustavy rovnic zarovnané na rovnítko. Uživatel musí mít možnost nastavení zarovnání každého sloupce doleva, doprava nebo na střed podle nejširší buňky ve sloupci matice. Dále Samotné realizaci předcházelo několik fází.

První fází bylo prozkoumání Software Development Kitu programu Adobe InDesign CS a nastudování příslušné dokumentace. Součástí této fáze byla také tvorba jednoduchého pluginu do programu InDesign. Další část se týkala samotného zprovoznění pluginu pana Jeřábka, pochopení funkčnosti samotného pluginu a jeho následná modifikace. Poznatky z této fáze se nachází v kapitole Těžké začátky.

Poslední částí byla samotná implemenatce sazby matic do pluginu. Této části je věnována největší část práce. Úkolem této části bylo také rozšíření stávající gramatiky. Na závěr jsou uvedeny výsledky implementace.

2 Těžké začátky

2.1 Porozumění SDK

Nedílnou součástí práce bylo proniknutí do SDK programu Adobe InDesign CS a nastudování dokumnetace. Její rozsah je opravdu e[illegible]ní a nebylo v mých silách vše přečíst a nastudovat. Panem Jeřábkem mi byly doporučeny tutorialy vydané přímo firmou Adobe, které velmi naučnou a přívětivou formou shrnují ty nejdůležitější a nejzákladnější poznatky nutné k vytváření funkčních pluginů pro program Adobe InDesign CS. Tyto tutoriály odkazovaly většinou přímo do nejdůležitějších částí dokumentace dodávané spolu s SDK.

K psaní pluginů je pro operační systém Windows doporučeno vývojové prostředí Microsoft Visual Studio ~~NET.~~ 2003. Důvody jsou popsány v následující části. Samotné vytvoření minimálního pluginu, který obsahuje jen formulář s talčítkem a žádnou funkcionalitou, je velmi pracná a časově náročná záležitost. V dokumentaci je pro tuto úlohu vyčleněno bezmála 33 stránek plných nastavení Visual Studia NET. 2003 a vytvoření základních souborů kódu a resource. Součástí SDK je program DollyXs, který tento zdlouhavý proces vykoná během pár chvil a je nutné zadat jen několik údajů jako jméno pluginu, jméno autora a identifikační číslo pluginu přidělené firmou Adobe. Program je napsán v jazyce Java.

SDK programu obsahuje také veledůležité ukázky již hotových řešení, takzvané Code Snippets. Je zde například ukázka jak vytvořit základní dialogové okno, menu, panel, tabulky a mnoho dalších. Jsou to projekty pro Visual Studio NET. 2003, ze kterých začínající programátor pluginů pro InDesign velmi rychle pochytí správné návyky a hlavně strukturu pluginu.

Součástí této přípravné části bylo vytvoření jednoduchého pluginu do InDesignu. Začal jsem postupovat podle doporučených tutoriálů od firmy Adobe. Ty v několika prvních částech odkazovali na příznačně nazvanou část dokumentace SDK „firstplugin“. Podle tohoto návodu nebyl problém pochopit základní konstrukce a mechanizmy fungování pluginů InDesignu a dobrat se k celkem uspokojivému výsledku. Největším problémem bylo psaní souborů s koncovkou *.fr a *.fh, ~~jejich~~ popis se nachází v následující části. Tyto soubory nejsou Visula Studiem podporovány a proto zde nebyla možnost automatického doplňování kódu. To při často krkolomných a dlouhých názvech komponent způsobovalo velké množství překlepů a chyb. Jinak si myslím, že tvorba pluginů pro InDesign není nikteratk složité a po určité praxi zjistíte, že máte přístup k většíně funkcionality kterou nabízí samotné jádro InDesingu a nemusíte si vše programovat od začátku sami. V tomto ohledu je firma

Adobe vůči programátorům velmi vstřícná.

Co již ale nepovažuji za rozumné, byla nemožnost běžíci plugin spustit v debugovacím módu Visula Studia. Jediná možnost se naskytla jen v případě, když plugin obsahoval tak závažnou chybu, která způsobila pád InDesignu a naskytla se možnost „Ladit“ přímo z operačního systému Windows. Poté si bylo možno prohlédnout část programu, která se vykonala před pádem a částečně tak odhalit zdroj chyby. Jinak všechno ladění probíhalo ve formě výpisu do mnou vytvořeného logovacího souboru. ~~To taky není zrovna pohodlné, ale lepší něž nic.~~ Při každé změně kódu a následném překladu bylo nutné ukončit InDesign, aby se plugin mohl aktualizovat a znovu InDesign spustit, což zabíralo velké množství času. Program nepatří zrovna k nejrychleji se spouštějícím a na mém pracovním nooteboku často start trval kolem dvou až tří minut.

2.2 Zprovoznění pluginu

První úskalí mé prace, které jsem musel podstoupit, bylo samotné zprovoznění projektu pluginu. V rámci předmětu PRJ5 jsem zpracoval důležitý návod, jak správně nainstalovat SDK programu Adobe InDesign CS, jak správně nastavit Visula studio a jak zprovoznit plugin samotný.

Myslím si, že jestli někdo bude v naší práci pokračovat, bude tento návod dobrým zdrojem informací a ušetří spoustu bezesných nocí. K dispozici jsem dostal jen soubory se zdrojovými kódy pluginu. Bohužel v této podobě se mi nepovedlo program ani přeložit. Od pana Jeřábka jsem následně získal projekt do Visual Studia NET. 2003. Projekt již měl nastaveny všechny cesty, které se lišily od ~~defaultních~~ ~~instalační~~ ~~cesty~~ SDK InDesignu. Proto bylo nutné přeinstalovat SDK do správného adresáře a nastavit Visual Studiu přístup k potřebným externím nástrojům. Poté již nebyl problém získat fungující plugin. Bohužel než se vše povedlo, zabralo toto řešení mnoho cenného času.

