VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

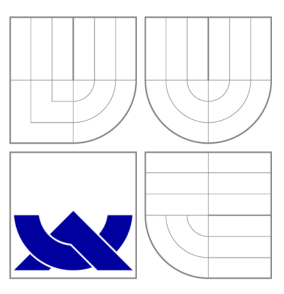
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

KONVERZE PREZENTACÍ MEZI PLATFORMOU LATEX A MICROSOFT POWER POINT

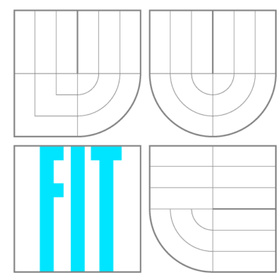
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE | Bc. LUKÁŠ ČERNÝ
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

KONVERZE PREZENTACÍ MEZI PLATFORMOU LATEX A MICROSOFT POWER POINT

LATEX AND MICROSOFT POWER POINT PRESENTATIONS CONVERTOR

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE / AUTHOR — Bc. LUKÁŠ ČERNÝ

VEDOUCÍ PRÁCE / SUPERVISOR — Ing. ZBYNĚK KŘIVKA, Ph.D.

BRNO 2011

Abstrakt

Tato práce se zabývá teoretickým a praktickým základem pro vytvoření převaděče mezi platformou LaTeX a Microsoft PowerPoint. Postupně rozebírá použití LaTeXu a třídy Beamer pro prezentace, programovou tvorbu PowerPoint dokumentů pomocí PowerPoint Primary Interop Assemblies a PresentationML. Nastiňuje použití generátorů lexikálního a syntaktického analyzátoru GPLEX a GPPG. Věnuje se vlastnostem dnešních nástrojů pro převod dokumentů. A nakonec se postupně věnuje jednotlivým úskalím návrhu, implementace a testování aplikace pro převod dokumentů mezi platformou LaTeX a Microsoft PowerPoint.

Abstract

This work deals with theoretical and practical basis for the creation of converter between LaTeX and Microsoft PowerPoint presentations. It discusses the use of LaTeX and the Beamer class for presentations, programmatic creation of PowerPoint documents using PowerPoint Primary Interop Assemblies and PresentationML. It outlines the use of the scanners and parsers generators GPLEX and GPPG respectively. It deals with the characteristics of today's tools for document conversion. And finally deals with the particular pitfalls of the design, implementation and testing of the application for conversion of documents between LaTeX platform and Microsoft PowerPoint.

Klíčová slova

LaTeX, Beamer, PowerPoint PIA, PresentationML, překladač, GPLEX, GPPG, převod dokumentů

Keywords

LaTeX, Beamer, PowerPoint PIA, PresentationML, convertor, GPLEX, GPPG, document conversion

Citace

Lukáš Černý: Konverze prezentací mezi platformou LaTeX a Microsoft Power Point, diplomová práce, Brno, FIT VUT v Brně, 2011

Konverze prezentací mezi platformou LaTeX a Microsoft Power Point

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Lukáš Černý
25. května 2011

Poděkování

Děkuji Ing. Zbyňku Křivkovi, Ph.D. za odborné vedení a návrhy pro vylepšení jak technické zprávy tak i samotné aplikace.

© Lukáš Černý, 2011.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 LaTeX a jeho rozšíření Beamer **5**

2.1 Preambule . . . 5
2.2 Zápis příkazů . . . 6
2.3 Základy smíšené sazby . . . 7
2.4 Výčtová prostředí . . . 7
2.5 Základy tvorby tabulek . . . 8
2.6 Tvorba prezentace s třídou Beamer . . . 10
2.6.1 Snímky prezentace . . . 10
2.6.2 Overlay aneb překrývání snímků . . . 10
2.7 Obrázky . . . 11
2.8 Shrnutí . . . 12

3 Programová tvorba PowerPoint dokumentů **13**

3.1 Automatizace pomocí PowerPoint PIA . . . 14
3.2 PresentationML - Open XML SDK . . . 18
3.3 Shrnutí . . . 21

4 GPLEX a GPPG **22**

4.1 The Gardens Point Scanner Generator (GPLEX) . . . 22
4.2 The Gardens Point Parser Generator (GPPG) . . . 23
4.3 Shrnutí . . . 24

5 Návrh **25**

5.1 Existující nástroje, jejich klady a zápory . . . 25
5.2 Stanovení požadavků na vytvářenou aplikaci . . . 26
5.3 Struktura aplikace . . . 27
5.4 Převod dokumentu z třídy Beamer na PowerPoint . . . 29
5.4.1 Lexikální analýza . . . 29
5.4.2 Generování abstraktního syntaktického stromu . . . 30
5.4.3 Převod . . . 31
5.5 Uživatelské rozhraní . . . 32
5.5.1 Hlavní dialog aplikace . . . 32
5.5.2 Nastavení parametrů převodu . . . 33
5.5.3 Rozhraní pro konzolový přístup . . . 35
5.6 Shrnutí . . . 36

6 Implementace **37**
6.1 Struktura aplikace a použité technologie . . . 37
6.2 Lexikální analýza a generování stromu dokumentu . . . 38
6.3 Převod . . . 39
6.3.1 Formátování textu . . . 39
6.3.2 Tabulky . . . 40
6.3.3 Výčtové typy . . . 41
6.3.4 Obrázky . . . 42
6.3.5 Overlay a `\pause` . . . 43
6.3.6 Přerovnání tvarů . . . 44
6.3.7 Řešení extrakce vnořených blokových prvků . . . 45
6.4 Uživatelské rozhraní . . . 46
6.5 Shrnutí . . . 48

7 Testování **49**
7.1 Omezení výsledné aplikace . . . 49
7.2 Distribuce aplikace . . . 51
7.3 Shrnutí . . . 51

8 Závěr **52**

A Obsah CD **56**

B Manuál **57**
B.1 Instalace . . . 57
B.1.1 Závislosti . . . 57
B.2 Ovládání aplikace . . . 57
B.2.1 Výběr pluginu . . . 58
B.2.2 Seznam souborů k převodu . . . 58
B.2.3 Výstupní adresář . . . 58
B.2.4 Spuštění, průběh a zastavení převodu . . . 59
B.2.5 Konzolové rozhraní . . . 59
B.3 Plugin pro převod z třídy Beamer na PowerPoint . . . 60
B.3.1 Nastavení parametrů převodu . . . 60
B.3.2 Konzolové rozhraní . . . 61

C Regulární výrazy použité pro lexikální analýzu **62**

D Bezkontextová gramatika pro zpracování dokumentu **66**

Kapitola 1

Úvod

V posledních letech stoupá počet informací a dat ukládaných člověkem. S tím jde ruku v ruce také nutnost řešit způsob uložení dat. Pokud chceme například uložit formátovaný textový dokument, tak se nám nabízí hned několik různých typů formátů pro uložení daného dokumentu. Těchto formátů existuje, dovolím si říct nespočet, některé z nich jsou otevřené a ty ostatní bohužel proprietární. Každý si většinou vybere jeden z formátů, který je mu příjemnější, ať už proto, že mu jeho oblíbený editor nabízí pouze jeden či proto, že mu zvolený formát přináší rozsáhlé typografické možnosti. Problém však nastává v okamžiku, kdy potřebujeme například kolegovi předat upravitelnou verzi svého dokumentu, ale kolega si s formátem, který používáme, nerozumí. Ve většině případů dokáží editory exportovat dokument i v jiném formátu a náš problém tedy odpadá, ale jsou i případy kdy to prostě nejde. V tuto chvíli nastupují na scénu překladače.

Překladače nám umožní doslova přeložit dokument z jednoho formátu do zcela jiného. Takovýchto překladačů je veliké množství, některé z nich se specializují pouze na převod mezi dvěma formáty, avšak jiné dokáží převádět celou paletu formátů. Dokonce již existují překladače, které není třeba stahovat ani instalovat, ale stačí pouze nahrát soubor přes internet a na email nám následně přijde jeho přeložená podoba. Jsou však i výjimky, tedy formáty, pro které překladače nejsou, či neumí vyprodukovat upravitelnou verzi překladu. V mnoha případech nemusí být potřeba, avšak najde se i několik takových kdy by se mohly hodit. Například v době psaní této práce není dostupný překladač mezi rozšířením platformy LaTeX pro prezentace (např. Beamer) a Microsoft PowerPoint. Může se zdát, že toto je zrovna jeden z příkladů, kdy překladač není potřeba, ale na Internetu lze narazit na dotazy různých lidí právě ohledně možnosti převodu mezi těmito dvěma formáty. Na jejich dotazy je nejčastější odpovědí věta typu "Ulož si to jako obrázky a potom z toho udělej prezentaci.". Pokud budeme postupovat podle této rady tak jistě získáme dokument, který bude vypadat stejně, ovšem naprosto ztratíme možnost jakékoliv editace, což rozhodně nevyvažuje zisk daný vzhledem.

V této práci se budeme zabývat celým procesem tvorby právě překladače pro převod mezi rozšířením Beamer platformy LaTeX pro prezentace a Microsoft PowerPoint. Postupně

projdeme jednotlivé fáze od seznámení se s LaTeXem, jeho rozšířením Beamer a Microsoft PowerPoint API, přes návrh, až po implementaci a konečně testování.

Druhá kapitola se zabývá syntaxí a použitím balíčku LaTeX a třídy pro prezentace Beamer. Postupně si vysvětlíme postup tvorby jednotlivých prvků dokumentu, jako jsou například výčtová prostředí a tabulky či změnu řezu písma. Vysvětlíme si pojem overlay a nastíníme si jak vytvořit jednoduchou prezentaci.

V třetí kapitole si nastíníme dva různé způsoby jak programově vytvořit prezentaci ve formátu používaném aplikací PowerPoint.

Ve čtvrté kapitole si představíme nástroje pro generování lexikálního a syntaktického analyzátoru, jejichž výstupem je kód v jazyce C#.

V páté kapitole si stručně rozebereme vlastnosti dnešních překladačů dokumentů a stanovíme si požadavky na nově vytvářený překladač. Následně si popíšeme jednotlivé části návrhu aplikace.

Šestá kapitola se věnuje samotné implementaci aplikace. Postupně si projdeme způsob realizace jednotlivých částí a vysvětlíme si, proč byl v určitých situacích použit zrovna daný přístup a ne jiný.

Sedmá kapitola poté stručně popisuje způsob testování výsledné aplikace. Také si zde zmíníme různá omezení výsledné aplikace, a proč k nim dochází.

Kapitola 2

LaTeX a jeho rozšíření Beamer

LaTeX je balíček maker, vytvořených Leslie Lamportem, usnadňujících tvorbu dokumentů pro sázecí program TeX. Umožňuje tak mnohem příjemnější a jednodušší tvorbu dokumentů, než v samotném TeXu. Beamer je potom rozšířením samotného LaTeXu pro tvorbu profesionálních prezentací.

Beamer vytvořil v roce 2003 Till Tantau pro prezentaci obhajoby své Ph.D. práce a o měsíc později jej na žádost svých kolegů uvolnil na Internet. A právě to umožnilo jeho rychlé rozšíření. Ovšem v roce 2007 Till Tantau přestal kód udržovat a vývoj třídy Beamer tak uvázl na mrtvém bodě. Situace se ovšem změnila, když se v dubnu roku 2010 rozhodl předat jeho údržbu někomu jinému. Nyní se o údržbu, opravu chyb, přidávání novinek a pomoc uživatelům starají Joseph Wright a Vedran Miletić. Aktuální kompletní kódy včetně kódů uživatelské příručky jsou dostupné ve veřejném mercurial repozitáři na adrese `http://bitbucket.org/rivanvx/beamer`.

Informace uvedené v této kapitole jsou čerpány z [4], [12] a [15].

2.1 Preambule

Preambulí nazýváme část zdrojového dokumentu uvedenou mezi příkazy `\documentclass` a `\begin{document}`. Jedná se o část, která slouží k nastavení různých vlastností výstupního dokumentu, vložení pomocných balíčků, popřípadě tvorbě vlastních příkazů. Pro nás nejzajímavějšími informacemi získanými z preambule jsou:

1. použitá znaková sada vstupního dokumentu (např UTF-8, ISO 8859-2, CP-1250)
2. téma (grafický styl) prezentace třídy Beamer
3. titulek prezentace, jméno autora a datum vytvoření

Použitou znakovou sadu lze zjistit z příkazu `\usepackage[`*znaková sada*`]{inputenc}`, kde například UTF-8 je zapsáno pomocí pomocí *utf8*, ISO 8859-2 pomocí *latin2* a CP-1250 pomocí *cp1250*.

Téma prezentace třídy Beamer získáme z příkazu `\usetheme{`*název tématu*`}`. Kde název tématu je některé z předem definovaných témat dodávaných s třídou Beamer.

Titulek prezentace, jméno autora a datum vytvoření získáme z parametrů příkazů `\title`, `\author` a `\date` v tomto pořadí.

2.2 Zápis příkazů

Jelikož se LaTeX snaží co nejvíc usnadnit práci uživateli, je možné u některých příkazů dosáhnout stejného efektu pomocí několika různých zápisů. Toto se může zprvu zdát spíš jako zesložitění. Avšak vzhledem k tomu, že dané zápisy dosáhnou stejného efektu, můžeme si v podstatě vybrat, který z nich nám více vyhovuje, či který z nich je vhodnější a umožní nám napsat přehlednější zdrojový kód. Těmito zápisy jsou klasický příkaz, skupina a prostředí.

Klasický příkaz se zapisuje pomocí kombinace zpětného lomítka a názvu – `\prikaz`, kterou následují případné složené, hranaté či jednoduché závorky, které určují jeho parametry. Všeobecně používaná konvence je, že ve složených závorkách jsou uváděny parametry povinné a v hranatých parametry, které mohou být vynechány (včetně samotných závorek). Pořadí jednotlivých parametrů, jejich povinnost či nepovinnost, stejně tak jako použitý typ závorek je dán definicí daného příkazu. Existují také příkazy, které nemají parametry žádné. V tomto případě je nutné, aby byl příkaz od okolního textu oddělen prázdným znakem (např. mezera, nový řádek). V případě, že použitým prázdným znakem je mezera, tak překladač všechny tyto odstraní a ty se potom neobjeví na výstupu.

Skupina se používá k označení bloku textu a používá znaky `{` a `}`. Důvodem k použití skupin je fakt, že některé z klasických příkazů ovlivňují veškerý text právě až do konce aktuální skupiny, či prostředí.

Prostředí je vyznačováno pomocí značek `\begin{nazev}` a `\end{nazev}`. Používá se podobně jako skupina k označení bloku, ale navíc určuje, že obsah tohoto prostředí bude zpracován jinak než jeho okolí (kdežto samotná skupina slouží pouze k určení hranic bloku). Podobně jako klasické příkazy, i prostředí může mít definované parametry (například prostředí pro tvorbu tabulek `tabular` má jeden povinný parametr[1]).

Za zmínku rozhodně stojí speciální příkaz pro uvedení komentáře v kódu, který nebude do výsledného dokumentu vysázen. Mimo možnosti použití prostředí `comment` pro komentáře můžeme použít řádkový komentář uvozený znakem `%`. Při použití tohoto zápisu je ignorován zbytek textu na daném řádku. Zároveň se následně ignorují všechny mezery a znaky na novém řádku, až dokud se nenarazí na nebílý znak, teprve tehdy se opět začne text normálně zpracovávat.

[1] více o tabukách v kapitole 2.5

2.3 Základy smíšené sazby

Smíšená sazba se používá v případě, kdy je potřebné nějakým způsobem zvýraznit určitou část textu. Obsahuje tedy písma s různým stupněm a řezem.

Stupeň písma určuje jeho velikost. LaTeX definuje sadu příkazů, která změní stupeň písma na určitou velikost, jejich přehled je v tabulce 2.1 (tabulka byla převzata z [12, str. 42]). Uvedené příkazy jsou právě tím dříve zmiňovaným případem, kdy příkaz ovlivní veškerý text až do konce aktuální skupiny či prostředí. V tabulce jsou uvedené i údaje o stupni písma, který daný příkaz nastaví. Ty však platí pouze pro dokument se základním stupněm písma 10pt, pokud je základní stupeň písma dokumentu jiný (11pt, 12pt), tak je velikost proporcionálně přepočítána.

Vzorek	Velikost	Příkaz	Vzorek	Velikost	Příkaz
ABCdef	5pt	`\tiny`	ABCdef	12pt	`\large`
ABCdef	7pt	`\scriptsize`	ABCdef	14,4pt	`\Large`
ABCdef	8pt	`\footnotesize`	ABCdef	17,28pt	`\LARGE`
ABCdef	9pt	`\small`	ABCdef	20,74pt	`\huge`
ABCdef	10pt	`\normalsize`	ABCdef	24,88pt	`\Huge`

Tabulka 2.1: Tabulka příkazů pro různé stupně písma

Řez písma se používá k vyznačování a rozumíme jím například kurzívu, strojopisné písmo, tučné písmo a další. LaTeX stejně jako v případě stupňů písma nabízí hned několik předdefinovaných řezů. Některé z nejčastěji používaných jsou vypsány v tabulce 2.2. V prvním sloupci je uveden klasický zápis pomocí příkazu, kde parametrem je ovlivněný text. Ve druhém sloupci je zápis pomocí varianty, kdy je ovlivněn veškerý následující text až do konce skupiny či prostředí.

Příkaz	Alternativní zápis	Vzorek
`\textbf{}`	`\bfseries`	**Polotučný řez**
`\texttt{}`	`\ttfamily`	`Strojopisné písmo`
`\textit{}`	`\itshape`	*Kurzíva*
`\textsc{}`	`\scshape`	KAPITÁLKY

Tabulka 2.2: Tabulka některých z nejčastěji používaných příkazů pro různé řezy písma

2.4 Výčtová prostředí

LaTeX nám mimo jiné nabízí 3 výčtová prostředí a to `itemize`, `enumerate` a `description`.

Prostředí `itemize` slouží k vytvoření nečíslovaných seznamů. Jednotlivé položky seznamu jsou přidávány pomocí příkazu `\item`. Jako příklad se můžeme podívat na následující seznam:

- První položka
- Druhá položka
- Třetí položka

Tento seznam je vytvořen pomocí následujícího kódu:

```
\begin{itemize}
   \item První položka
   \item Druhá položka
   \item Třetí položka
\end{itemize}
```

Pro vytvoření číslovaných seznamů slouží prostředí `enumerate`. Zápis je obdobný jako v případě nečíslovaného seznamu i zde jsou jednotlivé položky seznamu přidávány pomocí příkazu `\item`. Jediným rozdílem je, že výsledkem bude seznam číslovaný.

Posledním je prostředí `description`, které slouží k vytvoření seznamu definic. Syntaxe je obdobná s jedním rozdílem a tím je, že příkaz `\item` nyní přijímá jeden parametr v hranatých závorkách a tím je název termínu (titulek). Opět si uvedeme krátký příklad:

Termín Vysvětlení termínu

Další termín Vysvětlení dalšího termínu

Poslední termín Vysvětlení posledního termínu

Tento seznam definic je vytvořen pomocí následujícího kódu:

```
\begin{description}
   \item[Termín] Vysvětlení termínu
   \item[Další termín] Vysvětlení dalšího termínu
   \item[Poslední termín] Vysvětlení posledního termínu
\end{description}
```

Zajímavostí je, že seznamy definic je možné vytvořit například v HTML[2], ale moderní kancelářské WYSIWYG[3] editory seznamy definic většinou vytvářet neumí a je nutné je simulovat například pomocí dvousloupcové tabulky.

2.5 Základy tvorby tabulek

Tabulky jsou bezpochyby výborným způsobem pro přehledné zobrazení dat. Můžeme i říct, že existují případy, kdy jedna tabulka je mnohem názornější než stránka textu. LaTeX nám

[2]Hypertext Markup Language
[3]What You See Is What You Get

pro tvorbu tabulek nabízí prostředí `tabular`, které nám umožní vytvářet i komplikované tabulky. My se však zaměříme pouze na základní práci s prostředím `tabular` a nebudeme zabíhat do přílišných detailů.

Prostředí `tabular` má jeden povinný a jeden nepovinný parametr. Nepovinný parametr určuje vertikální zarovnání tabulky v textu, a pokud jej vynecháme, tabulka je automaticky zarovnána na střed. Povinný parametr určuje horizontální zarovnání v jednotlivých sloupcích a čáry mezi nimi. Pevně daným způsobem zápisu také v podstatě určuje počet sloupců tabulky. Obsahem tohoto parametru jsou tedy znaky určující zarovnání v sloupci oddělené pomocí znaku `|`, který určuje čáry mezi sloupci. Jednotlivé znaky pro nastavení zarovnání jsou:

l – zarovnání vlevo

c – zarovnání na střed

r – zarovnání vpravo

p – zarovnání do bloku, kdy jako parametr ve složených závorkách je uvedena jeho šířka

Jelikož tento způsob zápisu může být pro širší tabulky poněkud zdlouhavý, existuje možnost zkráceného zápisu pro opakující se sloupce ve formátu `*{kolik}{co}`.

Ukázali jsme si jak nastavit prostředí `tabular` a nyní se podíváme na vytvoření samotné tabulky. Jednotlivé sloupce tabulky se oddělují pomocí znaku `&`, řádky oproti tomu pomocí kombinace znaků `\\` nebo pomocí příkazu `\cr`. Důležité je, aby počet položek v každém řádku souhlasil s nadefinovaným počtem položek v hlavičce prostředí `tabular`. Teď pravděpodobně vyvstává otázka jak, když musí souhlasit počty sloupců, vytvořit buňku tabulky přes dva a více sloupců. Právě pro tyto situace existuje příkaz `\multicolumn`.

