

překladové
možnosti
TEX^u

IVO JANČÍK

překladové
možnosti
TEX^u

BRNO 1999
ŠTERNBERK 2000

SLOVO K LASKAVÉMU ČTENÁŘI

Tento text byl zpracováván od října 1998 do května 1999 jako diplomová práce na provozně-ekonomické fakultě MZLU v Brně pod vedením Ing. Jiřího Rybičky, Dr., kterému na tomto místě děkuji za odborné metodické vedení, cenné rady a doporučení. Obhajoba práce proběhla v červnu roku 1999. Text práce je předložen v původním znění a úpravě. Pouze specifické úvodní stránky diplomové práce jsou nahrazeny jinými. Sazba byla provedena programem T_EX za použití písma ITC New Baskerville. Text byl přeložen do PDF verze dne 29. května 2000.

Ivo Jančík

OBSAH

1. Úvod a cíl práce.....	6
1.1. Úvod.....	6
1.2. Cíl práce.....	7
2. Soubory z hlediska uložení na paměťové médium.....	8
2.1. Textové a binární soubory.....	8
2.2. Přednosti textových souborů.....	10
2.3. Omezení textových souborů.....	11
3. Textové formáty.....	14
3.1. Obecně o textových formátech.....	14
3.2. Strukturované a nestrukturované textové formáty.....	15
4. Textový formát SGML.....	18
4.1. HTML jako praktická aplikace SGML.....	19
4.2. Struktura HTML.....	19
4.2.1. DTD deklarace.....	20
4.2.2. Hlavička HTML.....	20
4.2.3. Tělo HTML dokumentu.....	21
4.2.4. Elementy na úrovni bloků.....	22
4.2.5. Elementy na úrovni textu.....	24
5. Textový formát L ^A T _E X.....	26
5.1. Struktura dokumentu L ^A T _E X.....	27
6. Formátová konverze.....	31
6.1. Obecně o formátové konverzi.....	31
6.2. Možnosti konverze L ^A T _E X–HTML.....	33
6.3. Parametrizovaná konverze.....	35
7. Princip T _E Xového překladu.....	36
7.1. Procesory T _E Xu.....	36
7.2. Režimy T _E Xu.....	41
7.3. Práce se soubory na úrovni T _E Xu.....	42
7.4. Program T _E X jako nástroj konverze.....	43
8. Překlad se zápisem do souboru ve výstupní rutině.....	47
8.1. První fáze konverze.....	47
8.1.1. Makra pro zajištění konverze.....	47
8.1.2. Překlad s využitím mapovacího souboru.....	50
8.1.3. Makra pro zápis ve výstupní rutině.....	52
8.2. Druhá fáze konverze.....	54
8.3. Diskuse.....	57
9. Zpracování HTML souboru.....	58
9.1. T _E X jako překladač HTML.....	58
9.2. Zpracování elementu.....	58
9.3. Zpracování atributů.....	59
9.4. Zpracování znakových entit.....	65

9.5. Diskuse	67
10. Závěr	69
10.1. Praktická omezení T _E Xové konverze	69
10.2. Praktické využití	70
10.3. Summary	71
10.4. Literatura	72
11. Příloha. Demonstrace konverze L ^A T _E X–HTML	73

1.1. ÚVOD

Jednou z hlavních oblastí současného využití výpočetní techniky je publikační činnost (DTP, DeskTop Publishing). Podstatnou část publikační činnosti realizované na počítači představuje příprava zdrojového textu, která zahrnuje také vymezení konkrétních částí textu smluvenými, mnohdy automatizovaně vkládanými značkami. Účelů takového vymezení je několik (určení povahy textu, vyjádření způsobu zobrazení). Vzniká tak souborový formát pro publikační činnost.

Uživatel má v dnešní době k dispozici celou řadu navzájem často kategoricky odlišných publikačních formátů. Odlišuje je způsob zápisu textu a vymezení (interpretačních) značek, přenositelnost, oblasti použitelnosti a celkový cíl, pro nějž byl formát navržen. Formáty mohou být textové nebo binární. Binární jsou většinou tvořeny automatizovaným procesem, jejich syntaxe bývá skrytá (nebo alespoň obecně neznámá z komerčních důvodů), jejich vzájemná konverze se neobejde bez speciálních nástrojů (namnoze implementovaných do aplikací, které dané formáty podporují). Naproti tomu textové formáty může přímo zapisovat uživatel v textovém editoru, jejich syntaxe jsou obecně známé (nebo alespoň snadno detekovatelné) a jejich konverze je snazší než u formátů binárních.

Zatímco binární formáty bývají závislé na použité platformě, textové formáty jsou lépe přenositelné a jsou tak vhodným prostředkem pro meziplatformní výměnu dat. S narůstající intenzitou výměny datových souborů mezi jednotlivými počítači a platformami (např. Internet) narůstá potřeba konverze mezi běžně užívanými formáty pro publikační činnosti různých směrů.

Dnes běžně rozšířenými zástupci publikačních formátů jsou HTML (praktická aplikace SGML pro síťovou interaktivní publikaci) a L^AT_EX (formát, též nadstavba systému T_EX pro elektronickou sazbu). Konverze mezi těmito formáty je v praxi realizována manuálním přepisem, poloautomatizovanou nedokonalou záměnou strukturních značek nebo účelovým filtrem vytvořeným v některém programovacím jazyku. Jelikož v případě T_EXu je implementace syntaktické analýzy vzhledem k rozsáhlému aparátu tohoto jazyka náročnou záležitostí, konverze realizovaná pomocí filtru, který nedisponuje lexikálním

analyzátořem na úrovni programu $\text{\TeX}$, není dostatečná. Je proto opodstatněné uvažovat o konverzi realizované využitím překladových možností programu $\text{\TeX}$. V takovém případě alespoň při překladu z $\text{\TeX}$ ového formátu na jiný odpadne na vstupní straně nutnost dodatečné implementace syntaktické analýzy.

1.2. CÍL PRÁCE

V předloženém textu budou prozkoumány fundamentální způsoby náhledu na soubory uložené na paměťová média, vymezen pojem textový formát pro publikační činnost a analyzovány jeho vlastnosti. Následně budou z obecného hlediska zváženy možnosti a omezení konverze mezi textovými formáty. Formáty HTML a $\text{\LaTeX}$ budou v práci popsány podrobněji a zvážena možnost konverze mezi nimi. Vzhledem k tomu, že hlavním cílem práce bude *vytvoření konverzního nástroje na bázi jazykového aparátu programu $\text{\TeX}$* , bude prozkoumán princip zpracování a překladu zdrojového souboru tímto programem a nastíněny možnosti formátové konverze pomocí $\text{\TeX}$ ového překladu. Vybrané možnosti konverze budou pak na praktické aplikaci převodu mezi formáty $\text{\TeX}$ a HTML demonstrovány.

2.1. TEXTOVÉ A BINÁRNÍ SOUBORY

Obecně lze způsob ukládání dat na paměťová media, tedy do souborů, které jsou fyzicky realizovány nejrůznějším způsobem, rozdělit na dva druhy podle toho, jak budeme uložená data vnímat. Každý soubor představuje logickou posloupnost binárních informací, bitů, tedy záznamů typu *nula* a *jedna*, též *pravda* a *nepravda* — záznamy lze interpretovat různě. Setrváme-li na této binární úrovni a budeme-li vnímat zapsaný soubor jako posloupnost bitů, nazveme soubor *binárním*. Abstrahujeme-li od jednotlivých bitů a budeme-li soubor vnímat na úrovni pomyslných *atomů*, tedy pro naši potřebu dále nedělitelných posloupností bitů, jejichž délka je dána určitým pevně daným počtem elementárních bitů, a jejichž význam určuje posloupnost těchto bitů v rámci atomu, nazveme soubor *textovým*. Atomy textového souboru nazveme znaky. (Soubory, které označíme jako textové z hlediska jejich obsahu a nazveme je textovými formáty; viz dále.)

Z aplikace tohoto hlediska vnímání souborů nelze vyvodit závěr, že by každý textový soubor byl podmnožinou množiny binárních souborů. Každý binární soubor je totiž možné vnímat jako soubor textový (a naopak), připustíme-li, že informace předaná na úrovni textového souboru může na první pohled postrádat z hlediska sémantiky lidského jazyka reálný smysl. Avšak pouze na první pohled, protože je zřejmé, že zobrazení f , které zobrazí posloupnosti bitů binárního souboru na množinu znaků textového souboru, tzv. kódování, je bijektivní. Z toho plyne, že informace předaná textovou reprezentací binárního souboru je *potenciálně* táž, jako informace předaná binární reprezentací téhož souboru. Zda informaci přijme, či zda tato pro něj zůstane skryta, závisí na vnímatelově znalosti sémantiky jednotlivých dílčích posloupností bitů a schopnosti tyto posloupnosti v textové reprezentaci rozpoznat.

Je záležitostí operačního systému nebo podpůrných programů, jak posloupnost bitů binárního souboru reprezentuje ve znakové podobě. Ještě do nedávné doby byla často za atom textového souboru považována posloupnost sedmi bitů; od toho se vyvozoval počet od sebe různých znaků, které mohly být v textovém souboru obsaženy. Textový soubor tak mohl obsáhnout 126 znaků. Vzhledem k tomu, že takovýto počet znaků se především pro texty, obsahující text v jiném, než anglickém jazyce, začal jevit jako nedostačující, bylo toto omezení odstraněno osmibitovým kódováním textových souborů; dnes se

můžeme běžně setkat s kódováními, které pro atomy textového souboru rezervují ještě delší posloupnosti bitů (Unicode). Např. mějme zobrazení f , které zobrazí posloupnost šestnácti bitů na množinu znaků. Pak velikost této množiny znaků bude dána výrazem 2^{16} , což představuje 65 536 znaků. Takováto kódování textových souborů jsou nezbytná pro některé cizokrajné, většinou západoasijské jazyky, jakými je čínština, japonština apod.

Textové soubory mohou být vnímány běžným uživatelem v relativně čitelné podobě již na úrovni jednoduchých prohlížečích programů operačního systému. Naproti tomu aby mohly být binární soubory vnímány způsobem, který byl uvažován autorem při jejich vytvoření, je nutné, aby byly zpracovány programy či rutiny, které disponují znalostmi o významu posloupností bitů těchto souborů a způsobu jejich převodu na reprezentaci textovou, grafickou, databázovou či jinou. Na rozdíl od textových kódování, kde je celý soubor chápán jako posloupnost bitů fixních, tedy neměnných délek, nemusí být binární soubor strukturován takto triviálně. Data binárních souborů jsou ukládána v podobě, v jaké budou uložena na určitá místa v operační paměti, odkud je budou číst v tomtéž tvaru aplikační rutiny nebo přímo instrukce procesoru. Zpracování takových souborů aplikačními rutinami a samotným procesorem je jednodušší, neboť odpadá sémantická a mnohdy i syntaktická analýza čtených dat, o to je pak také celý proces zpracování kratší, tedy rychlejší.

Textové soubory jsou naproti tomu zapsány ve formě, která je pro člověka relativně čitelná. V textovém souboru již není informace předávána jako posloupnost bitů, ale jako posloupnost skupin bitů; dále nedekomponované skupiny bitů nazveme *znaky*. Na úrovni znaků jsou také textové soubory běžně zobrazovány. Zápis textového souboru se blíží způsobu zápisu lidské řeči v převážné většině světových jazyků, kdy určitá posloupnost znaků tvoří slova jazyka a ta pak předávají uživateli jazyka informaci. Syntaxe textových souborů je volena podle účelu, aby byla pokud možno výstižná, přehledná, jednoznačná, mnohdy strukturovaná (viz dále) a aby nebyla zbytečně komplikovaná. Zkušenější uživatel je schopen často již na první pohled odhadnout u formátu souboru, se kterým se dříve nesetkal, co v hrubých obrysech bude tento obsahovat. Vyplývá to do značné míry z toho, že v textových souborech jsou používány anglické či pseudoanglické výrazy, např. *read*, *write*, *goto*, *begin*, *end*, ustálené konstrukce typu *while—do*, *if—then*, výrazy tvořené skupinou spojených slov jako *errorstopmode*, *StartKernPairs* a ustálené zkratky, např. *it = item* (též *italic*), *txt = text*.

2.2. PŘEDNOSTI TEXTOVÝCH SOUBORŮ

Zvažme nyní výhody a nevýhody textových souborů. Mezi výhody textových souborů bezesporu patří *jednoduchá editovatelnost*, tj. že mohou být editovány v běžném textovém editoru, který nevkládá do textu formátovací značky, resp. kterému může uživatel takovou činnost zakázat. Uživatel obeznámený s formátem editovaného souboru pak ve chvíli, kdy nemá po ruce příslušnou aplikaci, která byla uzpůsobena pro úpravy daného formátu souboru, sáhne po textovém editoru, který je standardně implementován na každé platformě jako základní programové vybavení, a provede zamýšlené změny.

Z toho plyne další výhoda textového souboru — *přenositelnost*. Jelikož data nejsou uložena v binární podobě způsobem, kterým budou načítána do paměti, lze textové soubory bez větších obtíží přenášet mezi nejrozličnějšími platformami za předpokladu, že je na těchto implementován nástroj, kterým budou následně zpracovány. Pokud není takový nástroj přítomen, lze, jak bylo uvedeno výše, pouze zanechat textovým editorem do souboru potřebné změny a přenést soubor na jinou platformu či pracovní stanici, kde daný nástroj implementován je. Jediné dva problémy při přenosu textových souborů mezi dnes nejužívanějšími platformami, kterými jsou MS DOS (resp. MS Windows), Unix (a z něj odvozené systémy) a Macintosh, jsou s oddělovači řádků a s použitou kódovací znakovou sadou. V prvním případě musí proběhnout konverze, jelikož systém MS DOS používá k oddělení řádků souboru posloupnost znaků s kódy 13 a 10 (CR LF) z kódové sady znaků ASCII, systém Unix znak s kódem 10 (LF) a Macintosh třetí znak 13 (CR). Konverze oddělovačů řádků textových souborů je v praxi řešena buď přímo již při přenosu z platformy na platformu, např. *ftp*, nebo filtry implementovanými buď nad operačním systémem jako uživatelský software, např. *unix2dos* pana I. Lišky, nebo jako aplikační filtr při načítání souboru do aplikace. Konverze kódování je dodatečnou záležitostí. Většinou se provádí až před započatím práce s dokumentem, někdy je dokonce pouze dočasná pro vizuální účely. Pro konverzi dnes běžně používaných kódování existuje velké množství nástrojů, mnoho aplikací má konverzní filtry vestavěné, existují také systémové prostředky pro přizpůsobení systému danému kódování.

Třetí výhodou textového souboru je vlastnost, kterou označíme jako *průhlednost*. Souvisí se snadnou pochopitelností obsahu souboru, která byla uvedena výše, a s tím, že informace jsou v souboru ve většině případů zapsány nezkódované (výjimkou je třeba soubor s definicí řezu písma *Printer Font ASCII*, který je sice textový, ale informace je v něm zakódována do textově vyjádřené binární podoby; podobným příkladem je například způsob zasílání binárních souborů elektronickou poštou, kdy soubory jsou převedeny do textové podoby filtrem *uuencode*). Díky této průhlednosti může textový soubor

použít jako zdroj svých informací síťový nebo jiný vyhledávací program, který analyzuje podřetězce zapsané v souboru a na jejich základě si vytváří účelovou databázi nalezených záznamů. To u binárního souboru není možné bez implementace rutin, které zvládají dekodování obsahu do rozumné textové podoby.

2.3. OMEZENÍ TEXTOVÝCH SOUBORŮ

Nyní zvažme nevýhody textových souborů. První zjevnou nevýhodou je *potřeba syntaktické analýzy*, která umocňuje složitost zdrojového kódu aplikace určené ke zpracování souboru. Syntaktická analýza také prodlouží dobu zpracování souboru. To ovšem dnes již není tak markantním problémem, jako před několika lety, vzhledem k neustálému zvyšování rychlostí hardwarových prostředků. Mějme například dokument v některém z formátů sázecího systému $\text{\TeX}$ o rozsahu sta stran běžného textu. Jeho zpracování na stroji PC AT s procesorem Intel 80286 o taktovací frekvenci 12 MHz a 1 MB operační paměti proběhne programem `tex.exe` pod operačním systémem MS DOS přibližně za 9 minut. Na stroji s procesorem Pentium o taktovací frekvenci 100 MHz velikostí operační paměti 16 MB proběhne zpracování téhož dokumentu za 6 sekund. (Poznamenejme, že výkon druhého uvedeného stroje představuje v době psaní této práce v číselném vyjádření asi polovinu běžně rozšířeného výkonu osobního počítačů.)

Druhou nevýhodou textových souborů je potenciální *nárůst velikosti*. Uvažujme například množinu šestnácti informací, které je třeba uložit do souboru. Je-li použit binární zápis, stačí nám na uložení jedné informace čtyři bity ($2^4 = 16$). Použijeme-li textový zápis s osmibitovým kódováním textového znaku, je třeba minimálně osm bitů pro vyjádření jedné informace. Takto uplatníme 16 znaků, zbylých 240 zůstane nevyužito. Pokud budeme chtít informaci vyjádřit více znaky, což je u textových souborů zcela běžné, velikost souboru naroste. Pokud však nastane potřeba zápisu množiny 256 informací, bude lhostejno, zda upřednostníme zápis v binárním či textovém tvaru, velikost bude táž, pokud ovšem opět nebudeme chtít vyjádřit informaci více znaky. V praxi lze najít příklady, kdy použití textového souboru bude výhodnější, než použití binárního. Binární zápis je zcela evidentně vhodnější pro uložení bitových map. Textové soubory jsou na druhé straně nejvhodnějším prostředkem pro zápis algoritmů v programovacím jazyce.

Vlastnost, která byla výše uvedena mezi výhodami textového souboru a nazvána jako *průhlednost*, může být posuzována i jako nevýhoda tohoto způsobu uložení dat. Skutečnost, že každý si může přečíst informace obsažené v textovém souboru, kde jsou relativně mnohem čitelnější, než v souboru binárním,

může mít za následek zneužití zdrojového kódu algoritmu, programu, textu dokumentu nebo jiných výsledků duševní práce. Tento nedostatek lze řešit šifrováním binárních souborů, komprimací za použití hesla, kódovaným přenosem, fyzickým omezením přístupu k datovým souborům a jinými prostředky softwarové ochrany.

Uvedeme nyní některé příklady souborů běžně zapisovaných v textové podobě. Lze je rozdělit na *systémově závislé* a *systémově nezávislé*. Systémově závislé soubory jsou takové, které přenosem na jiný operační systém ztrácí své opodstatnění, protože jejich syntaxe, struktura a sémantika byla navržena pro uplatnění v konkrétním operačním systému, kde jsou rovněž implementovány rutiny pro jejich zpracování, a kde existují předpoklady pro jejich využití. V první řadě se jedná o *soubory, které vykonávají operace nad příkazovým interpretem operačního systému*. V systému MS DOS jsou označovány jako soubory *dávkové* (angl. batch) a končí rozšířením BAT. V operačním systému Unix a systémech odvozených většinou rozšíření nemají a jsou označovány jako *scripty*; zde se navíc rozlišují na více druhů, podle použitého Unixového příkazového interpretu, tzv. *shellu*. Vyjadřovací jazyky těchto souborů představují relativně jednoduchý zápis volání systémových a uživatelských programů kombinovaný s implementovanými strukturami a řídicími slovy, např. větvení, cykly apod. Prostředky systému Unix jsou v tomto případě bohatší než u systému MS DOS, lze říci, že jazyk scriptů je zjednodušenou podobou programovacího jazyka C.

Jako další příklad textových souborů systémově závislých lze uvést *soubory konfigurační*, a to jednak systémové a jednak aplikační. Systémové konfigurační soubory představují zápis nastavení parametrů operačního systému. Aplikační konfigurační soubory obsahují parametry nastavení konkrétní aplikace instalované nad operačním systémem. Syntaxe takových souborů se může v některých případech shodovat pro tytéž aplikace pod různými operačními systémy. Nad systémy MS DOS a MS Windows existují textové konfigurační soubory s rozšířením SYS (ovšem ne všechny soubory SYS jsou důsledně textové), soubory INI a menšina aplikačních konfiguračních souborů. Většina konfiguračních souborů nad těmito systémy je binární. Naopak Unix upřednostňuje konfigurační zápisy v textových souborech. Je to dáno jeho architekturou a obecným požadavkem na maximální parametrizaci systému i aplikací. Uživatel systému Unix obvykle před spuštěním aplikace, resp. po spuštění systému, vyvstane-li nutnost nějaké úpravy, edituje textovým editorem konfigurační soubor, změní v souladu se syntaxí souboru parametry aplikace nebo systému a spustí aplikaci, resp. nechá znovu zavést systém. Výhoda editace parametrů aplikace bez spuštění této aplikace vyvstane v okamžiku, kdy uživatel potřebuje znát určité parametry nastavení aplikace, ovšem nemá podmínky (vzdálený terminál, nedostatek operační paměti, nedostatek času, chybějící

datové soubory) k tomu, aby mohl aplikaci spustit a parametry nastavení si prohlédnout, resp. editovat až po spuštění z aplikačního prostředí. Textové konfigurační soubory jsou také vzhledem ke své průhlednosti optimální cestou k umožnění cizím aplikacím měnit zapsané nastavení.

Systémově nezávislé textové soubory jsou takové soubory, ve kterých jsou uloženy informace, které mohou být zpracovány se stejnými výsledky na různých operačních systémech. Jedná se o soubory se zápisy programů v programovacích jazycích (např. C, Pascal, Java), v jazycích pro řízení databází (SQL), vyznačovacích jazycích pro zpracování textů (SGML, také T_EXové formáty plain a L^AT_EX), jazyk pro popis vzhledu tiskové strany PostScript a mnohé jiné.

