Mendelova univerzita v Brně
Provozně ekonomická fakulta

Syntaktický analyzátor zdrojových textů ve formátu ConTEXt

Diplomová práce

Bc. Adam Hanuš

Vedoucí práce: RNDr. Tomáš Hála, Ph. D.

Brno 2015

NA MÍSTĚ TOHOTO LISTU
SE NACHÁZÍ ORIGINÁL
ZADÁNÍ PRÁCE

Děkuji RNDr. Tomáši Hálovi, PhD., za vedení, podporu a cenné připomínky při tvorbě této práce. Rád bych také poděkoval Tomáši Vaňátovi, za rady ohledně grafů syntaxe a gramatiky. Poděkování také patří mým nejbližším za skvělé studijní zázemí, které mi poskytují.

Čestné prohlášení

Prohlašuji, že jsem práci *Syntaktický analyzátor zdrojových textů ve formátu ConTEXt* vypracoval samostatně a veškeré použité prameny a informace uvádím v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisů a v souladu s platnou Směrnicí o zveřejňování vysokoškolských závěrečných prací.

Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 20. května 2015

.....
podpis

Abstract

HANUŠ, ADAM. *Syntax analyzer of the ConTEXt source texts.* Diploma thesis. Brno, 2015.

The theme of this thesis is syntax analysis of the source texts based on TEX. The work contains the summary of basic terms from the theory of formal languages, there is also explained the function of the translator in all the analytical phases of the translation and finally it contains the introduction of the Lua scripting language and the ConTEXt typesetting system. The aim of the thesis is implementation of the syntax analyser for the source texts in ConTEXt format. The implementation is accompanied by the description of the solution developement of the lexical analysis, syntax analysis, creating syntax diagrams and context-free grammar.

Key words

syntax analyzer, parser, LL(1), lexical analysis, syntax analysis, recursive descent

Abstrakt

Hanuš, Adam. *Syntaktický analyzátor zdrojových textů ve formátu ConTEXt.* Diplomová práce. Brno, 2015.

Tato diplomová práce se zabývá syntaktickou analýzou zdrojových textů na bázi TEX-u. V dokumentu je sestaven přehled základních pojmů teorie formálních jazyků, dále je zde vysvětlena práce překladače ve všech analytických fázích překladu a nesmí se zapomenout také na představení programovacího jazyka Lua a sázecího systém ConTEXt. Cílem práce je implementace syntaktického analyzátoru pro zdrojové texty formátu ConTEXt. Implementace je doprovázena popisem postupného vývoje řešení lexikální analýzy, syntaktické analýzy, tvorby diagramu syntaxí a bezkontextové gramatiky.

Klíčová slova

syntaktický analyzátor, parser, LL(1), lexikální analýza, syntaktická analýza, rekurzivní sestup

Obsah

1 Úvod a cíl práce

1.1 Úvod

V jednadvacátém století patří počítače neodmyslitelně k našemu všednímu životu. Díky jejich neustálé miniaturizaci a zlevňování pracují všude, často aniž bychom si to uvědomovali. Pro každý počítač nebo obecněji řečeno, pro každé hardwarové zařízení musí být vytvořen i patřičný software, aby mu dal nějaký smysl. Abychom mohli my, jako koncoví uživatelé, software aktivně využívat, je nejprve nutné program naprogramovat. Každý jednotlivý kus software je vytvořen jedním z několika desítek, možná i stovek programovacích jazyků. Společným jmenovatelem je pro většinu z nich fakt, že je nutností jejich zdrojový kód přeložit, neboli transformovat program, napsaný v programovacím jazyce do strojového kódu. Jak již název napovídá, je k tomuto úkonu pořebný překladač. Pojem překladač rozhodně není spjat pouze s programovacími jazyky. Obecně tento nástroj prostě jen provádí transformaci čehokoli na cokoli jiného. Anglicko-český slovník tranformuje jednu řeč na druhou. Morseova abeceda zase mění slova v čárky a tečky. Ten, pro nás nejzajímavější, strojový překladač se používá již od padesátých let minulého století. Angličtina má pro pojem překlad termín „compilation“, proto se počítačovým překladačům říká kompilátory. V době, kdy vznikaly první překladače pro jazyky FORTRAN a ALGOL 60 jistě ještě nikdo netušil, že se bude stejného principu využívat i o 60 let déle. Můžeme říct, že pracujeme-li s počítačem, setkáváme se s překladačem na každém kroku. Už jen při psaní dokumentu dochází k překladu. Program rozpoznává jednotlivá slova, snaží se je podle určitých pravidel uspořádat a také kontroluje správnost jejich zadání. Když je dokument dokončený a chceme jej vytisknout, nastává další překlad. Dokument je převeden do takového formátu, aby si rozuměl s tiskárnou.

V dnešní moderní době, kdy psaní čehokoliv bez použití klávesnice je dávná historie a téměř utopie, je až se podivem, že většina lidí nemá hlubší povědomí o kvalitě vysázeného textu. Možná proto tak málo lidí zná a využívá síly a krásy sázecích programů na bázi TEX-u. Je to možná dané nutností osvojit si alespoň základní pravidla těchto vcelku složitých programů nebo je důvodem absence intuitivních a uživatelsky přívětivých editorů, jak je tomu u komerčních kancelářských balíků. Proč by tedy vlastně měl uživatel chtít používat k sázení elektronických dokumentů TEX-ové nástroje, když mu to na první pohled zabere více času a někdy i nervů? Někteří potřebují ve svých textech složitější matematické vzorce, někdo je uchvácen možností oddělení obsahu od vizuálního uspořádání a také by se nemělo zapomínat na ty, kteří jednodušše požadují kvalitu. V této práci probírané programy TEX a ConTEXt jsou

neinteraktivní systémy, což znamená, že jsou dokumenty zpracovávány dávkově. Je to dáno hlavně historickou dobou vzniku, ale čas ukázal, že tento způsob není vůbec špatný a v mnoha směrech je i výhodnější nežli koncept populárních editorů typu WYSIWYG se složitým grafickým rozhraním. Pravda je, že většina lidí chce dokumenty psát, ne je programovat, ale komu zkrátka záleží na kvalitě provedení, musí pro to něco obětovat. Určitě nebudu v tomto názoru jediný, když budu tvrdit, že komerční editory s grafickým rozhraním jsou vhodné spíš pro jednoduché texty, u nichž záleží na rychlosti vysázení a složité texty s matematikou, definicemi, obrázky, případně celé knihy se vyplatí sázet v TEX-ových programech.

TEX-ové programy mají také svůj programovací jazyk s vlastní syntaxí. Proto, nežli se uživatel může radovat z vysázeného výsledku, je nutností napsaný kód nechat projít fázemi překladu.

V této práci jsou vyčerpávajícím způsobem probrány kromě teoretických základů formálních jazyků také principy práce překladačů od lexikální analýzy, syntaktické analýzy, až po koncovou optimalizaci kódu. Hlouběji se tato práce zabývá syntaktickou analýzou neboli parsováním, a to zejména s použitím LL(1) analýzy. V praktické části práce je implementován kontrolor syntaxe jedné z méně známých odnoží systému TEX (v popularitě se nemůže rovnat například LATEX-u) a sice systému, či jestli chcete balíku maker, známým jako ConTEXt.