Příkaz `\multicolumn` přijímá celkem tři parametry. Prvním parametrem je počet sloupců, jež spojíme dohromady. Druhým parametrem je zarovnání ve vytvořeném spojeném sloupci. Tento se řídí v podstatě stejnými pravidly jako parametr prostředí `tabular`, který slouží k nastavení horizontálního zarovnání, s tím rozdílem, že nyní nastavujeme pouze jeden sloupec. Posledním parametrem je potom obsah nově vytvořeného spojeného sloupce.

V předchozím textu jsme si uvedli, že znak `|` v parametru prostředí `tabular` určuje vertikální čáru mezi sloupci. Avšak jsme si již neuvedli jak vytvořit horizontální čáru mezi řádky. Ta se vytváří pomocí příkazu `\hline`. Příkaz nemá žádné parametry a umisťuje se na několik možných míst. V případě, že chceme ohraničení vrchního okraje tabulky, tak musíme `\hline` umístit ihned za definici prostředí `tabular`. Pro ohraničení mezi jednotlivými řádky tabulky se umísťuje za oddělovač řádků (`\\`). A nakonec pro ohraničení spodního okraje tabulky musí následovat za oddělovačem řádků (`\\`) za posledním řádkem tabulky (tedy za posledním řádkem uvedeme `\\` a následně `\hline`). Mimo tento případ by se oddělovač řádků za posledním řádkem tabulky uvádět neměl. Může však nastat situace, kdy potřebujeme vytvořit horizontální čáru mezi řádky, avšak ne přes celou šířku tabulky. V tom případě použijeme místo `\hline` příkaz `\cline{`x-y`}`, kde x značí pořadové číslo prvního sloupce a y pořadové číslo posledního sloupce, přes něž má čára vést.

2.6 Tvorba prezentace s třídou Beamer

Aby jsme mohli vytvořit prezentaci za pomocí třídy Beamer, musíme uvést, že použitá třída dokumentu má být `beamer` pomocí příkazu `\documentclass{beamer}` na úplném začátku zdrojového dokumentu. Tímto zajistíme, že výstupem nebude například článek, ale opravdu prezentace.

Dále můžeme v preambuli uvést název prezentace pomocí příkazu `\title`, jméno autora pomocí příkazu `\author` a v neposlední řadě také datum vytvoření prezentace pomocí příkazu `\date`. Z těchto údajů můžeme následně nechat vygenerovat titulní stránku (více v kapitole 2.6.1).

2.6.1 Snímky prezentace

Základním kamenem každé prezentace, pokud pomineme obsah, je nepochybně možnost tvorby jednotlivých snímků. Právě tvorba jednotlivých snímků prezentace je při použití třídy Beamer opravdu jednoduchá. Nejprve si uveďme příklad a následně si popíšeme, co který příkaz znamená:

```
\begin{frame}[c]
   \frametitle{Titulek snímku}
   Obsah snímku
\end{frame}
```

Jak můžeme na první pohled vidět, tak celý snímek je obalený prostředím `frame`. Beamer nám místo prostředí `frame` umožňuje použít zápis pomocí příkazu `\frame`, ale vzhledem k přehlednosti se tento v praxi příliš často nepoužívá. Nepovinný parametr prostředí slouží k nastavení vertikálního zarovnání obsahu snímku. Možné hodnoty, kterých může nabývat, jsou `c` pro vystředění (výchozí hodnota pokud není parametr uveden), `t` pro zarovnání k vrchnímu okraji a `b` pro zarovnání k okraji spodnímu.

Příkaz `\frametitle` slouží, jak již jeho název napovídá, k nastavení titulku daného snímku. Obsahem snímku je následně text, který může obsahovat i případné příkazy maker LaTeXu.

Ve většině případů, kdy vytváříme prezentaci, chceme, aby měla nějakou titulní stránku. Jelikož se jedná o snímek, který je součástí většiny prezentací a LaTeX, potažmo Beamer se snaží uživateli co nejvíce usnadnit práci, vytvoří tento snímek za nás. Jediné co musíme udělat je vložit snímek, který bude obsahovat pouze příkaz `\titlepage` a veškeré údaje se vyplní z údajů uvedených v preambuli (viz 2.6)

2.6.2 Overlay aneb překrývání snímků

Často se při tvorbě prezentace dostaneme do situace, kdy by bylo vhodné jednotlivé části snímku odkrýt či zvýraznit postupně. Tato technika se v Beameru nazývá *overlay* (volně přeloženo jako překrývání snímků).

Nejjednodušším způsobem jak docílit překrývání snímků je použitím příkazu `\pause`. Pro větší názornost si uvedeme příklad:

```
\begin{frame}
    \frametitle{Postupné zobrazení}
    První část textu.
    \pause

    Druhá část textu, která se zobrazí od druhého snímku.
    \pause

    Třetí část textu, která se zobrazí od třetího snímku.
\end{frame}
```

Můžeme si představit, že při použití příkazu `\pause` překladač v podstatě snímek v jeho místě rozdělí na dva "podsnímky", na první umístí vše z daného snímku, co se vyskytlo před příkazem `\pause` a na druhý umístí nejen pouze to, co se vyskytlo za příkazem, nýbrž celý snímek.

Je tedy možné tímto postupem docílit postupného zobrazování informací na snímku. Avšak není možné určit, aby například pouze na druhém podsnímku byl text zobrazen jiným řezem než na podsnímku prvním a třetím. Pro tyto situace definuje Beamer doplňující parametr (anglicky nazývaný *overlay specification*) uvedený mezi závorkami `<` a `>`. Obsahem tohoto parametru je číslo vytvořeného podsnímku, na kterém se má daná položka zobrazit. Jelikož možnost uvedení pouze jednoho čísla by byla dost limitující, je možné uvést výčet. Výčet může být zapsán buď jako `<1,2,4,5>`, nebo stejně pomocí zkratek jako `<-2,4-5>`. Tento doplňující parametr je možné uvést u velkého množství příkazů LaTeXu. Některé z nich jsou například `\textbf{}`, `\item`, `\color{}` a spousta dalších. Seznam podporovaných příkazů a prostředí je možné nalézt v [15, kapitoly 9.3 a 9.4]

2.7 Obrázky

Jak jsme si již zmínili u tabulek, tak občas je názorná ukázka mnohem lepší než zdlouhavý popis. Například použití grafu či jiného obrázku je mnohdy velmi účelné. Jak vložit do dokumentu obrázek si naznačíme právě v této podkapitole.

Pro možnost vkládání obrázků do dokumentu musíme nejdříve nastavit v preambuli příkazem `\usepackage{graphicx}`, aby se při překladu použil balíček `graphicx`. Následně můžeme již vkládat obrázky pomocí příkazu `\includegraphics`. Tento příkaz má jeden povinný a jeden nepovinný parametr.

Povinný parametr slouží k určení názvu souboru s obrázkem, který se použije. Název je možné zadávat bez přípony a překladač se jej pokusí nalézt sám. Pro obrázek je možné použít několik různých typů, avšak toto je silně závislé na použitém překladači. Při použití

překladače `latex` můžeme použít pouze obrázky ve formátu EPS. Jestliže však použijeme překladač `pdflatex`, lze použít obrázky ve formátech JPEG, PNG nebo PDF, avšak EPS již nelze[4]. Soubor s obrázkem je vždy hledán v aktuální složce, ve které je umístěn zdrojový dokument. Toto chování je možné upravit v preambuli použitím příkazu `\graphicspath`, jehož parametrem je seznam cest, kde se budou obrázky hledat.

Nepovinný parametr oproti tomu nastavuje vlastnosti vkládaného obrázku. Jedná se zejména o šířku, výšku či měřítko obrázku. Parametr se zapisuje jako seznam atributů s přiřazenými hodnotami oddělenými čárkou (`[attr1=value,attr2=value,...]`). Pro účely této práce jsou podstatné zejména tyto atributy:

width – šířka vloženého obrázku
height – výška vloženého obrázku
scale – měřítko vloženého obrázku
page – pokud je obrázek ve vícestránkovém PDF, tak `page` určuje stranu

2.8 Shrnutí

V této kapitole jsme se seznámili se základy používání LaTeXu pro tvorbu dokumentů a třídy Beamer pro tvorbu prezentací. Postupně jsme si nastínili, jak fungují a jak se zapisují příkazy LaTeXu, podívali jsme se na základní formátování textu, na tvorbu seznamů a tabulek, práci s obrázky a v neposlední řadě vytvoření prezentace. Co je ale důležitější v rámci této práce je, že všechny popsané postupy a příkazy nyní budeme považovat za minimální nutnou funkčnost našeho vytvářeného překladače.

[4]Pro účely této práce budeme uvažovat použití překladače `pdflatex`

Kapitola 3

Programová tvorba PowerPoint dokumentů

Microsoft PowerPoint je jednou z nejčastěji používaných komerčních aplikací pro tvorbu a zobrazení prezentací. Hlavní výhodou pro jeho rozšíření oproti LaTeXu a Beameru je pravděpodobně[1] velice jednoduchá tvorba dokumentů pomocí WYSIWYG rozhraní.

Programová tvorba dokumentů (tedy tvorba dokumentu automaticky bez zásahu uživatele) je od uvedení Microsoft PowerPoint 2007 možná pomocí dvou různých způsobů.

Prvním způsobem je takzvaná automatizace pomocí Primary Interop Assembly[2], která funguje na principu přímého ovládání aplikace PowerPoint. Tedy nejdříve se spustí instance aplikace PowerPoint a následně se v ní vytvoří zvolený dokument. Výhodou je, že tímto způsobem lze vytvořit jak dokumenty ve starším formátu (*.ppt, pro verze 97 - 2003), tak i dokumenty ve formátu používaném od verze 2007 (*.pptx). Nevýhodou poté očividně zůstává nutnost mít nainstalován PowerPoint a jeho spouštění na pozadí při tvorbě dokumentu.

Druhým způsobem, který se nabízí. Je použití Open XML SDK a jeho část pro práci s prezentacemi PresentationML. Tento způsob přímo upravuje soubor s prezentací, a tedy nepotřebuje spouštět PowerPoint. Jeho zásadní nevýhodou je ovšem fakt, že tímto způsobem lze vytvářet dokumenty pouze od verze 2007 (*.pptx). Vzhledem k tomu, že Office Open XML byl koncem roku 2006 standardizován sdružením ECMA a později také ISO/IEC je práce s ním podstatně lépe zdokumentována než přístup pomocí automatizace.

Informace v této kapitole jsou čerpány z MSDN knihovny [7], [8], [10] a [9]. Další podstatné informace zejména týkající se PowerPoint PIA a jeho praktického použití z článku a připojené aplikace [1].

V následujících dvou podkapitolách (3.1 a 3.2) si nastíníme postup tvorby krátké prezentace oběma výše zmíněnými způsoby v jazyce C#. Tato prezentace bude sestávat z celkem čtyř snímků, první bude obsahovat titulek a podtitulek prezentace, druhý snímek bude

[1] pokud neuvažujeme např. obchodní model a jiné faktory

[2] dále jako PIA

obsahovat ukázku práce s odrážkami, třetí snímek obrázek a ukázku různých řezů písma a poslední snímek jednoduchou tabulku.

3.1 Automatizace pomocí PowerPoint PIA

Primary Interop Assemblies jsou jakýmsi pomyslným mostem mezi naším kódem a COM (Component Object Model) aplikace PowerPoint. První verze těchto knihoven byla vydána pro Office XP, tedy již celkem dávno. Jako u většiny produktů společnosti Microsoft si nové verze udržovaly zpětnou kompatibilitu s verzemi staršími. Ovšem toto se změnilo s příchodem Office 2007, kdy padlo rozhodnutí kompletně přepracovat objektový model. Možná jedním z hlavních důvodů bylo, že verze pro Office 2003 nebyla nejspíš plně odladěna a tak se při práci s ní objevovala spousta chyb [1].

Prvním důležitým krokem, který musíme provést jestliže chceme vytvořit aplikaci využívající PowerPoint PIA je, ve vlastnostech projektu, přidání referencí na knihovny `Office` a `Microsoft.Office.Interop.PowerPoint`. Tím se nám k použití zpřístupní dva jmenné prostory (`Microsoft.Office.Core` a `Microsoft.Office.Interop.PowerPoint`) obsahující třídy a metody pro práci s objektovým modelem aplikace PowerPoint.

Jak již jsme si zmínili výše, tak automatizace pomocí PowerPoint PIA je v podstatě proces, kdy programově spustíme a ovládáme instanci aplikace PowerPoint a používáme ji ke tvorbě vlastního dokumentu. Dalším krokem tedy bude spuštění nové instance aplikace PowerPoint (dále v této kapitole budeme pod termínem instance aplikace rozumět instanci aplikace PowerPoint), kterou použijeme k vytvoření samotné prezentace. Následující kód spustí minimalizovaný PowerPoint a vytvoří v něm novou prázdnou prezentaci.

```
// spustíme minimalizovaný PowerPoint
var application = new PowerPoint.Application();
application.ShowWindowsInTaskbar = MsoTriState.msoFalse;
application.Visible = MsoTriState.msoTrue;
application.WindowState = PowerPoint.PpWindowState.ppWindowMinimized;

// vytvoříme novou prezentaci
var presentation = application.Presentations.Add(MsoTriState.msoFalse);
```

V kódu si můžeme všimnout, že vlastnost `Visible` objektu `application` nastavujeme na `true`, tedy nás může napadnout, že nastavením této vlastnosti na `false` dosáhneme skrytí okna aplikace. Není tomu tak. Jestliže tak učiníme, dočkáme se pouze výjimky říkající “Invalid request. Hiding the application window is not allowed.”. Existuje však způsob jak toto chování obejít a použít PowerPoint bez toho, aby běžel na popředí (vytvořením objektu application pomocí metody `Activator.CreateInstance`).

Tímto jsme si spustili instanci aplikace a vytvořili prázdnou prezentaci. Než se však pustíme do tvorby jednotlivých snímku, tak si nejdříve nastíníme, jak je vlastně použitý objektový model postaven. Na obrázku 3.1 (obrázek převzat z [13]) jsou stručně vyobrazeny jednotlivé vrstvy objektového modelu pro práci s prezentací. Nejvýše položenou vrstvou

PowerPoint

- Application
  - Presentations
    - Presentation [1...n]
      - Slides
        - Slide [1...n]
          - Shapes
            - Shape [1...n]

Obrázek 3.1: Objektový model architektury PowerPoint PIA

je objekt `Application` reprezentující samotnou instanci aplikace. Za ním následuje objekt `Presentations`, který je v podstatě kolekcí otevřených prezentací neboli objektů `Presentation` (těch lze otevřít několik zároveň). S těmito objekty jsme se setkali již v dříve uvedené ukázce kódu. Nyní se však dostáváme k pro nás zajímavější části. Následuje kolekce `Slides`, která obsahuje objekty `Slide` jednotlivých snímků prezentace. Každý z těchto snímků poté obsahuje jednotlivé tvary, které v podstatě určují jeho obsah. Těmito tvary jsou mimo jiné například tabulka, textové pole, obrázky či grafy a každý z nich je reprezentován objektem `Shape`.

Nyní, když jsme si již nastínili základní strukturu objektového modelu, se můžeme pustit do tvorby jednotlivých snímků prezentace. Následující kód přidá do prezentace snímek s indexem 1 (index určuje pozici snímku v prezentaci) a při vytváření snímku použije rozložení s názvem “Úvodní snímek”. Zároveň vyplní nadpis a podnadpis a provede úpravu použitého řezu písma.

```
// vytvoříme první snímek
var slide = presentation.Slides.AddSlide(1, presentation.SlideMaster.↩
    CustomLayouts[1]);
var textFrame = slide.Shapes[1].TextFrame2;
textFrame.TextRange.Text = "Ukázková prezentace";
// můžeme provést změnu fontu
textFrame.TextRange.Font.Name = "Impact";
textFrame.TextRange.Font.Size = 48;
textFrame.TextRange.Font.Smallcaps = MsoTriState.msoTrue;
textFrame = slide.Shapes[2].TextFrame2;
textFrame.TextRange.Text = "Pomocí PowerPoint PIA";
```

```
textFrame.TextRange.Font.Smallcaps = MsoTriState.msoTrue;
```

Kolekce `CustomLayouts` obsahuje jednotlivá rozložení, která můžeme použít v prezentaci. Tyto rozložení se, v případě že nezměníme téma prezentace, aplikují z výchozího motivu prezentace s názvem "Motiv sady Office". Nejspíše by bylo vhodné poznamenat, že indexy v kolekcích jsou většinou číslovány od jedné (na rozdíl například od jazyka C).

Vzhledem k tomu, že jsme při vytváření snímku určili použití rozložení s názvem "Úvodní snímek", tak snímek již obsahuje několik tímto rozložením předdefinovaných tvarů. Jelikož tyto tvary (`Shapes`) reprezentují textová pole, můžeme použít vlastnost `TextRange`, která nám vrátí objekt reprezentující obsah tohoto textového pole. Následně již není problém změnit text, který obsahuje (vlastnost `Text`) nebo upravit parametry použitého fontu (vlastnost `Font`) pro celé pole.

Druhý snímek bude obsahovat jednoduchou ukázku práce s odrážkami. Pro vytvoření tohoto snímku použijeme stejný postup jako u předchozího, pouze s tím rozdílem, že pro nový snímek použijeme v definici rozložení (kolekce `CustomLayouts`) index 2. To nám zajistí, že se vytvoří snímek s klasickým rozložením ("Nadpis a obsah") tedy tvarem pro titulek a tvarem pro obsah. Výchozí nastavení tohoto rozložení navíc říká, že každý odstavec v tvaru s obsahem bude uvozen nečíslovanou odrážkou. To nám práci značně usnadňuje. Ovšem nyní vyvstává otázka, co je myšleno odstavcem v obsahu textového pole. Za odstavce jsou považovány části textu oddělené znakem `CR` (návrat vozíku). Již tedy víme, že jeden odstavec reprezentuje jednu odrážku, ovšem všechny takto vytvořené odrážky jsou pouze na první úrovni a navíc jsou všechny nečíslované. Následující kód demonstruje změnu zanoření odrážek a přenastavení na číslování.

```
// nastavíme pro třetí odstavec úroveň odsazení 2 a typ odrážek na číslování
textFrame.TextRange.Paragraphs[3, 1].ParagraphFormat.IndentLevel = 2;
textFrame.TextRange.Paragraphs[3, 1].ParagraphFormat.Bullet.Type = ↩
    MsoBulletType.msoBulletNumbered;
```

Pomocí vlastnosti `Paragraphs` můžeme zvolit, na jaké odstavce se budou změny aplikovat. Jak je vidět vlastnosti se předávají dva indexy, prvním je číslo odstavce, od kterého se budou nastavené atributy aplikovat, a druhým je potom počet odstavců, na které se budou aplikovat. Následně pomocí vlastnosti `ParagraphFormat` už můžeme nastavit požadované atributy (v tomto případě větší úroveň odsazení a typ odrážek na číslování).

Třetí snímek bude obsahovat obrázek a ukázku různých řezů písma. Opět budeme postupovat obdobně jako v případě prvního a druhého snímku, avšak nyní zvolíme rozložení s indexem 4 s názvem "Dva obsahy". Jak již název napovídá, jedná se o rozložení, kde je titulek a dvě textová pole pro obsah. Co provedeme je, že nahradíme levé textové pole obrázkem. Následující kód vloží do prezentace obrázek na specifikovanou pozici a s danou velikostí. Zároveň nahradí první tvar (mimo titulku), v našem případě tedy textové pole pro levý obsah.

```
// vložíme obrázek do prezentace
```

```
slide.Shapes.AddPicture(picture, MsoTriState.msoFalse, MsoTriState.msoTrue, ↩
    30, 150, 300, 300);
```

Potom co do snímku vložíme obrázek, tak nám již zbývá jen vložit do textového pole pro druhý obsah formátovaný text. Změnu řezu písma jsme si již ukázali (u prvního snímku), avšak pouze pro celé textové pole. Jestliže chceme změnit řez pouze pro určitou část textu, je třeba použít mírně odlišný způsob. Nejdříve si opět uveďme názorný příklad, který si následně vysvětlíme.

```
// nastavíme kurzívu třetímu slovu
textFrame.TextRange.Words[3, 1].Font.Italic = MsoTriState.msoTrue;

// nastavíme velikost písma na 18 od 30tého znaku po 6 znaků
textFrame.TextRange.Characters[30, 6].Font.Size = 18;
```

Co se provede, si můžeme představit jako to, že v podstatě instanci aplikace řekneme (pro první případ) "Označ třetí slovo a aplikuj na něj atributy". Těchto vlastností pro výběr části textu je dostupných hned několik, patří mezi ně například i `Paragraphs` pro výběr odstavců, `Lines` pro výběr řádků a `Sentences` pro výběr vět. Tímto jsme si pokryli všechny důležité informace a postupy pro vytvoření třetího snímku.