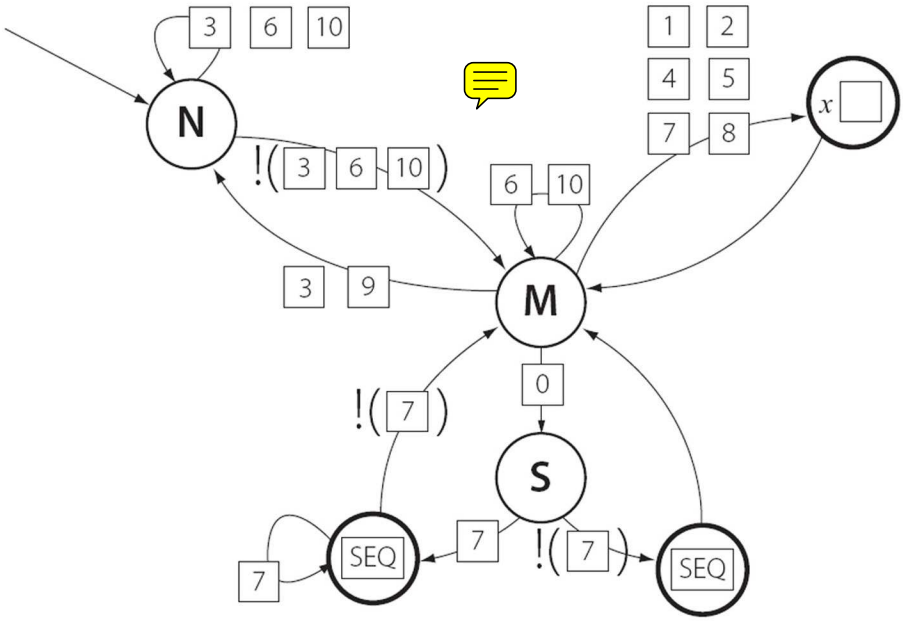
Program byl psán ve vývojovém prostředí Visual Studi NET. 2003 a jazyce C++. Nutnost programovat v tomto dnes již relaticně zas lém vývojovém prostředí byla z důvodu kompatibility s SDK programu Adobe InDesign CS verze 1, pro kterou byl plugin původně vytvořen. Pro korektní překlad a vytvoření souborů pluginu je nutný ~~platformě~~ nezávislý překladač ODFRC, který ve verzi pro SDK CS1 podporuje právě jen Visual Studi NET. 2003. ODFRC je nutný pro kompilaci souborů tvořící platformě nezávislé definice resource. Jsou napsány v jazyce ODFRez a typicky obsahují definice boss tříd, definice akcí, tabulky lokalizovaných řetězců, a dále pak definice panelů, dialogů, menu a ostatních grafických prvků. Tyto soubory mají zpravidla koncovku *.fr, nebo *.fh.

Plugin je rozdělen na dvě části a to část výkoného jádra a GUI. Výkonné jádro pluginu je samo o sobě již velmi rozsáhlé. Obsahuje kolem 6500 řádek v přibližně 60 třídách. Prezentační plugin obsahuje dalších téměř 2000 řádek kódu. Ze začátku vše vypadalo velmi náročně a nepochopitelně. Nad texty diplomové práce pana Jeřábka jsem strávil dlouhé večery než jsem pochopil správnou funkčnost programu. Dále jsem hodně času strávil ve zdrojových kódech programu, než jsem odhalil správný tok a chod programu. Další velmi užitečné inforamce jsem získal přímo při osobních setkáních s panem Jeřábkem. Při konzultacích s panem ing. Petrem Lobazem jsem hlavně získal poznatky ohledně teoretické části matematické sazby a práce s gramatikami.

3 Sazba matic

3.1 Analýza vstupní formule

Před samotným popisem sazby matic je nutné popsat práci samotného výkonného jádra pluginu, syntaktickou analýzu vstupu. Poté co uživatel vloží matematickou formuli, kterou chce vysázet do dokumnetu, a zmačkne tlačítko pro vysázení, načte jádro tuto formuli jako textový řetězec. Dvě nejduležitější komponenty pro analýzu jsou **TokenProcessor** a **SyntaxProcessor**. Funkcí TokenPorcessoru je parsovat na požádání vstupní formuli a získávat z ní důležité objekty typu **Token** pro SyntaxProcessor. TokenProcessor pracuje s jednotlivými řádky vstupní formule. Vždy když dojde na konec řádky, vyžádá si od InputProcessoru další řádku, pokud tedy na vstupu je další řádka. TokenProcessor pracuje na principu konečného automatu, který je na obrázku 1.



Obrázek 1: Konečný automat reprezentující funkčnost TokenProcessoru

Jednotlivá čísla reprezentují jednotlivé kategorie vstupních znaků popsaných níže. Popis jednotlivých stavů: N - nový řádek, M - uprostřed řádku, S - řídící sekvence, SEQ - vytvoření tokenu kategorie řídící sekvence, x - vytvoření tokenu kategorií 1, 2, 4, 5, 7 nebo 8.

TokenProcessor ze vstupního řetězce načítá znaky a dělí je do následujících kategorií:

1. **OpenGroup** – znak otevírací složené závorky `'{'`
2. **CloseGroup** – znak uzavírací složené závorky `'}'`
3. **EOL** – znak konce řádku `'\n'`
4. **SupIndex** – znak stříšky `'^'`
5. **SubIndex** – znak podtržítka `'_'`
6. **Space** – znaky `\t`; `\r`; ;
7. **Letter** – znaky s ASCII hodnotou od 65 do 90 a od 97 do 122
8. **OtherChar** – znaky které nelze zařadit do žádne ze zbylých kategorií.
9. **Comment** – znak komentáře `'%'`
10. **InvalidChar** – znaky `\f`; `\b`; `\a`

Znaky z kategorií 0, 1, 2, 4, 5, 7, 8 vytváří objekty typu Token, které dále putují do SyntaxProcessoru. TokenProcessor nevytvoří Tokeny najednou z celého vstupního řetězce, ale vždy vytvoří jeden. Když si SyntaxProcessor vyžádá další Token, pamět vyhrazená pro předchozí Token se uvolní a vytvoří se nový. Z tokenů jsou s pomocí gramatiky vytvářeny hlavní prvky matematické sazby – **Atomy**. Základní popis atomů a gramtiky je obsažen v následující části.

3.2 Změna gramatiky

Základní stavební jednotkou sazby pluginu je objekt typu **Atom**. Těch je dvanáct různých typů a všechny dědí od základní třídy Atom. Pro bližší popis jednotlivých atomů odkazuji na práci pana Jeřábka.

Dalším důležitým prvkem je objekt typu **MathList**. Ten obsahuje typovaný vector<Atom*>, do kterého se postupně jednotlivé atomy ukládají a je možno nad nimi provádět důležité operace ohledně výsledné metriky uložených atomů. Před započetím implementování řešení bylo nutné přidat do původní vstupní gramatiky nová slova pro matice.