1.2 Cíl práce

Cílem této práce je navrhnout a vytvořit synaktický analyzátor zdrojových textů formátu ConTEXt. Jedné se prakticky o před-překladač, který umožní zredukovat většinu syntaktických chyb před samotným překladem. Tudíž ušetří čas, který by byl vynaložen chybnými překladovými běhy. TEX-ový překladač je navíc známý svým časově náročnějším překladem, proto by mohl analyzátor uživateli strávený čas překladem výrazně zredukovat. Prostředkem pro tvorbu analyzátoru je programovací skriptovací jazyk Lua. Před započetím implementace je nutné nejprve navrhnout příslušnou bezkontextovou gramatiku jazyka ConTEXt, k čemuž se dají velice výhodně využit diagramy syntaxe.

Konstrukce analyzátoru se dá rozdělit na dvě části, které na sebe navazují. Nejprve je nutné implementovat lexikální analyzátor, který vstupní text identifikuje a rozdělí na elementy. Ty jsou dále rozpoznávány syntaktickým analyzátorem, který má na starost rozhodnutí o správnosti syntaxe jazyka. Syntaktický analyzátor je vytvořen pomocí LL(1) analýzy. tudíž je nutné zajistit všchny náležitosti jako výpočet množin FIRST a FOLLOW a vytvoření rozkladové tabulky. Program bude spustitelný z příkazové řádky s několika volitelnými parametry a jedním povinným parametrem, kterým je název analyzovaného souboru.

2 Teorie formálních jazyků

Tato část textu má za úkol objasnit a v případě potřeby formalizovat základní pojmy teorie formálních jazyků a automatů. Pochopení základní problematiky je naproto klíčové k navázání na téma strojového překladu. Teorie formálních jazyků je jedna z nejdůležitějších částí informatiky a bývá také proto součástí studijních plánů informatiky na všech vysokých školách, zabývající se touto oblastí. Původ má v lingvistice a filosofii jazyka, ale největší základy tvoří matematika s algebrou. Partie z této problematiky bývají úzce svázány s potřebami praxe při vývoji software i hardware. V roce 1956 se podařilo americkému matematikovi Noamu Chomskému vytvořit jako prvnímu matematický model jazyka a vybudoval tak základ pro formalizaci popisu přirozeného jazyka. Původní představy o tom, že se dokáže zautomatizovat překlad z jednoho přirozeného jazyka do druhého, se ukázaly býti velmi nepřesné a nereálné. Ani dnes po několika desítkách let vývoje nemůžeme tvrdit, že by byly výsledky nějak uspokojivé. Pravdou je, že poznatky z teorie formálních jazyků mají význam pro různá odvětví od dokazování matematických tvrzení po kontrukci kybernetických automatů a jsou obzvláště důležité pro aplikace zabývající se interpretací textu či strojovým čtením.

V návaznosti na teorii formálních jazyků jsou čtenáři představeny informace o strojovém překladu, historii překladačů a fázích překladu.

2.1 Abeceda a řetězec

Naprosto základními pojmy pro pokročilé zkoumání a vymezení jazyka, gramatik a automatů jsou abeceda a řetězec.

Pojmem abeceda se rozumí libovolná, konečná a neprázdná množina, složená ze znaků (písmena, symboly, číslice) abecedy. Příkladem abecedy může být třeba množina číslic od 0 do 9, azbuka, nebo také prázdná množina. Jednotlivý symbol abecedy je část řetězce nebo slova (Češka, 1992).

Řetězec nad abecedou je konečná posloupnost znaků abecedy. Velikost nebo také délku takové posloupnosti značíme $|v|$. Počet výskytů znaku a ve slově v značíme $\#a(v)$. Prázné posloupnosti znaků o nulové délce odpovídá tzv. prázné slovo, označované ε (Češka, 1992).

V literárních pramenech se také můžeme setkat s pojmem množina řetězců:

- Σ^* označuje množinu všech řetězců nad abecedou, včetne prázdného řetězce.
- Σ^+ označuje množinu všech neprázdných řetězců nad abecedou.

Jediný rozdíl mezi těmito dvěma zápisy tkví v platnosti vztahu $\Sigma^* = \Sigma^+ \cup \emptyset$.

Řetězec z je podřetězcem w, pokud existují takové x a y, že platí $w = xzy$. Pokud navíc $x = \emptyset$, říkáme, že slovo z je předponou (nebo také prefixem) slova w. Analogicky můžeme mluvit také o příponě (sufixu). Prázdný řetězec je podřetězcem, prefixem i sufixem každého řetězce. Řetězec nebo podřetězec, který obsahuje právě k výskytů symbolu a se symbolicky značí a^k, například $a^3 = aaa$ (Rybička, 2015).

S řetězci lze provést i následující operace (Rybička, 2015):

- zřetězení — připojení řetězce x na konec řetězce y, operace je asociativní;
- reverze — zrcadlový obraz řetězce x, přičemž symboly jsou zapsané v opačném pořadí;

2.2 Jazyk

Jazyk nad abecedou Σ je libovolná množina slov nad Σ. Jedná se tedy o podmnožinu všech řetězců (i prázdných) nad abecedou Σ. Prázdná množina je jazyk na libovolnou abecedou. Jazyky mohou také být i nekonečné. Můžeme si představit jazyk nad abecedou $[a, b]$, obsahující všechna slova, ve kterých se a a b vyskytují ve stejném počtu (Černá, Křetínský a Kučera, 2002).

2.2.1 Operace nad jazyky

Operace prováděné nad jazyky abecedy Σ lze rozdělit na množinové operace (vycházejí z faktu, že je jazyk množina) a na operace řetězcové (protože prvky jazyka jsou řetězce). Je vždy velice důležité rozlišit operace nad slovy a operace nad jazyky, ikdyž jsou některé z nich stejně značeny (zřetězení, mocnina). Množinové operace rozlišujeme tyto (Rybička, 2015):

- sjednocení $A \cup B = \{x | x \in A \vee x \in B\}$;
- průnik $A \cap B = \{x | x \in A \wedge x \in B\}$;
- rozdíl $A - B = \{x \in A \wedge x \notin B\}$;

- doplněk $\bar{A} = \Sigma^* - A$;
- substituce f je zobrazení abecedy Σ do podmnožin množiny abecedy Δ. Každému $x \in \Sigma$ je přiřazen jazyk $f(a) \subseteq \Delta^*$;

Řetězcové operace:

- konkatenace (součin) $A_1 \cdot A_2 = \{xy | x \in A_1 \wedge y \in A_2\}$;
- iterace $A^k = A \cdot A \cdot ... \cdot A$;

2.2.2 Reprezentace jazyka

Jazyky lze zapsat několika možnými způsoby, přičemž každý způsob se hodí pro jinak složitý jazyk (Rybička, 2015):

- výčet všech řetězců — hodí se pouze pro konečné jazyky;
- matematický zápis množiny — způsob vhodný pro jazyky s jednoduchou strukturou;
- formální gramatika — popis pomocí množiny pravidel, pomocí kterých lze jazyk generovat;
- automat — algoritmus, který rozhoduje, zda je libovolná věta součástí jazyka;

2.2.3 Konečná reprezentace jazyka

Pokud bude jazyk obsahovat nekonečný počet slov, okamžitě vyvstávají důležité otázky. Jak konečným způsobem reprezentovat nekonečný jazyk, tj. specifikovat množinu všech jeho slov. Pokud je jazyk konečný, tak je situace jasná. Vystačíme si s obyčejným výčtem všech jeho slov. V případě nekonečného jazyka je konečná reprezentace velice podstatná. Konečná reprezentace by měla být opět řetězcem symbolů, který budeme vhodně interpretovat tak, aby danému jazyku odpovídala nějaká konkrétní konečná reprezentace. Obráceně pak, aby každé konkrétní reprezentaci odpovídal jediný konkrétní jazyk. Tímto narážím na další důležité pojmy, kterými jsou gramatiky a automaty.