Poslední snímek bude obsahovat titulek a jednoduchou tabulku, tedy stejně jako u druhého snímku zvolíme rozložení s názvem "Nadpis a obsah". Co provedeme je, že podobně jako v případě obrázku nahradíme textové pole tabulkou. Opět začněme praktickou ukázkou.

```
// vytvoříme tabulku 4x4
slide.Shapes.AddTable(4, 4);
var table = slide.Shapes[2].Table;

// předvyplníme si celou tabulku pro ilustraci nějakými daty
for (int row = 1; row <= 4; row++)
    for (int cell = 1; cell <= 4; cell++) {
        table.Rows[row].Cells[cell].Shape.TextFrame2.TextRange.Text = "buňka" ↩
            + cell + ", řádek " + row;

// spojíme dvě buňky nad sebou a změníme obsah této spojené buňky
table.Cell(2, 2).Merge(table.Cell(3, 2));
table.Cell(2, 2).Shape.TextFrame2.TextRange.Text = "Spojené buňky";
```

Z ukázkového kódu je dobře vidět, že práce s tabulkami je opravdu velice jednoduchá. Nejprve si vytvoříme pomocí metody `AddTable` tabulku o rozměrech 4×4, získáme objekt tuto tabulku reprezentující a následně s ní již můžeme přímo manipulovat. Dále si můžeme povšimnout, že tabulka obsahuje kolekci řádků a každý z nich poté obsahuje kolekci buněk. Tento přístup se hodí zejména pro zpracování v cyklu. Avšak pro přímý přístup k buňce lze použít i další způsob a tím je metoda `Cell` objektu `table` reprezentujícího tabulku. Důležitou informací může být i to, že každá buňka obsahuje právě jeden textový tvar. Tedy buňka nemůže obsahovat nic jiného než formátovaný text (například obrázky). Jednotlivé buňky je možné spojovat. K tomu slouží metoda `Merge` objektu reprezentujícího buňku, jejím parametrem je objekt reprezentující jinou buňku. Tato metoda poté spojí všechny

buňky umístěné v obdélníku, jehož úhlopříčka je dána právě těmito dvěma buňkami (tedy buňkou, na jejímž objektu jsme volali metodu `Merge`, a buňkou, jejíž objekt jsme této metodě předali jako parametr).

Posledním krokem, který provedeme, je uložení nově vytvořené prezentace, její uzavření v instanci aplikace a nakonec ukončení samotné aplikace. Všechny tyto tři kroky jsou v podstatě triviální.

```
// uložíme výstup prezentace
presentation.SaveAs(outputFile);
// uzavřeme prezentaci
presentation.Close();
// ukončíme PowerPoint
application.Quit();
```

Co v předchozím příkladu není uvedeno, je že metoda `SaveAs` přijímá volitelný druhý parametr, kterým je formát ukládané prezentace. Mezi formáty dostupnými pro uložení jsou mimo klasických (.ppt, .pptx) také například možnost uložení jako obrázků či HTML.

3.2 PresentationML - Open XML SDK

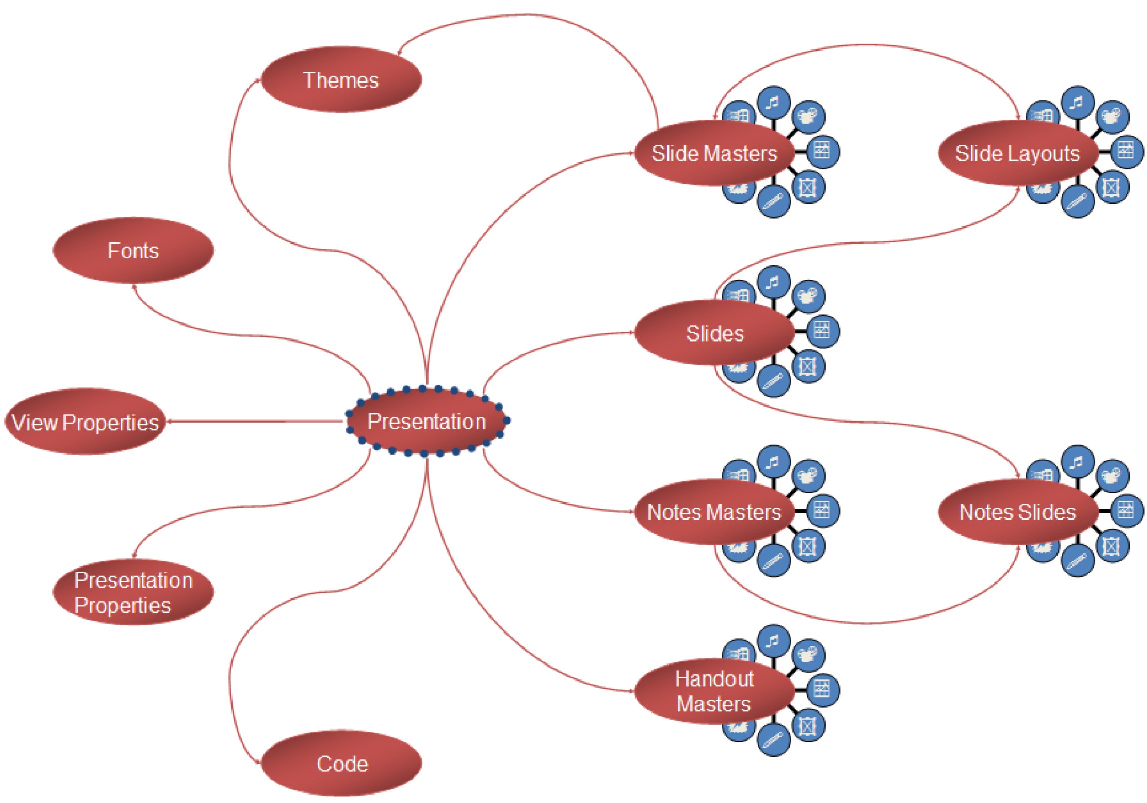
Open XML SDK obsahuje silně typované třídy obalující jednotlivé části Office Open XML a tím umožňuje práci (vytvoření či úpravu) s dokumenty na tomto standardu založenými. V době psaní této práce, je pro operační systémy Microsoft Windows XP Service Pack 3 a novější, volně dostupná ke stažení, na stránkách Microsoft Download Center[3], verze Open XML SDK 2.0. Část PresentationML tohoto standardu se poté týká samotných prezentací. Jak již je z názvu standardu patrné, tak je postaven nad XML[4]. Tedy celý prezentační soubor[5] se v podstatě skládá z jednotlivých částí popsaných pomocí XML, které jsou následně spojeny pomocí kompresního algoritmu ZIP do jednoho souboru. Co tento soubor obsahuje si můžeme prohlédnout na obrázku 3.2 (obrázek převzat z [6]).

Kořenem dokumentu je část `Presentation`, ta obsahuje jednu čí více částí `SlideMaster`, `SlideLayout`, `Slide` a `Theme`. Tyto části jsou zároveň nejmenší podmnožinou PresentationML, která dohromady tvoří validní dokument. Tedy pro vytvoření validního dokumentu tento musí obsahovat všechny tyto části. Část `Presentation` obsahuje souhrnné informace o dokumentu, mezi něž patří například seznam snímků, seznam vložených písem, nebo také nastavení směru textu zprava doleva či způsob komprese obrázků při ukládání. `SlideMaster` určuje pomyslný hlavní snímek (může jich být několik), který definuje formátování, text a objekty, které se objeví na každém snímku, který je od tohoto hlavního snímku odvozen. Další částí je `SlideLayout`, tato (opět jich dokument může obsahovat více než pouze jednu) obsahuje definice výchozího vzhledu a rozložení objektů na snímku pro všechny snímky, které jsou od něj odvozeny. Část `Slide`, jak již její název napovídá,

[3] `http://www.microsoft.com/downloads/en/default.aspx`

[4] eXtensible Markup Language

[5] v rámci této podkapitoly uvažujeme soubor uložený ve formátu Office Open XML



Obrázek 3.2: Diagram PresentationML dokumentu

obsahuje definici právě jednoho snímku prezentace. Poslední částí je `Theme`. Ta obsahuje definici použitého tématu prezentace (barevné schéma, apod.). Ostatní části jsou volitelné a obsahují například poznámky k jednotlivým snímkům či definice stylů a rozložení pro tisk.

Stejně, jako v případě automatizace pomocí PowerPoint PIA, si ukážeme základní postupy pro vytvoření jednoduché prezentace. Ovšem s tím rozdílem, že se zaměříme především na podstatné kroky a obecný popis tvorby jednotlivých elementů (snímek, textové pole, tabulka, apod.). Je tak zejména proto, že na rozdíl od automatizace, kdy ovládáme instanci aplikace PowerPoint, při práci s PresentationML musíme celý výsledný dokument vytvořit sami. Tedy objem prováděných činností je podstatně větší. Praktické ukázky práce s PresentationML dokumentem je možné nalézt například na [11].

Při použití Open XML SDK musíme vytvořit úplně celý dokument sami, tedy i části jako je `SlideMaster` či `SlideLayout`. Jelikož definice těchto částí bývá většinou značně rozsáhlá, je často vhodné vydat se nejprve menší oklikou. Tou je vytvoření prázdné prezentace (neobsahující žádné snímky) v aplikaci PowerPoint, její uložení a následné použití pro vytvoření prezentace nové pomocí Open XML SDK (z této prázdné prezentace si lze v kódu vždy vytvořit kopii, kterou budeme upravovat, a tedy ji lze používat stále dokola). Tato prázdná prezentace obsahuje již nadefinované všechny potřebné části, a tedy se můžeme

naplno věnovat tvorbě samotných snímků.

Stejně jako v případě automatizace je snímek tvořen různými tvary. Tyto tvary jsou v podstatě uzly, které jsou potomkem daného snímku. Ty poté obsahují další potomky, kteří určují obsah daného tvaru. Tedy práce se samotným dokumentem se příliš neliší od zpracování surového XML dokumentu pomocí přístupu k jeho elementům.

Postup tvorby nového snímku tedy sestává z vytvoření snímku, nastavení jeho vlastností, připojení tvarů a nakonec přidání reference na použitý `SlideLayout` a připojení snímku do prezentace. Opět si uveďme krátký příklad, který tyto činnosti demonstruje.

```
// vytvoříme část Slide pro nový snímek
var slidePart = presentationPart.AddNewPart<SlidePart>("rel257");
// nastavíme a vygenerujeme obsah snímku (GenerateTitleSlide je pomocná metoda↩
    vracející nový snímek)
GenerateTitleSlide().Save(slidePart);

// přidáme referenci na použitý SlideLayout
slidePart.AddPart<SlideLayoutPart>(slideLayoutTitle);

// přidáme odkaz na nový snímek do seznamu snímků
var slideId = new SlideId();
slideId.RelationshipId = "rel257";
slideId.Id = 257;
presentationPart.Presentation.SlideIdList.Append(slideId);
```

Důležitou informací je, že číslování částí `Slide` musí začínat číslem větším než 255 (a zároveň být menší než 2147483648). Toto číslo v podstatě udává unikátní identifikátor každého snímku. Nastínili jsme si tedy, jak se přidává nový snímek do prezentace. Nyní přeskočíme dále a podíváme se na změnu řezu písma (formátování textu).

Abychom lépe pochopili způsob, jakým se provádí formátování textu, tak si nejdříve nastíníme, jak je vlastně tento text v dokumentu ukládán a jakým způsobem se s ním pracuje. Představme si, že máme element textového pole, do kterého chceme vložit požadovaný text. Tento element (kromě potomků pro nastavení jeho vlastností) obsahuje potomky reprezentující jednotlivé odstavce. Každý z těchto odstavců poté obsahuje takzvané `Run` elementy, jejichž potomci již konečně definují samotný text a jeho formátování. Pro lepší pochopení si opět uveďme krátký příklad.

```
titleShape.TextBody = new TextBody(new Drawing.BodyProperties(new Drawing.↩
    NormalAutoFit()),
    new Drawing.ListStyle(),
    new Drawing.Paragraph(
        new Drawing.Run(new Drawing.RunProperties(
        new Drawing.LatinFont() { Typeface = "Impact" }) { Language = "cs-CZ",↩
            Capital = Drawing.TextCapsValues.Small },
        new Drawing.Text() { Text = "Ukázková prezentace" })));
```

Každý odstavec (`Paragraph`) může obsahovat několik `Run` částí, což umožňuje nastavit řez písma podle potřeby i pouze pro určitou část textu a ne jen pro celý odstavec. Výčtová prostředí, tedy text s odrážkami, jsou podobně jako u automatizace řešena odstavci, které

mají nastaveno použití odrážky.

Vložení obrázku do prezentace se řeší přidáním speciálního tvaru (místo klasického `Shape` použijeme tvar `Picture`) do požadovaného snímku, nakopírováním binární podoby obrázku do výsledného dokumentu a uložením reference na něj (pro spárování tvaru a binární podoby). Vlastnosti jako šířka, výška a umístění vloženého obrázku se specifikují přímo při vytváření tvaru `Picture` v jeho potomku `ShapeProperties`.

Poslední částí, kterou se budeme zabývat, jsou tabulky. Podobně jako při vkládání obrázku použijeme místo klasického tvaru `Shape` tvar jiný, a tím je `GraphicFrame`. V něm je poté zanořených několik různých elementů určujících jeho vlastnosti a až přibližně kolem třetí úrovně zanoření najdeme element `Table`, který reprezentuje tabulku (nutno podotknout, že všechny tyto elementy musíme vytvořit a přidat do tvaru `GraphicFrame` my). Element `Table` obsahuje poté řadu potomků. Jsou jimi nepovinný `TableProperties`, který určuje například styl tabulky, povinný `TableGrid`, který definuje počet a šířku sloupců tabulky. Poté následují elementy definující jednotlivé řádky tabulky (`TableRow`), které obsahují jednotlivé buňky (`TableCell`). V následující ukázce je uvedeno jak vytvořit řádek tabulky obsahující jednu buňku.

```
var tableRow1 = new Drawing.TableRow(new Drawing.TableCell(
    new Drawing.TextBody(new Drawing.BodyProperties(),
    new Drawing.ListStyle(),
    new Drawing.Paragraph(new Drawing.Run(new Drawing.RunProperties() { ↩
        Language = "cs-CZ", SmartTagClean = false },
    new Drawing.Text() { Text = "buňka1, řádek 1"}))),
    new Drawing.TableCellProperties())) { Height = 370840L };
```

Spojování buněk se poté provádí pomocí nastavení vlastnosti `RowSpan` (pro vertikální sloučení) či `GridSpan` (pro horizontální sloučení) první ze spojovaných buněk, na počet buněk, které budou spojeny. A následně nastavení odpovídající vlastnosti `VerticalMerge` či `HorizontalMerge` na `true` všem následujícím buňkám, které budou spojeny s touto první. Obsah takto spojených buněk je umístěn pouze v první buňce a ostatní se ponechávají prázdné.

3.3 Shrnutí

Nastínili jsme si postup tvorby jednoduché prezentace a zároveň také práci s nejčastěji používanými prvky prezentace (formátovaný text, obrázek, tabulka) dvěma různými způsoby. Každý z nich má své klady i zápory a tedy je pouze na nás zhodnotit, který z těchto způsobů si zvolit. Jestliže chceme vytvořit aplikaci, která je schopná prezentaci uložit nejenom v klasickém formátu, ale také například jako HTML a nevadí nám nutnost mít nainstalován PowerPoint, tak se nabízí automatizace. Jestliže však chceme vytvořit aplikaci, která nevyžaduje pro tvorbu prezentací žádný dodatečný software a spokojíme se pouze s jedním výstupním formátem tak se pravděpodobně uchýlíme k Open XML SDK.

Kapitola 4

GPLEX a GPPG

Důležitými součástmi překladače jsou nepochybně lexikální a syntaktický analyzátor. Doba, kdy bylo nutné, abychom si tyto analyzátory museli naprogramovat sami, je již dávno minulostí. Dnes již existují spousty nástrojů, které automaticky vygenerují analyzátor na základě námi udaných pravidel. Navíc díky tomu, že je těchto nástrojů dostupné velké množství, je možné najít si takový, jehož výstup je v námi požadovaném programovacím jazyce.

GPLEX je generátorem lexikálního a GPPG generátorem syntaktického analyzátoru. Jejich výstupem jsou třídy v jazyce C# implementující funkcionalitu těchto analyzátorů.

Informace uvedené v této kapitole jsou čerpány z dokumentací k oběma uvedeným generátorům, tedy z [2] a [3].

4.1 The Gardens Point Scanner Generator (GPLEX)

GPLEX je generátor lexikálních analyzátorů založených na konečných automatech. Vygenerovaný konečný automat je reprezentován tabulkou a má minimalizovaný počet stavů. Generátor navíc umožňuje vybrat si z různých schémat komprese jeho tabulky. Výstupem celého procesu je následně jeden C# zdrojový soubor, který obsahuje třídu `Scanner` implementující lexikální analyzátor. Podstatnou výhodou (zejména v našem národním prostředí) je podpora znakové sady Unicode.

Syntaxe definice lexikálního analyzátoru (tedy vstupu generátoru) je podobná té, kterou používá unixový generátor LEX (více informací o nástroji LEX v [5]). Tento soubor je rozdělen na tři části následovně:

```
// definice
%%
// pravidla
%%
// uživatelské metody
```

První část obsahuje různé druhy definicí a deklarací, zde je například možné pojmenovat

si různé regulární výrazy, nadefinovat startovací podmínky či dokonce specifikovat v jakém jmenném prostoru se bude výsledný zdrojový kód lexikálního analyzátoru nacházet.

Druhá část obsahuje definice pravidel použitých pro rozpoznávání vzorů. Každé z těchto pravidel může obsahovat nějakou sémantickou akci, která je provedena v případě, že vstup odpovídá danému pravidlu. Podobně jako v nástroji LEX jsou dostupné určité proměnné, které můžeme při definici sémantických akcí použít. Patří mezi ně například `yytext`, která obsahuje rozpoznaný vzor pravidla, jehož sémantická akce je právě prováděna či `yypos` obsahující informace o aktuální pozici (další lze nalézt v dokumentaci [2, str. 85]).

Poslední část je volitelná (tedy je možné ji vynechat i s uvozujícími '`%%`'). Tato část obsahuje definice uživatelských metod, které můžeme použít v rámci sémantických akcí. Kód použitý v této sekci je při generování lexikálního analyzátoru v podstatě přímo zkopírován do výsledné třídy.

4.2 The Gardens Point Parser Generator (GPPG)

GPPG je generátor syntaktických analyzátorů pro LALR(1) jazyky. Jeho vstupem je specifikace podobná té, kterou používá unixový generátor YACC (více informací o nástroji YACC v [5]) a výstupem je podobně jako u nástroje GPLEX zdrojový kód v jazyce C# obsahující třídu `Parser`, která implementuje daný syntaktický analyzátor. Na rozdíl od nástroje GPLEX však zdrojový kód neuloží do souboru, nýbrž jej vypíše na svůj standardní výstup.

Soubor obsahující specifikaci dané gramatiky je opět rozdělen na tři části (toto rozdělení v podstatě odpovídá tomu, které jsme si uvedli u nástroje GPLEX).

První částí jsou uživatelské definice jako například definice tokenů (terminálů), jejich priorit a asociativity či deklarace datových typů pro předávání hodnot z lexikálního analyzátoru. Způsob definice asociativity tokenů a jejich priorit je v podstatě obdobný s tím, který používá YACC, tedy pomocí klíčových slov `%left`, `%right`, `%noassoc` a `%token` určíme asociativitu daných tokenů a jejich priorita je dána pořadím jejich deklarace (později deklarované tokeny mají větší prioritu).

Druhá část obsahuje pravidla použité gramatiky a k nim přiřazené sémantické akce. Přiřazené sémantické akce se provádějí při redukci daného pravidla. V těle těchto akcí je opět dostupných několik proměnných, s kterými můžeme pracovat. Mezi nejdůležitější z nich patří `$$`, do které ukládáme hodnotu naší sémantické akce, a `$1`. . . `$N`, které obsahují sémantické hodnoty jednotlivých symbolů na pravé straně pravidla. Seznam dostupných metod a proměnných pro sémantické akce lze nalézt v dokumentaci [3, kapitola 8.2].

Třetí a poslední část je stejně jako v případě nástroje GPLEX volitelná a obsahuje definice uživatelských metod.

4.3 Shrnutí

Seznámili jsme se s dvěma nástroji pro generování lexikálního a syntaktického analyzátoru. Zároveň jsme si naznačili i způsob jejich použití. Důvodem, proč jsme si představili z nepřeberného množství generátorů právě GPLEX a GPPG, je skutečnost, že již od počátku se jejich vývoj soustředil na platformu .NET a jazyk C#. Jsou tedy implementovány tak, aby kód, který produkují, využíval plně vlastností a výhod tohoto jazyka (například `partial` třídy).

Kapitola 5

Návrh

Návrh je nepochybně jednou z podstatných etap vývoje jakéhokoliv software. Dobře zpracovaný návrh dokáže v mnoha případech značně usnadnit a urychlit implementaci daného produktu. Návrh sestává z několika různých etap. Nejprve provedeme stručné zhodnocení existujících nástrojů, což nám umožní vytvořit si obecný pohled na to, jak se tyto nástroje ovládají a jak fungují, ale zejména jakým směrem se při návrhu nového nástroje vydat. Následně si stanovíme požadavky na námi vytvářenou aplikaci. Navrhneme strukturu výsledné aplikace. Rozebereme si jednotlivé části převodu dokumentu mezi cílovými platformami. A nakonec provedeme návrh jednotlivých částí uživatelského rozhraní.

5.1 Existující nástroje, jejich klady a zápory

Jak již jsme si uvedli v úvodu tak nástroj provádějící převod mezi platformou LaTeX a Microsoft PowerPoint v době psaní této práce není dostupný. Tedy abychom získali přehled o tom, jak vypadají dnešní převaděče dokumentů a díky tomu mohli navrhnout konkurenceschopný nástroj, musíme se zaměřit alespoň na podobné nástroje. Vzhledem k povaze vytvářeného nástroje se vyhneme tzv. online převaděčům a také takovým, jejichž výstupem je spustitelný program (či video) a ne dokument.

Prvním nástrojem, na který se zaměříme je *Convert PowerPoint* od společnosti Softinterface, Inc.[1] Tento nástroj provádí převod z PowerPoint dokumentu. Je možné jej ovládat jak pomocí uživatelského rozhraní, tak i přes příkazovou řádku. Uživatelské rozhraní programu je na první pohled jednoduché a intuitivní. Mimo jiné také umožňuje dávkové zpracování. Je možné vybrat z mnoha výstupních formátů, mezi které patří například HTML, JPEG, GIF, RTF, ale dokonce i PDF. Nevýhodou však zůstává nutnost mít nainstalován PowerPoint. Při pohledu na výstup tohoto nástroje lze snadno odhadnout, že používá automatizaci. Tedy převod do HTML, RTF či obrázků je pravděpodobně řešen pouhým otevřením dokumentu v instanci aplikace PowerPoint a uložením v požadovaném formátu (viz popis způsobu uložení souboru v PowerPoint PIA v kapitole 3.1).