3.1. OBECNĚ O TEXTOVÝCH FORMÁTECH

Až doposud jsme soubory vnímali jako textové z hlediska způsobu uložení dat v těchto souborech. Nyní přejdeme k souborům, které by bylo lze nazvat soubory textovými z hlediska informace, kterou nesou. Aby však byla obě dvě hlediska důsledně odlišena, nazveme soubory, které obsahují data reprezentující text, *textovými formáty*. Definice však není dostačující. Soubor, obsahující zápis algoritmu v programovacím jazyce, může být také chápán jako reprezentace textu a přitom se nejedná o textový formát.

Definujme tedy textový formát poněkud precizněji. Nechť je dána množina terminálních symbolů T . Tuto množinu bude v praxi představovat množina přípustných znaků výchozího kódování textových souborů, např. množina znaků v kódování ASCII, ISO Latin-2 atp. Dále nechť jsou dány množiny neterminálních symbolů N_1 a N_2 , množiny přepisovacích pravidel P_1 a P_2 a počáteční symboly $S_1 \in N_1$ a $S_2 \in N_2$. Nazvěme jazyk $L_1 = L(T, N_1, P_1, S_1)$ jazykem *interpretačních výrazů* a jazyk $L_2 = L(T, N_2, P_2, S_2)$ jazykem obsahující jiné než interpretační výrazy. Potom jazyk L_T , nazveme *textovým formátem* za předpokladu, že množina řetězců (slov) tohoto jazyka je sjednocením množin řetězců jazyků L_1 a L_2 . Množina terminálních symbolů jazyka L_T bude T , množina neterminálních symbolů a množina přepisovacích pravidel budou kromě prvků množin jazyků L_1 a L_2 obsahovat nové prvky, které zajistí, aby při generování výrazů nedocházelo k výskytu nových řetězců, které nenáleží do žádného z jazyků L_1 a L_2 . Můžeme též předpokládat, že z počátečního symbolu S_T jazyka L_T bude možno derivovat symboly S_1 a S_2 .

Interpretační výrazy, nebo symboly, jsou řetězce, které v textu, tedy posloupnosti řetězců jazyka L_2 , označují jakýmkoliv způsobem úseky, aby pak tyto úseky mohly být identifikovány překladačem jazyka L_T . Jde o vyjádření logické struktury a členění textu, způsobu zobrazení textu nebo jiný povel pro zpracující aplikaci. V dalším textu budeme prvky jazyka L_2 také zkráceně označovat jako *příkazy* nebo *řídící sekvence*.

U textových formátů se předpokládá zpracování dalším programem, který pracuje na několika úrovních. Prakticky vždy provede syntaktickou analýzu. Dále převede daný textový formát na jiný formát. Může jít o vnitřní formát

programu, formát tiskové úlohy, jiný používaný textový formát, přenosný formát popisující vzhled tištěné strany (PostScript, DVI), nebo může překladač pouze vyčlenit stranou řetězce jazyka L_2 a vytvořit tzv. *čistý* textový formát.

Čistý textový formát je zvláštním případem textového formátu. Jde o takový textový formát, pro který platí, že množina řetězců jazyka vyznačovacích výrazů L_1 je prázdná. Soubor v čistém textovém formátu neobsahuje vyznačovací symboly. Chybí tedy informace o logické struktuře textu. Přítomny jsou pouze řetězce jazyka L_2 .

Jazyk L_2 bude typicky představovat jakési sjednocení všech jazyků, kterými se na Zemi lidé mezi sebou dorozumívají. Může obsahovat prakticky jakékoliv výrazy, které uživatel jazyka potřebuje zapsat, přesněji tedy jakoukoliv posloupnost znaků, která nepředstavuje řetězec jazyka L_1 . Pro případ situací, kdy uživatel potřebuje vyjádřit v textové podobě řetězec, který patří do jazyka L_1 , jsou do tohoto jazyka implementovány zástupné sekvence, které po zpracování souboru překladačem požadovaný řetězec generují.

3.2. STRUKTUROVANÉ A NESTRUKTUROVANÉ TEXTOVÉ FORMÁTY

Textové formáty lze rozdělit na nestrukturované, částečně strukturované a strukturované.

Strukturované textové formáty jsou takové formáty, které vyčlení a jednoznačně určí v textu dokumentu bloky, nesoucí ucelenou logickou informaci, resp. skupinu informací téhož druhu, což označíme jako *logickou strukturu textového formátu*. Zdůrazněme, že jde o textové formáty a tedy struktura by měla být charakteru, který souvisí s významovým obsahem textové části struktury. Obecně je za strukturu možné považovat i příkaz pro zobrazení znaku na dané pozici spolu s konkrétním znakem. Takovýto příkaz je však vhodná struktura pro jiné formáty než textové, např. grafické, protože nenese informaci o povaze textu. Nebudeme proto chápat takové struktury jako textové.

Strukturní bloky mohou být do sebe násobně vnořeny. U dokonale strukturovaného formátu je nezbytné, aby žádná část čistého textu dokumentu nebyla zapsána mimo strukturní blok. Logickou textovou strukturou může být hlavička dokumentu, nadpis, odstavcový text, výčet, položka výčtu, ale také zvýrazněné písmo ve smyslu obecného zvýraznění textu, nikoliv konkrétní změny písma, která nevypovídá nic o povaze textu.

Struktura textového formátu by měla být volena tak, aby byla pokud možno nezávislá na zobrazovacím nástroji. Zobrazení dokumentu je pak ponecháno

na tomto nástroji — aplikaci, která jej bude zpracovávat, nebo na definičním souboru, který bude spolu s dokumentem načten a který určí způsoby zobrazení jednotlivých logických struktur. Je pak možné, aby též soubor pro různé potřeby byl zobrazen různě, ať už se jedná o rozdíly ve vizuální podobě, nebo o rozdíly v úplnosti předání v souboru obsažených dat. Logické strukturování dokumentu totiž umožňuje, aby aplikace v souboru vyhledala pouze požadované bloky struktur, např. nadpisy, a zobrazila pouze tyto podle potřeby, přičemž zbytek ignoruje.

Nestrukturovaný textový formát je takový formát, který nese kromě samotného textu pouze informace o způsobu zobrazení dokumentu a abstrahuje od logického členění textu. Mezi části textu jsou vloženy příkazové sekvence, které definují vzhled obsahové části souboru. Jde např. o umístění objektu v souřadnicovém systému zobrazovacího zařízení, přesun aktuálního kreslicího bodu na jinou pozici, volba konkrétního řezu písma pro zobrazení textu, vykreslení grafického prvku apod. Takovéto příkazy jsou často závislé na použitém zobrazovacím zařízení.

Částečně strukturovaný textový formát pak obsahuje jak řídicí příkazy, které definují logickou strukturu dokumentu, tak i příkazy, které určují vzhled dokumentu. Tyto formáty mohou být strukturované na různém stupni.

Demonstrujeme typický rozdíl mezi textovým formátem strukturovaným a nestrukturovaným. Použijeme k tomu běžné syntaxe zápisu řídicích sekvencí v jazyku programu T_EX (syntaxe T_EXového formátu bude rozebrána dále). Předpokládáme, že je potřeba v dokumentu vymežit nadpis od okolního textu. V nestrukturovaném formátu bude například něco takového:

```
\volneradky 2
\zacnizkraje
\pismo \tucne \velike
Zpracování kávy za sucha
\volneradky 1
\pismo \bezne
```

Uvedené řídicí sekvence jsou ilustrační. Aplikace zpracující takovýto dokument vynechá dva volné řádky (může být také uvedeno 2 cicera nebo 1,5 cm), zvolí požadované písmo, zobrazí text nadpisu, vynechá volný řádek a nastaví původní řez písma. Celý sled příkazů postrádá jakoukoliv informaci, že se jedná o nadpis. To pochopí až člověk, který bude dokument vnímat ve zpracované podobě. Naopak zápis ve strukturovaném textovém formátu bude podstatně kratší:

Způsob zobrazení je přenechán aplikaci. Ta může načíst definiční soubor, v němž bude řečeno, že vyskytne-li se sekvence `\nadpis`, provede sled příkazů uvedených výše pro nestrukturovaný formát. Jiná aplikace pak provede zobrazení jinak a třetí aplikace pouze vyhledá všechny takové nadpisy a vytvoří z nich obsah.

Uvedme některé příklady strukturovaných a nestrukturovaných textových formátů. Budeme-li chápat jazyk PostScript jako textový formát, lze jej považovat za zcela nestrukturovaný. Také ve formátu RTF (Rich Text Format) téměř zcela chybí strukturní informace. Nestrukturované, nebo jen velice málo strukturované, jsou také formáty českých textových editorů T602 a MAT. Za nestrukturovaný lze považovat i čistý textový formát.

Mezi textové formáty, které jsou relativně dobře strukturované jsou formáty SGML a $\text{\LaTeX}$. Jelikož se tato práce zabývá konverzí mezi textovým formátem $\text{\LaTeX}$ a formátem HTML, který je podmnožinou — praktickou aplikací SGML, budou oba zmíněné formáty popsány blíže.

4 / TEXTOVÝ FORMÁT SGML

Formát SGML (Information Processing ..., ISO 8879) je typickým představitelem strukturovaného jazyka. Dokument v čistém SGML formátu neobsahuje žádné řídicí sekvence zabývající se vzhledem dokumentu. Většina dodatečných informací uložená do souboru s textem se soustředí na vyčlenění struktury obsaženého textu (např. u HTML se o užší specifikaci grafického vyznění dokumentu starají atributy, viz dále). SGML je metajazykem, tzn. že neobsahuje žádný konkrétní výčet vyznačovacích symbolů (značek), které mohou být pro vymezení struktury textu použity.

Úplný dokument SGML se skládá ze dvou částí. První je tzv. DTD (Document Type Definition), soubor ve kterém se definují *elementy* a *entity*, které smí být použity. Druhou částí dokumentu je samotný SGML soubor, který obsahuje na prvním místě odkaz na DTD soubor, ve kterém jsou použité elementy a entity definovány, a samotný text, kde jsou tyto symboly použity. Celá DTD definice nebo jen definice některých nových používaných symbolů může být uvedena přímo v SGML souboru. Popis syntaxe DTD definice přesahuje rámec této práce, viz (Information Processing ..., ISO 8879).

Element SGML představuje část textu dokumentu uzavřenou mezi uvozující a ukončující značku (angl. tag). Jako element lze zapouzdřit i celý dokument. U některých elementů, pokud je tak deklarováno v DTD dokumentu, a pokud je zřejmé, kde daný element končí, může být ukončující značka vynechána. Jsou také tzv. prázdné elementy, které neobsahují ani textovou část, ani ukončující značku. Jejich předmětem není vyznačení textové informace.

Značka elementu je vymezena rezervovanými znaky „<“ a „>“. Mezi tyto znaky se zapíše jméno elementu podle DTD, na velikosti písmen nezáleží. Ukončující značka má před jménem uveden znak „/“. V některých případech je možné použití zkrácené ukončující značky v podobě </>. Úplný SGML element může vypadat např. takto:

```
<nadpis>Zpracování kávy za sucha</nadpis>
```

V DTD definici lze stanovit, že význam, resp. zpracování elementu lze upravit výčtem vlastností, kterým se říká *atributy* (angl. attributes). Atributy se zapisují v uvozující značce mezi jméno elementu a znak „>“. Jsou uvedeny ve tvaru `atribut="hodnota"`. Některé atributy jsou určující nikoliv svou hodnotou, ale přítomností, pak se zapisují pouze ve tvaru `atribut`, nebo ve

tvary atribut="atribut". Jednoduchý příklad použití atributu uroven pro vyjádření úrovně nadpisu:

```
<nadpis uroven="3">Zpracování kávy za sucha</nadpis>
```

Specifickým případem syntaxe SGML je variantní možnost vkládání znakových entit pomocí zápisu &#číslo;, kde číslo je pozice znaku v použité kódové sadě. Každý znak v kódových sadách používaných v HTML má své jméno, proto lze psát také &název;. Například znak „é“ lze pro kódování podle ISO Latin-1 zapsat jako é, nebo é.

4.1. HTML JAKO PRAKTICKÁ APLIKACE SGML

HTML (HyperText Markup Language) představuje konkrétní aplikaci SGML jazyka. Umožňuje vytvářet hypertextové dokumenty, které jsou přenositelné mezi různými platformami. V současné době je platné doporučení organizace W3C z dubna roku 1998, které předepisuje HTML ve verzi 4.0 (HTML 4.0 Specification, 1998), která nahrazuje a rozšiřuje předešlou verzi 3.2. Předpokládá se zpětná kompatibilita verzí.

Mezi přednosti a přínosy HTML patří strukturování dokumentu plynoucí mj. z použití SGML, jednoduchá syntaxe a srozumitelná sémantika, možnost tvorby hypertextových odkazů, multimediální podpora. HTML deklarace úmyslně postrádá větší množství prostředků pro formátování dokumentu a jeho grafické vyznění. Aplikace, zpracující HTML dokument, si sama stanoví, jak jednotlivé elementy prezentuje. Taková aplikace nemusí být pouze vizuálně založená, může jít také o zvukové zprostředkování informace nebo jiné. Nejrozšířenějším typem aplikací, které zpracovávají HTML dokumenty, je v současné době tzv. WWW prohlížeč, což je volný překlad anglického termínu *browser*. Běžnými takovými prohlížeči dnes používanými jsou Netscape Communicator a MS Internet Explorer.

4.2. STRUKTURA HTML

Dokument HTML se skládá ze tří částí. Jsou to:

- deklarace <!DOCTYPE>;
- hlavička dokumentu;
- tělo dokumentu.

4.2.1. DTD deklarace

<!DOCTYPE> deklarace — nebo také *DTD deklarace* — je povinná podle doporučení W3C, je však v mnoha dokumentech absentována. Zmiňuje verzi HTML, které dokument odpovídá, a odkazuje na DTD deklaraci, jejíž elementy jsou v HTML dokumentu použity. Verze HTML 4.0 může obsahovat tři druhy <!DOCTYPE> deklarace. *Strict* DTD deklarace obsahuje takové HTML elementy, které jsou, jednoduše řečeno, graficky nezávislé. *Transitional* DTD deklarace (přechodná) obsahuje všechny elementy deklarované v předchozí deklaraci a navíc elementy graficky závislé, označované anglickým termínem *deprecated*, tedy nedoporučené. Použití těchto elementů může způsobit nepřenositelnost dokumentů. *Frameset* DTD deklarace obsahuje elementy předchozí deklarace a přidává definici tzv. *framů*, tedy grafických prostředků pro rozdělení vizuální plochy grafického prohlížeče na samostatné části (v češtině se používá výraz *rámečky* nebo *rámcce*). Jednotlivé DTD deklarace vypadají takto:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
    "http://www.w3.org/TR/REC-html40/strict.dtd">
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
    "http://www.w3.org/TR/REC-html40/loose.dtd">
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN"
    "http://www.w3.org/TR/REC-html40/frameset.dtd">
```

Uvedené internetové lokace umožňují zpracující aplikaci, aby načetla DTD dokument v případě, že jej nezná, a mohla tak správně vyhodnotit syntaxi HTML souboru.

4.2.2. Hlavička HTML

Hlavička dokumentu a tělo jsou uzavřeny do elementu <HTML>. *Hlavička* HTML dokumentu je vymezena elementem <HEAD>. Obsahuje doplňující informace k dokumentu. Nejběžnější elementy této sekce jsou:

- Element <TITLE>, který obsahuje stručný výstižný název celého dokumentu.
- Prázdný element <BASE>, který uvádí výchozí adresu, k níž se vztahují relativní odkazy v dokumentu použité. Obsahuje atribut `href`.
- Prázdný element <META> udávající dodatečné informace o dokumentu. Obsahuje atributy `name`, `content` a `http-equiv`.

- Element `<LINK>`. Používá se pro vyjádření vztahů mezi více dokumenty. Umožňuje provázat dokument s dokumentem předchozím a následujícím, s obsahem celé struktury dokumentů, rejstříkem, první stránkou apod. Reprezentace těchto odkazů je ponechána zcela na zpracující aplikaci. Odkazy jsou realizovány pomocí atributů `href`, `rel`, `rev`, `title`.
- Element `<STYLE>`, který slouží k odkazu na použité *style*s, angl. style sheets. Styley jsou vhodným nástrojem pro oddělení grafické reprezentace dokumentu HTML do zvláštních stylových souborů. Styley jsou psány zvláštním jazykem a patřičné elementy, které je v dokumentu využívají, pouze odkáží na soubor, kde jsou definice těchto stylů uloženy. Grafické úpravy jednoho stylu se pak mohou promítnout okamžitě na všechny elementy, které tento styl použijí. Odpadne tak pracné přepisování HTML kódu na mnoha místech.
- Elementy `<SCRIPT>`, `<ISINDEX>` aj.

Příklad hlavičky HTML dokumentu:

```
<HEAD>
  <TITLE>Zpracování kávy</TITLE>
  <BASE href="http://www.kava.cz">
  <META name="Autor" content="Jan Šálek">
  <LINK rel="Contents" href="contents.html" title="Obsah">
</HEAD>
```

4.2.3. Tělo HTML dokumentu

Tělo HTML dokumentu je vymezeno elementem `<BODY>`. Element může obsahovat atributy, popisující některé vlastnosti vizuální reprezentace dokumentu. Jde o barvu textu, hypertextových odkazů a možný obrázek na pozadí. Je rozšířeno mnoho dalších atributů, které však jsou ve verzi 4.0 uvedeny jako nedoporučené.

Elementy obsažené v elementu `<BODY>` spadají do dvou skupin. Jsou to *elementy na úrovni bloků* (angl. block level elements) a *elementy na úrovni textu* (text level elements). Prvně zmíněné elementy způsobují odstavcový zlom (ve vizuálně orientovaných HTML prohlížečích). Označují bloky textu jako celky. Obvykle obsahují text a elementy na úrovni textu. Ty nezpůsobují odstavcový zlom a slouží k jiným účelům, než označení textových bloků.

4.2.4. Elementy na úrovni bloků

Mezi nejběžnější elementy na úrovni bloků patří nadpisy, a dále elementy pro práci s odstavci, jmenovitě výčtová prostředí, tabulky, formuláře a také vodorovné linky.

Nadpisy jsou realizovány elementy <H1> až <H6> podle úrovně od nejvyšší k nejnižší. Prohlížeč obvykle zobrazí nadpis vyšší úrovně větším písmem, než nadpis úrovně nižší. Elementy vyžadují uvozující i ukončující značku. Mohou obsahovat atribut `align`, který vymezuje zarovnání nadpisu vpravo, vlevo, nebo na střed (right, left, center). Např.:

```
<H3 align="left">Zpracování kávy za sucha</H3>
```

Mezi nejběžnější *elementy pro práci s odstavci* patří element <P>. Uvozuje odstavec, ukončující značka není nutná. Obsahuje atribut `align`, který podobně jako u nadpisu deklaruje zarovnání odstavce. Odstavec je formátován zpracovávající aplikací do dostupných okrajů. Například:

```
<BODY>
<P>Při zpracování kávy za sucha se sklizené plody nejprve
propírají ve velkých nádržích, kde se zbaví hlíny a
nečistot. Oprané se suší na slunci v tenkých vrstvách
...
</BODY>
```

Chce-li uživatel použít již předformátovaný text, uzavře jej do elementu <PRE>. Pro text v tomto elementu je typicky zvolen neproporcionální řez písma. Zmiňme dále element pro vymezení vystředěného textu, <CENTER>, a element deklarující citaci, <BLOCKQUOTE>, typicky zobrazený v zúžených okrajích. Jednoduchým grafickým prvkem je vodorovná čára, oddělující od sebe textové bloky. Deklaruje ji prázdný element <HR> s atributy určujícími její vizuální podobu (např. `size`, `width`).

Elementy pro práci s *výčty* jsou a pro nečíslovaný a číslovaný výčet. Pro označení položky výčtu slouží element . Všechny tři elementy mohou obsahovat atributy pro vymezení typu odrážky (u nečíslovaného grafický symbol, u číslovaného způsob zápisu číselné konstanty), u číslovaného výčtu navíc počáteční hodnotu první položky výčtu. Jsou-li atributy použity v elementu , jsou platné pouze pro danou položku. V opačném případě jsou platné pro celý výčet. Příklad jednoduchého výčtu:

```
<P>Druhy zpracování kávy
```

```

<UL>
  <LI>Zpracování kávy za sucha
  <LI>Zpracování kávy za mokra
</UL>

```

Zvláštním druhem výčtu je *definiční seznam*. Je dán elementem <DL> a obsahuje elementy <DT> pro název definovaného výrazu a <DD> pro jeho popis.

Výčtová prostředí jsou obvykle zobrazována s odsazeným levým okrajem. Odřádková položka výčtu je předsazena před první řádek položky, u definičního seznamu je předsazen definovaný výraz, který je někdy též ukončen přechodem na nový řádek.

Pro definici *tabulek* slouží element <TABLE>. Tabulka v HTML 4.0 je rozdělena na tři části. Záhlaví, které vymezuje element <THEAD>, patu tabulky, vymezenou elementem <TFOOT>, a vlastní tělo, <TBODY>. Zpracující aplikace má tak možnost, aby od sebe jednotlivá části jednoznačně odlišila a mohla je takto prezentovat uživateli. Pro označení řádku tabulky slouží element <TR>. Řádek tabulky pak obsahuje elementy s vlastními políčky tabulky. Jsou to <TD> pro běžná políčka a <TH> pro políčka v záhlaví tabulky, která mohou být zobrazena vizuálně odlišně. Z mnoha atributů, které mohou políčka tabulky obsahovat jmenujme *rowspan* a *colspan*, které určují, kolik řádků, resp. sloupců bude dané políčko tabulky zabírat, *width* a *height*, které udávají požadovanou šířku a výšku políčka. Poslední dva atributy jsou nedoporučeny. Z dalších elementů použitelných v tabulce je vhodné zmínit ještě element <CAPTION>, který zaznamenává název tabulek. Název může být uvedený pod zobrazenou tabulkou. Tabulky mohou být násobně vnořeny. Příklad jednoduché tabulky:

```

<TABLE>
  <CAPTION>Spotřeba kávy na obyvatele v~kg</CAPTION>
  <THEADER>
    <TR>
      <TH>Stát</TH>
      <TH>1970/1971</TH>
    </THEADER>
  <TBODY>
    <TR>
      <TD>ČSSR</TD>
      <TD>0,84</TD>
    <TR>
      <TD>Dánsko</TD>
      <TD>12,74</TD>
  </TBODY>
</TABLE>

```

</TBODY>
</TABLE>

Formuláře, které se dnes hojně používají v dokumentech HTML zmíníme jen okrajově. Celý formulář HTML je vymezen elementem <FORM>. Jeho vizuální podoba je silně závislá na použitém prostředku pro zpracování HTML. Elementy deklarující prvky formuláře jsou odkázány na použití softwarových prostředků k jejich zpracování, běžně např. CGI nebo Java skripty, možnosti kterých překračují dalece možnosti HTML. Konverze formulářů HTML do textových formátů, které předpokládají jinou, než interaktivní reprezentaci dokumentu (např. tisk na papír), je zbytečná z toho hlediska, že repertoár vizuálních prvků HTML formuláře byl navrhován pro interaktivní použití na obrazovce a obsahuje mj. takové grafické prvky, které při jiném použití ztrácejí svůj smysl (např. tlačítka). Lze si však představit konverzi, která převádí HTML formulář na formulář určený pro tisk na papír, kde grafické prvky interaktivní budou nahrazeny grafickými poli, které uživatel vhodným způsobem vyplní perem.

4.2.5. Elementy na úrovni textu

První skupinou elementů na úrovni textu jsou elementy pro odlišení části textu, většinou volbou jiného řezu písma. Jsou přímé a nepřímé, oba druhy vyžadují uvozující i ukončující značku. Mezi přímé, které jsou nedoporučovány vzhledem ke své absolutní povaze, patří např. pro tučné písmo, <I> pro kurzívu, <TT> pro strojopisný neproporcionální řez nebo <SUP> a <SUB> pro horní a dolní index. Nepřímé naopak určují pouze obecně, jak má být daný text odlišen, neváží se na konkrétní druh řezu a předávají i informaci o povaze obsaženého textu. Patří sem elementy pro zvýrazněný text, pro silně zvýrazněný text, <CODE> pro vložený programový kód aj.

Jedním ze základních elementů HTML je element <A>. Umožňuje ohraničit text nebo jiný grafický prvek, který bude sloužit jako odkaz na další hypertextový dokument nebo jiný objekt. Uvozující i ukončující značka je povinná, vnoření elementů se nepovoluje. Běžným atributem tohoto elementu je href, který udává cílové uložení objektu, na nějž je odkazováno. U vizuálních prohlížečů HTML je běžné zobrazit textový hyperodkaz jinou barvou a umístit pod něj horizontální linku.

Elementem, který do dokumentu vloží obrázek, je prázdný element . Zdroj obrázku se specifikuje povinným atributem src. Z dalších atributů lze použít width a height, jejichž uvedení je vhodné proto, aby HTML

prohlížeč mohl naformátovat danou oblast dokumentu a pokračovat v dalším formátování, zatímco bude zdrojový obrázek natahován.

Z dalších elementů textové úrovně zmiňme `<BR>`, který zalomí řádek, element `<APPLET>`, který umožní vložit do dokumentu applet v jazyce Java, `<MAP>`, který definuje oblasti v obrázku, na nichž poklepání myši ve vizuálním prohlížeči způsobí uživatelem definovanou akci, a diskutovaný element `<FONT>`, umožňující uživateli změnit explicitně velikost použitého písma, v některých případech dokonce zvolit konkrétní font, kterým má být text zobrazen. Uvedené elementy, kromě `<BR>`, jsou silně závislé na použitých prohlížečích HTML dokumentů a způsobují problémy při přenosu souborů mezi různými platformami a aplikacemi, které tyto elementy nepodporují.

Výčet elementů lze ještě doplnit zmínkou o existenci elementů pro prvky formuláře, které spadají do textové úrovně.

T_EX je systém pro sazbu dokumentů vyvinutý Donaldem Knuthem (KNUTH, 1986). Jde o systém flexibilní a silně parametrický. Disponuje širokou množinou vestavěných řídicích sekvencí, které se označují jako *primitives* (řídčeji primitiva, j. č. primitivum). Pružnost systému spočívá v možnosti deklarací nových řídicích sekvencí a tak přizpůsobení konkrétním uživatelským potřebám. Sazba dokumentů pouze pomocí vestavěných primitivů by byla totiž poněkud těžkopádná, ač možná. Deklarace nových řídicích sekvencí, které jsou označovány jako *makra*, je možné psát přímo do zdrojového dokumentu, nebo je lze uložit do samostatného souboru. Z takového souboru lze pro zvýšení rychlosti zpracování maker vytvořit binární soubor, který se označuje jako *formát* T_EXu a který lze načíst při spuštění systému a určitým způsobem tak rozšířit množinu vestavěných primitivů.

Autor systému T_EX vytvořil základní formát, který se označuje jako *plain* (nepřekládá se). Mnoho uživatelů T_EXu s tímto formátem vystačí, jde však o formát nestrukturovaný a uživatel je nucen spoustu technických záležitostí řešit definicí vlastních maker. To se neobejde bez poměrně dobré znalosti práce primitivů. Mimo jiné z těchto důvodů byl Leslie Lamportem vytvořen formát T_EXu, který se označuje jako L^AT_EX. V jeho formátu je kladen silný důraz na strukturování dokumentu. Svojí podstatou se blíží strukturování HTML, na rozdíl od něj je však určen především k produkci kvalitní sazby. Obsahuje tudíž širší škálu vyjadřovacích prostředků.

Ve formátu L^AT_EX a celé řadě nástrojů s ním souvisejících (proto budeme hovořit o L^AT_EXu spíše jako o systému) je zavedena jednotná syntaxe definovaných maker, která odlišuje od uživatele původně volnější syntaxi primitivů a maker *plainu*. Platí zásada, že použití většiny původních primitivních řídicích sekvencí je narušením koncepce L^AT_EXu. Na rozdíl od maker formátu *plain* se makra formátu L^AT_EX zabývají spíše vymezením jednotlivých částí dokumentu. Vyjádření způsobu, jakým budou tyto části sázeny, není (obvykle) přímou součástí textu. Tyto definice je možné psát buď v deklarční části dokumentu, nebo lépe v samostatných souborech, které se označují jako *styly* (angl. style, nověji též *class*), pokud představují logicky ucelený soubor řídicích sekvencí, definující celkovou grafickou podobu dokumentu, nebo *balíky*, (angl. packages), pokud obsahují makra pro specifické záležitosti, např. dvojsloupcovou sazbu.

Syntaxe L^AT_EXových maker vychází do jisté míry z původní syntaxe zvolené D. Knuthem pro formát plain. Nutno uvést, že samotný T_EX umožňuje pomocí tzv. kategorií přiřazených každému jednotlivému znaku vstupního souboru změnit význam toho kterého znaku a tak zcela změnit syntaxi zápisu zdrojového souboru (viz dále o principech T_EXového překladu).

Kromě samotného textu obsahuje zdrojový soubor pro L^AT_EX příkazy, které se mohou vyskytovat ve dvou formách. Jednodušší formou je tzv. *aktivní znak*, jehož výskyt představuje zápis určitého povelu tomuto znaku přiřazenému. Např. znak tilda (~) představuje povel pro vysazení nezlomitelné mezery. Druhou, častější variantou zápisu povelu je zápis *řídící sekvence* (také jednoduše příkaz či makro). Ta je uvozena znakem „\“, za nímž následuje jméno řídící sekvence. Může to být posloupnost písmen (de facto znaků, jejichž kategorie je formátem nebo stylem nastavena na kategorii *písmeno*), nebo jediný znak, ten potom může být i jiný než písmeno. Např. sekvence \large je povel pro volbu písma o velikosti 12 pt. Sekvence \, je povel pro sazbu zúžené mezery, v které nesmí být proveden řádkový zlom. Řídící sekvence mohou doplňovat parametry (též argumenty, atributy). Parametry jsou v L^AT_EXu povinné a nepovinné, což je záležitost definice formátu; původní T_EX nepoužívá nepovinné parametry. Povinné parametry se zapisují do složených závorek, {parametr}, nepovinné se zapisují do hranatých závorek, [parametr]. Zvláštním případem povinných parametrů jsou parametry pro polohu grafického prvku při vytváření obrázků, ty jsou zapisovány do kulatých závorek, (10,20).

5.1. STRUKTURA DOKUMENTU L^AT_EX

Struktura dokumentu zapsaného ve formátu L^AT_EX je blízká struktuře dokumentu v HTML. Nejprve je uvedena *hlavička* dokumentu. Ta se uvozuje, od verze L^AT_EX2_ε, sekvencí \documentclass, která má dva nepovinné parametry a jeden povinný v pořadí nepovinný, kterým je název souboru doplňujícího zvolený styl, povinný, kterým je zvolený styl, v praxi soubor s příponou .cls a nepovinný, kterým je požadované nejdřívejší datum vytvoření stylového souboru. V hlavičce se dále může vyskytovat příkaz pro načtení balíku \usepackage, příkazy pro definici nových řídících sekvencí pomocí \newcommand nebo \renewcommand, příkazy stylu nebo balíku, které nepředstavují okamžité vysazení textu, např. \title, \author, a které mohou způsobit vysazení textu až na požadovaném místě ve vlastním dokumentu, a některé další. Platí zásada, že použití jakéhokoliv příkazu v hlavičce, který by znamenal okamžité vysazení textu, způsobí při překladu dokumentu chybu překládajícího programu (pod systémem MS DOS např. tex386.exe).

Hlavička L^AT_EXového dokumentu je ukončena příkazem `\begin{document}`. Tímto příkazem začíná hlavní, textová část dokumentu, kterou je vhodné v návaznosti na popsanou strukturu HTML označovat jako *tělo* dokumentu. Řídící sekvence `\begin` s jedním povinným parametrem představuje uvozující sekvenci jednoho z významných nástrojů L^AT_EXu, kterým je tzv. *prostředí*. Je vymežováno sekvencemi `\begin{jmeno}` a `\end{jmeno}` a ohraničuje logický celek textu, se kterým je zacházeno specifickým způsobem a který může obsahovat řídící sekvence mimo dané prostředí nebo v jiných prostředích nepoužitelné. Celý dokument v L^AT_EXu končí příkazem `\end{document}`. Prostředí `document` je nejvyšší úrovní prostředí a ohraničuje veškerý text, který bude sázen.

Jednoduchý dokument v L^AT_EXu může vypadat takto:

```
\documentclass[12pt]{report}
\pagestyle{empty}
\begin{document}
Při zpracování kávy za sucha se sklizené plody nejprve
propírají ve velkých nádržích, kde se zbaví hlíny a
nečistot. Uprané se suší na slunci v~tenkých vrstvách
...
\end{document}
```

Podobně jako u HTML můžeme rozlišit příkazy použité ve zdrojovém souboru L^AT_EXu na příkazy na úrovni bloků, kterými zde budou výše zmíněná prostředí, k nimž můžeme ještě přiřadit příkazy pro rozdělení textu na oddíly, a příkazy na úrovni textu, kam spadají ostatní příkazy, použité v těle dokumentu.

Mezi příkazy pro specifikaci oddílů textu patří řídící sekvence `\part`, `\chapter`, `\section`, `\subsection`, `\subsubsection`, `\paragraph` a `\subparagraph`. Jejich vizuální reprezentace je závislá na použitém stylu. Navíc v každém běžném L^AT_EXovém stylu nejsou zastoupeny z logických důvodů všechny oddíly, v některém pak může být definován některý další, např. `\appendix`. Uvedené příkazy mají první parametr nepovinný, jde o alternativní název, který bude reprezentovat oddíl v automaticky generovaném obsahu, druhý je povinný a vyjadřuje jméno oddílu. Každý příkaz má ještě alternativní tvar s hvězdičkou, který generuje název oddílu bez automaticky vkládaného číslování, např. `\chapter*{Název}`. Příklad:

```
\chapter{Zpracování kávy}
\section*{Zpracování kávy za sucha}
```

Textová prostředí ohraničují text o určitých specifických vlastnostech. Výčtová prostředí jsou podobně jako u HTML číslovaná a nečíslovaná. Pro číslovaný výčet slouží prostředí `enumerate`, pro nečíslovaný výčet prostředí `itemize`. Položka ve výčtu se označí příkazem `\item` zapisovaným před položku, která však poněkud nedůsledně netvoří parametr příkazu. Naproti tomu může příkaz `\item` obsahovat nepovinný parametr, který nahradí implicitně generovanou odrážku. Třetím výčtovým prostředím je prostředí `description`, které je obdobou definičního seznamu u HTML. Zde je nepovinná položka příkazu `\item` mlčky pokládána za povinnou, jelikož bez ní by výčet postrádal smysl. Je obvykle sázena tučným řezem. Příklad číslovaného výčtového prostředí:

```
\begin{enumerate}
  \item Zpracování kávy za sucha
  \item Zpracování kávy za mokra
\end{enumerate}
```

V uvedených třech prostředích lze uvést některá další makra pro úpravu vzhledu sazby.

Dalšími prostředími L^AT_EXu jsou prostředí `quote` a `quotation` pro krátké a dlouhé citáty, sázené běžně jako text v užších okrajích. Jsou obdobou `<BLOCKQUOTE>` v HTML. Dále prostředí `verbatim` pro sazbu zdrojového kódu (obdoba `<PRE>`) a `verse` pro jednoduchou sazbu poezie.

K rozsáhlejšímu nastavení zarovnávání textu slouží prostředí `flushleft`, `flushright` a `center`, postupně pro zarovnání vlevo, vpravo a na střed.

K sazbě tabulek slouží dvě prostředí, a sice `tabbing` a `tabular`. Prostředí `tabbing` je principiálně jednodušší, je určeno pro vertikální zarovnání textu bez nároku na vytvoření vizuálně efektní tabulky. Představuje obdobu tabelátorů psacích strojů. V HTML nemá obdobu. Na tomto místě se nebudeme tímto prostředím zabývat podrobněji.

Prostředí `tabular` přidává k řídicí sekvenci `\begin` další dva parametry. První nepovinný vyjadřuje připojení tabulky k okolnímu textu. Nabývá hodnot „t“, vrchem a „b“, spodem, není-li přítomen, je tabulka připojena středem. Druhý parametr deklaruje vzhled sloupců tabulky. Každý sloupec je zastoupen jedním písmenem, které udává jeho horizontální zarovnání, „l“ doleva, „r“ doprava, „c“ na střed, „p“ do bloku (zde je nutno specifikovat parametr udávající šířku tabulky). Vertikální linka v tabulce se deklaruje znakem `|`. Místo znaku `|` lze použít znak „@“ s jedním parametrem, který pak bude sázen v každém řádku tabulky mezi danými sloupci. Jednotlivé řádky tabulky jsou odděleny sekvencí `\\`, jednotlivé buňky znakem „&“. Horizontální linky

jsou vkládány do tabulky příkazem `\hline`. Chce-li uživatel vložit do tabulky buňku, která zabírá více sloupců, použije příkaz `\multicolumn`, který má tři povinné parametry v pořadí počet zabraných sloupců, způsob zarovnání, který je obdobný jako u deklarace sloupců celého prostředí, a samotný text buňky. Příklad jednoduché tabulky:

```
\begin{tabular}{|l|c|}
\hline
Stát & 1970/1971\\
\hline
ČSSR & 0,84\\
Dánsko & 12,74\\
\hline
\end{tabular}
```

Opět existuje řada dalších příkazů, které upravují vzhled sázené tabulky.

Tabulky lze ještě zapouzdřit do prostředí `table`. Je definováno proto, aby se uživatel nemusel zabývat umístěním tabulky v textu podle toho, kolik místa na stránce v dané chvíli zbývá. Systém sám umístí zapouzdřený objekt, kterým vůbec nemusí být tabulka, buď na aktuální stránku, nebo na stránku další, pokud již na dané stránce není dost místa. Prostředí `table` má nepovinné parametry, které určují jednopísmennými zkratkami „h“, „t“, „p“ a „b“ uživatelské preference umístění tabulky na stránce.

Specifickými prostředími $\text{\TeX}$ u jsou prostředí pro sazbu matematiky. Jde o prostředí `math`, `displaymath` a některá další. Jelikož tato prostředí postrádají svůj protějšek v HTML, které zobrazení složitějších matematických výrazů řeší většinou vložením obrázku nebo použitím skriptovacího jazyka, a tudíž konverze $\text{\TeX}$ ového matematického seznamu do jiného jazyka není obecně triviální záležitostí, nebudeme se těmito prostředími na tomto místě zabývat.

6.1. OBECNĚ O FORMÁTOVÉ KONVERZI

Vyjdeme z předchozí definice textového formátu. Při konverzi mezi dvěma textovými formáty půjde o převod jazyka jednoho textového formátu na jazyk druhého textového formátu. Rozdělili jsme vyjadřovací prostředky textového formátu na vlastní text a interpretační výrazy, nebo také příkazy. Při konverzi textového formátu lze většinou předpokládat, že vlastní text nepodlehne podstatným změnám, tedy že konverze jej zanechá v původním tvaru. Pak lze proces konverze vyjádřit jako nahrazení interpretačních výrazů výchozího formátu interpretačními výrazy cílového formátu. Obecně však prosté nahrazení není dostačující. Syntaxe jednoho formátu může deklarovat jiné řazení jednotlivých řetězců textu a interpretačních výrazů, které tvoří celek konvertované struktury. V praxi jde například o parametry příkazů, jejichž syntaxe zápisu je u různých formátů různá. Potom musí konverze zajistit i správné umístění parametrů mezi symboly cílového formátu, aby jejich požadovaný význam zůstal zachován.

Rozdělili jsme textové formáty na strukturované a nestrukturované z textového hlediska. Provedme další rozdělení. Nazvěme struktury, které obsahují strukturované textové soubory, textovými strukturami, nebo též strukturami vyšší úrovně, a struktury nestrukturovaných souborů, pokud jsou nějaké takové přítomny a nelze je označit za textové struktury, strukturami netextovými, nebo strukturami nižší úrovně. Nižší proto, že v praxi je běžné, že při zobrazování textového formátu jsou textové struktury vyšší úrovně, tedy obecné textové struktury, převedeny do vyjadřovacích prostředků jazyka zobrazovacího zařízení, resp. jazyka obsahující povely vhodné k vyjádření vizuální podoby dokumentu, což jsou většinou struktury jiné než textové. Pro jednoduchost budeme mezi netextové struktury řadit i jiné prvky jazyka interpretačních výrazů nestrukturovaného formátu, jakým může být např. posun aktuálního bodu zobrazovacího zařízení apod. Kombinaci takového příkazu s následným textem nelze totiž považovat za strukturu netextovou ani jinou, protože se jedná o příkaz platný sám o sobě a text, který s tímto příkazem souvisí jen proto, že po něm bezprostředně následuje. V dalším textu budeme strukturou pro zjednodušení rozumět prvek jazyka interpretačních výrazů, který v tomto jazyku strukturu vyjadřuje, a nebudeme uvažovat vlastní text, který struktura obsahuje.

Pak mohou nastat tyto případy konverze struktury textového formátu:

1. Struktura zdrojového formátu má svůj protějšek ve struktuře cílového formátu. To znamená, že cílový formát obsahuje strukturu, která má stejný nebo velice podobný význam jako struktura zdrojového formátu. Pak je konverze triviální a spočívá v nahrazení struktury jinou v patřičné syntaxi. V tomto případě nezáleží na tom, zda se jedná o struktury nižší úrovně, nebo o strukturu vyšší úrovně.

2. Struktura zdrojového formátu nemá svůj protějšek ve struktuře cílového formátu, avšak lze ji nahradit kombinací struktur cílového formátu. Jedná se často o konverzi strukturovaného textového formátu na méně strukturovaný, kde je přítomnost určité struktury v textu řešena přítomností struktur nižší úrovně. Tento proces je ve většině případů nevratný. Lze si představit, že opačná konverze proběhne za předpokladu, že existuje deklarace posloupnosti struktur nižší úrovně, jejíž výskyt indikuje přítomnost struktury vyšší úrovně. Pak stačí zaměnit tuto posloupnost danou textovou strukturou. Ovšem obecně výskyt takovéto posloupnosti není dostačující k indikaci přítomnosti vyšší struktury. Ta je totiž spjata s obsahem textové části struktury, jejíž analýza není, jak bylo uvedeno výše, triviální záležitostí. Druhou námitkou proti vratnosti konverze je, že posloupnost struktur nižší úrovně nemusí být fixní, totiž že pořadí struktur v posloupnosti lze různě obměňovat. V takovém případě by musely existovat deklarace pro všechna možná pořadí těchto struktur.

3. Struktura zdrojového formátu nemá svůj protějšek v cílovém formátu a nelze tuto strukturu nahradit kombinací struktur cílového formátu. Pak je konverze nemožná. Struktura musí být vynechána, nebo je nahrazena strukturou nesoucí informaci o výskytu nepřevoditelné struktury. V praxi se může jednat například o konverzi obrázku, přítomného ve formátu, který není čistě textově strukturovaný do formátu, který nedisponuje prostředky pro obsazení obrázku. Pak lze v takovémto formátu uvést běžnou textovou poznámku, že na konkrétním místě v textu se nacházel obrázek a zmínit ještě zdroj.

Probíhá-li konverze mezi formáty téže úrovně, většinou existují podobné struktury v obou formátech, a proto je konverze obecně s většími či menšími obtížemi možná. Ideální případ, kdy by oba formáty obsahovaly tytéž struktury a konverze by pak mohla být provedena jako aplikace bijektivní funkce jedním i druhým směrem, je však patrně prakticky nereálný.

Větší potíže také nenastávají, má-li být provedena konverze z dobře strukturovaného formátu na nestrukturovaný formát. Zde je však nutné mít představu o tom, jak ta která struktura by měla být zobrazena, jelikož nestrukturované

formáty jsou obvykle přímo vázány na konkrétní způsob zobrazení textu. Pokud se stanoví posloupnosti struktur nižší úrovně, kterými se provede nahrazení struktur vyšší úrovně, konverze proběhne bez problémů. V praxi je takováto konverze běžně aplikována při zobrazování a tisku SGML dokumentů.

Naproti tomu konverze z nestrukturovaného formátu na strukturovaný vyžaduje obecně lidský faktor a nemůže být provedena automatizovaně. Princip byl nastíněn v bodu 2. výše. Za současného stavu výpočetní techniky je jen stěží představitelná implementace algoritmů, které v nestrukturovaném formátu naleznou a vymezí konkrétní logické struktury, které jsou závislé na významu jazykových — z lidského hlediska — celků textu (slov a vět), obsaženého v daném dokumentu.

6.2. MOŽNOSTI KONVERZE L^AT_EX–HTML

Zvažme nyní možnosti konverze mezi textovými formáty L^AT_EX a HTML.

L^AT_EX je formát určený primárně pro pořizování dokumentů, u kterých se předpokládá výstup na tiskárně, resp. osvitové jednotce. Kromě struktur textových obsahuje i jiné, které se váží buď na specifikaci vzhledu textových struktur, nebo na vyjádření jiných struktur, než textových. Struktury a příkazy pro upřesnění vizuální podoby nejsou jednoznačně odděleny od textu, ve mnoha případech se nacházejí na téže příkazové úrovni, jako hlavní textové struktury. Ve výjimečných případech nalezneme v L^AT_EXu nestrukturované prvky, často se jedná o prvky které si deklaruje nad formátem sám uživatel.

Formát HTML je primárně používán jako prostředek hypertextové komunikace v rámci fenoménu WWW (World Wide Web). Ačkoli jsou zde přinejmenším srovnatelné předpoklady pro vznik formátu, který před textem upřednostní vizuální podobu dokumentu, jsou prostředky pro vyjádření grafických prvků v rámci samotného HTML silně omezené. Specifikace vzhledu textových struktur je chápána jako dodatečná a tak je také zapisována v rámci dokumentu, konkrétně jako parametry v rámci struktury. Důraz na strukturovanost formátu tak omezuje možnosti grafického vyjádření, které jsou proto řešeny buď vhodnými, často spornými kombinacemi dostupných prvků, např. tabulek, nebo použitím dodatečných prostředků, které nejsou přímou součástí HTML, např. tzv. Java scriptů.

V obou formátech existují relativně podobné nebo ztotožnitelné struktury, jejichž konverze je možná. Například struktura <BODY> z HTML může být

plně ztotožněna se strukturou prostředí `document` z $\text{\LaTeX}$ u. Podobně struktura `<UL>` je ztotožnitelná se strukturou prostředí `itemize`.

Existují však také struktury, které svůj protějšek v druhém formátu nemají. V případě formátu $\text{\LaTeX}$ je to takřka celý složitý matematický aparát nebo prostředí `picture`. Dále ryze $\text{\TeX}$ ová práce s boxy a jejich možné umísťování kamkoliv na tisknutelnou oblast.

V případě HTML jde například o prvky interaktivní prezentace, jako možnost vložení hyperodkazu nebo vizuální formuláře.

Do třetice existují struktury, jejichž konverze je v zásadě možná s určitými úpravami, které přizpůsobí strukturu jednoho formátu struktuře druhého formátu. Tak lze teoreticky převést $\text{\LaTeX}$ ové prostředí `tabbing` na element `<TABLE>` v HTML, nebo širokou škálu použitelných písem $\text{\LaTeX}$ u na omezený aparát elementu `<FONT>`, jehož použití je nedoporučeno.

Specifickým rysem konverze $\text{\LaTeX}$ –HTML zůstane převod nově definovaných struktur a řídicích sekvencí formátu $\text{\LaTeX}$. Každý uživatel tohoto formátu si může vytvořit vlastní deklarace, jejichž použití snižuje pravděpodobnost úspěchu jakékoliv konverze, která disponuje konstantním výčtem převodových transformací.

U konverze mezi dvěma formáty je ovšem třeba vymezit smysl vlastního převodu. Na jedné straně může jít o dosažení téhož nebo podobného vizuálního efektu, který by byl dosažen běžným zpracováním druhého formátu. Například provést konverzi $\text{\LaTeX}$ ového dokumentu tak, aby jeho vystavení na síti Internet ve formátu HTML bylo adekvátní a předalo uživateli tytéž informace podobnou vizuální cestou, jako kdyby uživatel četl původní dokument.

Na druhé straně může být konverze prováděna proto, aby uživateli usnadnila práci při tvorbě nového dokumentu v jiném formátu, který bude obsahovat víceméně shodné textové části. Existuje-li např. dokument ve formátu HTML a předpokládá se jeho tisk, který by měl být reprezentativnější, než je tisk z běžného prohlížeče (např. Netscape Communicator), je vhodné, aby proběhla konverze do formátu $\text{\LaTeX}$, která provede základní nahrazení původních struktur na nové a vykoná tak rutinní práci za uživatele. Ten pak vzhled dokumentu „doladí“ přidáním dalších struktur a redefinicí původních.

6.3. PARAMETRIZOVANÁ KONVERZE

Na tomto místě je vhodné vnést myšlenku *parametrizace konverze*. Ve výše uvedeném příkladě by bylo velmi praktické, aby uživatel sám stanovil, na jaké L^AT_EXové sekvence se mají elementy HTML převádět. Díky programovatelnosti T_EXu pak uživatel pouze deklaruje význam použitých sekvencí, které již v textu vymezují původní struktury. Např. může chtít, aby element <A> konvertoval takovým způsobem, že v místě výskytu ukončující značky se zapíše sekvence `\footnote`, jejímž parametrem bude řetězec z atributu `href` elementu <A>. Výsledkem bude zobrazení hyperodkazu jako poznámky pod čarou. Místo `\footnote` může jiný uživatel uvést sekvenci `\alist`, kterou si posléze sám deklaruje a která vytvoří a včlení někam za dokument rejstřík odkazů v dokumentu nalezených.

Parametrizaci konverze je možné řešit mapovacím souborem, v němž uživatel sám deklaruje převod jednotlivých sekvencí mezi formáty. Takový mapovací soubor pak lze pružně měnit podle povahy konvertovaného dokumentu a podle použitého cílového aparátu od uživatele k uživateli. Vhodný formát mapovacího souboru také umožní určité struktury nekonvertovat vůbec — vynechat je, a některé konvertovat jako pouhý text bez řídicích symbolů. Spolupráce konverzního programu a uživatele pak může dát optimální výsledek v případě, kdy konverze je vzhledem k povaze dotčených formátů nejednoznačná, kdy existuje více možností konverze a nelze objektivně stanovit jednu jako nejlepší.

Již byla řeč o programu T_EX, jeho primitivních řídicích sekvencích a formátech. Nyní se podívejme blíže, jakým způsobem tento program zpracovává zdrojový vstupní text. Řeč již nebude o formátu L^AT_EX, ale o fundamentálním způsobu vnímání programu T_EX. Co je zde uvedeno, platí samozřejmě i pro formáty, definované nad základními T_EXovými primitivami, neboť se vždy jedná jen o soubor nově definovaných maker, která budou při zpracování vždy rozvinuta na posloupnosti primitivů, pomocí nichž jsou definována.

Autor programu T_EX, Donald Knuth, když popisuje ve své knize (KNUTH, 1986) způsob zpracování zdrojového textu, přirovnává algoritmy programu T_EX k uspořádání živočišného trávicího traktu a hovoří o jednotlivých fázích zpracování jako o očích, ústech, hltanu a žaludku. Petr Olšák (OLŠÁK, 1998) zavedl poněkud exaktnější definice a rozděluje proces zpracování na pomyslný input procesor, token procesor, expand procesor a hlavní procesor. Přidržíme se těchto termínů i v této práci.

7.1. PROCESORY T_EXU

Nejprve je na vstupu připraven zdrojový soubor. *Input procesor* slouží k tomu, aby ze zdrojového souboru četl jednotlivé řádky. Hlavní náplní činnosti tohoto procesoru je úprava řádků do tvaru, v jakém jsou požadovány na úrovni T_EXu, jde tedy o konverzi mezi způsobem zápisu řádků v daném operačním systému, na němž je T_EX implementován, a způsobem zápisu řádků, v jakém je chtějí mít algoritmy T_EXu. V praxi jde o úpravu posloupnosti znaků, vyjadřující konec řádku, a překódování mezi kódováním použitým v operačním systému a vnitřním kódováním T_EXu, tedy ASCII. Při úpravě konce řádku jsou odstraněny všechny mezery za posledním znakem jiným, než mezera, a je doplněn znak podle hodnoty uložené do primitivu `\endlinechar`. Vidíme, že znak pro konec řádku může být pružně měněn. Obvykle je však nastaven na hodnotu `^M` (znak s kódem 13 v kódování ASCII, viz dále).

Takto zpracovaný řádek je následně čten tzv. *token procesorem*. Token je výraz, který se z angličtiny nepřekládá. Je dvojího druhu: buď se jedná o řídicí sekvenci, nebo se jedná o dvojici (znak, kategorie). Znak je reprezentován svou ASCII hodnotou. Kategorie je číslo, které je přiděleno každému znaku z vnitřní znakové sady a které má nějaký význam. Kategorii je celkem 15. Následující tabulka uvádí jejich význam (OLŠÁK, 1998):

0	uvození řídicí sekvence	\
1	otevření skupiny	{
2	zavření skupiny	}
3	přepínač matematického režimu	\$
4	separátor v tabulkách	&
5	konec řádku	^~M
6	označení parametrů maker	#
7	konstruktor mocniny	^
8	konstruktor indexu	-
9	ignorovaný znak	^^@
10	mezera	
11	písmeno	A–Z, a–z
12	ostatní znaky	<i>ostatní</i>
13	aktivní znaky	např. ~
14	znak uvozující komentář	%
15	nedovolený znak	^^?

Input procesor dodává token procesoru jednotlivé znaky a ten je převádí na posloupnost tokenů. Token typu řídicí sekvence vznikne v případě, že je na vstupu znak s kategorií 0. Všechny znaky s kategorií 11, které následují za tímto znakem, až po první znak s jinou kategorií jsou přečteny a tvoří jméno řídicí sekvence. Tak např. přečtením sekvence `\par` vznikne token *par*, který je dále brán jako nedělitelný symbol. Tokenům typu řídicí sekvence není přiřazena kategorie. Nevznikne-li token typu řídicí sekvence, vznikne token typu uspořádaná dvojice (znak, kategorie). Např. přečtením znaku L vznikne token L_{11} . Tento token představuje povel pro vysazení písmene L. Přečtením znaku { vznikne token $\{_1$, který reprezentuje příkaz k otevření skupiny.

Existují speciální případy, které token procesor ošetří jiným než uvedeným způsobem. Je-li na vstupu více mezer, je vytvořen pouze jeden token s kategorií 10, tedy jediná mezera. Je-li na vstupu znak s kategorií 14, je zbytek řádku input procesoru ignorován a žádný token není vytvořen. Následují-li po sobě dva konce řádku, tedy je na vstupu prázdný řádek, je vytvořen token *par*. Explicitně vznikne stejný token zápisem primitivu `\par`. Dále jsou ignorovány všechny mezery po výskytu řídicí sekvence. Např. následující dva zápisy budou mít shodný efekt.

```
\noindent \par
\noindent\par
```

U prvního zápisu jakoby mezera nebyla vůbec zapsána, bude odstraněna na úrovni token procesoru. Mezerou se v tomto případě myslí i konec řádku a znak tabulátor. A nakonec, je-li na vstupu znak s kategorií 7, běžně znak „^“,

a následuje-li znak se stejným ASCII kódem, jsou načteny další dva znaky v případě, že náleží mezi a–f nebo 0–9, v jiném případě je načten znak jeden, pakliže jeho ASCII hodnota je nižší než 128. V prvním případě je posloupnost pokládána za zápis znaku, jehož ASCII hodnota je hexadecimálně rovna uvedenému dvojpísmennému výrazu. V druhém případě je vytvořen znak, jehož ASCII hodnota je o 64 nižší, než ASCII hodnota uvedeného znaku. Znaky podlehnou výše uvedenému zpracování token procesorem.

V *expand processoru* probíhá rozvoj (expanze) řídicích sekvencí na primitivní řídicí sekvence. Jak bylo již výše při popisu formátu $\text{\LaTeX}$ řečeno, $\text{\TeX}$ disponuje určitým počtem zabudovaných řídicích sekvencí, primitivů. Mezi tyto sekvence patří i několik sekvencí, s jejichž pomocí je možné deklarovat nové řídicí sekvence, které sestávají z primitivních, nebo dříve deklarovaných řídicích sekvencí. Úkolem *expand processoru* je, aby do hlavního procesoru vstupovaly pouze ty řídicí sekvence a tokeny, které již nejdou dále rozvinout. Existují drobné výjimky z tohoto pravidla, které však nebudeme zmiňovat.

Nové řídicí sekvence jako posloupnosti jiných tokenů mohou být deklarovány pomocí primitivů `\def`, `\edef`, `\gdef` a `\xdef`. Příklad:

```
\def\radek{12dd}  
\def\skok{\vskip\radek}  
\edef\jinyskok{\vskip\radek}
```

Na prvním řádku v ukázce se deklaruje řídicí sekvence `\radek`, která po expanzi v místě, kde bude zapsána, dá posloupnost 12dd. Rozdíl mezi `\def` a `\edef` je v tom, že při `\edef` se provádí ihned při deklaraci expanze těla definice, tedy posloupnosti, která následuje za jménem definice mezi dvěma složenými závorkami. Při deklaraci pomocí `\def` se provádí expanze až při výskytu deklarované sekvence. Tak po provedení definic v naší ukázce bude zápis sekvence `\skok` znamenat totéž, co `\vskip\radek`, zatímco zápis `\jinyskok` bude znamenat `\vskip12dd`. Sekvence `\radek` při použití `\edef` podlehne rozvoji, zatímco `\vskip` nikoliv, jedná se totiž o primitivní sekvenci. Primitivy `\gdef` a `\xdef` znamenají po řadě totéž, ovšem deklarace jsou globální; v prvním případě totiž platily pouze v rámci skupiny, ve které byly zapsány. Skupina se vymezí např. znaky `{}`.

Deklarovaná řídicí sekvence, neboli makro, může mít *parametry*. Např.:

```
\def\skok#1{\vskip#1}  
\skok{12pt}
```

Zde je `\skok` makro s jedním parametrem. Výraz `#1` mezi deklarovanou sekvencí a první složenou závorkou znamená, kam bude parametr zapisován. Druhý výskyt tohoto výrazu v ukázce značí, odkud bude parametr volán při expanzi těla definice. Jako parametr se do makra dosadí implicitně vždy pouze jeden token. Chceme-li, jako v naší ukázce, dosadit delší posloupnost, je třeba ji uzavřít do skupiny, v tomto případě mezi znaky `{}`. Jinou možností je deklarovat parametry se separátory. Např.:

```
\def\kniha#1--#2.{Autor: #1, název: #2.}
\kniha Z.~Žáček--Nad šálkem plným vůně.
```

Zde sekvence `--` ukončuje zápis prvního parametru a uvozuje zápis druhého parametru. Tečka ukončuje zápis druhého parametru. Tyto sekvence jsou označovány jako separátory. Separátorem může být prakticky cokoli, nesmí to být ale např. složené závorky nebo jiné znaky s kategorií 1 a 2. Dále platí, že separátor se nesmí vyskytovat uvnitř vnořené skupiny. Proto zápis `\kniha A{--}B.` nepředstavuje ukončené načítání prvního parametru makra.

Řídící sekvence může být deklarována maximálně s devíti parametry. U definic, které budou vnořené do jiných definic, se pro vyjádření parametru zdvojuje znak „`#`“. Proto správný tvar vnořené definice je např.:

```
\def\aa#1{a#1\def\b##1{b##1\def\c####1{b####1}}}
```

Jiným způsobem, jak deklarovat řídící sekvenci, je primitiv `\let`. Např. zápis `\let\odst=\par` deklaruje novou řídící sekvenci `\odst`, která bude mít též význam, jako sekvence `\par` v době, kdy proběhlo přiřazení. Pozdější redefinice sekvence `\par` nemá na význam sekvence `\odst` vliv.

Ne všechny neexpandovatelné tokeny z `expand` procesoru přejdou do hlavního procesoru. Jedná se především o tokeny ovlivňující proces expanze. Jsou to primitivy `\expandafter`, `\noexpand`, primitivy pro implementaci větvení a některé jiné. Primitiv `\expandafter` je nástrojem k oddálení nechtěné expanze. Token, který následuje za tímto primitivem, nepodlehne expanzi hned, ale až po expanzi tokenu, který následuje jako druhý. Při zápisu `\expandafter\aa\b` tak napřed bude expandovat sekvence `\b` a teprve po ní sekvence `\aa`. Většinou se počítá s tím, že takovéto chování nějakým způsobem ovlivní sekvenci `\aa`. Primitiv `\noexpand` umožňuje označit token, který se nebude expandovat v aktuálním průchodu `expand` procesorem. S výhodou toho využijeme při použití `\edef`, kdy nechceme, aby se určité části makra rozvinuly. Primitivy pro větvení zpracovacího procesu jsou primitivy, jejichž jméno začíná slovem *if*. Sekvence typu *if* obsahují tělo, které je uzavřeno mezi uvozující sekvenci, např. `\ifx` a ukončující sekvenci `\fi`, která je pro

všechny primitivy stejná. Toto tělo může obsahovat sekvenci `\else`. V tom případě se provede část těla před `\else`, pokud je podmínka vyhodnocena pozitivně, jinak se provede část těla za `\else`. Pokud se `\else` nevyskytuje, provede se celé tělo pouze v případě, že je podmínka vyhodnocena kladně.

Primitivů typu *if* je v $\TeX$ u celá řada, existuje dokonce mechanismus, jak deklarovat nové. Nebudeme je všechny jmenovat, na tomto místě uvedeme pouze příklad. Sekvence `\ifx` testuje, zda jsou dvě makra definována shodným způsobem. Podle toho větví zpracování. Např.:

```
\def\aa{a}
\def\bb{b}
\let\c=\a
\ifx\aa\bb ano\else ne\fi
\ifx\aa\c ano\else ne\fi
```

Do hlavního procesoru poputuje `ne` `ano`, protože `\a` není definováno stejně, jako `\b`, zatímco `\c` `ano`.

Výkonným prvkem $\TeX$ ového zpracování souboru je tzv. *hlavní procesor*. Pokud se až sem dostane token, jehož význam není definován, nastane vstupní syntaktická chyba. V opačném případě je token od expand procesoru přebrán a vykonána akce, která je tokenu přiřazena. Token může reprezentovat úplnou akci sám o sobě, nebo může být upřesněn parametry. V druhém případě si hlavní procesor token uschová a vyžádá si od expand procesoru potřebné parametry. Opět, nejsou-li na vstupu připraveny, vznikne syntaktická chyba.

Tokeny, které vstoupí do hlavního procesoru, již jsou většinou dále neexpandovatelné. Jde o primitivy, přepínač fontu (`\font`), registry a jiné tokeny.

Registry jsou paměťová místa, kde je uložena určitá hodnota. Přístup k registrům je realizován zápisem řídicí sekvence, které je přiřazen význam registru buď implicitně, nebo explicitně, např. `\parskip`, nebo zápisem primitivu s číslem registru `\box255`. Odkazem na registr je i zápis `\fontdimen3\ptmr` (zjištění údaje o fontu). Lze definovat nové registry různých typů, např. numerické, délkové, výplňkové (s hodnotou roztažení a stažení). Zvláštním registrem je registr obsahující posloupnost tokenů (může expandovat na úrovni expand procesoru) a registr obsahující tzv. *box* (viz dále).

Jakou akci $\TeX$ vykoná po obdržení tokenu záleží na tom, v jakém *režimu* se nachází. Rozlišuje šest režimů:

- hlavní vertikální režim;
- vnitřní vertikální režim;
- horizontální režim;
- vnitřní horizontální režim;
- vnitřní matematický režim;
- vnější matematický režim.

7.2. REŽIMY T_EXU

Hlavní vertikální režim je výchozí. Prakticky představuje sestavenou stránku dokumentu. Obsahuje kompletované řádky, odstavce, vertikální výplňky, boxy a vše co bylo umístěno na stránku mimo jiný režim. Prvky vertikálního seznamu, tj. posloupnost vertikálního materiálu vytvořená ve vertikálním režimu, jsou řazeny pod sebe a zarovnáván podle referenčních bodů kompletovaných boxů. Vertikální seznam podléhá stránkovému zlomu.

Horizontální (odstavcový) režim slouží k sestavení řádků odstavce. Sestává z horizontálního materiálu, tvoří horizontální seznam, který podléhá algoritmu sestavení odstavce, tedy řádkového zlomu. Typicky obsahuje vysázená písmena, mezislovní mezery, odstavcové zarážky a boxy. Elementy jsou sestavovány vedle sebe podle referenčního bodu na účaří. Režim je obvykle ukončen výskytem sekvence `\par` nebo tokenem se stejným významem. Jindy je ukončen vynuceným přechodem do hlavního vertikálního režimu. Může být také přerušen přechodem do některého vnitřního režimu.

Vnitřní vertikální režim je spuštěn při kompletaci vnitřního vertikálního seznamu, typicky sekvencí `\vbox{...}`. Vnitřní horizontální režim je spuštěn při kompletaci vnitřního horizontálního seznamu, typicky sekvencí `\hbox{...}`.

Matematický režim má speciální pravidla a princip zpracování tokenů se v něm podstatně liší od zpracování tokenů v jiných režimech. Nejprve je načten celý obsah režimu a převeden na posloupnost relativních elementů, které nemají ještě pevnou velikost (matematický seznam), teprve poté je celý matematický seznam převeden buď na horizontální seznam, jedná-li se o vnitřní matematický režim, nebo na horizontální seznam ve vertikálním seznamu, jedná-li se o vnější matematický režim.

Vnější vertikální seznam je hlavní, z něj T_EX přechází do ostatních režimů. Do těch pak mohou být vnořeny ostatní režimy, kromě vnějšího vertikálního. Režimy, resp. jejich seznamy mohou být vnořeny libovolně hluboko, pokud to dovolí paměťová omezení implementace T_EXu.

Základním stavebním kamenem $\TeX$ u je *box* (nepřekládá se). Boxem je stránka, je jím řádek, tabulka, za speciální druh boxu můžeme považovat i vysázený znak, nejde však přímo o box. Do boxu lze explicitně pomocí k tomu určených primitivů uzavřít cokoliv. Jde o paměťové místo, kam jsou uloženy elementy, tvořící obsah boxu. S boxem je pak zacházeno jako s nedělitelným elementem. Box je dán čtyřmi charakteristikami, z nichž tři jsou rozměrové veličiny a čtvrtá představuje samotný obsah boxu. Každý box má referenční bod, k němuž jsou vztaženy jeho rozměry a podle nějž se daný box umísťuje na aktuální pozici v horizontálním nebo vertikálním seznamu. Rozměry boxu udává jeho šířka (napravo od referenčního bodu), výška (nahoru) a hloubka (dolů). Boxy mohou být násobně vnořeny.

Hlavní procesor $\TeX$ u sestavuje tedy boxy, znaky, pružné a pevné výplňky a některý další materiál; celá tato činnost je determinována tokeny, které jsou brány z expand procesoru, a řízena zabudovanými algoritmy, např. algoritmem stránkového nebo řádkového zlomu.

Jakmile je kompletováno dost materiálu v hlavním vertikálním seznamu, je spuštěn algoritmus stránkového zlomu. Ten zkompletuje `\box255`, který obsahuje optimální variantu zalomené strany. Pak je volána výstupní rutina `\output` a proveden její obsah, který uživatel sám stanoví, běžně je však rutina deklarována jako součást použitých formátů. Typicky obsahuje povel `\shipout`, který slouží pro zápis boxu do DVI souboru, výstupního binárního souboru $\TeX$ ového překladu.

7.3. PRÁCE SE SOUBORY NA ÚROVNI $\TeX$ U

Kromě samotného zpracování textu umožňuje $\TeX$ některé speciální činnosti, jako je práce se soubory, jejich otevření, načtení celých souborů nebo jednotlivých řádků, a také zápis do souboru. Pro potřeby této práce zmíníme některé primitivy.

K načtení souboru jako celku se použije primitiv `\input`. Např. `\input data.tex` načte soubor `data.tex`. Přípona $\TeX$ je implicitní, nemusí se psát. Chceme-li číst soubor po řádcích, je nutné jej nejprve otevřít primitivem `\openin`. Za primitivem se uvede číslo otevřeného souboru a přiřadí se mu fyzické jméno. Aby nebyla čísla souborům přiřazována stochasticky, bylo ve formátu `plain` deklarováno makro `\newread`, které uživateli řídící sekvenci přiřadí význam čísla, které pak zastupuje soubor. Uzavření souboru se provede pomocí `\closein`. Jeden řádek souboru se přečte sekvencí `\read\cislousub to`

`\radek`. Prove se akce, která je ekvivalentní k `\def\radek{přečtený řádek}`. Pokud při načítání řádku dojde k otevření skupiny znakem „{“ a před koncem řádku není skupina ukončena, probíhá načítání do doby, než je skupina uzavřena znakem „}“. Jednoduchý příklad:

```
\newread\soubor
\openin\soubor=data.txt
\read\soubor to \prvniradek
Toto je první řádek: \radek \par
\closein\soubor
```

Podobně primitivy `\openout`, `\closeout`, `\write` a makro `\newwrite` slouží pro práci při zápisu do souboru. Primitiv `\write` je však vykonán až při provádění příkazu `\shipout` ve výstupní rutině. Aby byl proveden okamžitě, je třeba uvést jej prefixem `\immediate`. Např.:

```
\newwrite\soubor
\openout\soubor=data.txt
\immediate\write\soubor{Toto je první řádek souboru.}
\closeout\soubor
```

7.4. PROGRAM T_EX JAKO NÁSTROJ KONVERZE

Stávající konvertory mezi formáty HTML a L^AT_EX jsou většinou psány ve vyšších programovacích jazycích, např. C nebo Pascal. Jakkoliv je takový jazyk výhodný pro tuto činnost, je nutné implementovat do daného konvertoru algoritmy pro syntaktickou analýzu zdrojového textu. Tyto mohou být poměrně složité, a v případě konverze formátu typu T_EX, který je sám o sobě výkonným programovacím jazykem se mohou minout účinkem. Propracovaná struktura T_EXového překladu totiž umožňuje nejrozumnější triky, které bez hlubší analýzy zdrojového kódu za znalosti vestavěných algoritmů nejsou na první pohled zřejmé. Formát L^AT_EX do jisté míry abstrahuje od podobných praktik. Přesto je v konečné fázi na uživateli, zda včlenění do dokumentu něco ze zmíněných triků a způsobí tak dokument těžce převoditelným do jiného formátu.

Samotný program T_EX však obsahuje algoritmy, které dokáží rozkrýt smysl použitých řídicích sekvencí. Umožňují dokonalou syntaktickou analýzu T_EXového formátu, jelikož daný formát s těmito algoritmy počítá a byl pro ně navržen. Někde uvnitř pomyslného hlavního procesoru T_EXu jsou tak všechna uživatelská makra a nově definované řídicí struktury textové i jiné převedeny

na množinu obsahující konečný počet řídicích sekvencí — primitivů, a následně poté na text, který bude zapsán do výstupního DVI souboru. To je důvod, proč by bylo vhodné pro konverzi mezi $\text{\TeX}$ ovým a jiným formátem vytvořit konvertor na bázi programu $\text{\TeX}$. Předpokládáme-li konverzi $\text{\LaTeX}$ –HTML, pak na vstupní straně by nemusela být programována žádná syntaktická analýza.

Na rozdíl od programovacího jazyka není však $\text{\TeX}$ stavěn na formátovou konverzi. Disponuje omezeným mechanismem pro práci se soubory, s řetězci, s proměnnými. Nepředpokládá rozsáhlejší výstup zpracovávaného textu do jiného souboru (kromě takových záležitostí, jako je např. tvorba obsahu apod.). Možnosti konverze jsou proto u programu $\text{\TeX}$ omezené právě z těchto hledisek.

Uvedené nedostatky lze odstranit vestavěním patřičných algoritmů do vlastního zdrojového kódu programu $\text{\TeX}$, který je veřejně k dispozici, a jehož úpravy pro nejrůznější účely existují (např. konvertor $\text{pdf2}\text{\TeX}$). Přidáním schopnosti patřičné práce s řetězci a soubory a změnou výstupních algoritmů by bylo lze vytvořit optimální konvertor mezi $\text{\TeX}$ ovým formátem a jiným. To by však představovalo zásadní změnu programu, což je činnost relativně náročná a předpokládá vytvoření nových instalací programu pro patřičné platformy.

Demonstrujme schopnosti $\text{\TeX}$ u v oblasti formátové konverze i bez zásahů do samotného programu, pouze s využitím aparátu $\text{\TeX}$ ového zpracování vstupního souboru, syntaktické analýzy a práce vestavěných primitivů. Možné způsoby $\text{\TeX}$ ové konverze lze rozdělit následovně:

- překlad s přímým zápisem do souboru;
- překlad se zápisem ve výstupní rutině;
- překlad s výstupem do DVI.

V prvních dvou případech funguje $\text{\TeX}$ jako konvertor (filtr), tzn. že vstupní textový soubor ve vstupním textovém formátu překládá do textového výstupního souboru ve výstupním textovém formátu. Budeme uvažovat případ, kdy na vstupu bude soubor ve formátu $\text{\LaTeX}$ a na výstupu soubor HTML a zvážíme i možnost cesty opačné. Ve třetím případě je výstupem $\text{\TeX}$ ového zpracování běžný binární DVI soubor. Na vstupu budeme předpokládat soubor ve formátu HTML. $\text{\TeX}$ tak zastoupí prohlížeč HTML.

Překladem s přímým zápisem do souboru je rozuměn takový způsob konverze, kdy $\text{\TeX}$ provádí zápis jednotlivých překládaných struktur během vlastního procesu zpracování vstupního souboru. Jakmile je znám nějaký prvek, jehož

konverzi již neovlivní další načtená data, je proveden překlad a výstup do souboru. Vzhledem k možnostem $\TeX$ u je tato cesta prakticky neschůdná. Během zpracování vstupu je totiž token procesorem generováno velké množství tokenů, které nereprezentují finální text dokumentu, a které podléhají expanzi až v okamžiku, kdy je v hlavním procesoru kompletován z těchto tokenů box. Primitiv pro zápis do souboru `\write` však potřebuje jako svůj parametr řetězec a nikoliv box. Proto provedeme-li konverzi posloupnosti jen částečně expandovaných (např. pomocí `\edef`) tokenů a vložíme-li je do parametru primitivu `\edef`, bude výstupní formát obsahovat nežádoucí znaky a prvky, tzn. že konverze nebude dokonalá. Řešit lze tuto situaci předchozím kompletováním boxu, extrakcí jeho obsahu (`\unvbox`, `\unhbox`) a následným zápisem do souboru, což je složitá cesta, nebo ještě složitější analýzou vstupního textu, která je však v rozporu s předpokladem využití zabudovaných algoritmů pro syntaktickou analýzu.

Vzhledem ke skutečnostem, uvedeným výše, je vhodné zvolit následující postup. Řídící sekvence vstupního souboru budou deklarovány tak, aby expandovaly na řídící sekvence výstupního souboru (ve významu běžného textu) a braly přitom do úvahy parametry, které umístí při rozvoji řídících sekvencí na patřičné místo (provedou konverzi). Výsledkem po úplné expanzi bude posloupnost znaků, které bude obsahovat výstupní soubor. Poté $\TeX$ zpracuje takto expandovaný text. Podrobí jej vestavěným algoritmům a naláme jej do boxů. Je triviální nechat vypsát obsah boxu do LOG souboru. Výpis je jednoznačný, proto jej lze načíst $\TeX$ em podruhé, filtrovat pouze finální znaky výstupního souboru do řetězců a ty pak vypsát do výstupního souboru. Aby byla činnost zápisu boxu do souboru automatizována, byl jako vhodný box zvolen `\box255`, který obsahuje kompletovanou stránku, a který samočinně po kompletaci zpracuje výstupní rutina `\output`. Pro jednoduchost je vhodné nastavit velikost stránky na jediný řádek. Výstupní rutina pak pomocí `\showbox` zapíše obsah stránky, zde jediného řádku, do souboru. Po zápisu všech stran se proces konverze ukončí. Jelikož je LOG soubor v době překladu otevřen pro zápis, musí proces načtení tohoto souboru a jeho převod na výstupní formát proběhnout při dalším běhu programu $\TeX$. Proto budeme tento způsob konverze označovat jako *dvojfázový překlad*. Vedlejším efektem této cesty překladu je přehledně do jednotlivých řádků zapsaný výstupní soubor, což lze ovlivnit způsobem nalámání řádků konvertovaného textu $\TeX$ em při kompletaci boxu 255.

Takto lze provést překlad sekvencí, jejichž syntaktická analýza je implementována do $\TeX$ u samotného nebo do sekvencí stávajících formátů, na nichž budou deklarovaná makra implementována. Může být např. provedena konverze $\LaTeX$ –HTML, jelikož syntaxe formátu $\LaTeX$ je do značné míry uzpůsobena implicitnímu způsobu „vnímání“ programu $\TeX$. Tento způsob lze

však významně modifikovat vzhledem ke skutečnostem uvedeným v sekci o $\text{\TeX}$ ovém překladu. Patříčnými deklaracemi lze přimět $\text{\TeX}$ k tomu, aby provedl načtení a syntaktickou analýzu souboru zapsaného v ne $\text{\TeX}$ ovém formátu, např. HTML. Následně je možné provést konverzi opačnou než v předešlém případě — HTML– $\text{\LaTeX}$. Aniž by však byla provedena konverze, lze také načtený formát zpracovat v $\text{\TeX}$ u běžným způsobem a připravit sazbu do souboru DVI, u náročnějších aplikací pak následně do PostScriptu. Lze tak simulovat prohlížeč HTML a provádět pomocí $\text{\TeX}$ u tisk internetových stránek.

Dále budou uvedené skutečnosti demonstrovány na praktických příkladech. Nebudeme analyzovat přímý překlad (jak byl zmíněn výše) vzhledem k jeho malé praktické užitečnosti. Nejprve ukážeme na překladu se zápisem ve výstupní rutině konverzi z formátu $\text{\LaTeX}$ do HTML. Následně deklarujeme makra, která umožní syntaktickou analýzu souboru ve formátu HTML a demonstrováme jejich princip na vytvoření sazby ze zdrojového HTML souboru. Zvážíme také možnost konverze HTML– $\text{\LaTeX}$.

Překlad formátu, který bude realizovat výstup konverze při činnosti výstupní rutiny, má dvě fáze. První fáze sestává ze samotné konverze a zápisu výsledků do souboru LOG. Druhá fáze pak spočívá v načtení souboru LOG a úpravě jeho obsahu na konečný soubor ve výstupním textovém formátu.

Demonstrujeme celý proces na konverzi z formátu L^AT_EX do formátu HTML. Nejprve budeme uvažovat fixní konverzní definice, tzn. že konverzní makra budou obsahovat pevně stanovené výstupní sekvence, na které budou vstupní sekvence transformovány. Kvůli přehlednosti a vzhledem k faktu, že je naším cílem pouze demonstrovat možnosti překladu, provedeme konverzi pouze několika vybraných sekvencí.

8.1. PRVNÍ FÁZE KONVERZE

V první fázi sledují překladová makra dva úkoly. Prvním z nich je konverze vstupních řídicích sekvencí na výstupní. Druhým je jejich výstup do souboru. Probereme postupně oba.

8.1.1. Makra pro zajištění konverze

Konverze vstupních řídicích sekvencí na výstupní je provedena redefinicí vstupních, v tomto případě L^AT_EXových řídicích sekvencí. Ideálním případem je, aby každá řídicí sekvence, která se může na vstupu vyskytnout, byla definována tak, aby expandovala na posloupnost znaků, tedy tokenů s kategorií, která znamená pro hlavní procesor povel pro vysázení znaku. Sekvence výstupních znaků je taková, jaká bude zapsána do cílového formátu.

To, že znaky projdou v sestaveném boxu 255 až do výstupní rutiny, zajistí úplnou expanzi všech možných elementů, a zaručí následně přítomnost pouze určité množiny elementů na výstupu. Tato množina bude typicky obsahovat pouze znaky, výplňky, primitivy `\discretionary` a některé boxy.

Je úlohou konverzních maker, aby nebyly obsaženy žádné jiné elementy a text nebyl strukturován do násobně vnořených boxů. Dále je hlavní úlohou konverzních maker, aby zajistila transformaci strukturní textové informace,

obsažené ve vstupním formátu. Informace bude konvertována okamžitě při rozvoji makra do posloupnosti znaků a dále již nebude detekovatelná.

Uvedme několik příkladů transformačních maker. $\text{\LaTeX}$ ový dokument obvykle začíná sekvencí `\documentclass` s několika parametry. Pro jednoduchost předpokládejme, že makro bude mít vždy jen jeden povinný parametr a ostatní budou vkládány třeba přes `\usepackage`. Dokument HTML začíná, pomineme-li DTD deklaraci, pravidelně uvozovací značkou elementu `<HTML>` a elementem `<HEAD>`. Budeme chtít, aby se sekvence `\documentclass` nahradila následujícím textem:

```
<HTML>
<HEAD>
<TITLE>LaTeXový dokument</TITLE>
</HEAD>
```

Pak definujeme konverzní makro následovně:

```
\def\documentclass#1{<HTML>\par<HEAD>\par
                      <TITLE>LaTeXový dokument<TITLE>\par
                      </HEAD>\par}
```

Pokud nebudou znaky „<“, „>“ a „/“ mít nastaveny nějaké speciální kategorie, expandují vždy na povel, který představuje sazbu znaků se stejným kódem ve výstupním řezu písma. Na výstupu v DVI bude tedy požadovaný text. Uvedené znaky mohou být při volbě řezu s jiným kódováním nahrazeny — z důvodu odlišného kódování použitého řezu písma — jinými znaky. Na výstupu do souboru LOG budou však v námi požadovaném tvaru díky vnitřním konverzním rutinám $\text{\TeX}$ u, které zajistí totožnost vstupního a výstupního kódování textových souborů, s nimiž uživatel pracuje. Povely `\par` jsou vloženy mezi text proto, aby byl výstup přehledně zarovnán. Mohou být bez negativního efektu vypuštěny. Jejich účinek závisí na nastavení sazby odstavců (vhodné nastavení viz dále).

Definujme další konverzní makro pro sekvenci `\begin{...}`. Bude testovat, co obsahuje první povinný parametr, a podle toho provede další činnost. Makro podle $\text{\LaTeX}$ ové deklarace pro některá prostředí obsahuje další parametry. Jejich načtení lze provést v závislosti na povinném parametru, např. když jeho hodnota bude `table`. Zde deklarujeme makro pro parametr `document`. Zároveň uvedeme deklaraci pro makro `\end{...}`, opět pouze pro prostředí `document`.

```
\def\document{document}
```

```

\def\begin#1{\def\co{#1}\ifx\co\document<BODY>\par\fi}
\def\end#1{\def\co{#1}\ifx\co\document
\par</BODY>\par</HTML>\fi}

```

Na prvním řádku deklarujeme pomocné makro pro testovací účely. V rámci makra `\begin` pak deklarujeme sekvenci `\co`, která bude obsahovat řetězec zapsaný v parametru `\begin`. Následuje porovnání sekvence `\co` a sekvence `\document`. Pakliže se shodují, je provedena konverze. Pro různá prostředí lze soustavou podmínek typu `\ifx`, která slouží k testu totožnosti definice makra, větvit zpracování prostředí, uvozeného sekvencí `\begin`, podle obsahu prvního parametru a zajistit tak patřičnou konverzi. Makro `\end` je deklarováno obdobným způsobem. Kromě expanze na ukončující značku elementu `<BODY>` obsahuje ještě ukončení elementu `<HTML>`, protože sekvencí `\end{document}` obvykle končí celý L^AT_EXový dokument.

Vlastní odstavcový text v L^AT_EXu je oddělen buď prázdnými řádky, nebo explicitně zapsanou sekvencí `\par`. Oba způsoby nařizují T_EXu ukončit zpracováváný odstavec. Zpracování odstavce začíná buď explicitně povel, např. `\noindent`, nebo implicitně výskytem znaku ve vertikálním režimu. Není tedy vyčleněna řídicí sekvence, která by bezpodmínečně uvozovala odstavec. Naproti tomu HTML vyžaduje, aby každý odstavec začínal uvozující značkou elementu `<P>`. Aby byla zajištěna konverze a mohla být před každý odstavec vložena tato značka, je třeba využít primitivního registru T_EXu `\everypar`. Ten zajistí vložení v něm obsažených tokenů před začátek každého odstavce.

```
\everypar{<P>}
```

Uvedená definice způsobí, že sekvence `<P>` bude vložena před *každý* odstavec. To ovšem znamená, že například expanze sekvence `\documentclass{article}` podle výše uvedené deklarace provede následující výstup:

```

<P><HTML>
<P><HEAD>
<P><TITLE>LaTeXovy dokument</TITLE>
<P></HEAD>

```

Je to proto, že i text typu `<HTML>` nebo `<HEAD>` je na úrovni hlavního procesoru chápán jako běžný odstavcový text, a jelikož těmito texty se uvozují nové odstavce, je před ně vložen text `<P>`. Možné řešení je následující. Deklarujeme pomocné makro `\x`, kterým budeme uvozovat všechny elementy, před které se nebude `<P>` vkládat.