2.3 Gramatika

Jak již bylo nastíněno v předešlé kapitole, gramatika splňuje požadavky kladené na reprezentaci konečných i nekonečných jazyků. Gramatikou je určena syntaxe jazyka, což znamená, že je vymezena určitá struktura vět jazyka. Pokud libovolná věta splní podmínku gramatiky, pak lze tvrdit, že patří do daného jazyka. V definici gramatiky vymezíme dvě konečné disjunktní množiny abecedy.

- množina neterminálních symbolů N — označuje určité syntaktické celky;
- množina terminálních symbolů Σ — je identická s abecedou, nad níž je definován jazyk;

Vztahem $N \cup \Sigma$ definujeme slovník gramatiky. Jádrem gramatiky je množina pře-

pisovacích pravidel, nazývaná také produkce. Každé přepisovací pravidlo zapisujeme jako uspořádanou dvojici (α, β) řetězců. Formálně vyjádřeno, množina přepisovacích pravidel P je podmnožinou kartézského součinu:

$$(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

Nyní si gramatiku definujeme formálním způsobem (Rybička, 2015). Gramatika G jazyka L je čtveřice $G_L = (N, \Sigma, P, S)$, kde

- N je konečná neprázdná množnina neterminálních symbolů.
- Σ je konečná neprázdná množina terminálních symbolů.
- P je konečná množina přepisovacích pravidel.
- S je startovací symbol gramatiky (kořen gramatiky).

Budeme předpokládat, že gramatika má vždy alespoň jeden neterminál (kořen). Mohli bychom samozřejmě uvažovat i gramatiku bez neterminálů, ale ta by potom generovala jen prázdný jazyk. Gramatika může mít rovněž i prázdnou množinu terminálů, ale to by opět vedlo k prázdnému jazyku nebo k jazyku s právě jedním slovem - prázdným. V praxi se proto s takovými typy gramatik moc nesetkáme. Na levé straně gramatiky je řetězec symbolů, z nichž musí být alespoň jeden neterminál. Na pravé straně může být libovolný řetězec nebo jen prázdné slovo.

2.3.1 Zápis gramatik

Velice důležité je sjednotit konvence značení všech prvků gramatiky. V praxi je víceméně jedno, jak si symboly označíme, důležité je vybrané značení bezmezně dodržovat. Nejčastější značení prvků gramatiky je následující (Dusíková, 2011):

- Terminální symboly jsou značeny malým tiskacím písmem (a, b, c).
- Pro terminální symboly jsou vymezena velká tiskací písmena (A, B, C).
- Pro řetězce složené z terminálů i neterminálů to jsou symboly řecké abecedy (α, β, γ).
- Nakonec řetězce složené pouze z terminálních symbolů se značí malou kurzívou (*a,b,c).*

Přepisovací pravidla budou zapisována ve tvaru $\alpha \rightarrow \beta$. Pro správnou představu značení jednotlivých symbolů je zde uveden jeden příklad gramatiky. Gramatika se třemi přepisovacími pravidly může vypadat takto:

$$G = (\{A, B, C\}, \{a, b, ..., z\}, P, A)$$
$$A \rightarrow B|AC$$
$$B \rightarrow a$$
$$C \rightarrow a|b$$

2.3.2 Klasifikace gramatik

Gramatiky lze rozdělit dle omezení tvaru přepisovacích pravidel. Při porovnání dvou gramatik nezáleží na druhu , tvaru či počtu přepisovacích pravidel, rozhodující jsou odvozovací možnosti. Již v roce 1956 se touto věcí zabýval lingvista Noam Chomsky a podle něho nyní rozlišujeme čtyři třídy gramatik a jazyků jimi generovaných. Chomského hierarchie hierarchicky postupuje od nikterak omezených přepisovacích pravidel a postupně přidává omezující požadavky (Rybička, 1999).

- Gramatika typu 0 — na tvar pravidel nejsou kladeny žádné omezující podmínky. Někdy se také tyto gramatiky označují jako frázové. Tento typ gramatiky akceptuje Turingův stroj.
- Gramatika typu 1 (kontextová gramatika) — pro každé její pravidlo platí, že $|\alpha| \leqslant |\beta|$ s eventuální výjimkou pravidla $S \to \varepsilon$, pokud se S nevyskytuje na pravé straně žádného pravidla.
- Gramatika typu 2 (bezkontextová gramatika) — každé její pravidlo je ve tvaru $A \to \alpha$, kde $|\alpha| \geqslant 1$ s eventuální výjimkou pravidla $S \to \varepsilon$, pokud se S nevyskytuje na pravé straně žádného pravidla.
- Gramatika typu 3 (regulární gramatika) — každé její pravidlo je tvaru $A \to aB|Aa$ s eventuální výjimkou pravidla $S \to \varepsilon$.

Toto rozdělení také určuje příslušné pořadí jazyků. Například jazyk L je regulární, pokud existuje regulární gramatika G taková, že $L(G) = L$. Velice důležité je povšimnout si smyslu výjimky u pravidla $S \to \varepsilon$. Kdyby neexistovala, z libovolného jazyka by se stal po přidání prázdného slova jazyk typu 0, který nelze popsat ani kontextovou gramatikou (Černá, Křetínský a Kučera, 2002).

V otázce vzhahu mezi jenotlivými třídami jazyků platí, že každá další třída je obsažená v předchozí.

$$L_0 \subset L_1 \subset L_2 \subset L_3$$

Lehce si rozebereme regulární gramatiky a podrobněji se budeme zabývat gramatikami bezkontextovými, protože mají v dnešní době nejvíce praktického využití, z velké části díky syntaktické analýze, což je hlavní náplní této práce.

2.3.3 Gramatika typu 3 (regulární)

Nyní se blíže podívejme na gramatiku typu 3, tedy gramatiku s nejvíce omezenými přepisovacími pravidly. Gramatika se též označuje jako lineární a v zásvislosti na které straně přepisovacího pravidla se nachází jediný neterminální symbol hovoříme o pravé nebo o levé lineární gramatice.

Gramatika typu 3 lze vyjádřit několika navzájem ekvivalentními způsoby. Jedná se o lineární gramatiku, regulární gramatiku, regulární výraz a konečný automat. Všechny tyto způsoby jsou na sebe algoritmicky převoditelné, jsou tedy ekvivalentní. Z definice ekvivalence dvou gramatik platí, že gramatiky G_1 a G_2 jsou navzájem ekvivalentní, jestliže generují jeden stejný jazyk (Černá, Křetínský a Kučera, 2002).