[1] Dostupný na: `http://www.softinterface.com/Convert-PowerPoint%5CConvert-PowerPoint.htm`

Dalším nástrojem je *PowerPoint Converter* od společnosti VeryDOC.com Company[2]. Podobně jako předchozí nástroj provádí převod z PowerPoint dokumentu a umožňuje ovládání jak pomocí uživatelského rozhraní, tak i přes příkazovou řádku. Všechna možná nastavení jsou vidět hned v hlavním oknu programu (formou zaškrtávacích políček). Je možné vybrat si z mnoha výstupních formátů, opět je možný export do HTML, obrázku, PDF, ale dokonce i jako Flash. Stále však přetrvává nutnost mít nainstalován PowerPoint (a stejně jako u předchozího nástroje pohledem na výstup zjistíme, že používá automatizaci).

Poslední nástroj, který si zmíníme, provádí opačný převod a to z PDF na editovatelnou prezentaci PowerPoint. Tímto nástrojem je *Able2Extract PDF Converter* od společnosti Investintech.com Inc.[3] Nástroj umožňuje konverzi PDF do různých formátů používaných Microsoft Office, ale také Open Office. Uživatelské rozhraní je příjemně zpracováno, celým procesem konfigurace převodu nás provádí průvodce, který zejména začátečníkům práci s tímto nástrojem významně ulehčí. Podporuje také dávkový převod dokumentů. Vyzkoušíme-li si převést například PDF vytvořené pomocí aplikace PowerPoint, dosáhneme u výstupu dobrých výsledků. Pokud vyzkoušíme převést PDF s prezentací vytvořenou za pomocí třídy Beamer, dosáhneme pro styl *default* také dobrých výsledků, avšak u jiných barevnějších stylů nástroj většinou nezpracuje správně obrázek umisťovaný na pozadí. Pokud si blíže prohlédneme způsob vytváření výstupní prezentace, můžeme si všimnout, že nástroj rozděluje text do různých tvarů, které následně absolutně umisťuje v rámci daného snímku.

5.2 Stanovení požadavků na vytvářenou aplikaci

Nyní si již můžeme stanovit požadavky na vytvářenou aplikaci. Pro větší přehlednost si je rozdělíme do dvou různých kategorií. Těmi jsou požadavky na překlad a požadavky na funkcionalitu. U požadavků na funkcionalitu budeme částečně vycházet z kladů dnešních překladačů dokumentů.

Požadavky na překlad se týkají směru překladu či implementovaných elementů dokumentu, v našem případě jsou požadavky následující:

- Primárně zaměření na překlad z platformy LaTeX na platformu Microsoft PowerPoint
- Implementace překladu elementů probraných v kapitole 2, jedná se zejména o
  - Smíšenou sazbu
  - Výčtová prostředí
  - Tabulky
  - Obrázky

[2]Dostupný na: `http://www.verypdf.com/ppt-converter/index.html`
[3]Dostupný na: `http://www.investintech.com/prod_a2e.htm`

– Snímky prezentace
– Overlay

Požadavky na funkcionalitu se týkají samotné aplikace, jejího chování, uživatelského rozhraní apod. V našem případě se jedná o následující body:

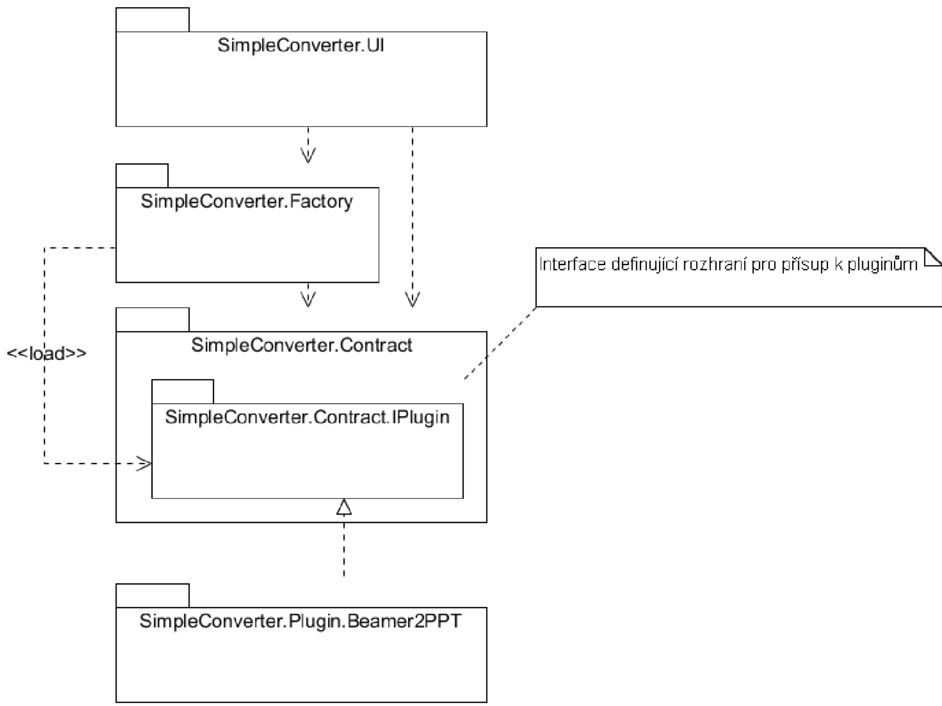
- Možnost budoucího rozšiřování aplikace i pro ostatní formáty (tedy překlad dokumentu provádět například pomocí vstupně-výstupních filtrů)
- Možnost použití jak přes uživatelské rozhraní, tak i jako konzolovou aplikaci
- Možnost dávkového zpracování

Tyto uvedené požadavky jsou minimem, které budeme požadovat od námi vyvíjené aplikace.

5.3 Struktura aplikace

Rozvržení aplikace do logických celků je, zejména v případě modulárního přístupu, nutností. Toto rozdělení nejenže zpřehlední vývoj, ale také nám zjednoduší pozdější údržbu aplikace.

V námi vytvářené aplikaci, je těchto logických celků několik. Jejich přehledné zobrazení včetně vazeb mezi nimi, je pomocí UML diagramu balíčků uvedeno na obrázku 5.1.



Obrázek 5.1: UML diagram balíčků struktury aplikace.

Prvním logickým celkem je uživatelské rozhraní (`SimpleConverter.UI`). Již z názvu je zřejmé, že tento celek bude zajišťovat obsluhu uživatelského rozhraní hlavní aplikace.

Dalším důležitým celkem je takzvaný kontrakt (`SimpleConverter.Contract`). Tento celek je důležitý zejména z pohledu modularity námi vytvářené aplikace. Jedná se v podstatě o pomyslný balíček rozhraní, které definují způsob, jakým jsou provázány jednotlivé dynamicky načítané pluginy s hlavní aplikací. Tedy je důležité, aby se tento celek co nejméně měnil.

Prostředníkem mezi uživatelským rozhraním a kontraktem je logický celek pojmenovaný `SimpleConverter.Factory`. Tento se stará o dynamické načítání jednotlivých pluginů definovaných za pomoci kontraktu a zprostředkování komunikace mezi nimi a uživatelským rozhraním.

Posledním celkem jsou samotné realizace rozhraní definovaných v kontraktu, tedy jednotlivé pluginy. Každý z pluginů lze považovat za jeden logický celek (v UML diagramu na obrázku 5.1 je uveden pouze jeden plugin a to `SimpleConverter.Plugin.Beamer2PPT`). Jeden plugin se tedy stará o celý proces převodu dokumentu.

Pro návrh pluginů se nám nabízí dva hlavní způsoby. Prvním je oddělení zpracování vstupu a výstupu, tedy jeden plugin vstupní a druhý plugin výstupní. Druhým způsobem je zpracování jak vstupu, tak i výstupu v jednom pluginu. V našem případě jsme nakonec zvolili variantu druhou. Důvody pro tuto volbu jsou, že v případě první varianty v podstatě uživateli umožníme převod z jakéhokoli formátu do jakéhokoli jiného formátu a to i tehdy, kdy jsou tyto formáty absolutně nekompatibilní. Navíc by uživatel musel nastavovat separátně parametry převodu jak vstupu, tak i výstupu, což vede k dalšímu zesložitění práce s aplikací. Tato varianta by zejména pro informaticky méně vzdělané uživatele mohla být odrazující. Oproti tomu varianta druhá umožňuje převod pouze mezi kompatibilními typy dokumentů, je možné nastavení jak pro vstup, tak i pro výstup sloučit do jednoho a je tedy uživatelsky přívětivější. Avšak za cenu nutnosti implementace různých dalších kombinací převodu, kterých se dalo v případě varianty první dosáhnout pouze s minimálním úsilím. Další výhodou druhé varianty je, zejména z pohledu programátora, že řešení mezikódu použitého pro převod mezi dvěma formáty je čistě v režii samotného pluginu, tedy není nutné specifikovat univerzální mezikód na úrovni samotné aplikace.

Dalším problémem, který nám vzniká z pohledu struktury aplikace, je že ve většině případů budeme chtít převod dokumentu ovlivnit nějakými dodatečnými parametry. Avšak o převod dokumentu se starají samotné pluginy, které nemají přístup k uživatelskému rozhraní, kde by uživatel mohl tyto parametry zadat. Řešením je, aby každý plugin definoval určitou oblast uživatelského rozhraní, která bude následně při zvolení daného pluginu vložena do hlavního okna (více v kapitole 5.5).

5.4 Převod dokumentu z třídy Beamer na PowerPoint

Složitost převodu dokumentu mezi třídou Beamer a platformou PowerPoint tkví zejména v tom, že každý z těchto formátů k dokumentu přistupuje jiným způsobem. Zatímco třída Beamer je založena na formátování textu, tak PowerPoint funguje na principu rozmístění netextových tvarů na snímku. Teprve obsahem těchto tvarů je samotný text. Dalším problémem je, že LaTeX (jehož rozšířením Beamer je) není bezkontextový jazyk, tedy není možné jej zpracovat pomocí bezkontextové gramatiky. Ovšem tento problém lze jednoduše odstranit tím, že povolíme pouze podmnožinu příkazů, což je v našem případě dostačující řešení (základ této podmnožiny jsme si již určili v kapitole 2).

5.4.1 Lexikální analýza

Prvním krokem převodu z třídy Beamer je rozdělení textu na tokeny, tedy lexikální analýza. Abychom toto mohli provést, musíme si nejdříve ujasnit, jaké části zdrojového textu jsou pro nás významné z pohledu jeho zpracování.

Jako první se můžeme například zaměřit na speciální znaky (symboly):

```
# % $ ^ & _ { } ~ \
```

Jelikož tyto znaky mají v LaTeXu speciální význam, můžeme je přímo označit za tokeny.

Následují příkazy a prostředí. Jak jsme si uvedli v kapitole 2.2, tak příkazů existují dva různé typy. Při bližším prozkoumání však zjistíme, že co se týče lexikální analýzy, nemusíme dělat rozdíly mezi příkazy ovlivňujícími jejich parametr, či příkazy ovlivňujícími zbytek aktuálního bloku. Lze je tedy obecně popsat regulárním výrazem `\[a-zA-Z]+`. Jelikož však zpracováváme pouze podmnožinu příkazů LaTeXu a třídy Beamer a zároveň potřebujeme rozlišit použité příkazy, můžeme si dovolit označit různé příkazy za různé tokeny. U prostředí je situace poněkud odlišná. Začátek a konec prostředí jsou v podstatě příkazy s (nejméně) jedním povinným parametrem, který udává typ prostředí. Příkazy pro počátek a konec prostředí lze popsat regulárními výrazy `\begin[ \t]*\r?\n?[ \t]*{[a-zA-Z]+}` a `\end[ \t]*\r?\n?[ \t]*{[a-zA-Z]+}`. Stejně jako v případě klasických příkazů si pro lepší zpracování označíme různá prostředí různými tokeny.

Příkazů a prostředí se společně týkají i specifikace overlay a volitelných parametrů. Jak již jsme si dříve zmínili, tak parametry overlay jsou ohraničeny znaky `<` a `>`, zatímco volitelné parametry pomocí znaků `[` a `]`. Jelikož tyto znaky nejsou součástí speciálních znaků používaných LaTeXem a zvláštní význam mají pouze v situaci, kdy jsou použity ihned za příkazem, musíme tuto situaci zohlednit. Tuto situaci lze řešit například vstupem do speciálního stavu po detekci příkazu a kontrolou právě volitelných parametrů a overlay specifikace v tomto stavu. U tohoto řešení je však nutné dbát na pořadí těchto parametrů, což však není příliš velkým problémem. Dokumentace třídy Beamer totiž stanovuje ([15, kap. 9.3]), že overlay specifikace by měla, až na pár výjimek, vždy předcházet definici volitelného parametru.

Poslední z hlavních částí, které nás zajímají je obyčejný text (*plaintext*). Při jeho zpracování je třeba věnovat pozornost zejména způsobu zpracování bílých znaků a escape sekvencím speciálních symbolů. Zpracování bílých znaků zjednodušeně spočívá v ignorování více mezer (či tabulátorů) za sebou a vložení nového odstavce po naražení na prázdný řádek. Právě zpracování obyčejného textu je dalším případem, kdy použití speciálního (ať již inkluzivního či exkluzivního) stavu nám umožní upravit vstupní řetězec způsobem, jakým potřebujeme.

Za zmínku stojí i komentáře do konce řádku, které je nutné při zpracování ignorovat. Ty je možné popsat jednoduchým regulárním výrazem: `%.*\n?[ \t]*`.

Výsledný seznam regulárních výrazů použitých pro lexikální analýzu s přiřazením jednotlivých tokenů je uveden v příloze C.

5.4.2 Generování abstraktního syntaktického stromu

Dalším krokem po rozdělení vstupního dokumentu na tokeny je sestavení stromu, který bude tento dokument reprezentovat, a s kterým již můžeme dále pracovat v rámci převodu. Pro syntaktickou analýzu předpokládejme validní dokument na vstupu.

Musíme tedy analyzovat použití jednotlivých příkazů, kde se v rámci dokumentu mohou vyskytovat, jakým způsobem se zapisují, zdali mají povinný parametr, zda je možné je zanořovat a další. Na základě těchto zjištění již můžeme sestavit bezkontextovou gramatiku, kterou použijeme k sestavení abstraktního syntaktického stromu zpracovávaného dokumentu. Zde si pouze stručně rozebereme jednotlivé části dokumentu.

Začneme tedy s analýzou dokumentu pro sestavení bezkontextové gramatiky, a to postupně směrem shora dolů, jelikož tento je pro člověka pro daný typ dokumentu přirozenější a tím pádem i přehlednější.

Prvním příkazem, na který narazíme, je specifikace třídy dokumentu. Ta je následována dvěma pomyslnými bloky. Prvním je preambule, která může obsahovat příkazy pro použití dodatečných balíčků, nastavení pro titulní stranu a další příkazy, které, můžeme říci, neovlivňují *obsah* dokumentu přímo.

Druhým blokem je blok definující dokument a jeho obsah. Tento blok může obsahovat definice jednotlivých snímků, ale také například opět nastavení pro titulní stranu či nastavení sekcí a podsekcí. Každý snímek může mít jak titulek, tak i podtitulek. Podle použitého příkazu pro zápis snímku je titulek (či podtitulek) buď parametrem, nebo je uveden příkazem kdekoli v těle snímku. Tělo snímku dále může obsahovat např. formátovaný text, tabulky, výčtová prostředí a obrázky[4], které se mohou dále zanořovat.

Tabulky mají parametrem stanovený počet sloupců, které jsou oddělovány znakem `&` a poté jednotlivé řádky oddělovány příkazem pro nový odstavec. Každá z buněk tabulky může obsahovat jak formátovaný text, tak další blokové prvky (tedy i tabulku).

Výčtová prostředí mají společnou vlastnost, a to, že obsahují seznam položek. Tyto

[4]tabulky, výčtová prostředí a obrázky budeme dále označovat jako blokové prvky

položky následně mohou obsahovat stejně jako buňky tabulky formátovaný text a blokové prvky.

Důležitou informací je také, že některé z příkazů přijímají pouze formátovaný text a ne blokové prvky. Mezi tyto patří již zmíněné nastavení titulní strany či titulku a podtitulku snímku.

Stručně jsme si naznačili postup pro vytvoření bezkontextové gramatiky popisující námi zvolenou podmnožinu příkazů. Její výslednou podobu je možné nalézt v příloze D.

5.4.3 Převod

Posledním a pravděpodobně i nejsložitějším krokem je převod získaného stromu dokumentu do požadovaného výstupního formátu. V rámci jeho návrhu je nutné vyřešit několik bodů týkajících se například použitých postupů, technologií či řešení rozdílnosti obou platforem.

Prvním bodem je, zda vytvářet výsledný dokument pomocí automatizace nebo pomocí Open XML SDK. Výhody i nevýhody obou přístupů jsme si již dostatečně popsali v kapitole 3. My si zvolíme přístup pomocí automatizace, a to hned z několika důvodů. Jednak tvorba dokumentu je podstatně jednodušší, ale hlavně je v podstatě nemožné, aby se nám podařilo vytvořit nevalidní dokument, který nepůjde otevřít. Navíc jelikož požadovaným výstupním formátem je PowerPoint dokument, můžeme předpokládat, že uživatel má nainstalován Microsoft Office. Malým plus pro tento přístup je možnost uložit výstupní dokument nejen jako prezentaci ve formátu pro PowerPoint, ale také například jako PDF dokument či obrázky.

Dalším bodem k řešení je, kdy budeme vytvářet jednotlivé snímky prezentace. Nabízí se nám dvě různé možnosti. Buď můžeme jednotlivé snímky vytvářet již během generovaní stromu dokumentu, nebo si nejdříve vygenerujeme celý strom a snímky budeme vytvářet až poté. První přístup není příliš výhodný ať již kvůli přehlednosti implementace, tak i z důvodu efektivity. Srovnáme-li čas potřebný k vygenerování stromu ze vstupního dokumentu s časem potřebným pro vytvoření dokumentu výstupního, zjistíme, že vytvoření výstupního dokumentu je podstatně časově náročnější, než zpracování dokumentu vstupního. Tedy je výhodnější nejdříve vygenerovat celý strom a v případě, že narazíme na chybu skončit ihned, než provádět obě činnosti zároveň a v případě chyby skončit s částečně vygenerovanou prezentací, která je v podstatě nepoužitelná. Proto tedy zvolíme přístup druhý.

V neposlední řadě je také nutné řešit rozdílnost obou platforem. Například pokud v prezentaci vytvářené za pomoci třídy Beamer vložíme do textu malý obrázek, vysází se do textu a text poté pokračuje za ním (za předpokladu, že je za ním dostatečný prostor), tedy obrázek se nám vloží takzvaně *inline*. Stejná situace platí i pro malé tabulky, které nejsou odděleny odstavcem. Tohoto však v aplikaci PowerPoint přímo dosáhnout nelze. Je to proto, že v prezentaci aplikace PowerPoint jsou snímky tvořeny jednotlivými tvary. Ty mohou být pro text, tabulku či obrázek (pomiňme teď ostatní tvary) a mohou obsahovat pouze to, pro co jsou určeny. Tedy situaci, kdy potřebujeme obrázek či tabulku vložit inline budeme nuceni řešit například postupným dělením tvaru pro text a následným pře-

souváním tvarů. S nemožností zanořování různých tvarů souvisí další problém. Tím je, že je v prezentaci tvořené pomocí třídy Beamer možné zanořovat blokové elementy. Jelikož, tohoto opět není možné dosáhnout, musíme se rozhodnout, jak tuto situaci řešit. Nabízí se dvě možnosti realizace. První možností je, že z vnořených elementů budeme zpracovávat pouze formátovaný text. Druhou možností je, že vnořené elementy vygenerujeme například na konci snímku a necháme uživatele, aby si je sám přeorganizoval. V tomto případě je nejlepším řešením realizovat obě varianty a uživateli dát možnost zvolit si tu, která mu více vyhovuje.

Rozdílnosti obou platforem se týká také velikost výsledného snímku. Zatímco u prezentací vytvářených za pomocí třídy Beamer je výchozí velikost snímku $12,8$ cm na šířku a $9,6$ cm na výšku (tedy cca 362×272 bodů při 72 bodech na palec), tak prezentace tvořené v aplikaci PowerPoint mají výchozí rozlišení 720×540 bodů, tedy přibližně dvojnásobek. Zde je nejvhodnějším řešením automaticky upravit velikost textu a v případě obrázků a definované šířky sloupců tabulek se uživatele zeptat, zda si přeje velikost upravit, či nikoliv (s tím, že výchozí chování bude úprava).

5.5 Uživatelské rozhraní

Při návrhu uživatelského rozhraní musíme dbát zejména na jeho přehlednost a celkovou přívětivost. Nepřehledné a zbytečně složité uživatelské rozhraní může uživatele odradit od používání aplikace a to i v případě, že její funkčnost je lepší než u jejích alternativ.

Před samotným návrhem dialogů si musíme ujasnit, jaké informace potřebujeme od uživatele získat a jaké máme v úmyslu mu sdělit. Jakmile si toto ujasníme, stačí již zvolit správné prvky pro získání či předání dané informace a poté se již můžeme pustit do návrhu.

5.5.1 Hlavní dialog aplikace

V případě námi vytvářené aplikace potřebujeme od uživatele získat následující informace (uvedeno i s rozborem jaký prvek použít):

- **seznam souborů, které budeme převádět** - jelikož se jedná o seznam, tak nejvhodnějším řešením je prvek typu `ListBox` či `ListView`. Dále také dvě tlačítka pro přidání a odebrání souboru z tohoto seznamu.