```
\def\x{\ifvmode\def\par{\endgraf\egroup\let\par=\endgraf}}%
```

```
\bgroup\everypar{}\leavevmode\fi}
```

Makro `\x` uzavře následující odstavec do skupiny a lokálně vyprázdní obsah registru `\everypar`. Vyskytne-li se makro v již sestavovaném odstavci, pak neprovede díky podmínce `\ifvmode` nic. Předchozí definice je třeba upravit následovně:

```
\def\documentclass#1{\x<HTML>\par\x<HEAD>\par
    \x<TITLE>LaTeXový dokument<TITLE>\par
    \x</HEAD>\par}
\def\begin#1{\def\co{#1}\ifx\co\document\x<BODY>\par\fi}
\def\end#1{\def\co{#1}\ifx\co\document\par
    \x</BODY>\par\x</HTML>\fi}
```

Nyní je zajištěn překlad odstavců do patřičného tvaru. Lze již přeložit jednoduchý dokument. Většinu ostatních $\text{\LaTeX}$ ových řídicích sekvencí je možné deklarovat podobným způsobem tak, aby provedly konverzi na požadovaný text. Jsou ale sekvence, jejichž konverze není zcela jednoznačná. Např. uzavření textu do prostředí `flushright` vyžaduje, aby každý element `<P>` tohoto odstavce obsahoval parametr `align="right"`. Toho lze docílit následovně. Deklarujeme pomocné makro `\paralign`, které se bude expandovat mezi tokeny registru `\everypar` podle nastaveného zarovnání. Dále deklarujeme další větvení v makru `\begin`. Pro přehlednost předchozí větvení podle parametru `document` není uvedeno.

```
\everypar{<P\paralign>}
\def\paralign{}
\def\flushright{flushright}
\def\begin#1{\def\co{#1}\ifx\co\flushright
    \def\paralign{ align="right"}\fi}
\def\end#1{\def\co{#1}\ifx\co\flushright\def\paralign{}\fi}
```

Makro `\end` zruší nastavení zapsané v `\paralign`. U dokonalé aplikace by bylo vhodnější, aby makro `\begin` uložilo do pomocné sekvence předchozí obsah `\paralign` a makro `\end` jeho hodnotu obnovilo. Podobně lze deklarovat prostředí `center`, které lze mimo uvedený způsob konvertovat i na element `<CENTER>`.

8.1.2. Překlad s využitím mapovacího souboru

Uvedli jsme již, že množina řídicích sekvencí, která se potenciálně může na vstupu vyskytnout při překladu z $\text{\TeX}$ ového zdroje, není konečná, což

znemožňuje dokonalou konverzi. Je proto rozumné dát uživateli možnost, aby sám určil, které sekvence se na vstupu vyskytnou a jakým způsobem mají konvertovat. Tímto způsobem lze také dosáhnout, aby uživatel, pokud není spokojen s vestavěným způsobem převodu, sám určil alternativní konverzi.

Nastíníme možnost takového postupu. Využijeme T_EXový způsob práce se soubory a deklarujeme *mapovací soubor*, kam bude uživatel zapisovat vlastní alternativy konverze. Při tvorbě takového souboru je třeba vždy zvážit syntaxi, jakou bude formát tohoto souboru mít. Uživatelsky *příjemná* syntaxe může mít za následek nárůst složitosti algoritmů pro zpracování mapovacího souboru. Může také značně zpomalit proces konverze. Naproti tomu programátorsky *příjemná* syntaxe může vyvolat nevoli u odborně nezpůsobilého uživatele. V našem případě zvolme druhou variantu a při návrhu syntaxe mapovacího souboru vyjděme z běžné T_EXové syntaxe. Můžeme ostatně předpokládat, že uživateli, který používá T_EX jako nástroj pro formátovou konverzi, nebude tato okolnost v nejmenším překážet.

Záznam v mapovacím souboru tak může být zapsán na jediném řádku a mít tento tvar:

```
\vyraz#1[#2]#3{<VYRAZ prvni=#1>#2</VYRAZ>}
```

Jako první je uvedeno jméno konvertované řídicí sekvence. Následuje maska parametrů v běžné podobě, jak je zapisována ve formátu `plain`. Jelikož L^AT_EX používá také nepovinné parametry, je nutno jejich přítomnost vyznačit hranatými závorkami. Z tohoto jednoduchého přístupu ovšem vyplyne omezení takové, že všechny parametry deklarované v mapovacím souboru musí být povinně přítomny. (Deklarace nepovinných parametrů sama o sobě není triviální záležitostí, viz *diskuse*.) Za deklarací parametrů následuje v složených závorkách řetězec, kterým bude sekvence nahrazena. Do řetězce jsou na patřičná místa předány parametry. Jejich přítomnost není nutná.

Předpokládejme, že mapovací soubor je uložen pod jménem `konverze.map`. Aby byl snadno detekovatelný konec souboru, který T_EX generuje až po vytvoření tokenu `\par` za posledním znakem jiným než mezera, budeme zapisovat na první pozici posledního řádku souboru znak zavináč (`@`).

```
\catcode'\@=12
\def\zavinac{@}
\newif\ifdone
\newif\ifnodefs
\donefalse
\nodefsfalse
```

```

\newread\mapfile
\immediate\openin\mapfile=konverze.map
\def\nactiradek{\immediate\read\mapfile to \work
  \def\test##1##2~^X{\def\prvni{##1}\ifx\prvni\zavinac
    \nodefstrue\fi}
  \expandafter\test\work~^X\relax
  \ifnodefs\donetrue\else\long\expandafter\def\work\fi}
\def\nactisoubor{\let\next=\prectidalsi \next}
\def\prectidalsi{\ifdone\let\next=\relax
  \else\nactiradek\fi\next}

\nactisoubor
\immediate\closein\mapfile

```

Zmíněná syntaxe mapovacího souboru počítá s použitím, kterému podlehe při zpracování T_EXem. Řádek je po načtení rozvinut a je před něj doplněn primitiv `\def`. Rozvinutí ve správný okamžik zařídí primitiv `\expandafter`. Definicí je vytvořena řídicí sekvence s příslušným obsahem. Při dalším výskytu této sekvence, pakliže budou dodrženy stanovené parametry, proběhne požadovaná konverze. Uživatel, obeznámý blíže s činností T_EXových maker, může v mapovacím souboru využít také doplňujících efektů, např. vloží výše deklarované makro `\x` na místo, kam nechce, aby byla vložena značka `<P>`. Zkušený uživatel může po zralé úvaze použít celý dostupný aparát T_EXu, v čemž lze spatřovat výhodu dané volby způsobu zápisu mapovacího souboru.

8.1.3. Makra pro zápis ve výstupní rutině

Nyní uvedeme makra, která se postarají o výstup přeloženého textu do LOG souboru. Bylo již řečeno, že vhodné místo pro zápis je automaticky volaná výstupní rutina. Aby byl zápis v LOG souboru co nejpřehlednější, nastavíme stránkové zrcadlo tak, aby každá stránka obsahovala pouze jeden řádek. Výstup do LOG souboru nebude obsahovat nadbytečný meziodstavcový materiál a snadno poznáme, kde začíná nový řádek. Tak můžeme naformátovat i výstupní soubor do přehledného tvaru. Dále jako vhodné písmo, kterým budou řádky kompletovány, zvolíme písmo neproporcionální. Tak zabráníme vytvoření ligatur a docílíme poněkud stejnoměrnějších délek řádků co do počtu písmen, než při použití písma proporcionálního. Zamezíme dále tvorbě hlavičky a zápatí dokumentu a nastavíme zarovnání textu doprava. Rovněž zrušíme odstavcovou zarážku.

```

\font\text=cstt10 at 10dd \text
\baselineskip=12dd
\vsizel=\baselineskip

```

```

\hsize=30em
\rightskip=Odd plus 1fill
\raggedbottom
\footline={}
\headline={}
\parindentOdd
\parskipOdd

```

Nyní předefinujeme výstupní rutinu tak aby provedla výstup kompletované strany do LOG souboru. Ještě předtím je nezbytné nastavit registr `\showboxdepth`, který udává, do jaké hloubky vnoření boxů se má provádět výpis (pro naše potřeby postačí s rezevou 8), a registr `\showboxbreadth` udávající maximální počet záznamů, které budou vypsané. Uvažujeme-li 30–40 znaků na řádek včetně mezer, vnoření řádku na třetí úroveň boxu a některé další vyskytující se prvky, jako `\discretionary` nebo prázdná odstavcová zarážka (vkládá se i při nulové velikosti), měl by dostačující počet vypsaných prvků být 60. Výpis je zajištěn primitivem `\showbox`.

```

\showboxbreadth60
\showboxdepth8
\output={\showbox255\shipout\vbox{\box255}}

```

Výstupní rutina provádí také zápis do DVI souboru (`\shipout`). Je to ze dvou důvodů. Jednak má uživatel možnost si zkontrolovat v přehledné podobě na obrazovce, jaké řádky byly do LOG souboru zapsány. Druhým důvodem je vnitřní mechanismus `TEXu`, který pokládá za podezřelou každou výstupní rutinu, která do DVI nic nezapíše. Existuje primitivní registr `\maxdeadcycles`, jehož hodnota udává počet po sobě následujících volání výstupní rutiny, během nichž není proveden zápis do DVI, a při jehož překročení `TEX` havaruje. Jelikož lze při větším vstupním souboru předpokládat velký počet jednořádkových stran, je lepší si s tímto registrem „nezahrávat“.

Jako poslední akci je potřeba upravit způsob zápisu do LOG souboru a běhu programu `TEX`. Běžně je při činnosti akce `\showbox` zobrazena na terminálu informace o výpisu boxu do LOG souboru a běh programu se zastaví. V LOG souboru je dále kromě boxu vypsané ještě bezprostřední okolí tokenu, který způsobil přerušení běhu programu. Zápis vypadá následovně:

```

! OK.
<to be read again>
                                \shipout
<output> {\showbox 255\shipout
                                \vbox {\box 255}}

```

Na prvním řádku `TeX` oznamuje, že nenastala chyba, ale byl proveden výpis. Na dalších řádcích je token, který je připraven ve vstupní frontě k dalšímu čtení. Následuje okolí primitivu `\showbox` a číslo řádku, na kterém nastala chyba. Jelikož by čtení takových dat primitivem `\read` mohlo způsobit havárii programu kvůli řádku, na němž není uzavřena skupina, je vhodné omezit výpis příkazem `\errorcontextlines0`. Aby program běžel plynule a nevypisoval spoustu textu na terminál, nastavíme ještě tzv. dávkový režim zpracování vstupního souboru.

```
\errorcontextlines0
\batchmode
```

Vzorový LOG soubor je vzhledem ke svému rozsahu uveden v příloze. Po úvodních informacích následuje první výpis obsahu boxu 255. Je uvozen znakem „>“. Nejprve jsou uvedeny rozměry celého boxu. Pak na dalších řádcích je seznam elementů, které box obsahuje. Úroveň vnoření je vyznačena počtem teček, které informaci předcházejí. Pro nás jsou směrodatné dvě tečky, které označují třetí úroveň vnoření, jíž je obsah řádkového boxu. Obsahuje písmena a jiné znaky uvozené sekvencí `\text` (podle použitého označení zvoleného písma), výplňky uvozené sekvencí `\glue`, může obsahovat ještě sekvence `\hbox`, `\discretionary`, `\penalty` a některé jiné v závislosti na expanzi konverzních maker. Aby mohla být makra, která budou načítat LOG soubor, deklarována co nejjednodušeji, je vhodné, aby konverzní makra expandovala na konečnou (pokud možno malou) množinu prvků.

8.2. DRUHÁ FÁZE KONVERZE

Druhá fáze konverze zajišťuje tyto akce:

- načítá po řádcích LOG soubor;
- analyzuje řádky a vybírá ty, které začínají právě dvěma tečkami;
- obsahuje-li řádek sekvenci `\text`, je z řádku přečten znak výsledného textu a uložen do pracovního *bufferu*;
- obsahuje-li řádek sekvenci `\glue`, je do *bufferu* vložena mezera;
- je-li první znak řádku vykřičník, je obsah *bufferu* vyprázdněn jako samostatný řádek výstupního souboru.

Nyní uveďme makra, která tyto činnosti realizují. Nejprve otevřeme soubor LOG pro čtení a výstupní soubor pro zápis. Předpokládejme, že LOG sou-

bor byl po skončení první fáze překladu zapsán do souboru makelog.log. Výstupní soubor se bude jmenovat output.txt.

```
\newread\infile
\immediate\openin\infile=makelog.log
\newwrite\outfile
\immediate\openout\outfile=output.txt
```

Nyní deklarujeme makro `\validate`, které bude analyzovat vstupní řádek a určí, zda je vhodný pro další zpracování, či nikoliv.

```
\newif\iflinevalid
\linevalidfalse
\def\tecka{.}
\long\def\validate#1#2#3#4^^X{\linevalidfalse
  \def\prvni{#1}\def\druhy{#2}\def\treti{#3}%
  \ifx\prvni\tecka
    \ifx\druhy\tecka\linevalidtrue\fi\fi
    \ifx\treti\tecka\linevalidfalse\fi}
\long\def\zjistiprvni#1#2^^X{\def\cojeprvni{#1}}
```

Je deklarována nová podmínka `\iflinevalid`, udávající, zda má pro další proces řádek význam. Makro nejprve stanoví, že řádek nemá význam. Pak postupně zkoumá první tři znaky řádku. Jsou-li první dva znaky tečka, tj. pracovní definice, obsahující tyto znaky jsou totožné s makrem `\tecka`, nastaví podmínku `\iflinevalid` na `true`. Je-li ovšem také třetí znak tečka, řádek nemá pro další proces význam a podmínka je nastavena na `false`. Definice makra je uvozena prefixem `\long`, jelikož nastane-li na vstupu prázdný řádek, je vygenerován token `\par`, který je součástí načteného řádku. Zároveň je deklarováno makro `\zjistiprvni`, které do sekvence `\cojeprvni` uloží první znak řádku pro zjištění přítomnosti vykřičníku.

Nyní deklarujeme makro, které načte vstupní řádek. Makro bude obsahovat ve svých útrobách sekvenci `\zpracujradek`, která bude volána, pokud má zpracování řádku smysl.

```
\def\exclam{!}
\def\space{ }
\def\znaky{}
\def\nactiradek{\ifeof\infile\message{EOF reached}%
  \else\immediate\read\infile to \work
    \expandafter\validate\work ABCD^^X\relax
    \iflinevalid\expandafter\zpracujradek\work^^X\fi}
```

```

\expandafter\zjistiprvni\work^^X\relax
\ifx\cojeprvni\exclam
\immediate\write\outfile{\znaky}\def\znaky{}\fi
\fi}
\def\zpracujradek..#1^^X{{%
\def\text##1 ^^X{\xdef\znaky{\znaky##1}}%
\def\glue##1^^X{\xdef\znaky{\znaky\space}}%
\def\hbox##1^^X{}%
\def\discretionary^^X{}%
\def\penalty##1^^X{}%
#1^^X\relax}}

```

Nenastal-li na vstupu konec souboru, načte makro `\nactiradek` do sekvence `\work` řádek vstupního textu. Řádek podrobí makru `\validate`. Akce je provedena tak, že za expandovaný řádek je přidána posloupnost ABCD, která zajistí správnou funkci makra `\validate` i v případě, že vstupní řádek bude obsahovat méně než tři znaky. Pokud je řádek vhodný ke zpracování, je provedeno makro `\zpracujradek`. Pokud první znak na řádku je vykřičník, je vypsán obsah sekvence `\znaky`, která slouží pro ukládání načtených znaků výsledného textu, do výstupního souboru. Makro `\zpracujradek` nejprve deklaruje činnosti, které se provedou při výskytu jednotlivých řídicích sekvencí na řádku vstupního souboru. Při výskytu sekvence `\text` je přidán znak do posloupnosti `\znaky`. Při výskytu `\glue` je vložena do `\znaky` mezera (`\space`). Ostatní akce jsou deklarovány jako prázdné. Na závěr makra `\zpracujradek` je akce provedena tak, že je dalšímu procesu postoupen celý obsah načteného řádku, v němž se budou typicky vyskytovat právě deklarované sekvence.

Než bude provedeno samotné zpracování souboru, je třeba ještě ošetřit výskyt sekvence `\bye`, která je přítomná v LOG souboru a při načtení do parametru může způsobit havárii programu.

```

\let\konec=\bye
\let\bye=\relax

```

Nakonec provedeme načtení a zpracování celého souboru. Deklarujeme sekvenci `\next`, která se bude rozvíjet v požadované makro obsahující na svém konci další `\next` v případě, že nenastal konec vstupního souboru, a která bude mít význam `\relax` v případě, že je vstupní soubor u konce. Tím je zaručeno vytvoření cyklu. Posledními činnostmi jsou uzavření vstupního a výstupního souboru.

```

\def\nactisoubor{\let\next=\prectidalsi \next}
\def\prectidalsi{\ifeof\infile\let\next=\relax

```

```

\else\nactiradek\fi\next}

\nactisoubor
\immediate\closein\infile
\immediate\closeout\outfile
\let\bye=\konec
\bye

```

8.3. DISKUSE

Uvedený způsobu dvojfázového překladu se zápisem ve výstupní rutině umožňuje optimální využití zabudovaných možností T_EXového překladu a analýzy zdrojového textu. Lze poměrně přesně deklarovat, které vstupní symboly a sekvence projdou při překladu až do konečného textu a které budou na cestě k výstupu filtrovány a odstraněny, nebo převedeny na jiné symboly a sekvence. Přímý překlad, který by využíval okamžitého zápisu čtených dat do souboru, by takovými možnostmi nedisponoval bez složitého aparátu marker, pakliže vůbec. Skutečnost, že se jedná o překlad dvoufázový, může být před uživatelem skryta vhodným dávkovým zpracováním vstupního souboru. Efektem je tedy nakonec jeden výstupní soubor v cílovém formátu. Pomocí mapovacího souboru lze přizpůsobit konverzi uživatelským potřebám. Syntaxe mapovacího souboru je však diskutabilní. Větší parametrizace konverze má za následek nárůst složitosti rutin pro čtení mapovacího souboru. Je tedy třeba zvážit, zda je účelné vynakládat úsilí vložené do konstrukce takových rutin ve snaze o obecnost konverze za každou cenu. Veliké úsilí totiž může být vynaloženo na místě, kde bude vhodnější upravit zdrojový soubor ve formátu L^AT_EX (např. přidáním jednoho či dvou chybějících volitelných parametrů).

Konverze opačným směrem bude diskutována za následujícím oddílem.

9 / ZPRACOVÁNÍ HTML SOUBORU

Funkce výše uvedených algoritmů byla demonstrována na konverzi z formátu $\text{\LaTeX}$ do formátu HTML. Při takovém procesu odpadá implementace složitějších algoritmů vstupní syntaktické analýzy. Naproti tomu pokud budeme chtít provést konverzi opačným směrem, je nutno provést deklarace, které zaručí vhodné načtení vstupního souboru, který pro zápis řídicích sekvencí využívá relativně odlišný aparát, než jakým disponuje např. $\text{\TeX}$ ový formát `plain`. Syntaktickou analýzu souboru HTML lze však, bude-li jednou provedena, využít i jiným způsobem, než pro konverzi. Lze například implementovat rutiny, které podle obsahu elementů zajistí výstup obsaženého textu a grafických prvků do DVI souboru. $\text{\TeX}$ v takovém případě může zastoupit nástroj pro tisk HTML souborů.

9.1. $\text{\TeX}$ JAKO PŘEKLADAČ HTML

Možnosti $\text{\TeX}$ ového překladu umožňují, aby byla syntaxe vstupního souboru do značné míry přizpůsobena uživatelským potřebám. Může být deklarována zcela nová množina interpretačních výrazů, resp. řídicích sekvencí, které se budou od těch, jež jsou deklarovány ve stávajících formátech, v mnohém lišit.

Demonstrujeme tyto možnosti návrhem maker, které by byly schopny číst soubor ve formátu HTML a zajistit zpracování souboru způsobem, který se bude vizuálně blížit způsobu, jakým HTML zpracovávají dnes běžně používané prohlížeče.

Omezíme se na několik základních příkladů s tvrzením, že kompletní návrh formátu, který by na vstupu četl soubor HTML a výstupem by byl vhodný DVI soubor, by zajistil vhodnou deklaraci všech HTML elementů. Cílem této práce není vytvořit kompletní formát `htmlTeX`, nýbrž naznačit možnosti $\text{\TeX}$ ového překladu.

9.2. ZPRACOVÁNÍ ELEMENTU

Nejprve deklarujeme potřebné znakové kategorie. Značky HTML elementů začínají znakem „<“. Toho využijeme a nastavíme kategorii znaku „<“ na 0, tedy znak uzavírací řídicí sekvenci. Podobně znak & bude uzavírat řídicí

sekvence, kterými se zapisují některé nestandardní znaky. Kategorii znaku „/“ nastavíme na 11, písmeno, jelikož se bude vyskytovat v koncových značkách elementů.

```
\catcode'\<=0
\catcode'\/=11
\catcode'\&=0
```

Nyní lze se stejným efektem psát např. `\parskip` i `<parskip`. Syntaxe HTML nerozlišuje u řídicích sekvencí mezi velkými a malými písmeny. $\text{\TeX}$ naopak velikost písmen důsledně odlišuje. Budeme dále předpokládat, že všechny řídicí sekvence jsou zapsány velkými písmeny. Dodatečně by bylo možno deklarovat řídicí sekvence s malými písmeny nebo s kombinací malých a velkých písmen, kterým by byl přiřazen význam příslušných řídicích sekvencí s velkými písmeny. Situaci lze také řešit preprocesovým filtrem.

Nyní je možno již deklarovat jednoduchý element, který nepoužívá žádné atributy. Takovým je element pro řádkový zlom `<BR>`. Adekvátně by bylo lze deklarovat sekvenci `<br>`.