2.3.3.1 Lineární gramatika

Jak již bylo avizováno, jedná se o gramatiku svou výrazovou silou rovnou regulární gramatice s jediným rozdílem, a sice, že může mít ve svých pravidlech více terminálů, nežli jen jeden. Lineární gramatika je snadno převoditelná na regulární gramatiku pomocí algoritmu regularizace (Dusíková, 2011). Takto vypadá pravidlo pravé lineární gramatiky:

$$A \rightarrow \omega B|\omega$$

Takto vypadá pravidlo levé lineární gramatiky:

$$A \rightarrow B\omega|\omega$$

Analogicky se zapisují i pravidla regulárních gramatik.

2.3.4 Gramatika typu 2 (bezkontextová)

Bezkontextová gramatika nabízí silnější popisné prostředky nežli regulární gramatika. Její použití je proto nejvýhodnější pro popis současných programovacích jazyků. Model, který umí tuto gramatiku zpracovat, je zásobníkový automat. Tyto gramatiky nás díky praktické části této práce zajímají nejvíce, protože mají praktický význam pro oblast definice syntaxe programovacích jazyků, formalizaci pojmu syntaktická analýza, atp. Na formální úrovni jsme gramatiku již definovali jako $G = (N, \Sigma, P, S)$. Nutně ještě musíme doplnit, že přepisovací pravidla jsou zapsána ve tvaru $A \rightarrow \alpha, A \in N$ a $\alpha \in (N \cup \Sigma)^*$ (Habiballa, 2003).

Příklad jednoduchých pravidel bezkontextové gramatiky G může vypadat třeba takto:

$$S \rightarrow aSb|ab$$

Takový zápis by se dal ekvaivalentně na dva řádky rozepsat i takto:

$$S \to aSb$$
$$S \to ab$$

2.3.4.1 Vlastnosti a druhy bezkontextových gramatik

Často může nastat situace, že gramatika obsahuje zbytečné symboly, zbytečná pravidla nebo je zadána nejednoznačně. Právě pro takové případy existuje v souvislosti s bezkontextovými gramatikami mnoho algoritmů, které dokáží gramatiku nějak modifikovat, a tím splnit některé důležité vlastnosti dané jejich generovaným jazykem. Tyto speciální tvary gramatik pak nemusejí porušovat třídu jazyků a mohou mít řadu výhodných vlastností. Gramatiky je možné zpracovávat například ve smyslu odstranění nedostupných či zbytečných symbolů, odstranění prázdných pravidel nebo odstranění cyklů. Různě upravené či zredukované gramatiky rozeznáváme tyto (Rybička, 2015):

Jednoznačná gramatika: Některé gramatiky se dají těžko derivačně rozebrat a to tak může vést k obtížím v některých praktických aplikacích. Řeč je o nejednoznačných či víceznačných gramatikách, u kterých syntaktický analyzátor neví, které z přepisovacích pravidel má použít, čímž roste časová i prostorová složitost spojená s hledáním všech možných derivačních stromů. Jednoznačnost gramatiky je, jak si ukážeme dále, nutnou podmínkou pro syntaktickou analýzu.

Vlastní gramatika: Gramatice bez zbytečných symbolů, e-pravidel a bez cyklů říkáme vlastní gramatika. Platí, že každá bezkontextová gramatika má ekvivalentní vlastní gramatiku, tzn. každou bezkontextovou gramatiku můžeme po úpravě převést na vlastní.

Nevypouštějící gramatika: Bezkontextová gramatika se nazývá nevypouštějící, jestliže neobsahuje žádné prázdné pravidlo. I když se nedá obecně říct, že by byl výskyt těchto pravidel nežádoucí, občas mohou způsobovat komplikace při převodech.

Redukovaná gramatika: Tento typ gramatiky nebsahuje dva konkrétní typy symbolů. Jsou jimi symboly nenormované (není možné je přepsat na řetězec terminálů) a nedosažitelné (nedá se k nim dostat žadným odvozením z kořene gramatiky). Jinými slovy lze to znamená, že gramatika neobsahuje zbytečné a nedostupné symboly.

Levorekurzivní gramatika: Gramatika s levorekurzivním neterminálem. Obdobně můžeme definovat i pravorekurzivní gramatiku.

2.3.5 Normální formy gramatik

Bezkontextová gramatika nemůže generovat prázdný řetězec, ale může být transformována na takovou, která bude obsahovat pravidlo, jež to umožní. Každá bezkontextová gramatika bez prázdného symbolu se dá ekvivalentně k všeobecné formě zapsat ve dvou normálních formách (Rybička, 2015).

2.3.5.1 Chomského normální forma

Derivační strom popsaný touto formou gramatiky je vždy jednoznačně binární. Platí, že všechna přepisovací pravidla smějí být pouze ve tvaru (Rybička, 2015):

- $A \to BC, A, B, C \in N$;
- $A \to a, a \in \Sigma$;
- $S \to \varepsilon$ pokud $\varepsilon \in L(G)$ a S nesmí být na pravé straně žádného přepisovacího pravidla;

2.3.5.2 Greibachové normální forma

Tento tvar je vhodný zejména pro analýzu a generování jazyků. Každé pravidlo začíná terminálem, proto zde lze snadno odhadnout počet derivací podle délky věty. Platí tady tato pravidla (Rybička, 2015):

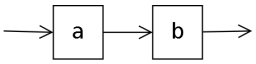
- Nejsou žádná ε pravidla.
- Přepisovací pravidla jsou ve tvaru $A \to a\alpha$, případně
- $S \to \varepsilon$, pokud $\varepsilon \in L(G)$ a S nesmí být na pravé straně žádného přepisovacího pravidla.

2.3.6 Grafy syntaxe

Pokud z nějakého důvodu potřebujeme graficky vyjádřit pravidla bezkontextové gramatiky, použijeme k tomu příslušný graf syntaxe. Tyto grafy se také často používají k popisu programovacích jazyků, protože se jedná velice přehlednou formu.

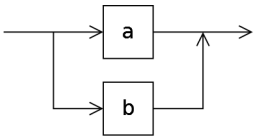
Uzly grafu představují obdélníky (neterminály - tj. vlastně odkaz na jiný graf) a ovály (terminály). Spojnice představují možnou posloupnost symbolů ve větné formě. Graf lze rozdělit na elementy, vyjadřující jednotlivé případy pravidel bezkontextové gramatiky (Rybička, 1999, s. 23-25):

– zřetězení symbolů — pravidlo gramatiky: $A \rightarrow a\ b$;



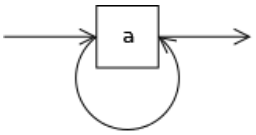
Obrázek 2.1 Zřetězení symbolů

– varianta — pravidlo gramatiky: $A \rightarrow a|b$;



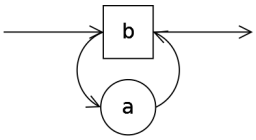
Obrázek 2.2 Varianta symbolů

– iterace — pravidlo gramatiky: $A \rightarrow aB$, $B \rightarrow aB|\varepsilon$. Klíčová je skutečnost, že prvek a se musí vyskytovat jedenkrát a musí být rekurzivní.