- **výstupní adresář** - zde je vhodné klasické řešení pomocí jednoho prvku typu `TextBox` a jednoho tlačítka pro otevření dialogu pro výběr složky.

- **pokyn k začátku převodu či ukončení probíhajícího** - naprosto zřejmé použití tlačítek

- **výběr pluginu pro převod** - pluginů může být v aplikaci načteno větší množství, avšak vždy může být zvolen pouze jeden. Navíc ve chvíli, kdy je zvolen, již není potřeba zobrazovat ostatní. Tedy nejvhodnějším typem použitého prvku je `ComboBox`.

- **nastavení parametrů převodu** - rozbor řešení v kapitole 5.5.2

Informace předávané uživateli se v našem případě týkají v podstatě pouze procesu překladu, jejich rozbor je následující:

- **informace o průběhu převodu (textová)** - během probíhajícího převodu dokumentů je vhodné uživatele, alespoň v omezené míře, informovat o prováděné činnosti, ale hlavně zobrazit případná varování či chyby. Máme tedy tři typy informace, kdy oznámení a varování se může vyskytovat ve větším množství (u nezotavitelné chyby předpokládejme ukončení překladu). Tedy jedná se o seznam textových zpráv. Avšak pro jejich odlišení je vhodné přidání například ikonky podle typu zprávy, tedy použitým prvkem je prvek typu `ListView`.
- **informace o průběhu převodu (grafická)** - jelikož aplikace umožní převod více než pouze jednoho dokumentu, je vhodné graficky zobrazit průběh převodu těchto dokumentů (kolik je již zpracováno × kolik jich zbývá zpracovat). K tomuto se nejlépe hodí prvek typu `ProgressBar`.

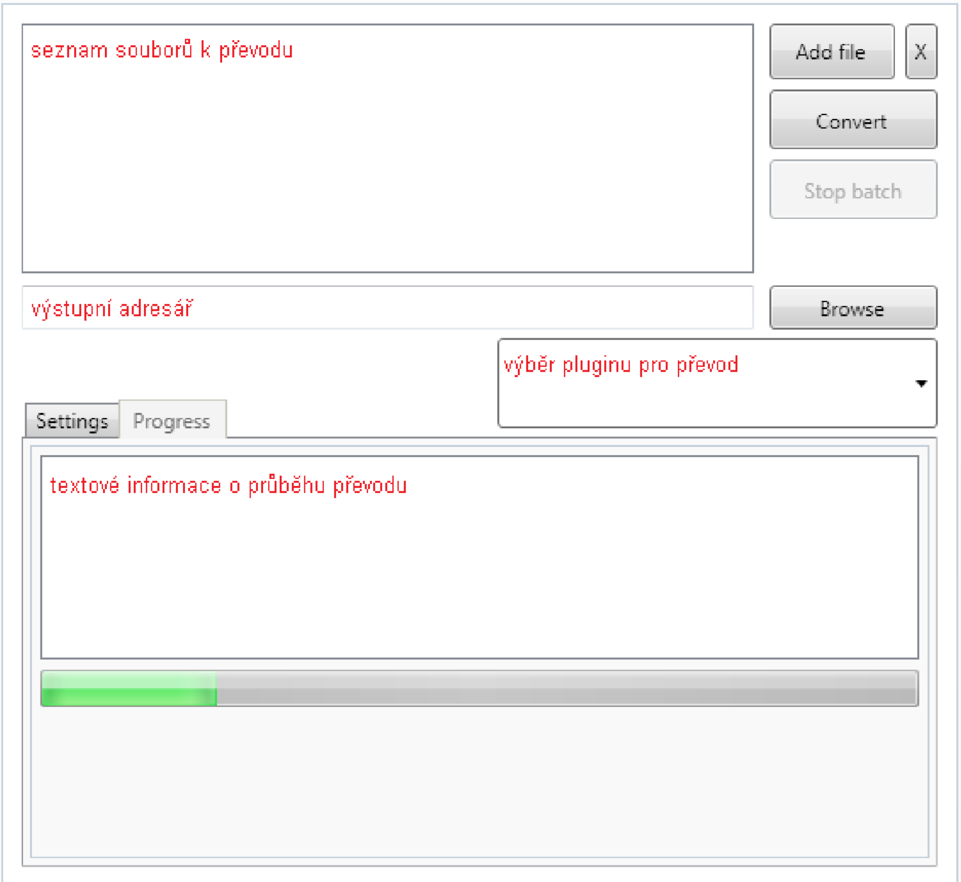
V tuto chvíli již víme, jaké prvky uživatelského rozhraní použijeme. Dalším krokem tedy je rozvržení těchto prvků v rámci okna aplikace. Návrh základního rozložení ovládacích prvků si můžeme prohlédnout na obrázku 5.2 (jedná se o návrh pouze rozložení ovládacích a informačních prvků, tedy neobsahuje popisky, které budou přidány pro lepší orientaci).

Dialog si lze pomyslně rozdělit do dvou oblastí. Horní část obsahuje prvky pro získání informací od uživatele, zatímco spodní část obsahuje prvky pro předání informace uživateli. Toto členění nám umožní dosažení větší přehlednosti. Dále si můžeme povšimnout, že byl použitý prvek `TabControl` pro rozdělení spodní části do dvou záložek. To opět slouží k zjednodušení a zvýšení přehlednosti uživatelského rozhraní. První záložka slouží k nastavení parametrů překladu, zatímco druhá záložka slouží k zobrazení průběhu převodu dokumentů. Důvodem pro rozdělení do dvou záložek je, že během převodu dokumentů nelze měnit parametry převodu a zároveň během úpravy parametrů převodu je zbytečné mít zobrazeny informace o neprobíhajícím převodu.

Kromě rozvržení je vhodné zvážit i další možnosti zvýšení uživatelské přívětivosti. Například zobrazení popisků při přejetí myší nad ovládacím prvkem, umožnění přidání dalšího souboru do seznamu přetažením myší (událost `drop`).

5.5.2 Nastavení parametrů převodu

Jak již jsme si naznačili v kapitole 5.3, tak část uživatelského rozhraní zajišťující nastavení parametrů převodu se vkládá z načteného pluginu. Toto vkládání je možné řešit tak, že v hlavním okně aplikace budeme mít definovanou určitou oblast, do které následně připojíme prvek typu `UserControl` definovaný v načteném pluginu. Tímto způsobem zpracování informací z tohoto prvku zůstává na samotném pluginu a jediná činnost, kterou musí provádět hlavní aplikace je načtení a připojení tohoto prvku.



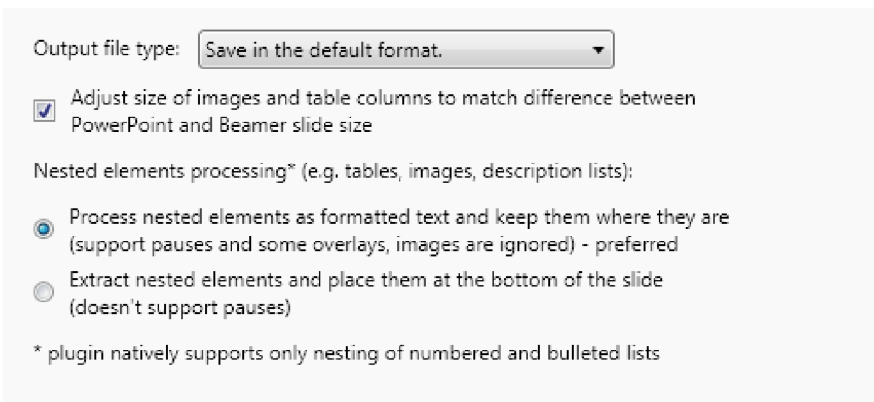
Obrázek 5.2: Návrh rozložení ovládacích prvků hlavního okna aplikace

Opět si proveďme rozbor, jaké informace chceme od uživatele získat a jaké prvky uživatelského rozhraní pro to využít. Z velké části budeme vycházet z informací uvedených v kapitole 5.4.3.

- **Formát výstupního souboru** - jelikož tvorba prezentace pomocí automatizace nám umožňuje výsledný dokument uložit v mnoha různých formátech, byla by škoda této možnosti nevyužít. Tedy dáme uživateli možnost výběru z několika různých formátů. K tomuto se nejvíce hodí prvek typu `ComboBox`, kde bude předvolený výchozí formát (`.pptx`).
- **Způsob práce se zanořenými blokovými elementy** - zde máme dvě možnosti, ze kterých si uživatel může jednu zvolit. Nejvhodnějším prvkem pro realizaci tohoto výběru je prvek typu `RadioList`.
- **Úprava velikosti obrázků a šířky sloupců tabulek** - v tomto případě jde o klasickou volbu typu Ano/Ne, proto je vhodné zvolit prvek typu `CheckBox`.

Informace předávané uživateli jsou, v případě pluginu, v podstatě pouze statické. Dynamicky zobrazované informace se předávají hlavní aplikaci, která je zobrazuje v seznamu

s informacemi o průběhu převodu.



Obrázek 5.3: Návrh rozložení ovládacích prvků pluginu

Na obrázku 5.3 si můžeme prohlédnout výsledné rozložení jednotlivých ovládacích prvků. Jak si můžeme všimnout, jsou ovládací prvky doplněny vcelku obsáhlými popisky. Je to zejména z důvodu, aby měly nejen popisný, ale částečně i vysvětlující charakter, jak jejich volba aktivně ovlivní výstup.

5.5.3 Rozhraní pro konzolový přístup

Jelikož jedním z požadavků na aplikaci, které jsme si stanovili, je možnost jejího použití jako konzolové aplikace, musíme si stanovit rozhraní, přes které se aplikace bude ovládat. Musíme vyřešit zejména, jakým způsobem bude probíhat výběr použitého pluginu, nastavení výstupního adresáře, parametrů převodu a nastavení vstupního souboru či dokonce seznamu souborů. Také je vhodné, aby bylo možné přepínače zadávat několika již obecně zažitými způsoby, tedy aby například použití přepínače `-h`, `/h` či `--help` provedlo stejnou akci. Podívejme se tedy na výsledný seznam přepínačů (pro zápis alternativ je použit znak `|`).

- `h|?|help` - vypíše nápovědu pro použití aplikace a skončí. V případě, že je definován pomocí příslušného přepínače použitý plugin, tak vypíše nápovědu pro něj.
- `l|list` - vypíše seznam dostupných pluginů včetně jejich klíčů a skončí.
- `o|output` - má jeden parametr a nastaví výstupní adresář. Tento přepínač je nepovinný, a pokud není uveden použije se podadresář `output` v aktuálním pracovním adresáři.
- `p|plugin` - má jeden parametr, který specifikuje použitý plugin pomocí jeho klíče.

V omezené míře také budeme moci nastavit parametry převodu, a to pomocí následujících přepínačů:

- `n|nadjust` - slouží k zakázání úpravy velikosti obrázků a tabulek (potlačuje výchozí chování)

- `e|extract` - slouží k zvolení extrakce zanořených blokových elementů (potlačuje výchozí chování)

Jako poslední budeme zadávat seznam souborů, které budou převáděny.

5.6 Shrnutí

V této kapitole jsme se seznámili s nástroji, které plní podobnou funkci a zároveň si stanovili požadavky na námi vytvářenou aplikaci. Po analýze možností vstupních dokumentů jsme si definovali základ pro lexikální a syntaktickou analýzu. Zběžně jsme si probrali úskalí převodu mezi cílovými platformami a možné způsoby jejich řešení. A v neposlední řadě jsme si definovali uživatelské rozhraní a rozhraní pro ovládání aplikace z konzole operačního systému. Máme tedy již kompletní návrh potřebný pro implementaci samotné aplikace.

Kapitola 6

Implementace

Během implementace převádíme myšlenky, znalosti a postupy získané v rámci návrhu na použitelný produkt. V rámci této kapitoly si popíšeme, jaké technologie byly použity a proč. Rozebereme si způsob implementace jednotlivých částí aplikace a vysvětlíme si, proč byl zvolen daný způsob a ne jiný. Tato kapitola tedy slouží jako vodítko k pochopení nejen stavby celé aplikace, ale i fungování jejích částí.

6.1 Struktura aplikace a použité technologie

Zaměřme se nejdříve na použité technologie. Aplikace je implementována v jazyce C# a pro běh vyžaduje nainstalovaný Microsoft .NET Framework 4 (stačí pouze Client Profile). Využívá několik knihoven třetích stran, které jsou dostupné pod různými variantami otevřených licencí. Jedná se o Managed Extensibility Framework[1] použitý pro implementaci požadované modularity aplikace. Knihovnu NDesk.Options[2] použitou pro zpracování parametrů příkazové řádky. A nakonec již v kapitole 4 zmiňované nástroje GPLEX a GPPG pro implementaci lexikálního a syntaktického analyzátoru. Pro vývoj bylo použito vývojové prostředí Microsoft Visual Studio 2010.

Celá aplikace je rozdělena do několika různých projektů, které v podstatě odpovídají diagramu 5.1, který jsme si připravili v rámci návrhu. Jednotlivé projekty používají vlastní jmenné prostory, které taktéž odpovídají stanovenému rozdělení. Jedinou výjimkou jsou projekty obsahující uživatelské rozhraní a konzolový přístup. Kdy projekt s grafickým uživatelským rozhraním je umístěn ve jmenném prostoru `SimpleConverter`, zatímco projekt pro konzolový přístup je umístěn ve jmenném prostoru `SimpleConverter.Console`.

Popišme si nyní způsob implementace a fungování pluginů. Pluginy jsou implementovány jako samostatné dynamicky linkované knihovny (`.dll`). Aby bylo možné je aplikačně načítat musí definovat určité rozhraní, které je součástí balíčku `Contract`. Ten slouží jako jediná vazba mezi pluginy a samotnou aplikací. Při spuštění aplikace poté objekt třídy `Loader`,

[1] Dostupný na: `http://mef.codeplex.com/`
[2] Dostupná na: `http://www.ndesk.org/Options`

umístěné v balíčku `Factory`, prohledá složku `/plugins`, kde najde všechny pluginy definující dané rozhraní a poté je načte (přesněji vytvoří instance tříd implementujících dané rozhraní v dané dynamicky linkované knihovně). Následně již může aplikace s těmito pluginy pracovat.

6.2 Lexikální analýza a generování stromu dokumentu

Základ pro implementaci lexikální a syntaktické analýzy jsme si položili již v minulé kapitole. Dalším krokem tedy bylo doplnit sémantická pravidla pro generování samotného stromu dokumentu. Než si však popíšeme použité struktury pro uchovávání tohoto stromu, je vhodné zmínit, že oproti návrhu bylo provedeno několik menších změn.

Z důvodu co nejlepší použitelnosti v praxi a toho, že aplikace implementuje pouze podmnožinu příkazů (maker) balíčku LaTeX, bylo nutné brát v úvahu situace, kdy aplikace narazí na neznámý příkaz. Tyto situace jsou řešeny částečně jak během lexikální, tak syntaktické analýzy vstupního dokumentu. V rámci lexikální analýzy odstraňujeme u neznámých příkazů samotný příkaz a jeho případné volitelné parametry a overlay specifikaci. Případný povinný parametr i s uvozujícími složenými závorkami je ponechán. Tedy v případě zápisu neznámého příkazu `\prikaz<3>[par]{text text}` dojde k odstranění červené části. Důvodem pro toto chování je, že povinný parametr ve většině případů udává část obsahu snímku, tedy jeho odstranění by bylo nevhodné. Stejný postup se aplikuje i na neznámá prostředí, u nichž je ponechán pouze jejich obsah. Tento přístup však řeší pouze situace, kdy je daný příkaz či prostředí uvnitř vykreslovaného snímku. Není však schopný vyřešit situaci, kdy například v preambuli definujeme nový příkaz pomocí příkazu `\newcommand`. V tuto chvíli vstupuje na scénu použití zotavení z chyb v rámci syntaktické analýzy. Zotavení z chyb je realizováno Hartmannovou metodou (*Panic-Mode recovery*). K zotavení může docházet celkem na dvou místech, a to v preambuli a mezi jednotlivými snímky. V případě, že se narazí na chybu uvnitř snímku, je zpracování ukončeno a o zotavení se syntaktický analyzátor nepokouší (jelikož po vykonání předchozích operací mezi možné syntaktické chyby patří většinou pouze chybějící složená závorka). Díky tomuto chování je aplikace schopná převádět i prezentace obsahující neimplementované příkazy.

Zaměřme se nyní na realizaci samotného stromu dokumentu. Jak uzly, tak i listy jsou objekty definované stejnou třídou (třída `Node`). Tato třída definuje vlastnosti pro typ uzlu (tučné písmo, nečíslovaný seznam, prostý text atd.), volitelný parametr, overlay specifikaci, potomky uzlu, či přímo obsah uzlu (např. prostý text nemá potomky, ale má obsah). Dle typu uzlu zůstávají některé z těchto vlastností nevyužité. Zároveň se v rámci této třídy provádí převedení případné textové reprezentace overlay specifikace na úplný seznam čísel snímků, na kterých se daný uzel zpracuje.

Ne však všechny informace jsou ukládány přímo do stromu dokumentu. Jedná se zejména o titulek a podtitulek snímku. Ty jsou ukládány do samostatné tabulky. Důvodem pro to je, že je možné titulek či podtitulek snímku zapsat kdekoliv v rámci samotného snímku,

tedy i na jeho konci. Jelikož však pro zpracování potřebujeme informaci o použití titulku znát ještě před samotným vytvořením snímku, je vhodné si tuto informaci uložit. Pokud bychom titulek a podtitulek zanesli do stromu dokumentu, nevyhnuli bychom se nutnosti několikanásobného průchodu snímkem před jeho samotným zpracováním.

6.3 Převod

Samotný převod dokumentu do výstupního formátu po získání syntaktického stromu je jedna z obtížnějších částí vývoje aplikace. Je totiž potřeba řešit rozdílnosti mezi jednotlivými platformami, zejména omezení platformy výstupní.

Prvním krokem je extrakce preambule a těla dokumentu ze získaného stromu. Tyto části jsou poté zpracovávány separátně. Nejdříve se provede zpracování preambule, dle které se nastaví vnitřní stav pro převod daného dokumentu (velikost použitého písma, nastavení údajů pro generování titulní strany, nastavení cest pro hledání obrázků atd.). V rámci tohoto zpracování také provádíme kontrolu použité znakové sady. Aplikace dokáže správně zpracovat znaky národních abeced pouze při použití kódování UTF-8. Pokud je tedy vstupní dokument v jiné znakové sadě, oznámíme uživateli, že některé znaky národních abeced nemusí být zobrazeny správně. Po zpracování preambule se provádí zpracování těla dokumentu. Omezme se nyní na to, že tělo dokumentu je tvořeno pouze kolekcí snímků (reálně může obsahovat také nastavení titulní strany, sekce, podsekce apod.).

Tělo dokumentu se postupně zpracovává po jednotlivých snímcích. Před samotným zpracováním snímku se nahlédne do tabulky s titulky (a podtitulky), zda daný snímek bude mít titulek. Pokud ano, vygenerujeme snímek s předpřipraveným tvarem pro titulek, pokud ne, vygenerujeme prázdný snímek. Následné zpracování titulku a obsahu snímku probíhá opět separátně (jelikož titulek může obsahovat pouze formátovaný text, kdežto obsah snímku může být podstatně komplexnější).

Jak obsah titulku, tak i obsah snímku jsou definovány podstromem. Tedy abychom mohli vytvářet výsledný dokument, musíme implementovat patřičný průchod tímto stromem. Pro tento průchod je používána iterativní varianta průchodu *preorder* pro n-ární strom.

Tvorba obsahu snímku je rozdělena, jak již princip tvarů v dokumentu aplikace PowerPoint napovídá, na zpracování formátovaného textu a oddělené zpracování jednotlivých blokových elementů. Podívejme se tedy, jak jsou realizovány jednotlivé části tvorby snímku prezentace.

6.3.1 Formátování textu

Základním problémem formátování textu je, že se musí veškeré formátování provádět zpětně. Tedy nejdříve musíme do daného tvaru vložit text a ten můžeme následně naformátovat, jak potřebujeme. Z toho důvodu je implementována třída `TextFormat`, nabízející rozhraní, díky kterému je tato činnost z hlediska programátora poněkud přirozenější. Tedy nejdříve nastavíme formát a až následně vkládáme text. Tato třída si po nastavení nového formátu

(řezu či stupně) písma uloží aktuální stav na zásobník a umožňuje tak se k němu po vynoření ze zpracovávaného podstromu vrátit. Zároveň pro příkazy, které mění stupeň písma (např. `\footnotesize`, `\Large`), přepočítává velikost v závislosti na nastavené velikosti základního písma. Objekt této třídy je použit k formátování textu nejen v textových tvarech, ale také uvnitř tabulek či výčtových typů.

Dalším bodem, na který musel být při implementaci brán zřetel je, že text musí být uvnitř textového tvaru. Představme si situaci, kdy máme na snímku nějaký text, následovaný tabulkou, po které opět následuje text. Tyto situace jsou řešeny v podstatě nejtriviálnějším možným způsobem. A to tak, že pokud poslední použitý tvar není textový a narazíme na text (a zároveň neprovádíme zpracování některého z blokových elementů), tak automaticky vytvoříme nový textový tvar pro tento text. Je však nutné poznamenat, že i když jsou výčtové typy realizovány pomocí textových tvarů (blíže si vysvětlíme v kapitole 6.3.3), tak je považujeme za netextové.