Původní gramatika:

Slovo:	Blok | Procedura | Text | Exponent | Limita | Závorky | Odmocnina | Znak
Blok:	**1** Slovo **2** | Zlomek
Procedura:	**A** Blok2
Text:	**I** **1** text **2**
Exponent:	Index | Index Index
Limita:	**C** Index
Index:	InSup | InSub | InSup InSub | InSub InSup
InSup:	**4** Blok2
InSub:	**5** Blok2
Závorky:	**D** Lzávorka Slovo **E** Pzávorka
Lzávorka:	znaky (, [, | a ostatní definované v CSGlyph.txt jako leftpar | nic
Pzávorka:	znaky),], | a ostatní definovaní v CSGlyph.txt jako rightpar | nic
Odmocnina:	**F** Blok2 **G** Blok2
Zlomek:	**1** Slovo **B** Slovo **2**
Blok2:	Blok | Sekvence | Znak
Sekvence:	Porcedura | Závorky | Odmocnina | Text | **H** | **J**
Znak:	**7** | **8** | **H** | **J**
text:	libovolný text bez znaku konce řádku

Tučně podtržená čísla odpovídají jednotlivým kategoriím znaků, které zpracovává TokenProcessor. Tučně podtržená písmena jsou kategorie jednotlivých řídících sekvencí. Na úrovni TokenProcessoru nejsou jednotlivé kategorie rozpoznávány. Do SyntaxProcessoru je předán Token s obdaženou ~~řídící~~ sekvencí. Ten ve spolupráci s CSProcessorem už určí správnou kategorii řídící sekvence.

Ty jsou následující:

A) **Procedure** – sqrt, overbrace, underbrace, oversquare, undersquare, overbrace, underbracket, overline, underline, stroke

B) **Fraction** – over, atop

C) **Limits** – limits

D) **LeftPar** – left

E) **RightPar** – right

F) **Root** – root

G) **Of** – of

H) **Glyph** – zde jsou obsaženy speciální znaky definované v CSGlyph.txt. Jsou zde například obsaženy písmena řecké abecedy, speciální matematické znaky a mnohé další.

I) **Text** – text

J) **Function** – jde o uživatelsky definovatelné názvy funkcí. Ty jsou definovány v souboru Function.txt, zde je například sin nebo cos.

Nová gramatika:

Slovo:	Blok | Procedura | Text | Exponent | Limita | Závorky | Odmocnina | Znak | Matice
Blok:	**1** Slovo **2** | Zlomek
Matice:	**K** **1** znaky ~~lyc,r~~ **2** **1** Slovo $\overline{\text{\& Slovo}}$ **L** Slovo $\overline{\text{\& Slovo}}$ **2**
Procedura:	**A** Blok2
Text:	**I** **1** text **2**
Exponent:	Index | Index Index
Limita:	**C** Index
Index:	InSup | InSub | InSup InSub | InSub InSup
InSup:	**4** Blok2
InSub:	**5** Blok2
Závorky:	**D** Lzávorka Slovo **E** Pzávorka
Lzávorka:	znaky (, [, | a ostatní definované v CSGlyph.txt jako leftpar | nic
Pzávorka:	znaky),], | a ostatní definovaní v CSGlyph.txt jako rightpar | nic
Odmocnina:	**F** Blok2 **G** Blok2
Zlomek:	**1** Slovo **B** Slovo **2**
Blok2:	Blok | Sekvence | Znak
Sekvence:	Procedura | Závorky | Odmocnina | Text | **H** | **J**
Znak:	**7** | **8** | **H** | **J**
text:	libovolný text bez znaku konce řádku

K) **Matrix** – matrix

L) **Cr** – cr

Pro nadtržené výrazy platí, že se mohou ve výrazu vyskytovat 1 až n-krát. Zároveň byly přidány nové typy atomů:

- **Matrix** – atom slouží k sazbě matic, vytváří se po nalezení kontrolní sekvence **matrix**. Obsahuje jeden typovaný kontejner vector <Cell* >, do kterého se ukládají všechny postupně načtené buňky ze vstupního řetězce. Atom musí zajistit detekci své metriky, všech načtených buněk a následně určit jejich správnou pozici při sazbě.
- **Cell** – atom složí k uchovávání obsahu jedné buňky matice. Vytváří se při nalezení znaku pro oddělování sloupců, pokud mám matice více jak jeden sloupec, jinak se do buňky načte obsah těla matice. Atom

obsahuje jeden objekt typu MathList. Ten za něj vykoná většinu práce a zjistí metriku jednotlivých atomů buňky. Dále každá buňky obsahuje index řádku a sloupce, ve kterém se nachází.

Pro nově přidanou řídící sekvenci **cr** se atom nevytváří stejně jako například pro řídící sekvenci **of** u odmocniny nebo **over** a **atop** u zlomků.

3.3 Syntaxe sazby matic

Syntaxe je z větší části inspirovaná ~~sazebním~~ jazykem TEX. Ukázka základní syntaxe:

`\matrix` {zarovnávací znaky} {tělo matice}

Kde `\matrix` je kontrolní seqence sázené matice, **{zarovnávací znaky}** je blok, který obsahuje zarovnávací znaky jednotlivých sloupců matice a **{tělo matice}** obsahuje jednotlivé buňky a řádky matice, které budou ve finále zobrazeny.

Nedílnou součástí musela být možnost uživatelsky nastavitelného zarovnání jednotlivých sloupců. Pro zarovnání je vyhrazen samostatný blok do kterého se vkládají zarovnávací znaky. Mezi znaky i mezi znak a závorku je možno vkládat pro přehlednost mezery, které jsou při zpracovávání vstupní fotmule vynechávány. Zarovnávací blok se zapisuje do bloku začínajícího otevřenou složenou závorkou za kontrolní sekvenci matice `\matrix`. Blok zarovnání je ukončen uzavírací složenou závorkou.

Zarovnávací znaky jsou následující:

- **l** – znak pro zarovnání buňky na levý okraj podle nejširší buňky v daném sloupci matice.
- **c** – znak pro zarovnání buňky na střed podle nejširší buňky v daném sloupci matice.
- **r** – znak pro zarovnání buňky na pravý okraj podle nejširší buňky v daném sloupci matice.

Při zpracovávání matice se nejprve načtou zarovnávací znaky a počet načtených zarovnávacích znaků se následně kontroluje s počtem buňek v jednotlivých řádcích matice. Pokud se počet buňek v řádku neshoduje s počtem zarovnávacích znaků, je vyvolána chyba a oznámena uživateli. Uživateli je oznámeno, zda je počet buněk v řádku větší nebo menší než počet zarovnávacích znaků.

Příklad zápisu:

`\matrix` **{ lcr }** {tělo matice}

Tělo matice začíná složenou otevírací závorkou stejně jako blok se zarovnáním sloupců. Pro tělo matice jsou vyhrazeny dva speciální příkazy `&` a kontrolní sekvence `cr`.

Jejich Význam je následjící:

- **&** – tento speciální znak slouží k oddělení jednotlivých buněk v řádku matice. První oddělovač buněk následuje vždy až za první buňkou matice, jinak se umisťuje před každou buňku. Před první a za poslední prvek se neumisťujě.
- **cr** – kontrolní sekvence značí konec řádku matice. Zapisuje se za poslední buňku řádky. Tato sekvence se neumisťuje v posledním řádku matice.