Obrázek 2.3 Iterace symbolu

– seznam oddělovačů — pravidlo gramatiky: $A \rightarrow bB$, $B \rightarrow abB|\varepsilon$. Prvek a v tomto případě hraje roli oddělovače.



Obrázek 2.4 Oddělovač symbolu

2.4 Automaty

Automaty jsou virtuální výpočetní stroje sloužící k rozpoznávání výše definovaných jazyků. Zatímco gramatiku můžeme chápat jako generativní systém, který postupně vytváří výslednou větu, automat je systém akceptační. Na vstupu je libovolná věta a výstupem je informace, jestli tato věta patří nebo nepatří do daného jazyka. Rozeznáváme hned několik variant automatů, které ale vždy patří do dvou hlavních nadmnožin, tedy automatů konečných a zásobníkových.

2.4.1 Konečný automat

Konečný automat je systém, který může nabývat konečně mnoho stavů. Přičemž daný stav se mění na jiný stav na základě vnějšího podnětu. Zapisovat lze pomocí stavového diagramu, grafu nebo tabulkou. Konečný automat je vybaven konečnou stavovou řídicí jednotkou, čtecí hlavou a páskou, na které je zapsáno vstupní slovo. Na startu je čtecí hlava nastavena na nejlevější stranu pásky. Automat dle momentálního stavu a přečteného symbolu mění svůj stav a posunuje se o jedno políčko na pásce doprava. Konec výpočtu je definován zablokováním automatu (výpočet není konečný) nebo přečtením celého vstupního slova. Pokud automat přečte celé slovo a dostane se do nějakého z akceptujících koncových stavů, říkáme, že automat slovo akceptuje. V opačném případě slovo zamítá. Množina všech akceptujících slov vytváří jazyk akceptovaný automatem M.

Nedeterministický konečný automat (Jančar, 2003) je definován jako pětice

$$M = (Q, \Sigma, \delta, q_0, F),$$

kde Q je konečná neprázdná množina stavů. Σ je vstupní abeceda. $\delta : Q \times \Sigma \to Q$ vyjadřuje přechodová funkce. q_0 znamená počáteční (iniciální) stav a $F \subseteq Q$ je množina přijímaných (koncových) stavů.

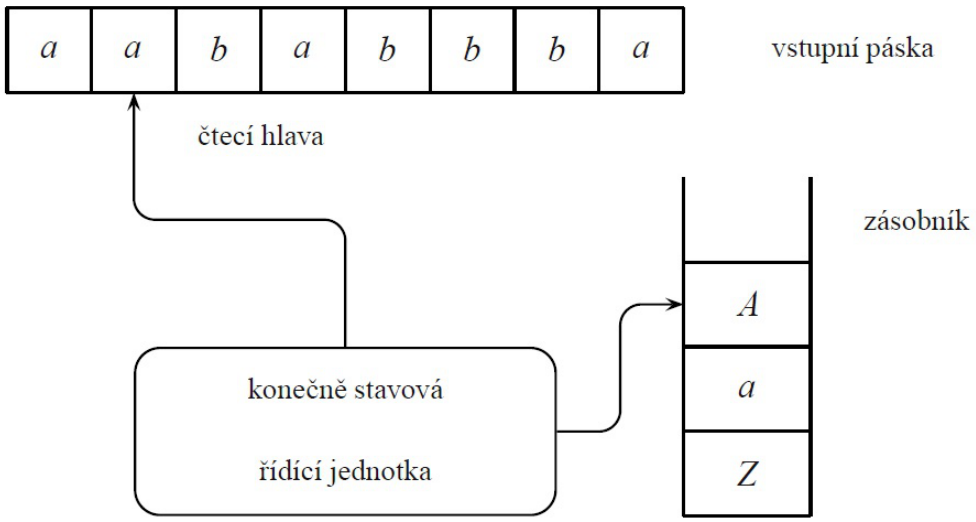
Pokud je pro automat vždy jednoznačně určeno, z kterého do kterého stavu přejít, mluvíme o deterministickém automatu. Pokud je tomu naopak a automat může v jednu chvíli přejít do více stavů, pak se jedná o automat nedeterministický.

Existují různé varianty konečných automatů. Kromě deterministického a nedeterministického se v odborné literatuře, (například) Jančar (2003), můžeme setkat s automaty s výstupem. Některý z nich, například Mealyho automat může v každém svém kroku produkovat výstup. Moorův automat zase výstup vyprodukuje na základě stavu ve kterém se právě nachází. Nutno podotknout, že i přes nemalé rozdíly mezi variantami konečných automatů, jsou na sebe všechny automaty převoditelné. Z toho plyne velice důležitý fakt, že každý konečný nedeterministický automat je

převoditelný na ekvivalentní deterministický automat.

2.4.2 Zásobníkový automat

Zásobníkové automaty rozpoznávají bezkontextové jazyky a lze si je představit jako nedeterministické konečné automaty s tím, že navíc obsahují pomocnou paměť, pracující jako zásobník (LIFO). Velikost tohoto zásobníku je potenciálně nekonečná v tomto smyslu: v každém okamžiku výpočtu je sice konečná (uloženo je konečné množství symbolů), ale kdykoliv ji lze rozšířit o další počet paměťových míst tím, že přidáme další symboly na vrchol zásobníku. Funkcionalitu zásobníkového automatu popisuje následující obrázek.



Obrázek 2.5 Zásobníkový automat

Ze vstupní pásky může automat pouze číst, přičemž čtecí hlava se smí pohybovat jen vpravo. Automat může na vrchol zásobníku ukládádat symboly a tyto symboly může následně číst, ovšem pouze ty z vrcholu zásobníku. Přečtený symbol se z vrcholu odstraní. Této akci se v odborné literatuře říká destruktivní čtení (Černá, Křetínský a Kučera, 2002).

Vyšší výpočetní síla zásobníkového automatu pramení z přidání právě jednoho zásobníku ke konečnému automatu. Zopakovaním tohoto rozšíření bychom dostali

automat se dvěma zásobníky, tedy stroj, který je svoji výpočetní silou rovný Turingově stroji. Rozdíl oproti konečnému automatu tkví také v možnostech rozpoznání slova. Zatímco konečný automat slovo rozpozná pouze tím, že skončí v koncovém stavu, zásobníkový automat toho může dosáhnout i tak, že vyprázdní celý svůj zásobník. Formální definice zásobníkového automatu vypadá takto (Černá, Křetínský a Kučera, 2002, s. 77):

Nedeterministický zásobníkový automat (PDA - z angl. push-down automata) je sedmice

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F), kde$$

- Q je množina stavů;
- Σ je vstupní abeceda;
- Γ je zásobníková abeceda
- δ je přechodová funkce pro kterou platí $\delta : Q \times (\Sigma \cup \{\varepsilon\} \times \Gamma)$;
- q_0 je počáteční stav;
- Z_0 je počáteční symbol zásobníku;
- $F \subseteq Q$ je množina koncových stavů;

2.4.3 Konfigurace a krok výpočtu

Činnost automatu je vyjádřena posloupností konfigurací. Konfigurace (q_i, ω, α), jakožto intuitivní představa o celkové situaci automatu, je dána vnitřním stavem, dosud nepřečtenou částí vstupní věty a obsahem zásobníku. Budeme-li mít automat M, který chce přijmout slovo w, pak se automat nachází v počáteční konfiguraci. Dále následuje několik kroků výpočtu, nežli se automat dostane do své koncové konfigurace, která může být, jak již bylo řečeno, akceptující nebo zamítající. Jednotlivý krok výpočtu se definuje jako binární relace mezi konfiguracemi automatu. Přechod mezi jednotlivými konfiguracemi se značí symbolem šipky se zarážkou (Jančar, 2003).