6.3.2 Tabulky

Pro vytvoření obyčejné tabulky je zapotřebí postupně provést několik kroků. Prvním krokem je zpracování parametru prostředí `\tabular` udávajícího počet, zarovnání a ohraničení jednotlivých sloupců. Kvůli možnosti použití zkráceného zápisu sloupců a nutnosti tento zápis rozgenerovat do použitelné formy, je tento parametr zpracováván pomocí jednoduchého konečného automatu. Ten během zpracování postupně do jedné struktury ukládá zarovnání a šířku daného sloupce a do druhé jeho ohraničení.

Poté, co již máme zpracovaný daný parametr prostředí `\tabular`, máme dostatek informací pro vytvoření nové tabulky (pro vytvoření je totiž potřeba znát počet sloupců). První činnost, kterou musíme po vytvoření nové tabulky provést, je nastavení jejího stylu. Je to nutné z toho důvodu, že výchozí styl tabulky obsahuje podbarvení jednotlivých řádků a také nastavení okrajů buněk. Po tomto nastavení již postupně přidáváme řádky a naplňujeme je daným obsahem. V rámci zpracování každého řádku také provádíme nastavení horního či dolního okraje buněk v závislosti na použití, či nepoužití příkazů `\hline` či `\cline`. V případě spojování buněk provedeme jejich spojení a přeskočíme zpracování všech sloupců, přes které byla buňka spojena.

Jako velký problém se ukázala činnost, jež je v aplikaci PowerPoint řešena pomocí pár kliknutí. Touto činností je automatické přizpůsobení šířky sloupce tabulky velikosti jeho obsahu. I když po prostudování API nalezneme několik různých metod a vlastností (například vlastnost `AutoSize` objektu `TextRange2`), které mají podobnou činnost realizovat, velice rychle narazíme. Polovina z nich skončí s výjimkou říkající, že daný argument nepřijímají a druhá polovina pro změnu s `NotImplementedException`. Bylo tedy nutné tuto situaci řešit jinak. Prvním pokusem o řešení bylo vytvořit tabulku s velice úzkými sloupci, které se následně postupně rozšiřují, až do chvíle, kdy jsou dostatečně velké pro pojmutí celého textu. Základní koncept byl takový, že se postupně zjišťoval počet řádků, na které byl text vysázen (pomocí kolekce `Lines` je možné přistupovat k jednotlivým vysázeným řádkům

textu, kdy se jako řádek počítá i zalomení) a porovnával se s počtem řádků, které vstupní text měl fyzicky obsahovat (tedy počet znaků nového řádku). Tento přístup však ztroskotal na tom, že počet prvků kolekce `Lines` počtu vysázených řádků neodpovídal. Bylo tedy nutné zvolit přístup jiný a to doslova opačný. Místo postupného zvětšování se provádí nárazové zmenšení. Při vytváření tabulky ji tedy vytvoříme s šířkou každého sloupce přibližně 1000 obrazových bodů (pro srovnání, šířka snímku je 720 bodů), následně pro každý řádek v každém sloupci zjistíme a přepočítáme velikost takzvaného ohraničujícího obdélníku textu (anglicky *bounding box*). Tento ohraničující obdélník je vlastně oblast, kterou text skutečně zabírá. Poté zmenšíme šířku každého sloupce na šířku největšího ohraničujícího obdélníku v tomto sloupci se vyskytujícím. Bohužel i tento přístup v některých případech selhává, jelikož PowerPoint z neznámého důvodu nedokáže správně určit velikost ohraničujícího obdélníku.

6.3.3 Výčtové typy

Výčtové typy[3] jsou jediné z blokových prvků, u kterých umožňujeme jejich zanoření (výjimkou je seznam definic, uvedeme si později proč). Opět je nutné věnovat se několika záludnostem. Vytváření odrážek trpí stejným neduhem jako formátování textu. A to, že je nejprve nutné vložit samotný text a až následně z něj vytvořit odrážku. Dalším řešeným problémem je, že v případě prezentací vytvářených za pomoci třídy Beamer, mohou jednotlivé odrážky výčtu obsahovat i více než pouze jeden odstavec. Kdežto u prezentací v aplikaci PowerPoint je situace poněkud složitější.

Podívejme se tedy na samotnou realizaci. Číslované a nečíslované seznamy jsou definovány pomocí textového tvaru. Jejich zanořování je poté řešeno pomocí rekurze. Ovšem se zanořováním souvisí i zmíněný problém několika odstavců v rámci jedné odrážky. Uveďme si praktický příklad:

- Odstavec.
  Další odstavec.

  1. Odrážka
  2. Odrážka s dvěma
     odstavci.

  Pokračování první odrážky.

- Odstavec v další odrážce.

Toto je naprosto normální seznam, dokonce i v dokumentu aplikace PowerPoint je možné ho vytvořit ručně (pro vytvoření dalšího odstavce, místo další číslované položky, použijeme klávesovou zkratku *Shift+Enter*). Problém nastává při automatizaci, kdy simulovat chování

[3] neboli seznamy

stisku kláves *Shift+Enter* není možné (nebo alespoň v žádné z dostupných dokumentací není popsáno, jak požadovaného efektu dosáhnout). Bylo tedy nutné opět přijít s řešením, které poskytuje co nejpoužitelnější výsledek.

První z možných situací je, že odrážka obsahuje několik odstavců, ale neobsahuje žádný seznam. V tomto případě se při přidávání textu počítá celkový počet odstavců. Následně se při nastavování odrážek nastaví daná odrážka pouze prvnímu odstavci, kdežto dalším se nastaví pouze odsazení. Je nutné podotknout, že díky tomuto přístupu si sami musíme udržovat číslování odrážek. Důvodem je právě to, že máme za sebou několik odrážek, které nejsou číslované, a tím pádem dojde k resetování číslování.

Druhá situace je rozšířením první, tedy odrážka navíc obsahuje seznam (či seznamy). V takovém případě, jelikož je zpracování podseznamů řešeno rekurzí, se zpracování odrážky rozdělí na několik částí. Nejprve se zpracuje text, který je před zanořeným seznamem, a to stejným způsobem jako v situaci první. Následně se rekurzivně vytvoří podseznam. Nyní je nutné do vytvořeného podseznamu nezasahovat, jelikož ten již je formátovaný tak, jak je nutné. A přidat blok textu následující zanořený seznam, kterému se nastaví příslušné odsazení bez použití odrážek. Tímto způsobem je možné vytvořit seznam vizuálně odpovídající tomu v naší ukázce.

Zatím jsme si popsali způsob, jakým jsou vytvářeny číslované a nečíslované seznamy. Podívejme se tedy nyní na řešení seznamu definic. Seznam definic je tak trochu speciálním případem. Důvodem je, že seznam definic nepatří mezi podporované prvky většiny dnešních WYSIWYG editorů a PowerPoint není výjimkou. Jelikož není možné nastavit jako typ odrážky text, bylo nutné zvolit jiný přístup. Nakonec bylo zvoleno řešení tabulkou s dvěma sloupci. V prvním sloupci je umístěn definovaný termín a v druhém jeho definice. Právě z důvodu použití tabulky není možné zanořování.

6.3.4 Obrázky

Zpracování obrázků patří mezi jednodušší části celé implementace. Za zmínku tak stojí snad pouze způsob vyhledávání daných obrázků.

Jak již jsme si uvedli v kapitole 2.7, tak obrázky lze zadávat bez přípony a jsou hledány v adresáři, kde je zdrojový dokument umístěn. Také je možné specifikovat případné další adresáře, kde obrázek hledat. Samotné vyhledávání je řešeno v cyklu, kdy se testují na existenci soubory pro jednotlivé přípony postupně ve všech daných adresářích. Jakmile najdeme požadovaný obrázek, cyklus ukončíme (tedy nehledáme možné další). Kvůli použití tohoto přístupu je také pevně dána priorita jednotlivých přípon, která je následující: `jpg`, `jpeg`, `png` a `gif`. Bohužel, z důvodu nedostupnosti kvalitní volně dostupné knihovny pro extrakci obrázků z PDF, nejsou PDF obrázky podporovány.

6.3.5 Overlay a \pause

Aplikace overlay specifikací a zpracování příkazu `\pause` zasahuje do všech částí zpracování snímku. Zejména způsob zpracování příkazu `\pause` se může na první pohled jevit jako ne příliš průhledný. Je tedy vhodné jim věnovat zvýšenou pozornost.

Pro řešení byl použit obdobný přístup, jaký je používán třídou Beamer. Tedy jeden snímek rozgenerujeme na několik podsnímků[4], v závislosti na použitých overlay parametrech či použití příkazu `\pause`. Přístup, kdy by se pro realizaci používaly animace v aplikaci PowerPoint, byl zavržen hned na začátku. Animaci lze totiž nastavit pouze celému tvaru, a to by v našem případě, kdy overlay specifikací můžeme například změnit barvu pouze jednoho slova uprostřed textu, bylo v podstatě nepoužitelné.

Podívejme se tedy na samotný způsob realizace. Abychom mohli jeden snímek rozgenerovat postupně na několik podsnímků, musíme tuto činnost provádět v cyklu, dokud nejsou splněny určité podmínky. Těmito podmínkami jsou, že v daném podstromu reprezentujícím zpracovávaný snímek se nesmí vyskytovat nezpracovaný příkaz `\pause` a zároveň v rámci všech overlay specifikací v daném podstromu se nesmí vyskytovat větší, než je číslo aktuálního průchodu (tedy číslo aktuálního podsnímku).

Zpracování overlay specifikace se provádí pomocí jednoduchého zjištění, zda použitá specifikace obsahuje číslo aktuálního průchodu. Jestliže ano, příkaz se aplikuje, pokud ne, tak se ignoruje. Zvláštním případem jsou příkazy, u kterých overlay specifikace potlačuje i vykreslení jejich obsahu (například overlay u příkazu `\textbf` způsobí pouze potlačení formátování, kdežto u příkazu `\item` potlačí vykreslení celé odrážky). V našem případě se tato situace týká titulku a podtitulku snímku a také položek seznamů. V případě titulku a podtitulku je situace jednoduchá, v podstatě pouze ignorujeme výpis. Ovšem v případě položek seznamu potřebujeme, aby daný prostor byl zabraný, i když se obsah odrážky nevykresluje. Toto je řešeno tak, že danou odrážku fyzicky vykreslíme, ovšem následně jí nastavíme, v rámci formátování textu, atribut `Visible` na `false`.

Příkaz `\pause` na rozdíl od overlay specifikace přímo vynucuje přechod na další podsnímek. Zajímavostí je, že i když se to může zdát nesmyslné, je možné používat příkaz `\pause` i v rámci titulku snímku. Proto je, z důvodu odděleného zpracování titulku a obsahu snímku, nutné udržovat si pro daný snímek počítadlo zpracovaných `\pause`. Kromě tohoto "globálního" počítadla pro daný snímek potřebujeme ještě "lokální" počítadlo pro daný průchod. Poté jakmile při zpracování snímku narazíme na `\pause` inkrementujeme lokální počítadlo a pokud je jeho hodnota větší, než hodnota globálního, tak inkrementujeme také globální počítadlo, ukončíme aktuální průchod a vynutíme si průchod další. Ukončení aktuálního průchodu se provádí většinou okamžitě, jedinou výjimkou je případ, kdy je daný `\pause` v tabulce. U tabulek je totiž nutné řešit změnu velikosti podle jejich obsahu a šířka jednotlivých sloupců musí odpovídat i textu z nevykreslených řádků. Proto se tabulka vytvoří celá, změní se její velikost, poté se odstraní řádky, které na výstupu již

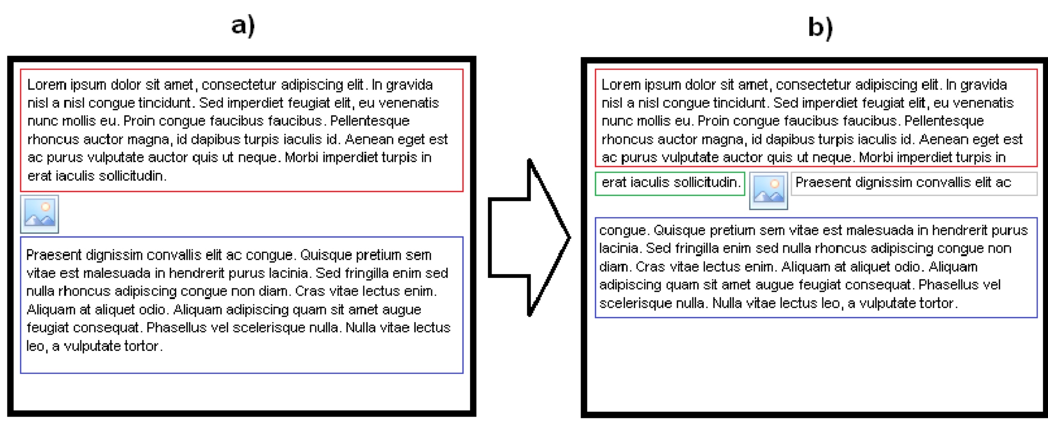
[4]jako podsnímek uvažujeme jednu z možných reprezentací podstromu definujícího daný snímek

být nemají, a teprve tehdy se provede ukončení aktuálního průchodu.

6.3.6 Přerovnání tvarů

Jak již jsme si osvětlili v rámci návrhu, tak PowerPoint neumí pracovat s elementy (obrázky, tabulky) vloženými doprostřed textu. Bylo tedy nutné toto základní[5] obtékání textem implementovat. Řešení však není tak jednoduché, jak by se mohlo na první pohled zdát. Abychom mohli vytvořit toto jednoduché obtékání, musíme v mnoha případech přistoupit k dělení příslušných textových tvarů a zároveň také k přesouvání jednotlivých tvarů v rámci snímku.

Generování jednotlivých tvarů v rámci snímku je implementováno tak, že nový tvar, ať již se jedná o obrázek, text, tabulku či seznam, se vloží vždy pod nejspodnější tvar (tedy pod poslední vložený). Příklad si můžeme prohlédnout na obrázku 6.1 a). Toto chování můžeme s jistotou označit za velmi vzdálené od požadovaného. Byla proto implementována metoda, pojmenovaná `Reshaper`, která se stará o rekonfiguraci[6] jednotlivých tvarů v rámci snímku.



Obrázek 6.1: Ukázka činnosti metody `Reshaper`; a) rozložení před, b) rozložení po

Metoda `Reshaper` je volána vždy, když je potřebné provést rekonfiguraci tvarů na snímku. Jelikož však umí zpracovat pouze dva tvary zároveň, je většinou volána v případě vložení nového tvaru do snímku. Tedy tvary na snímku rekonfigurujeme postupně během tvorby snímku, místo jedné velké rekonfigurace až po jeho vytvoření.

Podívejme se tedy nyní na způsob její činnosti. Jelikož toto základní obtékání se týká pouze textových tvarů, obrázků a tabulek (netextových tvarů), mohou nastat celkem tři různé situace pro přerovnání.

Prvním případem je situace, kdy první tvar je textový a druhý tvar je netextový. V tomto případě nejdříve zjistíme, zda se netextový tvar vejde za poslední řádek tvaru textového.

[5] základní proto, že text je obtékán v podstatě pouze jedním řádkem
[6] rozdělení, přesunutí, atd.

Ke zjištění této informace se využívá velikost ohraničujícího obdélníku posledního řádku textového tvaru a velikost netextového tvaru. Jestliže se netextový tvar na dané místo nevejde, tak již rekonfiguraci neprovádíme. Pokud však zjistíme, že se netextový tvar na dané místo vejde, musíme provést postupně několik kroků. Nejdříve extrahujeme poslední řádek z textového tvaru a umístíme jej do nového textového tvaru na stejnou pozici, kde se původně nacházel originální řádek. Poté tento nový tvar musíme zmenšit na šířku textu, který obsahuje. A jako poslední krok již přesuneme netextový tvar vedle nově vytvořeného textového.

Dalším případem je situace, kdy jak první, tak i druhý tvar je netextový. Tento případ je nejjednodušší ze všech realizovaných. V podstatě pouze zkontrolujeme, zda se druhý tvar vejde vedle tvaru prvního a v případě, že ano, tak jej přesuneme. Není tedy nutné počítat ohraničující obdélníky, či dělit tvary.

Třetí případ je variací případu prvního, tedy první tvar je netextový, zatímco druhý je textový. V tomto případě nejprve zjistíme, zda se první slovo z textového tvaru vejde za tvar netextový. Pokud nevejde, tak rekonfiguraci již neprovádíme. Pokud se však vejde, musíme opět provést několik kroků. Jako první krok provedeme přesunutí textového tvaru vedle netextového a upravení jeho šířky, tak aby zabíral pouze požadovanou část snímku (aby snímek nepřesahoval). Po tomto přesunutí PowerPoint automaticky upraví i rozložení textu do jednotlivých řádků. Díky tomu, můžeme vyjmout všechny řádky, kromě prvního a umístit je do nového tvaru, který je umístěn, jak pod netextovým, tak i pod textovým tvarem, který nyní obsahuje pouze jeden řádek textu.

Čtvrtým případem je situace, kdy potřebujeme přerovnat dva textové tvary umístěné za sebou. Avšak dříve jsme si uvedli, že mohou nastat pouze tři situace. Tento čtvrtý případ, který se logicky nabízí, nemůže kvůli principu implementace ve skutečnosti nastat, tedy jej vůbec neuvažujeme.

Pokud se nyní vrátíme k obrázku 6.1 b), můžeme si všimnout, že došlo celkem ke dvěma rekonfiguracím tvarů. Jako první byla provedena rekonfigurace popsaná prvním případem. Jak vidíme, tak z textového tvaru (ohraničen červenou barvou) byl odříznut poslední řádek a vložen do tvaru nového (ohraničen zelenou barvou). A následně byl přesunut obrázek za tento nový tvar. Poté byla provedena rekonfigurace popsaná třetím případem. Kdy z textového tvaru (ohraničen modrou barvou) byla přesunuta část prvního řádku do samostatného tvaru (ohraničen šedou barvou).

6.3.7 Řešení extrakce vnořených blokových prvků

Dříve jsme si uvedli, že z důvodu řešení rozdílnosti obou platforem, dovolíme uživateli vybrat si způsob zacházení s vnořenými blokovými prvky. Jeden z těchto způsobů je jejich extrakce a následné vložení na konec aktuálního snímku. Tato varianta je nabízena zejména uživatelům, kteří si plánují výslednou prezentaci ručně upravit. Popišme si tedy nyní, jak je řešena realizace této činnosti.

Provedení extrakce vnořených blokových prvků, je prováděno tak, že daný podstrom

nezpracujeme (tedy ani si nerozgenerujeme potomky na zásobník), ale místo toho jej uložíme do globálně udržované fronty. V této frontě jsou umístěny všechny extrahované blokové prvky. Poté již stačí pouze na konci zpracování snímku zpracovat prvky v této frontě umístěné. Důvodem pro použití fronty a ne jiné datové struktury pro udržení seznamu extrahovaných prvků, je zejména alespoň částečná snaha o zachování jejich pořadí. Může však nastat situace, kdy při zpracovávání extrahovaného prvku narazíme na další vnořený prvek (tedy prvek vnořený ve vnořeném prvku). V tomto případě tento vnořený prvek vložíme na konec fronty. Tedy již neřešíme pořadí (je velmi pravděpodobné, že uživatel si jednotlivé tvary sám přeorganizuje, tedy na jejich pořadí opravdu příliš nezáleží).

V rámci extrakce musíme řešit situace, kdy extrahovaný prvek obsahuje příkaz `\pause`. Tyto situace řešíme tak, že v rámci zpracování extrahovaných prvků příkazy `\pause` ignorujeme. Opět je brán zřetel na to, že pro uživatele je snadnější obsah odstranit, než jej přidávat.

6.4 Uživatelské rozhraní

Vzhled a rozložení ovládacích prvků uživatelského rozhraní jsme si již podrobně definovali v rámci návrhu. Avšak během implementace bylo učiněno dodatečné rozhodnutí o přidání dalšího prvku. Oproti návrhu, kdy jsme uvažovali použití pouze jednoho ukazatele průběhu, v rámci implementace přibyl další. Ve výsledku jsou tedy použity dva, jeden pro zobrazení průběhu převodu aktuálně zpracovávaného dokumentu a druhý pro zobrazení celkového průběhu (v rámci všech zpracovávaných dokumentů). Ostatní změny byly převážně kosmetického charakteru (přidání popisků). Zaměřme se tedy více na použité technologie a postupy.

Uživatelské rozhraní je realizováno pomocí *Windows Presentation Foundation* (WPF). Důvodem pro zvolení WPF, namísto *Windows Forms*, je zejména skutečnost, že umožňuje mnohem lepší oddělení prezentační vrstvy od aplikační. Jako malé plus také můžeme označit možnost měnit vzhled jednotlivých prvků uživatelského rozhraní pomocí různých témat.

Abychom dosáhli opravdu co nejlepšího oddělení jednotlivých vrstev byl pro implementaci zvolen návrhový vzor Model-View-ViewModel (MVVM). Jelikož se jedná o vcelku nový návrhový vzor a nemusí být příliš známý, popišme si stručně jeho princip (informace o MVVM byly čerpány z [14]). Podobně jako návrhový vzor MVC i MVVM dělí implementaci do tří pomyslných vrstev. První vrstvou je *Model*, ta se stará o práci s daty, popřípadě vykonává jiné nízko-úrovňové činnosti (v našem případě do této vrstvy patří načítání a práce s jednotlivými pluginy). Další vrstvou je *View*. Ta se stará o komunikaci s uživatelem. Poslední vrstvou je *ViewModel*, tato vrstva zprostředkovává komunikaci mezi ostatními vrstvami (*Model*, *View*). Tento přístup se na první pohled příliš neliší od toho používajícího MVC, ovšem je zde podstatný rozdíl. U návrhového vzoru MVVM *Model* ani *ViewModel* vůbec nepotřebují informaci, že nějaký *View* existuje. Místo toho se *View* naváže na veřejné vlastnosti, které poskytuje *ViewModel*. A následně se přes tyto vlast-

nosti automaticky synchronizují. Tímto způsobem je dosaženo velmi volné vazby mezi jednotlivými vrstvami, což přináší mnoho výhod.