```
\def<BR>{\ifvmode\leavevmode\fi\hbox{}}\hfil\break}
\let<br>=<BR>
```

Jelikož běžně funguje tento element i tehdy, není-li zpracováván řádek, tj. v terminologii $\text{\TeX}$ u ve vertikálním režimu, je deklarován tak, že pokud je zapsán ve vertikálním režimu, začne nejprve nový odstavec.

9.3. ZPRACOVÁNÍ ATRIBUTŮ

Většina elementů HTML však obsahuje parametry, které významným způsobem ovlivňují jejich zpracování. Stejně jako samotné názvy elementů, i parametry mohou být zapisovány malými i velkými písmeny. Zde však již je možné realizovat konverzi všech písmen na malá z důvodu jednoznačnosti. Makra, která konverzi provádějí, jsou upravena podle maker P. Olšáka, která dal veřejně k dispozici na Internetové konferenci společnosti $\text{\LaTeX}$ TUG(Archiv konference $\text{\LaTeX}$, URI: <http://www.cstug.cz>).

```
\def\defascii#1{\doascii#1\endascii}
\def\doascii#1{\ifx#1\endascii\else
  \expandafter\def\csname:\the\lccode'#1\endcsname{#1}\relax
  \expandafter\doascii\fi}
\defascii{abcdefghijklmnopqrstuvmwxyz}
```

```

\def\expandlc#1{\dolower#1\endlower}
\def\dolower#1{\ifx#1\endlower\else\lowchar#1%
\expandafter\dolower\fi}
\def\lowchar#1{\ifnum\lccode'#1>0
\csname:\the\lccode'#1\endcsname
\else #1\fi}

```

TeX sice disponuje primitivem `\lowercase`, který slouží ke konverzi textu na malá písmena, tento však pracuje až na úrovni hlavního procesoru. Zde však potřebujeme nástroj, který pracuje na úrovni expand procesoru. Tím je makro `\expandlc`. To pomocí cyklu makra `\dolower` postupně snímá jednotlivé znaky parametru `#1`, dokud není sejmut token `\endlower`. Znaky zpracuje makro `\lowchar`. Zjistí pomocí `\lccode`, zda se jedná skutečně o velké písmeno. Pokud ano, stejným `\lccode` vytvoří pomocí `\csname` řídící sekvenci `\:cislo`, kde číslo je příslušná hodnota `\lccode`. Např. „a“ bude nahrazeno sekvencí `\:97`. Pomocí `\defascii` jsou již předtím připraveny všechny sekvence, které mohou takto vzniknout, a jejich obsah je deklarován jako příslušné malé písmeno. Např.: `\def\csname:97\endcsname{a}`. Je tedy zajištěno, že na úrovni expand procesoru bude provedena patřičná konverze. Lze psát např. `\edef\what{\expandlc{ALIGN="RIGHT"}}`. Sekvence `\what` pak bude obsahovat `align="right"`.

Poněkud náročnější budou makra, která zpracují atributy elementu. Proces zpracování atributu provedeme tak, aby pak v rámci deklarace elementu mohla být použita sekvence `\zpracujatributy`, jejíž parametr bude řetězec obsahující všechny přítomné atributy HTML daného elementu. Někde uvnitř procesu se deklarují řídící sekvence, které budou mít též název, jako jméno atributu, a jejichž obsahem bude hodnota atributu. Provede se tedy něco jako `\def\align{right}`. Pak již bude snadné zjistit v těle deklarace elementu hodnotu atributu.

Atributy HTML elementů jsou zapisovány relativně volným způsobem. Mezi jménem atributu a rovnítkem může být více nebo žádná mezera, stejně jako mezi rovnítkem a hodnotou. (`align="right"` je totéž jako `align = "right"`.) Běžně je uváděno, že hodnota atributu má být uzavřena mezi dva znaky „““. Většina prohlížečů však tyto znaky nevyžaduje. A dále existují atributy, u nichž hodnota nemusí být přítomna (pracovně je nazveme jako prázdný atribut); pokud je, je shodná se jménem atributu (např. `compact` je totéž jako `compact="compact"`). V dále uvedených makrech jsou ošetřeny všechny eventuality kromě přítomnosti mezer okolo rovnítko.

Nejprve deklarujeme několik pomocných maker.

```

\def\attrs{}
\def\nomoreattrs{^^X}
\def\upravattrs#1{\gdef\attrs{#1 ^^X}}
\def\separe#1 #2^^X{\gdef\sepattr{#1}\gdef\attrs{#2^^X}}
\def\defattr#1=#2^^Y{\edef\what{\expandlc{#1}}%
  \def\act{\offuvoz#2"ABCD"^^X}%
  \expandafter\def\expandafter\val\expandafter{\act}%
  \expandafter\global\expandafter
    \let\csname\what\endcsname=\val}
\def\ABCD{ABCD}
\def\offuvoz#1"#2"#3^^X{\def\druhy{#2}\ifx\druhy\ABCD#1
  \else#2\fi}

```

Sekvence `\attrs` bude obsahovat všechny přítomné atributy. Z tohoto řetězce bude postupně odebíráno po jednom atributu, který podlehně dalšímu zpracování. Pro zjištění, zda jsou ještě nějaké nezpracované atributy, bude sloužit sekvence `\nomoreattrs`, použitá dále v testu typu `\ifx`. Makro `\upravattrs` přidá za řetězec obsahující atributy mezeru a token `^^X` pro účely výše uvedeného testu. Makro `\separe` se postará o oddělení prvního atributu z řetězce. Pracuje tak, že najde první mezeru v řetězci a vše, co jí předchází, uloží do definice `\sepattr`. Mechanismy $\TeX$ u se přitom postarají, aby nenastal případ, že mezera bude prvním znakem řetězce. Zbytkem za mezerou se přepíše hodnota `\attrs`. Přítomnost mezery i při výskytu jediného atributu je zaručena činností `\upravattrs`. Makro `\defattr` vytvoří definici, která bude požadována při deklaraci elementu. Sejmeme z celé deklarace atributu část před rovnítkem, převede ji na malá písmena a vytvoří z ní řídicí sekvenci, jejíž obsah deklaruje podle části za rovnítkem. Z této části jsou ještě předtím odstraněny případné uvozovky (znaky „“). K tomu slouží poslední dvě deklarace, jmenovitě makro `\offuvoz`. Nemá řešen případ, kdy je přítomen jen jeden z obou znaků. Aby splnilo svůj účel, musí být makro voláno tak, že do jeho parametru je vložen nejprve požadovaný řetězec, z něž mají být uvozovky odstraněny, a dále řetězec "ABCD"^^X. Makro hledá první řetězec v parametru, který je uzavřen do uvozovek. Je-li jím řetězec ABCD, pak testovaný řetězec není uzavřen do uvozovek a je ponechán v původním tvaru. Nemá uvažován případ, kdy se na vstupu vyskytne atribut ve tvaru `atribut="ABCD"`.

Nyní deklarujeme makro `\sejmijeden`, které provede sejmutí a zpracování jednoho atributu z řetězce `\attrs`.

```

\def\sejmijeden{\expandafter\separe\attrs
  \def\testuj##1=##2^^Y{\def\test{##2}}%
  \edef\testovany{\sepattr=%}
  \expandafter\testuj\testovany^^Y%

```

```
\ifx\test\empty\undef\sepattr{\sepattr=present}\fi
\expandafter\defattr\sepattr^^Y}
```

Nejprve je provedeno odebrání prvního atributu makrem `\separe`. Následuje test na přítomnost prázdného atributu. Pokud je test pozitivní, je atribut upraven na tvar `atribut=present`, aby mohl být dále zpracován tímž způsobem, jako ostatní atributy. Zpracování provede výše uvedené makro `\defattr`. Aby byly zpracovány všechny atributy, je třeba deklarovat cyklus, který projde celý řetězec a provede makro `\sejmijeden` postupně pro jeden každý atribut.

```
\def\cyklujatributy{\let\next=\zpracujjeden \next}
\def\zpracujjeden{\sejmijeden\ifx\attrs\nomoreattrs
\let\next=\relax\fi\next}
```

Nejprve je přiřazen sekvenci `\next` význam makra `\zpracujjeden`. To odečte a zpracuje jeden atribut řetězce a provede test na konec řetězce, který nastane, pokud je řetězec tvořen dvěma znaky — mezerou a `^^X` (tytéž znaky, které obsahuje sekvence `\nomoreattrs`). Nastal-li konec řetězce, je sekvenci `\next` přiřazen význam `\relax`. Zbývá jen deklarovat makro, které obsáhne celou činnost, a které bude voláno s jedním parametrem, do nějž vložíme řetězec atributů elementu.

```
\def\zpracujatributy#1{\def\param{#1}\ifx\empty\param
\else\upravattrs{#1}\cyklujatributy\fi}
```

Nezbytnou součástí makra `\zpracujatributy` je test na prázdný parametr, který udává, že uvozující značka elementu byla přítomna bez atributů.

Jsou-li definována makra pro zpracování atributů HTML elementů, lze deklarovat makra pro zpracování elementů samotných. Jako první příklad uveďme deklaraci elementu `<P>`, který obsahuje jeden atribut `align`, nabývající hodnot `right`, `left` a `center`. Budeme prozatím požadovat, aby text odstavce byl ukončen výskytem dalšího elementu `<P>`. Běžnému $\TeX$ ovému způsobu ukončení odstavce prázdným řádkem je třeba zamezit. Prázdný řádek v HTML je chápán jako mezerou. Proto deklarujeme sekvenci `\par`, která je výskytem prázdného řádku generována, jako `\relax` a explicitně budeme ukončovat odstavec zástupnou sekvencí `\endgraf`. Dále deklarujeme podmínku `\ifparopened`, pomocí níž budeme zjišťovat, zda se právě zpracovává odstavec, a podle toho větvit zpracování.

```
\let\par=\relax
\newif\ifparopened
```

\paropenedfalse

Dále deklarujeme pomocné řídicí sekvence. Z důvodů obecnosti je vhodné při nastavování jednotlivých zarovnání odstavce nevolat přímo primitivní registry T_EXu \leftskip a \rightskip, ale vytvořit nové sekvence, které budou volány ze všech použitých maker a které nastavují okraje a způsob zarovnání relativním způsobem. Je totiž potřeba zajistit správnou sazbu odstavce např. při vnořeném výčtu, kdy levý okraj doprava zarovnaného odstavce nesmí překročit hranici odsazení výčtu.

```
\newdimen\skplf \skplf0pt \def\strlf{0pt}
\newdimen\skprt \skprt0pt \def\strrt{0pt}
\def\plusskplf#1{\global\advance\skplf#1
  \edef\setskplf{\leftskip\skplf plus\strlf}\setskplf\relax}
\def\setstrlf#1{\gdef\strlf{#1}
  \edef\setskplf{\leftskip\skplf plus\strlf}\setskplf\relax}
\def\plusskprt#1{\global\advance\skprt#1%
  \edef\setskprt{\rightskip\skprt plus\strrt}\setskprt\relax}
\def\setstrrt#1{\gdef\strrt{#1}%
  \edef\setskprt{\rightskip\skprt plus\strrt}\setskprt\relax}
\def\parleft{\setstrlf{0dd}\setstrrt{1fill}%
  \parfillskip2em plus 1fill}
\def\parright{\setstrlf{1fill}\setstrrt{0dd}\parfillskip0pt}
\def\parcenter{\setstrlf{1fill}\setstrrt{1fill}\parfillskip0pt}
```

Jelikož tato práce sleduje překladové možnosti T_EXu, jsou předchozí makra, která slouží typografickým cílům překladu (vlastní sazby), uvedena pouze pro ilustraci kódu, který by mohl být deklarován při úplné definici T_EXového formátu, zabývajícího se vizuální prezentací HTML souborů. Zmíňme jen, že makra \parleft, \parright a \parcenter nastaví postupně (v našem případě jednoduchým a ne zcela typograficky vhodným způsobem) zarovnání na levý okraj, pravý okraj a na střed. Nyní už samotná deklarace elementu <P>.

```
\def\alleft{left}
\def\alright{right}
\def\alcenter{center}
\def<P#1>{\ifparopened\endgraf\egroup\fi\def\align{}}%
  \zpracujatributy{#1}\relax\paropenedtrue\unskip\bgroup
  \edef\whatalign{\expandafter\expandl\c\expandafter{\align}}%
  \parleft
  \ifx\whatalign\alleft\parleft\fi
  \ifx\whatalign\alcenter\parcenter\fi
```

`\ifx\whatalign\alright\parright\fi}`

Nejprve jsou definovány sekvence pro testovací účely zarovnání odstavce. Element `<P>` je deklarován s jedním parametrem. Znak „>“ slouží s výhodou jako separátor parametru, který může být prázdný. Je proveden test, zda probíhá zpracování odstavce. Pokud ano, makro počítá s tím, že je otevřena skupina. Odstavec kompletuje sekvencí `\endgraf` a skupinu uzavře. Poté proběhne zpracování atributů. Aby vše proběhlo hladce i v případě, že atributy nebudou přítomny, je deklarována sekvence `\align` jako prázdná. Tím je také zrušena předchozí deklarace. Makro dále otevře novou skupinu, uloží do sekvence `\whatalign` na malá písmena převedenou hodnotu zarovnání podle deklarované sekvence `\align`, která je výstupem makra `\zpracujatributy`, nastaví implicitní zarovnání a testuje, jak má být podle atributu odstavec zarovnán. V tomto případě je první test nadbytečný, protože zarovnání vlevo je implicitní. Je uveden *pro forma*.

Podobně jako odstavec lze definovat jiné elementy HTML. Ukážeme ještě deklaraci elementu pro nečíslovaný výčet. Nejprve deklarujeme pomocná makra pro sazbu odrážek, které mohou být trojího typu podle atributu `type`, a to `disc`, `circle` nebo `square`. Atribut `type` se zapisuje do elementu `<UL>`, kde platí pro celý výčet, i do elementu `<LI>`, kde platí pro všechny položky následující (v závislosti na použitém HTML prohlížeči může také platit lokálně pro jednu položku).

```
\font\sym=cmsy10 at 12dd
\newbox\discbox \setbox\discbox=\hbox{\sym\char15\hskip4dd}
\def\disc{\copy\discbox}
\newbox\circlebox \setbox\circlebox=\hbox{\sym\char14\hskip4dd}
\def\circle{\copy\circlebox}
\newbox\squarebox \setbox\squarebox=
    \hbox{\raise0.6dd\hbox{\vrule width4dd height4dd}\hskip5dd}
\def\square{\copy\squarebox}
\def\tpdisc{disc} \def\tpsquare{square} \def\tpcircle{circle}
\def\bullselect{\edef\whattype{\expandafter
    \expandlc\expandafter{\type}}}%
    \ifx\whattype\tpdisc\let\bulltype=\disc\fi
    \ifx\whattype\tpsquare\let\bulltype=\square\fi
    \ifx\whattype\tpcircle\let\bulltype=\circle\fi}
```

Pro jednotlivé odrážky jsou deklarována makra `\disc`, `\square` a `\circle`. Makro `\bullselect` je implementováno pro volbu správného typu odrážky. Přiřadí sekvenci `\bulltype` význam požadované odrážky z výše uvedených

podle hodnoty atributu `\type`. Nyní deklarujeme makra pro zpracování elementů `<UL>` a `<LI>`. Makro `\UL` uzavře kompletovaný odstavec, pokud existuje takový, vynechá prázdné místo, otevře skupinu, v rámci ní zvětší odsazení zleva, zvolí odrážku podle úrovně vnoření výčtu (ta je testována pomocí registru `\ulinner`), a dá možnost přepsání hodnoty odrážky voláním makra `\bullselect`. Makro `\UL` uzavře odstavec i skupinu a sníží hodnotu odsazení zleva. Také oznámí dalším procesům pomocí `\paropenedfalse`, že není kompletován odstavec. Makro `\LI` uzavře skupinu i odstavec a obojí znovu otevře. Dále zjistí deklaraci atributů a podle ní nastaví typ odrážky. Změna odrážky v `\LI` ovlivní i další položky, podobně jako u prohlížečů Netscape a Internet Explorer. Odstavec je zahájen bez vložení meziodstavcové mezery.

```
\newcount\ulinner \ulinner=0
\newdimen\listleftskip \listleftskip=2.5em
\def<UL#1>{\ifparopened\endgraf\egroup\paropenedfalse\fi
\zpracujatributy{#1}%
\ifnum\ulinner=0\vskip\baselineskip\fi
\advance\ulinner1\plusskplf{\listleftskip}%
\ifnum\ulinner=1\let\bulltype=\disc\fi
\ifnum\ulinner=2\let\bulltype=\circle\fi
\ifnum\ulinner=3\let\bulltype=\square\fi
\bullselect}
\def</UL>{\ifparopened\endgraf\egroup\fi
\egroup\plusskplf{-\listleftskip}%
\global\advance\ulinner-1\paropenedfalse}
\def<LI#1>{\ifparopened\endgraf\egroup\fi
\paropenedtrue\bgroup\zpracujatributy{#1}\bullselect
{\parskip0dd\noindent}\llap{\bulltype}}
```

9.4. ZPRACOVÁNÍ ZNAKOVÝCH ENTIT

Demonstrujme nyní vkládání speciálních HTML znaků. Jsou uvozeny znakem „&“, který jsme výše deklarovali jako znak uvozující řídící sekvenci. Jméno speciálního znaku lze uvést za znak „&“ buď přímo, např. `´`, nebo pomocí číselného kódu, který je uvozen znakem „#“. Tento znak je v $\text{\TeX}$ ovém formátu `plain` rezervován pro označení parametrů v makrech. Nezbyvá tedy než změnit jeho kategorii a zvolit jiný znak pro označení parametru. V následující ukázce to bude znak „\$“. Je však vhodné použít znak, který se na vstupu v HTML kódu vyskytnout nemůže. Může to být znak z množiny, která se používá pro podobné účely, kdy chce uživatel zamezit kolizi s náhodně se vyskytujícím znakem, tedy znaky typu `^^X`.

```

\catcode'\&=0
\catcode'\#=11
\catcode'\$=6
\def&nbsp;{\~}
\def&aacute;{á}
\def&#1;{\ifnum$1=193 Å\fi
\ifnum$1=167 \char36\fi}

```

Posloupnost ` `, která vyjadřuje v HTML nezlomitelnou mezeru, deklarujeme jako tzv. T_EXovou *vlnku*, tedy povel pro vložení tak vysoké penalty, že je v daném místě zamezeno řádkovému zlomu. Podobně přímo deklarujeme sekvenci `á`. Následuje deklarace makra `&#`. Znak středník funguje s výhodou v tomto makru jako separátor parametru, který udává číselný kód znaku v kódování podle ISO Latin-1. Jelikož T_EX často používá jiná kódování výstupních znakových sad, nelze se na znaky odvolat přímo přes hodnotu parametru, ale je třeba přiřadit podle použitého kódování každému vstupnímu znaku znak výstupní. Zde, podle kódování ISO Latin-2, se znak 193 přepíše na znak Å (též `Á`), znak 167 na znak s kódem 36, tedy \$ (`§`).

Na závěr demonstrace maker pro zpracování HTML souboru uveďme jednoduchou deklaraci maker, která zastřeší celý soubor.