Přechodová funkce automatu lze vyjádřit několikerým způsobem. Dle Rybičky (1999) lze přechodovou funkci přehledně napsat do tabulky. Další možností, jak přechodovou funkci zobrazit, je diagram přechodů automatu (Kollmanová, 2013). Uzly v diagramu představují vnitřní stavy automatu a hrany mezi nimi se rovnají terminálním symbolům čteným z pásky. Koncové stavy se pak vyznačují dvojitým zakroužkováním.

3 Strojový překlad

V této kapitole bude představen překladač, jakožto nástroj pro konverzi algoritmů zapsaných v programovacím jazyce do strojového kódu. Samozřejmě, že překladač nemusí převádět jen kód v programovacím jazyce. Existují překladače, které převádí text mezi různými formáty nebo překladače zdrojových textů v programech pro sazbu textu, jako je například TEX. Je zde také vysvětlen princip práce překladače. Čtenář je také seznámen s historií překladačů. Nejprve je na místě přesně definovat pojem překladač.

Překladač je program, který k libovolnému programu P_z(zdrojový program) v jazyku J_z(zdrojový jazyk) vytvoří program P_c(cílový program) v jazyku J_c(cílový jazyk) se stejným významem. Překladač tedy realizuje zobrazení jazyka J_z do jazyka J_c (Vavrečková, 2008). Překladače lze rozdělit na několik druhů z hlediska cílového kódu či z hlediska komunikace s uživatelem.

3.1 Historie a vývoj strojového překladu

První kompilátor sestrojila paní Grace Murray Hopperová již v roce 1952, zatímco první interpret byl implementován o 6 let později v roce 1958 Stevem Russellem. Jednalo se o překladače pro jazyky FORTRAN a COBOL. Tento fakt je zarážející, protože je známá skutečnost, že naprogramovat kompilátor je o hodně složitější než vytvořit interpret. Pro vysvětlení je nutné se dívat o několik desítek let zpět a představit si tehdejší podmínky. Interpretace programu potřebuje bezesporu více paměti nežli spuštění kompilovaného programu. V době, kdy byl luxus spravovat operační paměť 1kB, bylo šetření paměti klíčové. Procesory v šedesátých letech také nebyly tak složité jako ty dnešní. Tehdejší množství procesorových instrukcí bylo výrazně menší, tudíž bylo výrazně jednodušší i kompilování. Totéž víceméně platí i o moderních programovacích jazycích. Starší jazyky měly proto o hodně jednodušší kompilátory (Arstechnica, 2014).

Vývoj velmi ovlivnila i studie přirozených jazyků matematika Noama Chomského, které dala základ ke klasifikaci jazyků podle gramatik, které je postihují. Zlom v celé podstatě překladu přišel v šedesátých letech s vývojem jazyka Algol 60. Ten zavedl tzv. problémově orientovaný přístup, tj. princip „nejdříve vyřeš to důležité“. Tento princip je hlavní vlastností i dnešních vyšších programovacích jazyků. Další vyvíjené jazyky jako Pascal nebo Ada se snažily být nezávislé na konkrétní architektuře. V dnešní době je toto oddělení od architektury počítače ještě umocněno během ko-

nečných aplikací na virtuálních počítačích (typicky například Java nebo .NET).

3.2 Kompilátor versus Interpret

Všichni znalí informačních technologií mají povědomí o rozdílu mezi kompilátorem a interpretem, ale na první pohled tyto rozdíly nemusejí být tak úplně zřejmé. Velice vtipným a elegantním způsobem o rozdílech píše na svých stránkách Jiří Peterka (2011). *Pro názorné vymezení rozdílu mezi interpretací a kompilací by bylo možné použít analogii s jízdou taxíkem. Když do něj nastoupíte, můžete řidiče požádat, aby vás nejprve odvezl například na Václavské náměstí, pak na Staroměstské náměstí, pak na Vyšehrad apod. Nebo mu můžete postupně říkat, že má jet rovně, pak že má zahnout doleva, pak zase doprava atd. První případ odpovídá svojí logikou interpretaci programu ve vyšším programovacím jazyku, zatímco druhý případ naopak odpovídá kompilaci - v prvním případě specifikujete pouze čeho chcete dosáhnout, a ponecháváte na někom jiném (na interpretu), aby vámi požadované akce realizoval takovým způsobem, jaký považuje za nejvhodnější. Druhý případ pak odpovídá tomu, že si svůj záměr (projet se kolem příslušných pamětihodností) někde předem rozpracujete (přeložíte) do konkrétního itineráře, a pak postupně navigujete vozidlo taxislužby konkrétními ulicemi, uličkami atd.*

Každý druh překladače je vhodný pro svou specifickou situaci. Následující tabulka shrnuje výhody a nevýhody obou druhů v podstatných činnostech jejich práce (Vavrečková, 2008).

Tabulka 3.1 Srovnání vlastností kompilačního a interpretačního překladače

Vlastnost	Kompilátor	Interpret
Rychlost běhu cílového programu	lepší	
Rychlost spuštění cílového programu	lepší	
Rychlost překladu		lepší
Spotřeba paměti - operační	lepší	
Spotřeba paměti - cílový soubor na paměťovém médiu		lepší
Přenositelnosti kódu mezi platformami (Win, Linux,...)		lepší
Možnost optimalizaxe	lepší	
Nezávislost na překladači	lepší	

V praxi vznikají překladače, které se vzájemně překrývají a využívají výhod obou

jmenovaných.

3.3 Interpret

Interpret nevytváří generovaný program, vytváří jen vnitřní reprezentaci programu pro svou vlastní potřebu (tu můžeme chápat jako cílový jazyk). Celý proces spočívá v tom, že zdrojový text se nepřekládá do žádného jiného (a už vůbec ne spustitelného) tvaru jako je tomu u kompilátoru. Místo toho spoléhá na existenci jiného programu–interpretu. Když je pak potřeba program ve vyšším programovacím jazyku spustit, je ve skutečnosti spuštěn onen interpret, který na vstup dostane zdrojový text dotyčného programu. Interpret poté postupně načítá a vyhodnoducuje příkazy, na které reaguje příslušnými akcemi.