Vraťme se nyní k implementovaným vlastnostem a schopnostem uživatelského rozhraní. V rámci návrhu jsme si naznačili, že by bylo vhodné implementovat i různá menší vylepšení pro zvýšení uživatelské přívětivosti. V aplikaci se tedy po pozdržení ukazatele myši nad některým z ovládacích prvků zobrazují jednoduché popisky s jeho činností (angl. *tooltip*). Byla také implementována funkcionalita *drag and drop* pro přetažení souborů do seznamu ke zpracování. V rámci zvýšené použitelnosti se jednotlivé soubory vložené do seznamu ke zpracování validují, zda je daný plugin dokáže zpracovat, a podle výsledku validace se podbarví buď červenou, či zelenou barvou (červená - nepodporuje, zelená - podporuje). Vcelku významná část chování uživatelského rozhraní je, že v případě, kdy uživatel spustí převod daných dokumentů, se uživatelské rozhraní automaticky přepne na záložku zobrazující průběh převodu. Zároveň se také zablokují veškeré ovládací prvky, které souvisí s nastavením parametrů převodu, přidání dokumentu do seznamu ke zpracování či spuštění převodu. Toto chování je implementováno z důvodu, aby uživatel neměnil chování zvoleného pluginu během probíhajícího převodu. Jediným funkčním ovládacím prvkem během probíhajícího převodu, je pouze tlačítko pro ukončení aktuálně prováděné dávky.

Prozatím se všechny informace, které jsme si v této podkapitole uvedli, týkaly pouze grafického uživatelského rozhraní. Podívejme se tedy na realizaci konzolového přístupu. Nejspíš první zásadní vlastností, které si při pohledu na způsob realizace všimneme, je, že celý konzolový přístup je naprosto oddělen od grafického uživatelského rozhraní a je přesunut do samostatného spustitelného souboru. Důvodem pro toto oddělení je, že se ukázalo poněkud složité ve WPF aplikaci potlačit zobrazení uživatelského rozhraní. Ovšem přináší to i pár výhod. Například bylo možné zvolit krátké jméno spustitelného souboru a navíc se při spuštění nemusí načítat spousty knihoven zajišťujících uživatelské rozhraní.

Jak jsme si již uvedli v kapitole 6.1, byla pro zpracování parametrů příkazové řádky použita knihovna *NDesk.Options*. Díky tomu je možné parametry zadávat mnoha různými, již dobře zažitými způsoby. Jedná se zejména o uvození parametru pomocí jednoho z kombinace znaků `?`, `/`, `-` či `--`. Dále je například podporováno přepínání voleb připojením znaku `+` či `-` a spousty dalších konvencí zápisu parametrů.

I v případě implementace konzolového přístupu bylo potřeba vyřešit pár nepříjemností. Jde o způsob výběru pluginu a nastavení parametrů převodu.

Způsob výběru pluginu jsme si již částečně vyřešili v rámci návrhu. Jediné co zbývalo vyřešit, je způsob generování klíče. Právě na klíč pluginu je kladeno několik požadavků:

1. klíč pro daný plugin musí být stejný i po opětovném spuštění aplikace
2. každý plugin musí mít unikátní klíč
3. klíč by měl být generován automaticky
4. klíč musí být relativně krátký, z důvodu zadávání uživatelem

Nakonec je tedy generování klíče řešeno tak, že se nejdříve vygeneruje MD5 hash pro název a verzi pluginu. Tento hash se následně rozdělí na čtyři 32 bitů dlouhé části, z nichž se za pomoci operace XOR, vypočítá jedno 32 bitů dlouhé číslo. Jelikož však i toto číslo je pro uživatele příliš dlouhé, provede se nakonec převod do číselné soustavy o základu 36 (tedy výstup obsahuje znaky `0-9` a `a-z`). Díky tomu získáme klíč, jehož maximální délka je přibližně šest až sedm znaků, což již lze označit za přijatelné. Celý tento proces probíhá automaticky a programátoři pluginů tedy tuto činnost nemusí řešit.

V rámci nastavení parametrů převodu je nutné řešit způsob, jakým tyto parametry předat samotnému pluginu. V případě grafického uživatelského rozhraní byla situace jednoduchá, protože si veškeré nastavení obstarával plugin sám. Jelikož aplikace nemá a ani nemůže mít znalosti jaké parametry plugin přijímá, je řešení realizováno následujícím způsobem. V samotné aplikaci zpracujeme všechny známé parametry, následně, pokud je v rámci známých parametrů zvolen příslušný plugin, se všechny neznámé parametry předají pro zpracování pluginu. Ten z této množiny parametrů zpracuje jemu známé a všechny přebytečné vrátí zpět hlavní aplikaci. Důvod pro vracení přebytečných parametrů zpět hlavní aplikaci je ten, že mezi těmito přebytečnými parametry se vyskytují i názvy jednotlivých dokumentů určených k převodu (seznam souborů se uvádí jako poslední z parametrů, ale bez jakéhokoli uvození, tedy není jej možné zpracovat jako parametr).

6.5 Shrnutí

Postupně jsme si uvedli, jaké technologie a nástroje jsou použity pro implementaci. Vysvětlili jsme si, jakým způsobem je řešena lexikální a syntaktická analýza dokumentu. Osvětlili jsme si způsob převodu jednotlivých prvků dokumentu. Díky tomu bychom se nyní měli mnohem lépe orientovat ve zdrojových kódech celé aplikace a zároveň chápat způsob fungování jednotlivých jejich částí.

Kapitola 7

Testování

Testování je bezesporu jednou z důležitých oblastí životního cyklu software. Hlavně při vývoji větších aplikací jeho význam dále stoupá. V rámci vývoje námi vytvářeného překladače tedy bylo nutné se testování věnovat.

Celá aplikace a zejména převod dokumentů byl testován manuálně. K použití jednotkových testů se z důvodu samotné činnosti aplikace nepřistoupilo (výstup bylo nutné vizuálně kontrolovat, tedy použití jednotkových testů je k tomuto účelu ne příliš vhodné). Samotné testování probíhalo zároveň s vývojem aplikace, kdy po přidání nové funkcionality byla tato otestována. Tedy prováděly se testy na zpracování jednotlivých prvků dokumentu, a zda nově přidaná funkcionalita nekoliduje s již implementovanými částmi. Pro testování byla použita metoda *černé skříňky* (i když v některých případech by se dalo říct, že probíhalo testování i formou *šedé skříňky*). Nakonec bylo provedeno několik jednoduchých testů na zpracování jednotlivých prvků (tyto testy jsou přiloženy na CD).

Během testování se podařilo odhalit několik více i méně závažných problémů. Ze závažnějších si můžeme uvést například pád aplikace při pokusu o spojení pouze jednoho sloupce tabulky (tedy v podstatě žádné spojení), či nesprávné prohledávání cest k obrázkům. Z méně závažných, například neignorování mezery po uvedení příkazů pro změnu stupně písma (`\Large`, `\huge`, `\tiny` atd.).

Kromě odhalování chyb se v rámci testování přistoupilo také k refaktorizaci. Díky tomu, je výsledný kód aplikace přehlednější a lépe se v něm hledají chyby.

Bohužel, i přes důkladné testování jsou do výsledné aplikace zaneseny problémy, které jsou dané chováním aplikace PowerPoint a tak i přes veškerou snahu, je nebylo možné eliminovat. Část těchto problémů se snaží aplikace odstranit různými oklikami, ovšem ani tato řešení bohužel nejsou účinná vždy.

7.1 Omezení výsledné aplikace

Výsledná aplikace má několik omezení, které plynou ať již z implementace pouze podmnožiny vstupního jazyka, použití LALR(1) syntaktického analyzátoru pro analýzu vstupního

dokumentu, nedostupnosti prostředků pro realizaci, či z problémů zanesených samotnou aplikací PowerPoint. Popišme si tedy jednotlivá omezení.

Omezení plynoucí z implementace podmnožiny vstupního jazyka a použití LALR(1) syntaktického analyzátoru:

- není podporován příkaz `\newcommand` — možným řešením by například mohlo být přesunutí větší části zpracování do sémantických akcí. Toto řešení by zároveň vyžadovalo úpravu lexikální analýzy tak, aby neprováděla filtraci neznámých příkazů.
- v tabulce obsahující horní ohraničení musí být případný příkaz `\pause` umístěn až za definici tohoto ohraničení (tedy za příkaz `\hline`)
- není podporováno prostředí `math` — v případě, že se narazí na prostředí `math`, tak se celý jeho obsah zpracuje jako neformátovaný text. Tedy nezpracovávají se jednotlivé příkazy. Díky tomu je možné pomocí doplňků[1] pro PowerPoint dosáhnout přeložení na rovnice (ovšem tuto činnost je potřeba provádět manuálně ve výsledném dokumentu).

Omezení plynoucí z nedostupnosti prostředků pro realizaci:

- nejsou podporovány obrázky umístěné v PDF — důvodem je nedostupnost kvalitní a zejména udržované knihovny pro extrakci obrázků z PDF. Většina kvalitnějších knihoven je dostupná pouze za poplatek.

Omezení a problémy zanesené aplikací PowerPoint:

- jestliže otevřeme PowerPoint příliš brzo po skončení (či během) převodu, může dojít k tomu, že se po několika vteřinách sám ukončí. Toto je způsobeno tím, že aplikace spouští PowerPoint pro tvorbu výstupního dokumentu a po ukončení překladu jej opět vypíná. Avšak samotné vypnutí může být časově náročnější, a tedy tento proces, kdy se PowerPoint vypíná, může dobíhat i po skončení převodu. V závislosti na výkonnosti počítače by mělo dojít k vypnutí aplikace PowerPoint nejpozději přibližně do 30 vteřin od ukončení posledního převodu.
- jak již bylo uvedeno v kapitole 6.3.2, může dojít k situaci, kdy jsou tabulky nesprávně zmenšeny. Bohužel je implementované řešení jediné, které alespoň z části funguje. Chyba se v tomto případě projevuje v samotné aplikaci PowerPoint, kdy vrácená velikost ohraničujícího obdélníku neodpovídá skutečné (důvod, proč k tomuto chování dochází, se bohužel nepodařilo zjistit). Situaci, kdy k tomuto dojde, je možné řešit například nastavením zarovnání do bloku s uvedením velikosti pro daný sloupec a následným ručním upravením zarovnání ve výsledném dokumentu.

Toto jsou známé problémy a omezení.

[1] například TexPoint: `http://texpoint.necula.org/`

7.2 Distribuce aplikace

Po ukončení testování bylo nutné se zamyslet nad možnostmi distribuce výsledné aplikace. Jelikož primární platformou, pro kterou je aplikace určena, je Microsoft Windows. Bylo vhodné zvolit způsob distribuce, který je pro uživatele této platformy přirozený. Zvoleny byly způsoby dva. Prvním je klasický instalační balíček a druhým je distribuce pomocí archivu ZIP (tedy bez nutnosti instalace).

Instalační balíček je vytvářen v rámci samotného projektu pomocí *Visual Studio Installer*. Tato varianta zahrnuje i přidání zástupce na aplikaci do nabídky *Start* a na plochu uživatele.

Distribuce pomocí archivu ZIP, byla zvolena zejména pro uživatele, kteří chtějí aplikaci používat, avšak nemají potřebu si ji nainstalovat. Tento distribuční balíček je vytvářen automaticky při sestavení *Release* verze aplikace. Pro jeho sestavení se používá *post-build* událost v rámci hlavního projektu. Pro vytváření archivu je používán nástroj 7zip[2].

7.3 Shrnutí

V rámci této kapitoly jsme si popsali, jaké techniky testování byly použity a v jakých fázích vývoje se testování provádělo. Vyjmenovali jsme si známé problémy a omezení, které jsou důsledkem implementace pouze podmnožiny vstupního jazyka, či dokonce problémů se samotnou aplikací PowerPoint. A nakonec jsme si zvolili způsob distribuce finální aplikace. Mnohé může nyní napadnout, že tímto vývoj aplikace a testování končí. Avšak pokud vytváříme aplikaci, kterou mají lidé opravdu používat, pravděpodobně neukončíme vývoj po vydání první verze, ale budeme v něm pokračovat, dokud bude o ni ze strany uživatelů zájem.

[2]dostupný na: `http://www.7-zip.org`

Kapitola 8

Závěr

Vývoj kvalitního překladače, ať již se jedná o překladač dokumentů či zdrojových kódů, si žádá nejen znalosti v oblasti samotných překladačů, ale zejména také dobrou znalost jak vstupní, tak i výstupní platformy.

V rámci této práce jsme si postupně představili obě platformy, mezi kterými bude překlad probíhat. Naznačili jsme si různé postupy a příkazy, pomocí nichž lze vytvořit nejrůznější prezentace využívající třídu Beamer platformy LaTeX. Nastínili jsme si postup tvorby prezentace platformy Microsoft PowerPoint pomocí dvou různých technik, a to automatizace pomocí PowerPoint PIA a vytvoření dokumentu pomocí Open XML SDK obalující (nejen) PresentationML.

Také jsme si uvedli dva nástroje usnadňující vytvoření přední části překladače (tedy lexikální a syntaktickou analýzu), jejichž výstupem je kód v jazyce C#. Jsou jimi generátor kódu lexikálního analyzátoru GPLEX a generátor kódu syntaktického analyzátoru GPPG.

Představili jsme si několik nástrojů, provádějících podobnou činnost. A na základě požadavků, které jsme si stanovili po zvážení kladů a záporů existujících aplikací, jsme postupně provedli návrh, kde jsme si rozebrali různé problémy a jejich možná řešení. Popsali jsme implementaci, kde jsme se věnovali způsobu řešení jednotlivých částí aplikace. Nakonec bylo diskutováno testování výsledné aplikace, kde jsme si uvedli známá omezení a problémy.

V rámci návrhu a implementace může být velkým usnadněním využití znalostí zejména ze dvou předmětů magisterského studia. Jedná se o předmět Teoretická informatika, z kterého využijeme především znalosti z oblasti regulárních a bezkontextových gramatik a jazyků. A předmět Výstavba překladačů, který tyto znalosti přenáší do praktické roviny jako je návrh lexikálního a syntaktického analyzátoru, ale také nás seznámí se základem práce s nástroji typu LEX a YACC.

Výstupem této práce je funkční aplikace umožňující převod prezentací vytvořených pomocí třídy Beamer platformy LaTeX do formátu používaného aplikací Microsoft PowerPoint. Díky modularitě aplikace je nepřeberné množství možností dalšího vývoje, například vytvoření nových pluginů pro převod mezi dalšími formáty. Budoucí vývoj je však možné směřovat i na samotný plugin pro převod mezi třídou Beamer platformy LaTeX a Microsoft

PowerPoint. Vzhledem k rozsahu vstupní platformy se nabízí nepřeberné množství různých rozšíření, jako jsou například podpora sazby matematických rovnic, prostředí `picture` či dokonce prostředí `tikz`.

Literatura

[1] Elmue: *PowerPointCreator*. [online], 2009-01-19 [cit. 2010-11-29].
URL http://www.codeproject.com/KB/office/PowerPointCreator.aspx

[2] Gough, J.: *The GPLEX Scanner Generator (Version 1.1.4 May 2010)*. [online], 2010-11-01 [cit. 2010-12-28].
URL http://gplex.codeplex.com/releases/view/54942#DownloadId=162616

[3] Gough, J.; Kelly, W.: *The GPPG Parser Generator (Version 1.4.3 November 2010)*. [online], 2010-11-02 [cit. 2010-12-28].
URL http://gppg.codeplex.com/releases/view/54986#DownloadId=162824

[4] Grätzer, G.: *More Math Into LaTeX (4th Edition)*. Springer, 2007, ISBN 978-0-387-32289-6.

[5] John R. Levine, D. B., Tony Mason: *Lex & yacc. 2nd ed.* O'Reilly, 1992, ISBN 1-56592-000-7.

[6] Jones, B.: *Intro to PresentationML part 1 - Core architecture and the "presentation" part*. [online], 2006-04-11 [cit. 2010-12-26].
URL http://blogs.msdn.com/b/brian_jones/archive/2006/04/11/573529.aspx

[7] Microsoft Developer Network: *PowerPoint Solutions*. [online], 2010-05 [cit. 2010-11-29].
URL http://msdn.microsoft.com/en-us/library/bb772069.aspx

[8] Microsoft Developer Network: *PowerPoint Object Model Reference*. [online], [cit. 2010-11-29].
URL http://msdn.microsoft.com/en-us/library/bb265987(office.12).aspx

[9] Microsoft Developer Network: *Working with PresentationML Documents*. [online], [cit. 2010-11-30].
URL http://msdn.microsoft.com/cs-cz/library/gg278318(en-us).aspx

[10] Microsoft Developer Network: *Structure of a PresentationML Document*. [online], [cit. 2010-12-26].
URL http://msdn.microsoft.com/en-us/library/gg278335.aspx

[11] Microsoft Developer Network: *Presentations*. [online], [cit. 2010-12-27].
URL `http://msdn.microsoft.com/cs-cz/library/cc850828(en-us).aspx`

[12] Rybička, J.: *LaTeX pro začátečníky*. Konvoj, 2003, ISBN 80-7302-049-1.

[13] Singh, M.: *Demystifying Microsoft Office Object Model – Part 2*. [online], 2010-05-23 [cit. 2010-12-25].
URL `http://www.manvirsingh.net/index.php/demystifying-microsoft-office-object-model-part-2/`

[14] Smith, J.: *WPF Apps With The Model-View-ViewModel Design Pattern*. [online], 2009-02 [cit. 2011-04-27].
URL `http://msdn.microsoft.com/en-us/magazine/dd419663.aspx`

[15] Tantau, T.; Wright, J.; Miletić, V.: *The BEAMER class, User Guide for version 3.10*. [online], 2010-07-12 [cit. 2010-11-05].
URL `http://www.ctan.org/tex-archive/macros/latex/contrib/beamer/doc/beameruserguide.pdf`

Příloha A

Obsah CD

- Text této práce, včetně zdrojových kódů pro LaTeX
- Kompletní zdrojové kódy pro příklady uvedené v kapitole 3
- Zdrojové kódy několika ukázkových prezentací
- Zdrojové kódy aplikace
- Generovaná programová dokumentace ke zdrojovým kódům aplikace
- Generované diagramy tříd pro jednotlivé balíčky (projekty)
- Spustitelná aplikace
- Distribuční balíčky (instalátor a ZIP archiv)

Příloha B

Manuál

V rámci této uživatelské příručky se zaměříme především na ovládání samotné aplikace a možnosti nastavení parametrů převodu pluginu pro převod dokumentů mezi třídou Beamer a platformou Microsoft PowerPoint.

B.1 Instalace

Způsob instalace aplikace závisí na tom, jaký distribuční balíček jsme si zvolili. V případě, že jsme si zvolili klasický instalátor, tak nás celým procesem instalace provede jednoduchý průvodce. Pokud jsme si zvolili ZIP archiv, tak stačí program pouze extrahovat do zvolené lokace a je ihned připraven k použití.

B.1.1 Závislosti

Samotná aplikace i její pluginy potřebují k fungování na počítači nainstalované určité knihovny, či aplikace.

Pro spuštění aplikace je nutné mít nainstalován .NET 4.0 Client Profile. V případe, že jsme aplikaci instalovali pomocí instalátoru, tak ten kontroluje zda je na počítači .NET 4.0 Client Profile instalovaný. Pokud není, nabídne jeho instalaci. Jestliže jsme aplikaci extrahovali ze ZIP archivu, tak si jej musíme případně doinstalovat sami.

Pro umožnění převodu dokumentů pomocí pluginu pro převod z třídy Beamer na PowerPoint je nutné mít na počítači nainstalován kancelářský balík Microsoft Office 2007 či novější.

B.2 Ovládání aplikace

Podívejme se nyní na ovládání a práci se samotnou aplikací.

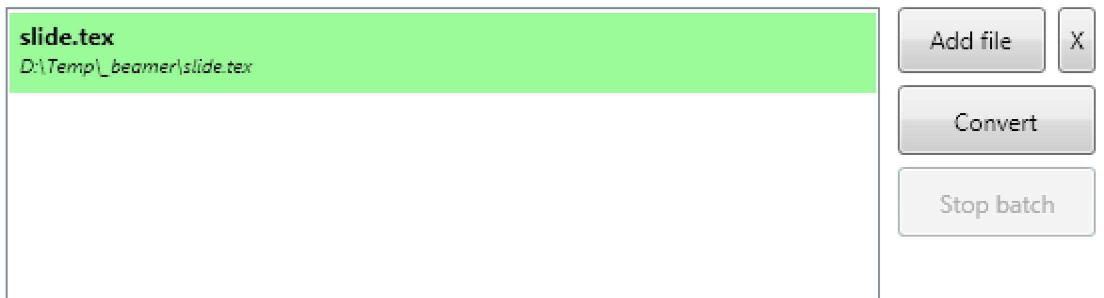
B.2.1 Výběr pluginu

Výběr pluginu se provádí jednoduchým výběrem ze seznamu dostupných pluginů. Seznam dostupných pluginů je zobrazen na obrázku B.1.

Obrázek B.1: Seznam dostupných pluginů

B.2.2 Seznam souborů k převodu

V seznamu souborů k převodu jsou umístěny soubory, které chceme převádět. Seznam souborů je zobrazen na obrázku B.2. Přidat soubor do seznamu můžeme buď pomocí tlačítka *Add file*, nebo jednoduchým přetažením souboru (popř. souborů). Soubor ze seznamu odstraníme tak, že jej označíme a buď použijeme tlačítko *X*, nebo stiskneme klávesu *Del*.