```

\def<BODY #1>{\rm}
\def</BODY>{\ifparopened\endgraf\egroup\fi}
\def<HTML>{}
\def</HTML>{\konec}

```

Makro `\HTML` je deklarováno jako prázdné. Lze na toto místo umístit deklaraci nastavení, která se postará o sazbu hlavičky, paty, a inicializují použité registry. Makro `\HTML` ukončí zpracování výše deklarovanou sekvencí `\konec`. Podobně makro `\BODY` nastaví základní řez písma a může obsahovat další nastavení jako `\HTML`. Makro `\BODY` ukončí zpracování odstavce a uzavře skupinu. O kompletaci poslední strany se postará již `\konec`.

Celé zpracování HTML souboru může proběhnout ve dvou variantách. Buď izolujeme všechna makra pro zpracování HTML elementů do formátového souboru, který ukončíme sekvencí `\dump`, a necháme iniT_EX vygenerovat nový formát. Pak lze při použití tohoto formátu číst HTML soubor přímo programem T_EX. Druhou možností je konstrukce řídicího souboru, který může mít typicky tento jednoduchý tvar:

```

\input htmltex
<input soubor.html

```

Jsou-li makra pro zpracování elementů uložena v souboru `htmltex.tex`, pak budou na prvním řádku souboru načtena. Jako znak uvozující řídicí sekvenci je na druhém řádku již použit znak „<“, jelikož lze předpokládat, že makra v `htmltex.tex` se postarají o to, aby znak „\“ mohl být načítán jako běžný znak bez zvláštního významu. Výhodou tohoto přístupu je možná parametrizace výstupu. V souboru `htmltex.tex` lze deklarovat makra, která umožní uživateli upravit celý proces zpracování, např. zamezit vkládání HTML obrázků, změnu velikosti výstupu apod. Takovéto deklarace lze vložit mezi uvedené dva řádky řídicího souboru. Lze tak vytvořit více souborů, které zpracují daný HTML zdroj vždy s jiným výsledkem. U prvního přístupu by bylo lze podobného efektu dosáhnout pouze změnou zdrojového kódu HTML souboru nebo formátovou implementací obecného konfiguračního souboru, např. `htmlcfg.tex`, kam by mohly být dohodnuté řídicí sekvence zapisovány uživatelem, a jehož načtení by bylo zajištěno v deklaraci formátu.

9.5. DISKUSE

Je zřejmé, že $\text{\TeX}$ disponuje algoritmy, které mu umožňují přizpůsobit se do značné míry syntaxi vstupního souboru. Lze pak načíst soubor v jiném textovém formátu, než jaký je $\text{\TeX}$ em běžně používán. Princip byl demonstrován na formátu HTML. Podobně by bylo lze načíst prakticky jakýkoliv do textové podoby konvertovatelný SGML dokument, nebo např. soubor ve formátu RTF (Rich Text Format). Běžná je spolupráce $\text{\TeX}$ u s jazykem PostScript, proto lze uvažovat i o vkládání nestandarních a hardwarově závislých grafických prvků, které např. formát HTML podporuje, do souboru generovaného $\text{\TeX}$ em a v rámci postprocesingu přeloženého do jazyka PostScript. Tak by bylo možné vkládat do přeložených souborů na místa, deklarovaná HTML elementy, obrázky, které vznikly vhodnou externí konverzí z GIF a JPEG souborů na soubory ve formátu EPS (Encapsulated PostScript).

Byla diskutována možnost konverze $\text{\LaTeX}$ ového dokumentu na HTML soubor. Za využití uvedených maker pro analýzu HTML elementů a maker procesu dvojfázového překladu lze deklarovat také opačný proces, tedy konverzi z formátu HTML do formátu $\text{\LaTeX}$. Poněkud problematičtější by byla deklarace mapovacího souboru. Také načítání LOG souboru by muselo být provedeno upravenou cestou. Běžně by se totiž na pozici znaku výstupního $\text{\LaTeX}$ ového souboru vyskytovaly speciální znaky, např. znak „\“ s kategorií 0, který uzavírá řídicí sekvenci. Makra pro čtení LOG souboru by byla upravena takovým způsobem, aby tyto znaky byly načteny jako znak s kategorií 11, resp. 12 a byly tak chápány jako běžný povel pro sazbu písmene.

Způsob konverze elementů na řídicí sekvence je otázkou potřeb uživatele a existence vhodných nástrojů formátu $\text{\LaTeX}$. Jelikož lze v $\text{\LaTeX}$ u deklarovat nové řídicí sekvence, skýtá konverze tímto směrem relativně méně úskalí. Elementy HTML mohou konvertovat na speciální množiny jim přizpůsobených elementů, jejichž deklarace bude k $\text{\LaTeX}$ ovému souboru připojena např. jako styl a vymezí dle uživatelských potřeb způsoby sazby dokumentu.

Bylo ukázáno, že z obecného hlediska je konverze mezi textovými formáty pro publikační činnost možná za předpokladu, že interpretační výrazy jednoho formátu lze převést na výrazy druhého formátu buď přímou náhradou adekvátním výrazem, nebo náhradou výrazu soustavou výrazů (struktur), které jako celek vytvoří strukturu shodnou se strukturou původního formátu. Analyzované formáty HTML a $\text{\LaTeX}$ obsahují velké množství stejných nebo podobných strukturních značek. Konverze mezi těmito formáty může proběhnout relativně úspěšně s několika omezeními, které vyplývají z odlišného určení obou formátů. Příkladem takového omezení je konverze interaktivních formulářů HTML nebo matematická sazba $\text{\TeX}$ u.

Silný překladový a jazykový aparát programu $\text{\TeX}$ umožňuje kromě přípravy sazby i jiné využití; to bylo demonstrováno na konstrukci konverzních marker pro převod mezi dokumenty ve formátech HTML a $\text{\LaTeX}$. Jelikož $\text{\TeX}$ disponuje nástroji pro práci s externími soubory, lze jej využít jako konverzního filtru. Na vstupu pak nemusí být pouze formát, který je navržen primárně pro zpracování programem $\text{\TeX}$ (např. $\text{\LaTeX}$), ale díky přizpůsobitelné vstupní syntaxi i formát využívající zcela odlišnou množinu strukturních značek (v našem případě HTML). Z praktických ukázek předvedených výše je zřejmé, že je možné konstruovat úplný překladový aparát na bázi primitivních řídicích sekvencí $\text{\TeX}$ u, který zajistí vhodnou konverzi většiny značek, jež se na vstupní straně $\text{\TeX}$ ového překladu vyskytnou, na strukturní značky výstupního formátu.

10.1. PRAKTICKÁ OMEZENÍ $\text{\TeX}$ OVÉ KONVERZE

Možná omezení, která plynou z konverze pomocí programu $\text{\TeX}$, jsou následující. V první řadě je to nutnost další editace vzhledem k tomu, že konverze neproběhla dokonale. To je však prakticky běžná vlastnost u jakékoliv formátové konverze a kromě nevhodnosti použitého konverzního nástroje je přímo úměrná také nekompatibilitě konvertovaných formátů.

Jazyk $\text{\TeX}$ u také nedisponuje tak širokou škálou prostředků pro práci s proměnnými a soubory, jako běžné programovací jazyky. Proto jsou některé efekty konverze dosažitelné pouze pomocí náročné deklarace složitých marker. Některých efektů je pak třeba se vzdát úplně (např. čtení binárních souborů, tvorba složitých datových struktur a práce s nimi).

Omezení také vyplýne z dvojfázového překladač, který byl demonstrován výše. Výstup do LOG souboru v první fázi tohoto způsobu konverze je totiž náročný časově — vzhledem k velkému množství izolovaných zápisů do LOG souboru — a paměťově — každý znak výstupního souboru je v LOG souboru zapsán jako posloupnost nejméně sedmi znaků, předpokládáme-li jednoznakové označení řídicí sekvence pro použitý font a počítáme-li také jeden znak pro konec řádku. Na strojích s extrémně nízkým výkonem proto tento způsob konverze může způsobit problémy, kterým však vzhledem k rapidně rostoucím výkonům počítačů netřeba přikládat větší význam.

10.2. PRAKTICKÉ VYUŽITÍ

Jazyk SGML je primárně určen k takovým účelům, kde uživatel upřednostní vymezení textových částí v dokumentu obsažených a nezabývá se jejich vzhledem. Vhodná SGML (i HTML) syntaxe je tedy optimální cestou k přípravě textu dokumentu. Dokument tak uchová strukturní informace a jeho vzhled při následné publikaci může být rozdílnou deklarací vzhledu těchto struktur podle aktuálních požadavků měněn.

Konverze SGML (HTML) dokumentu na formát $\text{\LaTeX}$ pak může být cestou pro sazbu takto připraveného textu. Jelikož jde o formát textový, je vhodný pro meziplatformní výměnu dokumentů k finálnímu tisku. Obecně i opačný případ konverze je možný vzhledem k tomu, že je mnohdy potřeba například vystavit dokumenty připravené ve formátu $\text{\LaTeX}$ na síti Internet ve formátu HTML, a formát $\text{\LaTeX}$ obsahuje (resp. předpokládá se, že bude obsahovat) dostačující strukturní informace o povaze částí dokumentu.

Nespornou výhodou způsobu konverze pomocí jazykového aparátu programu $\text{\TeX}$ je, že program samotný existuje v implementacích pro široké množství platform, a díky veřejné dostupnosti jeho zdrojového kódu v jazyce WEB jsou další nestandardní implementace možné. Jelikož je vyžadována naprostá shodnost výsledků činnosti programu na různých platformách, je zaručeno, že konverzní makra pro $\text{\TeX}$, zapsaná jako textový soubor, dají na různých platformách bez dalších úprav tytéž výsledky. Této vlastnosti se konverzní filtr implementovaný pro daný operační systém může jen stěží přiblížit.

Parametrizace konverze pomocí mapovacího souboru skýtá možnost spolupráce konverzního programu s uživatelem, který konverzi provádí, a lze tak lze dosáhnout maximálního efektu úpravou konverzních transformací podle požadovaného cíle. V praxi by bylo vhodné, aby většina běžných, standardních a předpokládaných konverzních transformací byla implementována do

konverzních maker, aby uživatel nebyl nucen psát složitě všechny transformace do mapovacího souboru sám. Lze si též představit uživatelskou výměnu mapovacích souborů pro nejrůznější účely a nejrůznější formáty.

Vzhledem k tomu, že běžný způsob $\TeX$ ového zpracování je dávkový, tj. takový, který provádí činnost na základě zdrojového a konfiguračních souborů a s uživatelem komunikuje minimálně, resp. vůbec, může konverze programem $\TeX$ probíhat jako u běžných filtrů automaticky. Uživatel ocení tuto skutečnost při zpracování dlouhých dokumentů, u nichž by interaktivní způsob konverze jen zvyšoval nároky na strojový čas a pracovní výkon uživatele.

Závěrem, vzhledem k výše uvedenému, *lze způsoby $\TeX$ ové konverze demonstrované v této práci považovat za vhodné a doporučit k praktickému využití.*

10.3. SUMMARY

This thesis deals with the DTP (DeskTop Publishing) data formats and the possibilities of conversion between them. The basic principles of data storing methods are analysed and the text (ASCII) files and the binary files are discussed. The advantages and the disadvantages of the text files are analysed; the advantages are seen in their easy editability, transparency, and capability of being transferred from a platform to another. Main disadvantages of the text files are the need of syntax analysis, possible size growth, and potential escape of information. text data formats are defined as a union of the text data and the interpretation tags, called control sequences, or commands, as well. The text data formats are structured, if they contain tags defining their logical structure, or non-structured, if they contain other tags. Two standard text formats are mentioned, SGML (HTML) and $\LaTeX$. The conversion between mentioned data formats is declared possible, since they both contain similar structures. The principles of $\TeX$ input files reading and processing are analysed from the point of format conversion. $\TeX$'s powerful capabilities of syntax analysis make it convenient for reading the text data formats and converting them to another format. The principles of $\TeX$ conversion are demonstrated by defining $\TeX$ macros for two-phase $\LaTeX$ –HTML conversion, optionally using the map file for further conversion transformations. $\TeX$'s capability of reading non- $\TeX$ format is demonstrated by defining macros for HTML reading. The advantages of $\TeX$ format conversion are its hardware independence, since $\TeX$ is implemented on many different platforms, as well as low requirements of syntax analysis due to $\TeX$'s processing algorithms.

10.4. LITERATURA

1. *HTML 4.0 Specification*. W3C Recommendation, 1998. URI: <http://www.w3.org/TR/1998/REC-html40-19980424>.
2. *Information Processing—Text and Office Systems—Standard Generalized Markup Language (SGML)*. ISO 8879:1986. URI: <http://www.iso.ch/cate/d16387.html>.
3. KNUTH, D. E. *The T_EXbook, volume A of Computers and Typesetting*. Reading, MA, USA: Addison-Wesley, 1986. ISBN 0–201–16447–0.
4. OLŠÁK, P. *Typografický systém T_EX*. Praha: c_sTUG, 1995. ISBN 80–901950–0–8.
5. OLŠÁK, P. *T_EXbook naruby*. Brno: Konvoj, 1998. ISBN 80–85615–64–9.
6. RYBIČKA, J. *L^AT_EX pro začátečníky*. Brno: Konvoj, 1995. 1. vydání. ISBN 80–85615–42–8.
7. RYBIČKA, J. *Úvod do teorie formálních jazyků*. Brno: Konvoj, 1999. ISBN 80–85915–25–8.
8. *Tags for the Identifications of Languages*. RGC 1766, Ed. H. Alvestrand, 1995. URI: <http://ds.internic.net/rfc/rfc1766.txt>.
9. CONNOLLY, D. *A Lexical Analyzer for HTML and Basic SGML*. 1996. URI: <http://www.w3.org/TR/WD-html-lex>.
10. WIRTH, N. *Algoritmy a štruktúry údajov*. Prel. Pavol Fischer. Bratislava: Alfa, 1989. Orig. Algorithms + Data Structures = Programs, ISBN 80–05–00153–3.

11 / PŘÍLOHA. DEMONSTRACE KONVERZE L^AT_EX–HTML

Uvedme ucelenou ukázkou realizace konverze mezi formáty L^AT_EX a HTML aplikací dvojfázového překladu. Mějme následující soubor ve formátu L^AT_EX uložený jako `arno.tex`.

```
\documentclass{article}
\begin{document}
Arno Anzenbacher píše:
\begin{quotation}
V~každodenním užívání jazyka se ukazuje velká {\em relativita \/\}
ve způsobu, jak něco chápeme a označujeme jako věc. To nás přivádí
k~ontologické otázce: {\em Co je vlastní jsoucno, jež je základem
našeho zkušenostního světa?}

Již ontologická kritika předsókratiků ukázala, že v~této otázce
jde o~problém {\em jednoty\/} a {\em rozdílnosti\/}.
\end{quotation}
\end{document}
```

Pro konverzí sekvenční `\em` vytvoříme záznam do mapovacího souboru, ostatní sekvenční budou deklarovány jako přednastavená konverzní makra. Mapovací soubor konverze.map bude vypadat takto:

```
\em{<EM>\gdef\endem{</EM>}\aftergroup\endem}
@
```

Dále mějme makra pro realizaci první fáze uložena v souboru `ltx2log.tex`, který má následující tvar:

```
\baselineskip=12dd \topskip=0dd
\font\text=cstt10 at 10dd \text
\vsizel=\baselineskip
\hsizel=30em \raggedbottom
\rightskip=0dd plus 1fill
\parindent0dd \footline={}
\showboxbreadth400
\showboxdepth8
\output={\showbox255\shipout\vbox{\box255}\advance\pageno1}
\errorcontextlines0 \batchmode
\let\konec=\end
\def\x{\ifvmode\def\par{\endgraf\egroup\let\par=\endgraf}%
\bgroupleverypar{\}\leavevmode\fi}
\def\documentclass#1{\x<HTML>\par\x<HEAD><TITLE>LaTeXový
dokument<TITLE></HEAD>\par}
\def\document{document}
\def\quotation{quotation}
\def\begin#1{\def\co{#1}\ifx\co\document\x<BODY>\par\fi}
```

```

\ifx\co\quotation\par\x<BLOCKQUOTE>\par\fi}
\def\end#1{\def\co{#1}\ifx\co\document\par
\</BODY>\par\x</HTML>\fi
\ifx\co\quotation\par\x</BLOCKQUOTE>\par\fi}
\everypar{<P>}

\catcode'\@=12 \def\zavinac{@}
\newif\ifdone \donefalse
\newif\ifnodefs \nodefsfalse
\newread\mapfile
\immediate\openin\mapfile=konverze.map
\def\nactiradek{\immediate\read\mapfile to \work
\def\test##1##2^^X{\def\prvni{##1}\ifx\prvni\zavinac
\defstrue\fi}
\expandafter\test\work^^X\relax
\ifnodefs\donetrue\else\long\expandafter\def\work\fi}
\def\nactisoubor{\let\next=\prectidalsi \next}
\def\prectidalsi{\ifdone\let\next=\relax
\else\nactiradek\fi\next}
\nactisoubor \immediate\closein\mapfile
\input arno
\konec

```

Provedením předposledního řádku souboru proběhne načtení překládaného L^AT_EXového souboru, jeho konverze a výstup do LOG souboru, který má tento tvar (ve zkrácené podobě):

```

This is emTeX (tex386), Version 3.14159 [4b] (preloaded format=csPLAIN ...
**&csPLAIN ltx2log.tex
(ltx2log.tex The format: csplain <Oct. 1998>.
The cs-fonts are preloaded and A4 size implicitly defined.
\mapfile=\read0
(arno.tex
> \box255=
\ vbox(12.8401+0.0)x337.0491, glue set 0.04733
.\ glue(\topskip) 3.50139 plus 60.0
.\ hbox(6.49861+0.0)x337.0491, glue set 303.34415fill
..\ hbox(0.0+0.0)x0.0
..\ text <
..\ text H
..\ text T
..\ text M
..\ text L
..\ text >
..\ penalty 10000
..\ glue(\parfillskip) 0.0 plus 1.0fil
..\ glue(\rightskip) 0.0 plus 1.0fill

! OK.
<to be read again>
\shipout

```

```
...
1.2 \begin{document}
```

```
[1]
> \box255=
\vbox(12.8401+2.37779)x337.0491, glue set 0.04733
.\glue(\topskip) 2.5694 plus 60.0
.\hbox(7.4306+2.37779)x337.0491, glue set 89.87943fill
..\hbox(0.0+0.0)x0.0
..\text <
..\text H
..\text E
..\text A
..\text D
..\text >
..\text <
..\text T
..\text I
..\text T
..\text L
..\text E
..\text >
..\text L
..\text a
..\text T
..\text e
..\text X
..\text o
..\text v
..\text ý
..\glue 5.6175
..\text d
..\text o
..\text k
..\text u
..\text m
..\text e
..\text n
..\text t
..\text <
..\text T
..\text I
..\text T
..\text L
..\text E
..\text >
..\text <
..\text /
..\text H
```

```

..\text E
..\text A
..\text D
..\text >
..\penalty 10000
..\glue(\parfillskip) 0.0 plus 1.0fil
..\glue(\rightskip) 0.0 plus 1.0fill

! OK.
<to be read again>
\shipout

...
1.3 A
    rno Anzenbacher píše:

[1]
> \box255=
\ vbox(12.8401+0.0)x337.0491, glue set 0.04733
.\ glue(\topskip) 3.50139 plus 60.0
.\ hbox(6.49861+0.0)x337.0491, glue set 303.34415fill
..\ hbox(0.0+0.0)x0.0
..\text <
..\text B
..\text 0
..\text D
..\text Y
..\text >
..\penalty 10000
..\glue(\parfillskip) 0.0 plus 1.0fil
..\glue(\rightskip) 0.0 plus 1.0fill

! OK.
<to be read again>
\shipout

...
1.4 \begin{quotation}

[1]
> \box255=
\ vbox(12.8401+2.37779)x337.0491, glue set 0.04733
.\ glue(\topskip) 3.50139 plus 60.0
.\ hbox(6.49861+2.37779)x337.0491, glue set 196.61179fill
..\ hbox(0.0+0.0)x0.0
..\text <
..\text P
..\text >
..\text A
..\text r
..\text n

```

```

..\text o
..\glue 5.6175
..\text A
..\text n
..\text z
..\text e
..\text n
..\text b
..\text a
..\text c
..\text h
..\text e
..\text r
..\glue 5.6175
..\text p
..\text í
..\text š
..\text e
..\text :
..\penalty 10000
..\glue(\parfillskip) 0.0 plus 1.0fil
..\glue(\rightskip) 0.0 plus 1.0fill

! OK.
<to be read again>
\shipout

...
1.5 V
~každodenním užívání jazyka se ukazuje velká {\em relativita \/>

... další strany v podobném tvaru

[1]
> \box255=
\ vbox(12.8401+0.0)x337.0491, glue set 1.94846fill
.\ glue(\topskip) 2.5694 plus 60.0
.\ hbox(7.4306+0.89165)x337.0491, glue set 297.72665fill
..\ hbox(0.0+0.0)x0.0
..\text <
..\text /
..\text H
..\text T
..\text M
..\text L
..\text >
..\penalty 10000
..\glue(\parfillskip) 0.0 plus 1.0fil
..\glue(\rightskip) 0.0 plus 1.0fill
.\ hbox(0.0+0.0)x337.0491
.\ glue 0.0 plus 1.0fill

```

! OK.
 <to be read again>
 \shipout
 ...
 1.40 \konec

[1])
 Output written on ltx2log.dvi (16 pages, 1548 bytes).

LOG soubor dále zpracujeme makry pro realizaci druhé fáze konverze, která budou uložena do souboru log2html.tex v tomto tvaru:

```
\newread\infile \immediate\openin\infile=log.txt
\newwrite\outfile \immediate\openout\outfile=vystup.htm
\newif\iflinevalid \linevalidfalse
\def\tecka{.} \def\znaky{} \def\exclam{!} \def\space{ }
\let\fine=\bye
\let\bye=\relax
\batchmode
\long\def\validate#1#2#3#4^^X{\linevalidfalse
  \def\prvni{#1}\def\druhy{#2}\def\treti{#3}%
  \ifx\prvni\tecka
    \ifx\druhy\tecka\linevalidtrue\fi\fi
    \ifx\treti\tecka\linevalidfalse\fi}
\long\def\zjistiprvni#1#2^^X{\def\cojeprvni{#1}}
\def\nactiradek{\ifeof\infile\message{EOF reached}%
  \else\immediate\read\infile to \work
    \expandafter\validate\work ABCD^^X\relax
    \iflinevalid\expandafter\zpracujradek\work^^X\fi
    \expandafter\zjistiprvni\work^^X\relax
    \ifx\cojeprvni\exclam
      \immediate\write\outfile{\znaky}\def\znaky{}\fi
    \fi}
\def\zpracujradek..#1^^X{%
  \def\text##1 ^^X{\xdef\znaky{\znaky##1}}%
  \def\glue##1^^X{\xdef\znaky{\znaky\space}}%
  \def\hbox##1^^X{%
    \def\discretionary^^X{%
      \def\penalty##1^^X{%
        #1^^X\relax}}
  \def\nactisoubor{\let\next=\prectidalsi \next}
\def\prectidalsi{\ifeof\infile\let\next=\relax
  \else\nactiradek\fi\next}
\nactisoubor
\immediate\closein\infile
\immediate\closeout\outfile
\fine
```

I v tomto souboru je nastaven režim zpracování na `\batchmode`, aby případné chyby, které nemají na konverzi vliv, nebyly zobrazeny na terminál a nepřerušovaly běh programu. Výsledkem celého procesu konverze bude soubor `vystup.html` v tomto tvaru:

```
<HTML>
<HEAD><TITLE>LaTeXový dokument<TITLE></HEAD>
<BODY>
<P>Arno Anzenbacher píše:
<BLOCKQUOTE>
<P>V každodenním užívání jazyka se ukazuje velká
<EM>relativita </EM> ve způsobu, jak něco chápeme a
označujeme jako věc. To nás přivádí k ontologické otázce:
<EM>Co je vlastní jsoucno, jež je základem našeho
zkušenostního světa?</EM>
<P>Již ontologická kritika předsókratiků ukázala, že v této
otázce jde o problém <EM>jednoty</EM> a
<EM>rozdílnosti</EM>.
</BLOCKQUOTE>
</BODY>
</HTML>
```

Celý proces lze realizovat dávkově zapouzdřením jednotlivých volání programu `TeX` do řídicího souboru, kde může být také provedeno smazání dále nepotřebných pracovních LOG souborů, aby efekt konverze byl úplný.