K výhodám interpretu řadíme (Vavrečková, 2008):

- rychlost — hlavně u velkých projektů je rychlejší cyklus editace-spuštění-debugging;
- laditelnost — snadné odhalování chyb, nevyžaduje se znalost strojového jazyka (assembleru);
- kompatibilita a přenositelnost — stejný program je možné spustit na různých platformách, nejsou rozdíly mezi operačními systémy;
- správa paměti — při běhu programu nemusí být proměnné vázány na fyzickou adresu v operační paměti;
- jednoduchost — obecně je snazší vytvořit interpret jazyka nežli jeho kompilátor;

3.4 Kompilátor

Kompilátor má na vstupu program ve vyšším programovacím jazyce (například C,C++) a výstup je strojový jazyk nebo jazyk symbolických instrukcí (Assembler). Většina kompilátorů překládá zdrojový kód psaný ve vyšším programovacím jazyce do strojového kódu, který lze následně přímo spustit počítačem či virtuálním strojem. Nicméně je také možná kompilace z nižšího jazyka do vyššího, pak takovému úkonu říkáme dekompilace. Kompilátor také může pracovat tak, že překládá z jednoho vyššího jazyka do druhého, potom se bavíme o cross-kompilátoru (Compilers.net, 2005).

Několik spíše experimentálních kompilátorů bylo vytvořeno již v padesátých letech, ale historické prvenství je připisováno týmu FORTRAN vedeného Johnem Backusem. První kompilátor předvedli v roce 1957. Následné roky došlo k velkému roz-

machu a směle lze označit šedesátá léta za zlatá z pohledu vývoje kompilačních principů. Sám kompilátor je program napsaný v některém z programovacích jazyků. První kompilátory byly psané v aesembleru. Prvním jazykem, pro který byl napsaný složitější kompilátor, byl jazyk Lisp a stalo se tak v roce 1962 (Compilers.net, 2005).

Zajímavý problém vyvstane, pokud chceme zkompilovat jazyk, jehož kompilátor je napsaný v témže jazyce. První takový kompilátor musí být totiž zkompilován kompilátorem vytvořeném v jiném jazyce nebo zkompilován spuštěním kompilátoru v interpretu (přesně tak to provedli pánové Hart a Levis při vývoji jejich kompilátoru jazyka Lisp v roce 1962.)

Kód, který dostaneme po spuštění kompilačního překladače nemusí být obecně ekvivalentní se strojovým kódem konkrétního počítače. Obecně můžeme cílové kódy rozdělit dle jejich vztahu k určitému procesoru či OS takto (Češka a Hruška, 2015):

- Čistý strojový kód — jedná o strojový kód počítače bez předpokladu existence určitého operačního systému nebo knihoven. Obsahuje pouze instrukce z instrukčního souboru počítače. Tato varinta je velmi málo běžná, někdy se používá například k vytváření systémových programů (jader OS, atp.).
- Rozšířený strojový kód — tento typ kódu obsahuje navíc od předešlého také podprogramy operačního systému a podpůrné knihovní podprogramy. Tento kód se dá považovat za kód virtuálního počítače, tvořeného kombinací konkrétního technického a programového vybavení, přičemž poměr obou složek se může měnit podle různých implementací.
- Virtuální strojový kód — jedná se o nejobecnější formát zdrojového kódu. Obsahuje pouze virtuální instrukce, které jsou nezávsislé na architektuře ani na operačním systému. Tato varianta umožňuje vytvářet přenositelné překladače.

3.5 Interpretový kompilátor (hybridní překladač)

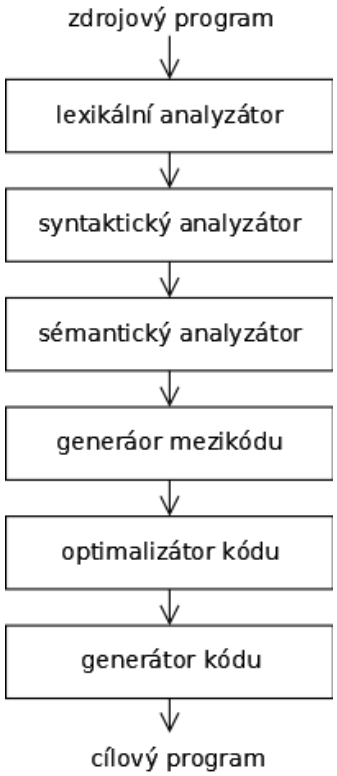
Interpretový kompilátor je zajímavý kompromis mezi interpretem a kompilátorem. Překládá zdrojový kód do kódu virtuálního stroje, který je následně interpretován. Ve funkcionalitě se snoubí rychlý překlad s průměrně až nadprůměrně rychlým spuštěním programu. Aby tomu tak opravdu bylo, musí platit dva důležité předpoklady (Habiballa, 2005).

- Kód virtuálního stroje je nižší úrovně nežli zdrojový kód programu, ale vyšší úrovně nežli nativní strojový kód.
- Kód virtuálního stroje je jednoduchého formátu, který lze lehce analyzovat a interpretovat.

Příkladem hybridního překladače je například JDK (Java Development Kit) pro jazyk Java.

3.6 Fáze překladu

V následujících kapitolách bude čtenář podrobněji seznámen s počátečními analytickými fázemi překladu, které sestávají zejména ze tří druhů analýzy zdrojového kódu. Všechny části procesu překladu jsou ve zjednodušeném podání znázorněny na následujícím schématu.



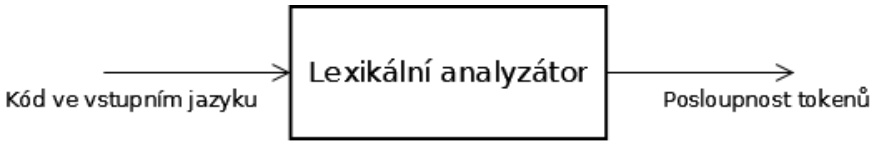
Obrázek 3.1 Jednotlivé části procesu překladu

3.6.1 Lexikální (lineární) analýza

Lexikální analýza je prvním z kroků k vytvoření překladače. Jejím úkolem je převést zdrojový text na posloupnost symbolů (lexémů), přičemž symbolem rozumíme část kódu s vlastním významem (číslo, závorka, operátor, proměnná atp.). Celý sou-

bor údajů předávaný z lexikálního do syntaktického analyzátoru označuje odborná literatura termínem token. Proces převedení jednotlivých lexémů do takové reprezentace, aby mohly být následně předány syntaktické analýze, pak nazývá tokenizace. Nutno podotknout, že lexikální analyzátor se zatím nestará o celkový smysl konstrukce (Vavrečková, 2008). Tedy z implementačního hlediska je ve většině programovacích jazyků vstupem lexikálního analyzátoru textový soubor a výstupem posloupnost symbolů (tokenů).

Vzhledem k tomu, že lexikální analyzátor je část překladače, která čte zdrojový text, může mít na uživatelském rozhraní i další úkoly. Jedním z nich je kupříkladu odstraňování poznámek a odsazovačů ze zdrojového programu. Dalším úkolem může být uchování počtu načtených znaků konce řádků, což se může hodit při výpisu chybového hlášení.



Obrázek 3.2 Jednoduché schéma lexikálního analyzátoru

Produkované tokeny jsou terminálními symboly bezkontextové gramatiky, která popisuje syntaxi vstupního jazyka.