Blok těla matice je ukončen uzavírací složenou závorkou. Ta následuje vždy za poslední buňkou posledního řádku matice. Nejduléžitější částí celého těla jsou buňky matice, které obsahují jednotlivé znaky a výrazy, které budou ve finále zobrazeny. Buňka řádku matice může obsahovat libovolná slova vstupní gramatiky.
Příklad zápisu jednořádkové matice:

```
\matrix { lcr } { buňka & buňka & buňka }
```

Příklad zápisu víceřádkové matice:

```
\matrix
{ lcr }
{ buňka & buňka & buňka \cr
  buňka & buňka & buňka \cr
  buňka & buňka & buňka }
```

3.4 Programová implementace sazby matic

Implementaci bylo potřeba rozdělit na několik částí. Bylo potřeba navrhnout nové typy atomů a správně překrýt metody, které dědí od rodičovské třídy Atom. Dále bylo nutné navrhnout metodu pro SyntaxProcessor, která se vykoná při nalezení kontorlní sekvence `matrix`. Tato metoda musí zajistit správné načtení zarovnání jednotlivých sloupců a všech buněk matice. Touto metodou bude popis zahájen.

3.4.1 Metoda csMatrix()

Prototyp metody **csMatrix** se nachází v hlavičkovém souboru SyntaxProcessor.h a její implementace v souboru SyntaxProcessor.cpp. První parametr metody je ukazatel na atom matrix typu CSMatrix, do kterého se budou postupně ukládat všechna důležitá data. První věcí, kterou je třeba vykonat, je načtení znaků zarovnání jednotlivých sloupců. Blok zarovnávacích znaků musí začínat otevírací závorkou. Pokud ne, je proces ukončen a uživateli je oznámeno chybové hlášení o špatně zadané syntaxi matice. Pokud je otevírací složená závorka v pořádku, může se pokračovat k načítání jednotlivých znaků. Metoda si ve smyčce žádá od TokenProcessoru Tokeny s jednotlivými znaky. Pro tyto tokeny není nutné vytvářet atomy jelikož nebudou již dále využity. Nám jde jen o získání ASCII hodnoty znaku, který je v příchozím Tokenu obsažen a to hodnoty 99 pro **c**, 108 pro **l** a 114 pro **r**. Dále je kontrolována hodnota 125 pro uzavírací složenou závorku, která ukončuje blok zarovnávacích znaků sloupců matice. Jednotlivé znaky zarovnání jsou ukládány do kontejneru vector, protože nemůžeme předem zjistit kolik znaků bude v bloku obsaženo. Pokud by se při načítání vyskytla v příchozím Tokenu jiná hodnota než výše uvedená, je uživateli oznámeno, které znaky jsou podporovány a na jaké pozici vstupní formule udělal konkrétně chybu. Informace o pozici chyby jsou získávány z právě zpracovávaného tokenu. Poslední věcí, kterou je nutno v této části udělat, je nastavit atomu matice počet sloupců podle počtu načtených zarovnávacích znaků. Tento počet je dále neměnný a bude dále použit k odchytávání chyb při načítání těla matice.

Další fází metody **csMatrix** je samotné načtení těla matice. Zde již dochází k vytváření atomů a to pro tyto účely vytvořeného atomu Cell, jehož implementace bude popsána dále. Tělo matice musí začínat a končit stejně jako blok pro znaky zarovnání sloupců a to složenými závorkami. Pokud se tak neděje, uživateli je zobrazeno patřičné chybové hlášení. Načtení celého těla probíhá v jednom cyklu while, který končí, pokud je na vstupu složená uzavírací závorka, která blok těla matice ukončí. Na začátku cyklu je nutné vyhradit paměť pro jeden objekt typu MathList a jeden objekt typu Cell. Objekt

typu MathList bude vždy tvořit tělo příslušné buňky Cell. Nyní je důležité toto tělo pro buňku získat. To obsatará jedna z nejdůležitějších metod **word**, které jako jeden z parametrů předáme MathList těla buňky. Pro bližší seznámení s touto metodou odkazuji na zdrojové kódy pluginu a diplomovou prácí pana Jeřábka. Důležité je, jakou hodnotu metoda vrátí. Pro načítání těla matice jsou důležité tři následující: oddělovač sloupců &, kontrolní sekvence `cr` značící konec řádku a uzavírací složená závorka značící konec těla matice. Pro všechny tyto návratové hodnoty se provádí téměř stejný kód, jsou zde ale důležité odlišnosti. První návratová hodnota – oddělovač sloupců – značí, že celé tělo buňky je správně načteno. Příslušné buňce je nastaven sloupcový a řádkový index, který bude využit při finálním vykreslování matice. Dále do buňky uložíme načtené tělo a celou buňku přidáme do spojového seznamu buněk atomu matrix. Nakonec je nutné zvýšit sloupcový index o jedna. Pro zbylé dvě návratové hodnoty – kontrolní sekvenci `cr` a uzavírací složenou závorku – je postup naprosto stejný. Obě hodnoty následují vždy po poslední buňce v řádku matice. Postup je podobný postupu pro hodnotu oddělovače sloupců. Rozdíl je v posledních řádcích kódu. V této části metody je první možnost odchytit chybu související se špatně zadaným počtem buněk řádku matice v porovnání s počtem načtených zarovnávacích znaků. Poté následujé kód který by bylo v případě chyby zbytečné vykonávat. Atomu matice je nastaven počet řádků odpovídající čítači řádků uvnitř metody. Počet řádků se tedy přenastavuje při každém nalezení kontrolní sekvence **cr** a není rovnou napevno nastaven jako počet sloupců po načtení zarovnávacích znaků. Poté je nutné vynulovat čítač sloupců a naopak zvýšit o jedna čítač řádků. Tento postup by již nebyl nutný pro koncovou uzavírací složenou závorku, ale zároveň mi přišlo zbytečné psát ten stejný kód jen s ubráním dvou řádků, které stejně po uložení poslední buňky nemají na další běh programu vliv. Pokud vše proběhne v pořádku bez chyb, vrátí celá metoda csMatrix nechybový návratový kód a atom matice je přidán do hlavního MathListu vstupní formule.