Obrázek B.2: Seznam souborů k převodu

Při přidání souboru do seznamu automaticky proběhne validace, zda aktuálně zvolený plugin dokáže daný soubor zpracovat. Jestliže ano, tak je soubor podbarven zelenou barvou, v případě že ne, je podbarven barvou červenou.

B.2.3 Výstupní adresář

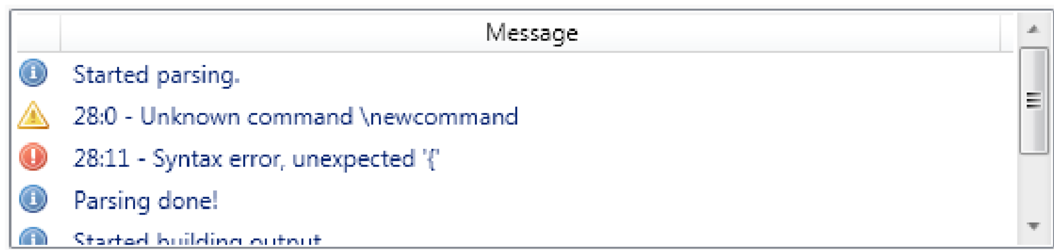
Výstupní dokumenty převodu se ukládají do výstupního adresáře, jejich název je vytvořen automaticky z názvu vstupního dokumentu a přípony výstupního formátu. Pozor, **jestliže dokument s daným jménem již existuje, tak je přepsán**.

Nastavení výstupního adresáře je možné provést manuálním zapsáním cesty, či použitím dialogu *Procházet* spuštěného tlačítkem *Browse*. Nastavení výstupního adresáře je zobrazeno na obrázku B.3.

Obrázek B.3: Nastavení výstupního adresáře

B.2.4 Spuštění, průběh a zastavení převodu

Spuštění převodu dokumentů se provádí pomocí tlačítka *Convert*. Jakmile se spustí převod, tak se aplikace automaticky přepne na záložku zobrazující průběh převodu. Kromě ukazatelů průběhu aplikace také informuje o průběhu převodu textově, a to pomocí krátkých zpráv (obrázek B.4).



Obrázek B.4: Seznam zpráv o průběhu převodu

Jednotlivé typy zpráv jsou rozlišeny použitou ikonou:

- popisuje informaci o spuštění, či dokončení určité fáze převodu a podobné zprávy informačního charakteru.
- popisuje varování např. o neznámém příkazu ve zdrojovém dokumentu
- popisuje chybu v rámci převodu dokumentu. V některých případech (viz obrázek B.4) se podaří provést zotavení z dané chyby a převod pokračuje dále.

Aplikace umožňuje ukončení provádění aktuální dávky. To se provádí kliknutím na tlačítko *Stop batch*. Po kliknutí na toto tlačítko, se po dokončení aktuálně probíhajícího převodu, již nebude provádět převod dalších dokumentů ze seznamu.

B.2.5 Konzolové rozhraní

Aplikace taktéž umožňuje práci pouze v konzolovém režimu. Na rozdíl od klasického uživatelského rozhraní se konzolová verze spouští pomocí souboru `sc.exe`. Je možné zvolit hned několik různých přepínačů, které se zadávají klasickým způsobem, tedy uvozené znakem `-` či `/`. Jejich seznam je následující (znak `|` odděluje možné varianty přepínače):

- `h|?|help` - vypíše nápovědu pro použití aplikace a skončí. V případě, že je definován pomocí příslušného přepínače použitý plugin, tak vypíše nápovědu pro něj.

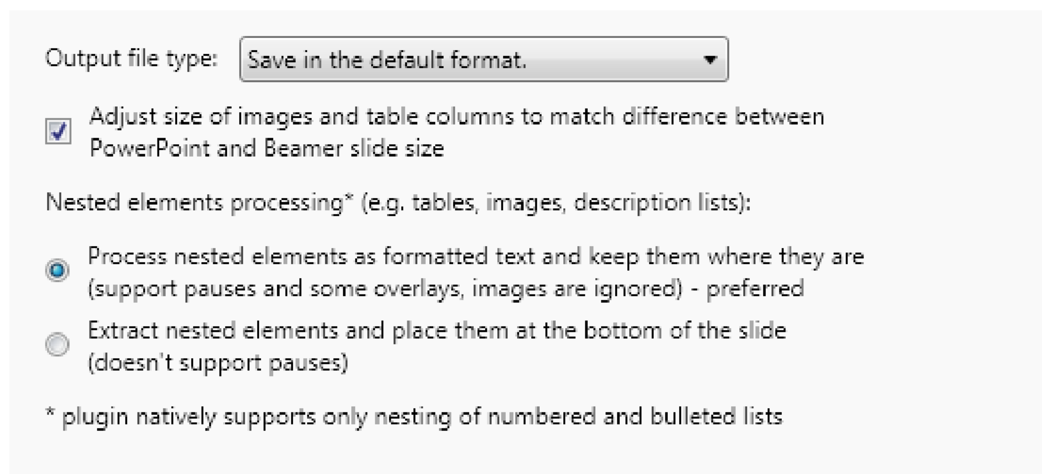
- `l|list` - vypíše seznam dostupných pluginů včetně jejich klíčů a skončí.
- `o|output` - má jeden parametr a nastaví výstupní adresář. Tento přepínač je nepovinný, a pokud není uveden použije se podadresář `output` v aktuálním pracovním adresáři.
- `p|plugin` - má jeden parametr, který specifikuje použitý plugin (klíč).

B.3 Plugin pro převod z třídy Beamer na PowerPoint

Zaměřme se nyní na práci s pluginem pro převod dokumentů mezi třídou Beamer a platformou Microsoft PowerPoint.

B.3.1 Nastavení parametrů převodu

Plugin umožňuje částečně ovlivnit podobu a formát výstupního dokumentu. Jak si můžeme na obrázku B.5 všimnout, tak lze vybrat formát výstupního souboru (prezentace, obrázky, HTML, PDF).



Obrázek B.5: Možnosti nastavení parametrů převodu

Dalším parametrem, který je možné zvolit (*Adjust size ...*), je zda má plugin přepočítávat velikosti zadávaných rozměrů tak, aby výsledek přibližně visuálně odpovídal prezentaci generované třídou Beamer. Důvodem pro toto přepočítávání je rozdílná velikost snímku používaná třídou Beamer a prezentací pro PowerPoint.

Posledním parametrem, je způsob zpracování vnořených elementů. PowerPoint nedokáže vnořené elementy zpracovat a plugin se tedy tuto situaci snaží řešit různými způsoby. Můžeme si tedy vybrat buď, že se vnořené elementy zpracují jako obyčejný formátovaný text, nebo se vyextrahují a umístí se na konec snímku.

B.3.2 Konzolové rozhraní

Plugin akceptuje některé parametry převodu při použití konzolového rozhraní. Jedná se o následující přepínače (opět znak `|` odděluje možné varianty):

- `n|nadjust` - slouží k zakázání úpravy velikosti obrázků a tabulek (potlačuje výchozí chování)
- `e|extract` - slouží k zvolení extrakce zanořených blokových elementů (potlačuje výchozí chování)

Příloha C

Regulární výrazy použité pro lexikální analýzu

```
// Zkratky, výraz uvedený vlevo slouží jako zkratka pro regulární výraz vpravo
// zkratka se poté uvádí ve složených závorkách
// ----------------------------------------------------------------------------
ws          [ \t]*
wsp         [ \t]+
wsl         {ws}\r?\n?{ws}
envBegin    \\begin{wsl}
envEnd      \\end{wsl}

// Preambule
// ----------------------------------------------------------------------------
\\documentclass { BEGIN(pre_optional); return DOCUMENTCLASS; }
\\usepackage    { BEGIN(pre_optional); return USEPACKAGE; }
\\title         { return TITLE; }
\\author        { BEGIN(pre_optional); return AUTHOR; }
\\institute     { BEGIN(pre_optional); return INSTITUTE; }
\\today         { return TODAY; }
\\date          { return DATE; }

// Prostředí ( + příkazy specifické pro prostředí)
// ----------------------------------------------------------------------------
{envBegin}\{document\}      inBody = true; BEGIN(body); return BEGIN_DOCUMENT;
{envEnd}\{document\}        inBody = false; BEGIN(INITIAL); return END_DOCUMENT;
{envBegin}\{frame\}         return BEGIN_FRAME;
{envEnd}\{frame\}           return END_FRAME;
{envBegin}\{itemize\}       BEGIN(pre_optional); return BEGIN_ITEMIZE;
{envEnd}\{itemize\}         return END_ITEMIZE;
{envBegin}\{enumerate\}     BEGIN(pre_optional); return BEGIN_ENUMERATE;
{envEnd}\{enumerate\}       return END_ENUMERATE;
{envBegin}\{description\}   BEGIN(pre_optional); return BEGIN_DESCRIPTION;
{envEnd}\{description\}     return END_DESCRIPTION;
{envBegin}\{tabular\}\{     tabular = true; tbl = 0; BEGIN(tabular_arg); return ↩
    BEGIN_TABULAR;
{envEnd}\{tabular\}         tabular = false; return END_TABULAR;
\\item                      BEGIN(pre_overlay); return ITEM;
\\multicolumn               return MULTICOLUMN;
```

```
// Specifické příkazy třídy Beamer
// ----------------------------------------------------------------
\\frame          { BEGIN(pre_optional); return FRAME; }
\\frametitle     { BEGIN(pre_overlay); return FRAMETITLE; }
\\framesubtitle  { BEGIN(pre_overlay); return FRAMESUBTITLE; }
\\pause          { return PAUSE; }
\\usetheme       { return USETHEME; }

// Formátování textu (smíšená sazba)
// ----------------------------------------------------------------
\\textbf         { BEGIN(pre_overlay); return TEXTBF; }
\\texttt         { BEGIN(pre_overlay); return TEXTTT; }
\\textit         { BEGIN(pre_overlay); return TEXTIT; }
\\textsc         { BEGIN(pre_overlay); return TEXTSC; }
\\bfseries       { return BFSERIES; }
\\ttfamily       { return TTFAMILY; }
\\itshape        { return ITSHAPE; }
\\scshape        { return SCSHAPE; }
\\tiny           { return TINY; }
\\scriptsize     { return SCRIPTSIZE; }
\\footnotesize   { return FOOTNOTESIZE; }
\\small          { return SMALL; }
\\normalsize     { return NORMALSIZE; }
\\large          { return LARGE; }
\\Large          { return LARGE2; }
\\LARGE          { return LARGE3; }
\\huge           { return HUGE; }
\\Huge           { return HUGE2; }
\\color          { BEGIN(pre_overlay); return COLOR; }
\\textcolor      { BEGIN(pre_overlay); return TEXTCOLOR; }
\\underline      { BEGIN(pre_overlay); return UNDERLINE; }
\\and            { return AND; }

// Obrázky
// ----------------------------------------------------------------
\\includegraphics    { BEGIN(pre_optional); return INCLUDEGRAPHICS; }
\\graphicspath       { return GRAPHICSPATH; }

// Ostatní
// ----------------------------------------------------------------
\\titlepage      { return TITLEPAGE; }
\\hline          { return HLINE; }
\\cline          { return CLINE; }
\\section        { return SECTION; }
\\subsection     { return SUBSECTION; }
\\subsubsection  { return SUBSUBSECTION; }
\\LaTeX[ ]?      { BEGIN(str); /* .. Zpracování textu */ }

// Nový odstavec
\\\\|\\cr     { BEGIN(pre_optional); if(tabular) return ENDROW; else return NL; }

// Escapované speciální znaky
\\#                      BEGIN(str); /* .. Zpracování textu */
\\\$                     BEGIN(str); /* .. Zpracování textu */
\\%                      BEGIN(str); /* .. Zpracování textu */
\\textasciicircum\{\}    BEGIN(str); /* .. Zpracování textu */
```

```
\\&                     BEGIN(str); /* .. Zpracování textu */
\\_                     BEGIN(str); /* .. Zpracování textu */
\\\{                    BEGIN(str); /* .. Zpracování textu */
\\\}                    BEGIN(str); /* .. Zpracování textu */
\\~\{\}                 BEGIN(str); /* .. Zpracování textu */
\\textbackslash\{\}     BEGIN(str); /* .. Zpracování textu */
\\textpipe(\{\})?       BEGIN(str); /* .. Zpracování textu */

// Začátek a konec bloku
\{                      { return '{'; }
\}                      { return '}'; }
&                       { return '&'; }

// Komentáře
%.*\r?\n?{ws}           // ignoruj

// Overlay specifikace
// ---------------------------------------------------------------
<pre_overlay> {
        {wsl}<              BEGIN(overlay); /* .. Zpracování textu */
        {wsl}[^<]           BEGIN(pre_optional);
    }
<overlay> {
        [^>]*               /* .. Zpracování textu */
        >                   BEGIN(pre_optional); return OVERLAY;
    }

// Volitelné parametry
// ---------------------------------------------------------------
<pre_optional> {
        {wsl}\[             BEGIN(optional); /* .. Zpracování textu */
        {wsl}[^[]           if(inBody) BEGIN(body); else BEGIN(INITIAL);
    }
<optional> {
        [^\]]*              /* .. Zpracování textu */
        \]                  if(inBody) BEGIN(body); else BEGIN(INITIAL); ↩
            return OPTIONAL;
    }

// Nastavení sloupců prostředí tabular
// ---------------------------------------------------------------
<tabular_arg> {
        \{                  tbl++; /* .. Zpracování textu */
        \}                  {
                                tbl--;
                                if(tbl < 0) {
                                    BEGIN(INITIAL);
                                    return STRING;
                                }
                            }
        [^{}]*              /* .. Zpracování textu */
}

// Obyčejný text
// ---------------------------------------------------------------
<body> [^#\$%\^&_\{\}\\]           BEGIN(str); /* .. Zpracování textu */
```

```
[^#\$%\^&_\{\}\\[:IsWhiteSpace:]]     BEGIN(str); /* .. Zpracování textu */

<str> {
        [^#\$%\^&_\{\}~\\ \n\t\r]*       /* .. Zpracování textu */
        " "|\t                           /* .. Zpracování textu */
        \r                               // ignoruj
        ~                                /* .. Zpracování textu */
        \n                               /* .. Zpracování textu */

        // Escapované speciální znaky
        \\#                              /* .. Zpracování textu */
        \\\$                             /* .. Zpracování textu */
        \\%                              /* .. Zpracování textu */
        \\textasciicircum\{\}            /* .. Zpracování textu */
        \\&                              /* .. Zpracování textu */
        \\_                              /* .. Zpracování textu */
        \\\{                             /* .. Zpracování textu */
        \\\}                             /* .. Zpracování textu */
        \\~\{\}                          /* .. Zpracování textu */
        \\textbackslash\{\}              /* .. Zpracování textu */
        \\textpipe(\{\})?                /* .. Zpracování textu */
        \\LaTeX[ ]?                      /* .. Zpracování textu */
        %.*\n?{ws}                       // ignoruj komentář

        // Konec obyčejného textu
        [#\$\^&_\{\}\\]              {
                                         if(inBody)
                                             BEGIN(body);
                                         else
                                             BEGIN(INITIAL);
                                         return STRING;
                                     }
    }
```

Jak si v definici regulárních výrazů použitých pro lexikální analýzu dokumentu můžeme všimnout, tak je použito několik různých stavů. Všechny stavy, kromě stavu `body` jsou *exkluzivní*, tedy jakmile do tohoto stavu přejdeme, zpracováváme pouze regulární výrazy v něm definované. Stav `body` je naopak *inkluzivní* a tedy zpracovává i regulární výrazy určené pro stav `INITIAL`. Je v podstatě startovací podmínkou pro zpracování řetězce textu a používá se ke zpracování bílých znaků v rámci těla dokumentu (jelikož bílé znaky mimo hlavní tělo dokumentu potřebujeme ignorovat, kdežto ty uvnitř potřebujeme zpracovat).

Příloha D

Bezkontextová gramatika pro zpracování dokumentu

⟨document⟩→ ⟨documentclass⟩⟨preambule⟩⟨body⟩

⟨documentclass⟩→ **DOCUMENTCLASS**⟨optional⟩**{STRING}**

⟨preambule⟩→ ϵ
| ⟨preambule⟩**USEPACKAGE**⟨optional⟩**{STRING}**
| ⟨preambule⟩⟨optional⟩**USETHEME{STRING}**
| ⟨preambule⟩⟨titlesettings⟩
| ⟨preambule⟩**GRAPHICSPATH{**⟨pathlist⟩**}**

⟨pathlist⟩→ **{STRING}** | ⟨pathlist⟩**{STRING}**

⟨titlesettings⟩→ **TITLE{**⟨simpleformtext⟩**}**
| **AUTHOR**⟨optional⟩**{**⟨simpleformtext⟩**}**
| **INSTITUTE**⟨optional⟩**{**⟨simpleformtext⟩**}**
| **DATE{**⟨simpleformtext⟩**}**

⟨sectionsettings⟩→ **SECTION{**⟨simpleformtext⟩**}**
| **SUBSECTION{**⟨simpleformtext⟩**}**
| **SUBSUBSECTION{**⟨simpleformtext⟩**}**

⟨body⟩→**BEGIN_DOCUMENT**⟨bodycontent⟩**END_DOCUMENT**

⟨bodycontent⟩ → ϵ | ⟨bodycontent⟩⟨titlesettings⟩
| ⟨bodycontent⟩⟨sectionsettings⟩
| ⟨bodycontent⟩⟨slide⟩

⟨slide⟩ → **BEGIN_FRAME**⟨slidecontent⟩**END_FRAME**
| **BEGIN_FRAME{**⟨simpleformtext⟩**}**⟨slidecontent⟩**END_FRAME**
| **BEGIN_FRAME{**⟨simpleformtext⟩**} {**⟨simpleformtext⟩**}**
⟨slidecontent⟩**END_FRAME**
| **FRAME**⟨optional⟩**{**⟨slidecontent⟩**}**

⟨slidecontent⟩ → ϵ
| ⟨slidecontent⟩**{**⟨slidecontent⟩**}**
| ⟨slidecontent⟩**STRING**
| ⟨slidecontent⟩⟨sectionsettings⟩
| ⟨slidecontent⟩⟨environment⟩
| ⟨slidecontent⟩⟨commands⟩
| ⟨slidecontent⟩⟨image⟩
| ⟨slidecontent⟩**TITLEPAGE**

⟨image⟩ → **INCLUDEGRAPHICS**⟨optional⟩**{STRING}**

⟨environment⟩ → **BEGIN_ITEMIZE**⟨optional⟩⟨itemslist⟩**END_ITEMIZE**
| **BEGIN_ENUMERATE**⟨optional⟩⟨itemslist⟩**END_ENUMERATE**
| **BEGIN_DESCRIPTION**⟨optional⟩⟨itemslist⟩**END_DESCRIPTION**
| **BEGIN_TABULAR STRING**⟨tablerows⟩**END_TABULAR**

⟨itemslist⟩ → **ITEM**⟨overlay⟩⟨optional⟩⟨slidecontent⟩
| ⟨itemslist⟩**ITEM**⟨overlay⟩⟨optional⟩⟨slidecontent⟩

⟨tablerows⟩ → ⟨tablecols⟩
| ⟨tableline⟩⟨tablecols⟩
| ⟨tablerows⟩**ENDROW**⟨tablecols⟩
| ⟨tablerows⟩**ENDROW**⟨tableline⟩⟨tablecols⟩

⟨tableline⟩ → **HLINE**
| **CLINE{STRING}**
| ⟨tableline⟩**HLINE**
| ⟨tableline⟩**CLINE{STRING}**

⟨tablecols⟩ → ⟨slidecontent⟩
| **MULTICOLUMN{STRING} {STRING}{**⟨slidecontent⟩**}**
| ⟨tablecols⟩**&**⟨slidecontent⟩
| ⟨tablecols⟩**&MULTICOLUMN{STRING}{STRING}{**⟨slidecontent⟩**}**

⟨commands⟩ → ⟨command⟩⟨overlay⟩⟨optional⟩`{`⟨slidecontent⟩`}`
| ⟨groupcommand⟩⟨slidecontent⟩
| ⟨standalonecommand⟩

⟨command⟩ → **TEXTBF** | **TEXTIT** | **TEXTTT** | **TEXTSC**
| **UNDERLINE** | **TEXTCOLOR**

⟨groupcommand⟩ → **BFSERIES** | **TTFAMILY** | **ITSHAPE** | **SCSHAPE**
| **TINY** | **SCRIPTSIZE** | **FOOTNOTESIZE** | **SMALL** | **NORMALSIZE**
| **LARGE** | **LARGE2** | **LARGE3** | **HUGE** | **HUGE2**
| **COLOR**⟨overlay⟩⟨optional⟩`{`**STRING**`}`

⟨standalonecommand⟩ → **TODAY** | **PAUSE**
| **FRAMETITLE**⟨overlay⟩⟨optional⟩`{`⟨simpleformtext⟩`}`
| **FRAMESUBTITLE**⟨overlay⟩⟨optional⟩`{`⟨simpleformtext⟩`}`
| **NL**

⟨optional⟩ → ϵ | **OPTIONAL**

⟨overlay⟩ → ϵ | **OVERLAY**

⟨simpleformtext⟩ → ϵ
| ⟨simpleformtext⟩⟨command⟩⟨overlay⟩⟨optional⟩`{`⟨simpleformtext⟩`}`
| ⟨simpleformtext⟩⟨groupcommand⟩⟨simpleformtext⟩
| ⟨simpleformtext⟩**STRING**
| ⟨simpleformtext⟩**NL**
| ⟨simpleformtext⟩`{`⟨simpleformtext⟩`}`
| ⟨simpleformtext⟩**TODAY**
| ⟨simpleformtext⟩**PAUSE**
| ⟨simpleformtext⟩**AND**