3.4.2 Nové atomy

Nyní se již podíváme na implementaci nově přidaných atomů **Matrix** a **Cell**. Všechny atomy dědí svoji rodičovskou třídu **Atom**. To přináší nutnost překrýt dvě důležité metody **compose()** a **detect()** sobě vlastním způsobem pro každý atom. Metody pomáhají s konečnou sazbou matematického výrazu. Poté, co jsou načteny všechny atomy ze vstupních řetězce a je postaven syntaktický strom, je nutné atomy převést na tzv. DrawingMethods, které již InDesign vykreslí. Převod atomů na tyto metody má dvě fáze, kde figurují ony dvě zděděné metody výše. V první fázi je nutné zjistit metrické údaje jednotlivých atomů pomocí metody **detect()**. Ta využívá jako svého pomocníka třídu StyleProvider, která obsahuje nejrůznější nastavení typu velikost a pozice exponentů, přesah zlomkové čáry, mezery uvnitř odmociny, typy potomků apod. Pokud byla metoda **detect()** zavolána na všechny atomy, následuje druhá fáze a to volání metody **compose()**. Z prvních fáze již známe metrické údaje všech atomů, můžeme tedy snadno určit pozice, na které se má každý atom vykreslit. Metoda pak následně volá metody třídy DrawingManager pro přidání tzv. DrawEngineMethods, která nedělá nic jineho než, že volají metodu třídy DrawEngine pro konečné vykreslení daného elementu. Třída DrawEngine je jediná třída, která přímo pracuje s InDesignem.

3.4.3 Atom Cell

Implementace atomu **Cell** se nachází v souborech Cell.h a Cell.cpp. Atom obsahuje jeden objekt typu MathList, který představuje tělo jedné buňky. Z toho plyne, že detekci jednotlivých atomů a jejich usazení zajistí přímo metody třídy MathList. Dále každý obejkt třídy Cell obsahuje index řádku a sloupce, pro následnou detekci ve třídě Matrix.

3.4.4 Atom CSMatrix

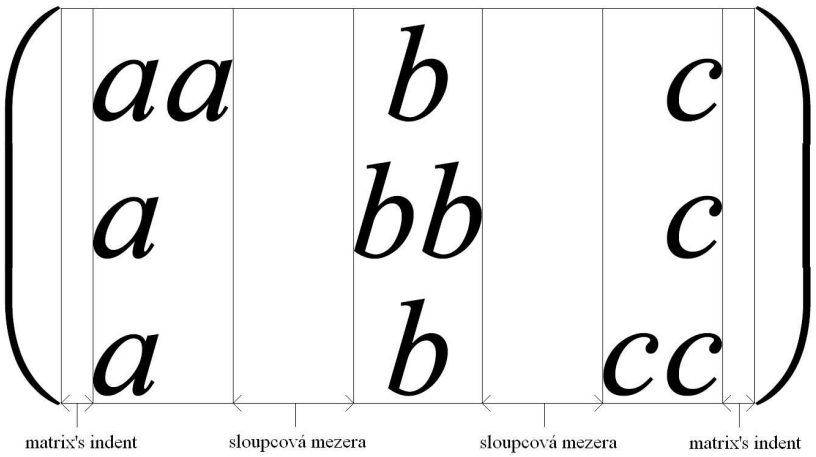
Implementace atomu **Matrix** se nachází v souborech CSMatrix.h a CSMatrix.cpp. Atom obsahuje typovaný kontejner vector<Cell* >, do kterého jsou ukládány všechny načtené buňky matice. Dále obsahuje kontejner pro načítání znaků zarovnání pro jednotlivé sloupce matice. Oba tyto kontejnery se naplní při načítání matice ze vstupní formule ve výše popisované metodě csMatrix(). Následuje metoda detect(). V této metodě je nutné zajistit detekci rozměrů jednotlivých buněk matice a také matice samotné. Ve for cyklu se prochází všechny načtené buňky. Pro každou buňku se volá její metoda detect(). Poté již u každé buňky známe jijí výšku a šířku. Tyto rozměry jsou důležité pro následovné zarovnávání buňek ve sloupcích matice a určení jejich pozice. Postup je následující: jelikož každá buňka má svůj řádkový a sloupcový index, tak vždy velmi jednoduše zjistíme, jestli se již přešlo na nový řádek matice. Do pole uložíme šířky buněk v prvním řádku. Pří zjišťování metriky buněk na dalším řádku porovnáváme nově zjištěné šířky buňek s již uloženými a pokud je nová šířka větší, nahradíme jí šířku na správném indexu reprezentujícím příslušný sloupec. Po projití všech buňek tedy získáme maximální šířky jednotlivých sloupců matice.

```
if (columnMaxWidth[columnIndex] < (*cell)->getWidth())
{
        width -= columnMaxWidth[columnIndex];

        columnMaxWidth[columnIndex] = (*cell)->getWidth();

        width += columnMaxWidth[columnIndex];
}
```

V ukázkovém kódu je proměnná **cell** iterátor, který prochází vector se všemi načtenými buňkami z metody csMatrix.

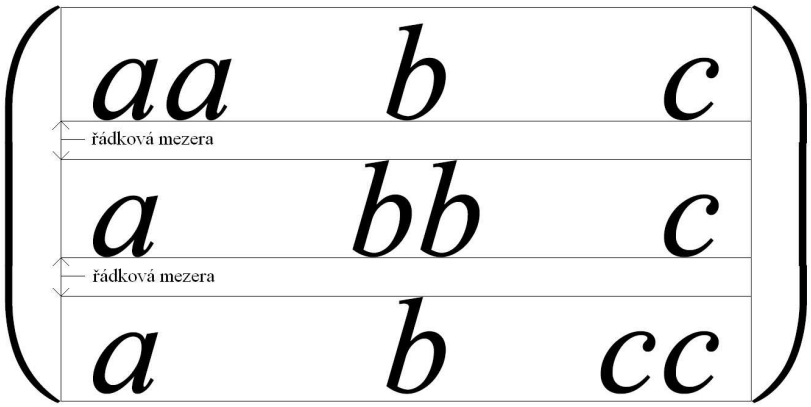
Podobný postup provedeme i pro zjištění maximálních výšek jednotlivých řádků matice. Na začátku vždy uložíme výšku první buňky a po detekci následující buňky v řádku porovnáme jejich výšky a pokud je nová výška větší, tak jí původní výšku nahradíme. Při zjišťování maximálních výšek řádků si budeme dále uchovávat hodnotu descentu nejvyšší buňky v každém řádku. Tu využijeme až v metodě **compose()** pro vertikální zarovnání buňek v řádku.

Nyní mame tedy zjištěny základní rozměry matice a je třeba je ještě upravit o uživatelsky nastavitelné hodnoty. Z výše získaných dat již není problém určit celkové rozměry matice.



Obrázek 2: Počítání šířky matice

K celkové šířce matice se musí dvakrát přičíst odsazení matice od závorek (matrix's indent) a také správný počet sloupcových mezer. Obě dvě tyto hodnoty jsou znázorněny na obrázku 2.



Obrázek 3: Počítání výšky matice

Celková výška matice se musí zvětšit o správný počet řádkových mezer, které jsou znázorněny na obrázku 3.

Nyní již známe finální hodnotu výšky a šířky matice a z nich je nutné určit ascen a descent matice. Tyto dvě hodnoty jsou nutné pro určení správné velikosti závorek kolem matice a také pro správné výškové zarovnání znaménka při operacích mezi maticemi a mezi maticemi a ostatními typy atomů. Po domluvě s panem Lobazem bylo vybráno zarovnání na střed výšky matice jak ukazuje následující obrázek 4. Zároveň jsou všechny hodnoty na obrázcích 2 a 3 uživatelsky nastavitelné v souboru StyleConfig.txt.

$$\sqrt{a+b}+\begin{pmatrix} aa & b & c \\ a & bb & c \\ a & b & cc \end{pmatrix}+\begin{vmatrix} a & b \\ c & d \\ e & f \\ g & h \end{vmatrix}$$

Obrázek 4: Vliv ascentu a descentu

Tímto krokem končí práce metody **detect()** a nastupuje metoda **compose()** atomu CSMatrix.

Metoda musí zajistit konečné pozice jednotlivých buňek pro vykreslení a zajistit také správné zarovnání každé buňky ve sloupci. Musíme opět projít všechny načtené buňky a zpracovat jednu po druhé. Při procházení buňek musíme hlídat řádkový index. Při přechodu na další řádek je nutné y-ovou souřadnici buňek v daném řádku zvětšit o proměnou **composeHieght**, ke které se přičítá maximální výška předchozího řádku a mezery mezi řádky, jak ukazuje následující kód.

```
if ((*cell)->getRowIndex() != rowIndex)
{
        composeHeight += (rowMaxHeight[rowIndex] +
                    spaceBetwenRows);
        rowIndex++;
        columnIndex = 0;
        composeWidth = 0;
}
```

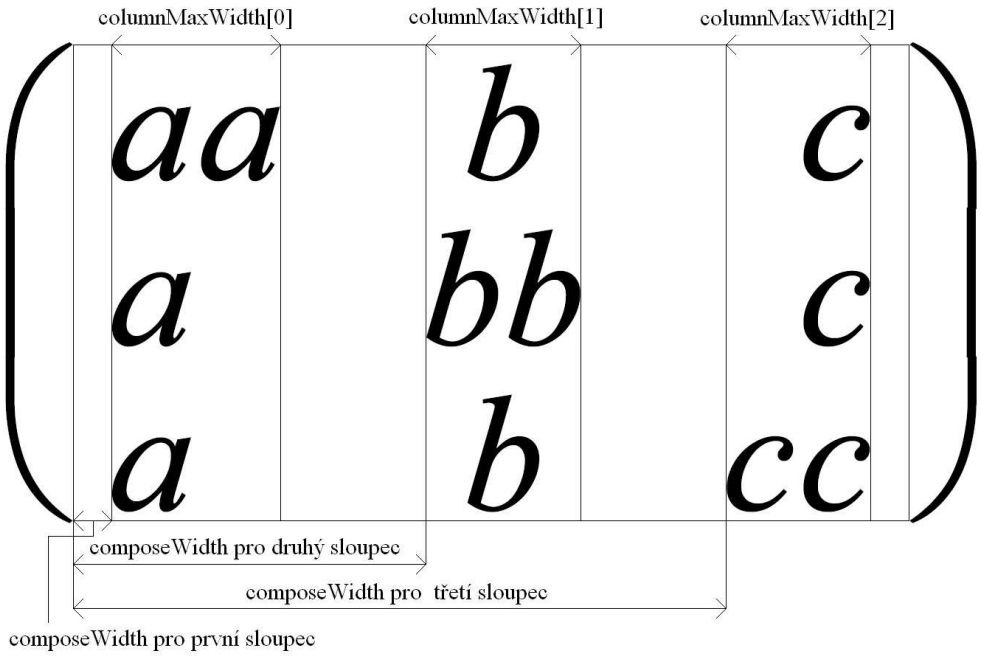
Podobně je nutné počítat proměnou **composeWidth**, která obsahuje hodnotu pro posun x-ové souřadnice následující buňku ve stejném řádku. K proměnné přičteme maximální šířku naposledy zpracovaného sloupce a velikost mezery mezi sloupci.

```
composeWidth += (columnMaxWidth[columnIndex] + spaceWidth);
```

Buňky v prvním sloupci se zpracovávají zvlášť, protože k jejich x-ové souřadnici je nutno připočíst ještě odsazení od závorky (matrix's indent).

```
composeWidth += (columnMaxWidth[columnIndex] + spaceWidth
                        + matrixsIndent);
```

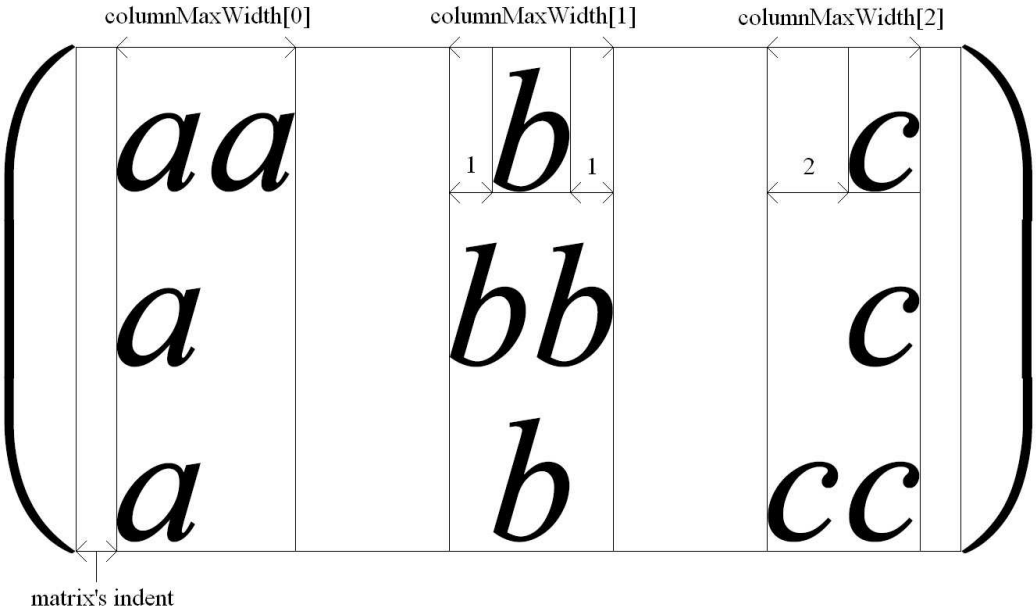
Obrázek ukazuje počítání proměnné composeWidth v jednotlivých krocích pro sloupce v jednom řádku.



Obrázek 5: Počítání proměnné composeWidth

Pokud nyní známe proměnnou composeWidth, můžeme začít se samotným zarovnáváním jednotlivých sloupců. Následující obrázek ukazuje hodnoty

označené čísly, které je nutné spočítat pro jednotlivé způsoby zarovnání. Pro zarovnání sloupců slouží proměnná **columnAlignment**. Tato hodnota se vždy přičte k x-ové souřadnici buňky.



Obrázek 6: Zarovnání sloupců matice

První slopec je zarovnán doleva. Hodnota columnAlignment je vždy zvětšena o hodnotu composeWidth, v tomto konkrétním případě je hodnota composeWidth rovna hodnotě odsazení(matrix's indent) matice od závorky.

```
columnAlignment = composeWidth;
```

Druhý slopec má zarovnání na střed. Pro buňku, jejíž šířka je menší než maximální šířka v daném sloupci se musí spočíst hodnota přídavného zarovnání 1. Výsledná hodnota proměnné columnAlignment je tedy následující:

```
columnAlignment = composeWidth + (columnMaxWidth[columnIndex]
                  - (*it)->getWidth())/2.0;
```

Pro poslední sloupec zarovnaný doprava je nutné připočítat rozdíl nejšířší buňky ve sloupci a šířky samotné buňky. Tato hodnota je na obrázku pod číslem 2.

Výsledná hodnota proměnné columnAlignment bude:

```
columnAlignment = composeWidth + (columnMaxWidth[columnIndex]
                  - (*cell)->getWidth());
```

Nyní jsme tedy získali všechny potřebné hodnoty pro horizontální zarovnání buněk v matici, ale je třeba také získat potřebné hodnoty pro vertikální zarovnání buňek. V úvahu je třeba brát také zarovnání podle nejvyšší buňky v rámci řádku, například správné zarovnání znaménka na zlomkovou čáru, jak ukazuje následující obrázek 7.

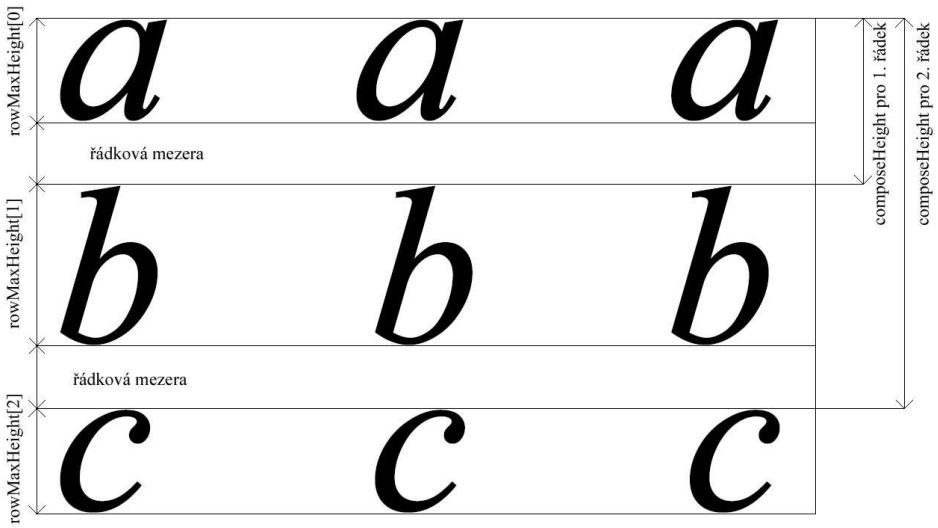
$$\begin{pmatrix} \frac{a}{c} & - & \frac{4}{d} \\ \frac{1}{b} & + & \frac{3}{2} \end{pmatrix}$$

Obrázek 7: Vertikální zarovnání buňek v řádku

K těmto účelům použijeme hodnoty descentů nejvyšších buněk v řádku, které jsme si ukládaly v metodě **detect()**. Každé buňce, která je nižší než nejvyšší buňka v řádku, přenastavíme descent na hodnotu právě té nejvyšší buňky jak ukazuje následující kód:

```
if ((*cell)->getHeight() < rowMaxHeight[rowIndex])
{
        (*cell)->setDescent(maxDescents[rowIndex]);
}
```

Nyní již stačí k y-ové souřadnici buněk přičítat hodnotu **composeHeight**, která byla popsána na začátku metody **compose()** v souvislosti s přechodem na následující řádek matice. Následující 8 ukazuje proměnnou composeHeight pro jednotlivé řádky.



Obrázek 8: Počítání proměnné composeHeight

Dále je nutné pro správné zarovnání odečíst descent buňky a jednu menší hodnotu pro lepší výsledek. Nyní již máme určené pozice všech buněk a můžeme matici poslat na vykreslení.

V rámci implementace se také povedlo opravit několik paměťových úniků původního pluginu. Není zaručeno, že jsou opraveny všechny, odstranil jsem jen ty, na které jsem narazil při testování.

4 Výsledky

V této kapitole uvedu výsledky sázených matic a soustav rovnic.

$$\begin{pmatrix} \frac{\sqrt{a}}{5+b} & a^3 & b^2 \\ 0 & \frac{a^4}{b} & a+b \end{pmatrix}$$

Obrázek 9: Matice

Jak je vidět, není problém do matice vložit již vytvořené atomy původního programu.

$$\begin{aligned} 3x^3 + 5x^2 - 10x &= 15 \\ \tfrac{6}{3}x^3 + 8x^2 + 2x &= 9 \\ 4x^3 - 5x^2 - x &= 45 \end{aligned}$$

Obrázek 10: Plná soustava rovnic

Program nyní tedy umožňuje sazbu soustav rovnic se zarovnáním na rovnítko bez vynechání prvků v řádcích nebo sloupcích.

$$\begin{aligned} 3x^3 & & - 10x &= 15 \\ \tfrac{6}{3}x^3 & & + 2x &= 9 \\ & - 5x^2 & &= 45 \end{aligned}$$

Obrázek 11: Soustava s chybějícími prvky

Lze samozřejmě sázet i soustavy rovnic s chybějícími prvky v řádcích nebo sloupcích a stále lze uchovat zarovnání na rovnítko.

$$\begin{aligned} 3x^3 - 10x &= 15 \\ \tfrac{6}{3}x^3 + 2x &= 9 \\ -5x^2 &= 45 \end{aligned}$$

Obrázek 12: Soustava rovnic bez zarovnání na rovnítko

Zde je ukázána i soustava s chybějícími prvky bez zarovnání na rovnítko.

5 Závěr

Použitá literatura

[1] KNUTH, D. E. *The ~~TextBook~~*
7. vydání, Massachusetts: Addison–Wesley, 1986,
ISBN 0–201–13447–0

[2] RYBIČKA J. *LaTeX pro začátečníky*
2. vydání, Brno: Konvoj, 1999, ISBN 80–85615–77–0

[3] KOPKA, H., DALY, P., W. *LaTeX – Podrobný průvodce*
Brno: Computer Press, 2004, ISBN 80–722–6973–9

[4] WICK, K. *Pravidla matematické sazby*
2. vydání, Praha: Academica, 1996, 21–144–66

[5] Adobe InDesign online developer training,[online]
http://www.adobe.com/devnet/indesign/training.html, říjen 2006

[6] LIBERTY, J. *Naučte se C++ za 21 dní*
1. vydání, Praha: Computer Press, 2002, ISBN 80–7226–774–4

[7] LIBERTY, J. *Naučte se C++ za 21 dní*
2. aktualizované vydání, Brno: Computer Press, 2007, ISBN 978–80–251–1583–1

[8] KOENIG, A., MOO, B. E. *Rozumíme C++*
Praha: Computer Press, 2003, ISBN 80-7226-656-X

A Uživatelský manual

Syntaktická pravidla

`\matrix` { zarovnání sloupců } { tělo matice }

Pro uživatele TEXu nebude problém pracovat s maticemi, syntaxe je téměř stejná. Pro všechny ostatní jsou zde ukázkové příklady.

Příklady

```
\left(
\matrix {lcr}
{
{\sqrt a \over 5+b} &a^3 &b^2\cr
0 &{a^4 \over b} &a+b
}
\right)
```

$$\left(\begin{matrix} \frac{\sqrt{a}}{5+b} & a^3 & b^2 \\ 0 & \frac{a^4}{b} & a+b \end{matrix}\right)$$

```
\matrix {lccccr}
{
3x^2 &+ &5x^2 &- &10x &= &15\cr
{6 \over 3}x^3 &+ &8x^2 &+ &2x &= &9\cr
4x^3 &- &5x^2 &- &x &= &45
}
```

$$\begin{matrix} 3x^3 & + & 5x^2 & - & 10x & = & 15 \\ \frac{6}{3}x^3 & + & 8x^2 & + & 2x & = & 9 \\ 4x^3 & - & 5x^2 & - & x & = & 45 \end{matrix}$$

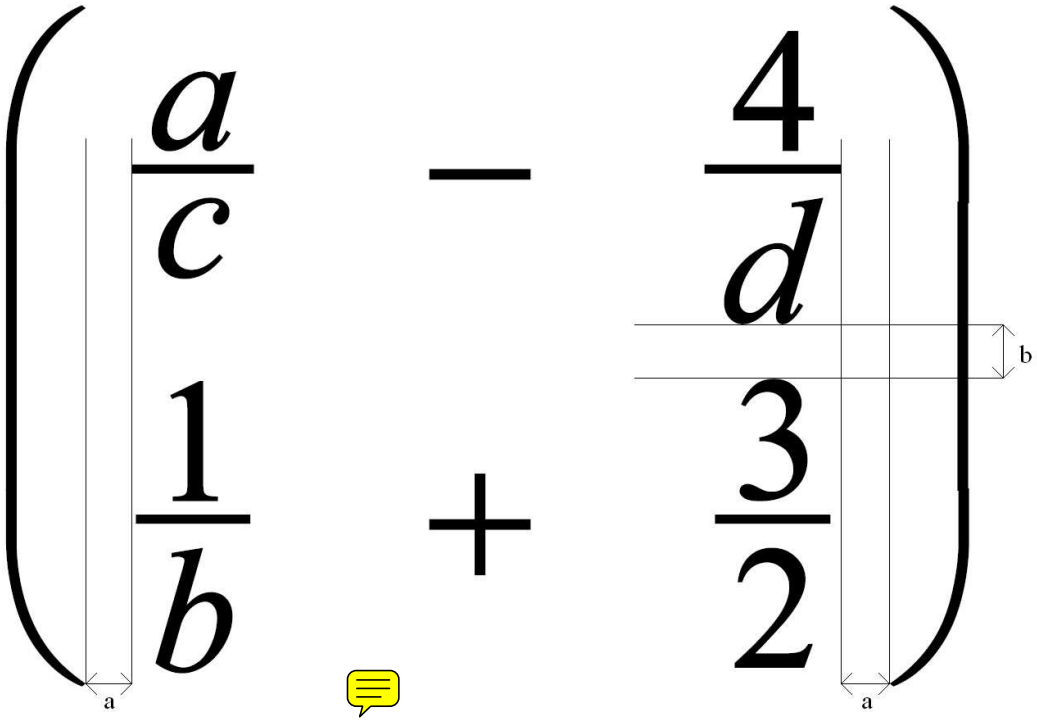
```
\matrix
{lccccr}
{
3x^3 & &- &10x &= &15\cr
{6 \over 3}x^3 & &+ &2x &= &9\cr
&-5x^2 &  & &= &45
}
```

$$\begin{matrix} 3x^3 & & - & 10x & = & 15 \\ \frac{6}{3}x^3 & & + & 2x & = & 9 \\ & - & 5x^2 & & = & 45 \end{matrix}$$

```
\matrix
{lccccr}
{
3x^3 &- &10x &= &15 &\cr
{6 \over 3}x^3 &+ &2x &= &9 &\cr
-5x^2 &= &45 & & &
}
```

$$\begin{matrix} 3x^3 & - & 10x & = & 15 \\ \frac{6}{3}x^3 & + & 2x & = & 9 \\ -5x^2 & = & 45 & & \end{matrix}$$

StyleConfig.txt



Hodnoty na obrázku jsou uživatelsky nastavitelné a jejich význam je následující:

- **a** – odsazení matice od závorky – indent
- **b** – mezera mezi řádky matice – spaceBettwenRows

Nový formát konfiguračního souboru *StyleConfig.txt*:
sprovider d d' t t' s s' ss ss'
sspace tiny thin medium thick cell
exponent a b c d e f g
limits a b
fraction a b
matrix a b
root a b c d e
parenthesis a b
Význam ostatních hodnot je popsán v uživatelském manuálu **UserMan.pdf** pana Jeřábka.

Space.txt

Do souboru pro nastavení velikostí mezer mezi jednotlivými atomy přibily dvě hodnoty pro matice a buňky matice.

22. matrix

23. cell

Popis a význam souboru s mezerami je popsán ve výše uvedeném manuálu.

B Obsah přiloženého CD