K rozpoznání slova daného jazyka podléhajícího lexikální analýze se využívá konečný automat. Postup zpracování textu je následující (Vavrečková, 2008):

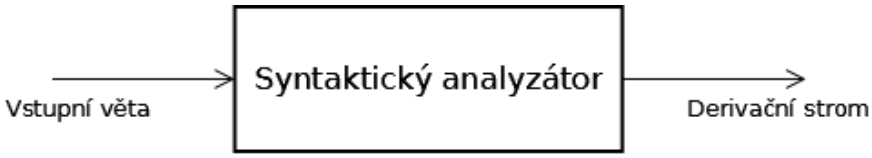
- Na vstupu je řetězec znaků, které chceme zanalyzovat.
- Automat čte postupě jednotlivé znaky, některé ukládá na výstupní pásku a mění svůj stav.
- Koncový stav automatu je pro každý typ symbolu jiný. Podle konečného stavu automatu poznáme o jaký symbol se jedná.

3.6.2 Syntaktická analýza (parsování)

Hlavním cílem syntaktického analyzátoru je rozpoznat syntaktické konstrukce a na jejich základě sestavit syntaktickou strukturu programu (derivační strom).

Úloha syntaktického analyzátoru je ve srovnání s lexikální analýzou složitější už jen z toho důvodu, že musí rozpoznávat místo regulárního jazyka lexikální analýzy složitější bezkontextový jazyk. Rozlišujeme tyto základní rozpoznatelné syntaktické konstukce (Beneš a Kolář, 2015):

- posloupnost - AB;
- alternativa - $A|B$;
- opakování - $AAA...$;
- hierarchie - $E \rightarrow (E)$;



Obrázek 3.3 Jednoduché schéma synaktického analyzátoru

3.6.2.1 Derivační strom

Výsledkem syntaktické analýzy je derivační strom (též označován jako syntaktický). To je taková konstrukce, která je schopna vyčerpávajícím způsobem popsat syntaktickou strukturu programu. Pro pořádek zde uvádím formální definici derivačního stromu (Vavrečková, 2011).

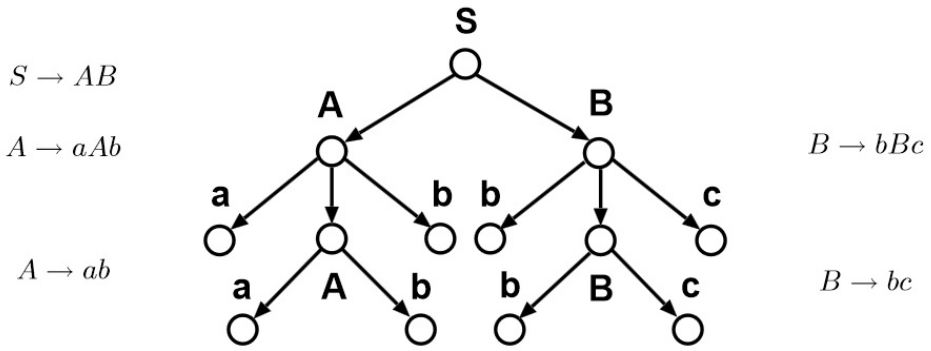
Derivační strom derivace podle gramatiky G je orientovaný acyklický graf, který má jediný kořen, do všech ostatních uzlů vstupuje právě jedna hrana, a dále má tyto vlastnosti:

- Kořen stromu je ohodnocen startovacím symbolem gramatiky.
- Listy jsou ohodnoceny terminálními symboly, všechny ostatní uzly jsou ohodnoceny neterminálními symboly.
- Všechny koncové uzly v jakékoliv fázi konstrukce tvoří větnou formu v gramatice G.

Pokud budeme mít gramatiku G a vygenerujeme z ní řetězec $aabbbcc$ derivací

$$S \rightarrow AB \rightarrow aAbB \rightarrow aAbbBc \rightarrow aAbbbcc \rightarrow aabbbbcc,$$

pak této derivaci odpovídá následující derivační strom (pravidla gramatiky jsou uvedená na stranách obrázku) (Češka, 1992):



Obrázek 3.4 Derivační strom s pravidly

K úspěšné derivaci je zapotřebí zajistit jednoznačnost vstupní gramatiky. Gramatika je jednoznačná, pokud pro každý terminální řetězec lze sestrojit právě jeden derivační strom. Samozřejmě existují gramatiky, kdy jednoznačnost nejsme schopni splnit. To má potom velmi, v negativním slova smyslu, významný dopad na efektivitu procestu analýzy. Pokud v určitou chvíli nevíme, které pravidlo v gramatice vybrat, obvykle jsme nuceni zkoušet v podstatě všechny možnosti, které existují a čekat, jestli dojdeme k požadovanému výsledku. Tento postup obecně vede k tzv. kombinatorické explozi, což je vytváření obrovského množství možností rostoucí geometrickou řadou. Je tedy jasné, že vhodný tvar gramatiky je pro výpočet reálných aplikací naprosto zásadní (Habiballa, 2013).

3.6.2.2 Metody syntaktické analýzy

Pokud mluvíme o bezkontextovém jazyce a gramatice přímo v souvislosti s rozkladem jazyka (parsováním), pak je nutné uvést dva základní způsoby provedení syntaktické analýzy, které jsou takto dělené podle způsobu konstrukce derivačního stromu. Obě tyto metody jsou obecně nedeterministické (Vavrečková, 2011).

- Syntaktická analýza shora dolů — při syntaktické analýze shora dolů začínáme derivační strom budovat od výchozího symbolu (kořene derivačního stromu) a postupnými přímými derivacemi dojdeme k terminálním symbolům, které tvoří analyzovanou větu (koncovým uzlům derivačního stromu). Tyto kroky, kdy přepisujeme od počátku S nazýváme expanze, protože neterminály rozšiřujeme na řetězce dle pravidel. Problém spočívá ve správnosti volby přímých derivací, tj. pořadí používání přepisovacích pravidel.

– Syntaktická analýza zdola nahoru — při syntaktické analýze zdola nahoru začínáme derivační strom budovat od koncových uzlů a postupnými přímými redukcemi dojdeme ke kořenu (výchozímu symbolu gramatiky). Vycházíme z myšlenky postupného zpětného odvozování, tedy principu opačnému k expanzi, kterému říkáme redukce.

K oběma metodám existují také deterministické verze implementované deterministickým zásobníkovým automatem. Číslo k uvedené v kulatých závorkách je rovno počtu dopředu prohlížených symbolů z dosud nepřečtené části vstupního řetězce (Vavrečková, 2011).

– LL(k) analýza — deterministická verze analýzy shora dolů;
– LR(k) analýza — deterministická verze analýzy zdola nahoru;

V této práci se budeme věnovat výhradně LL(k) analýze. Jedná se o jednoduchou a snadno rozšiřitelnou analýzu, která je navíc i dobře implementovatelná metodou rekurzivního sestupu. Pravidla pro expanzi určuje tzv. rozkladová tabulka. Lze tedy říct, že konstrukce LL analyzátoru je vlastně v přeneseném smyslu konstrukcí této tabulky.

3.6.2.3 Rozkladová tabulka

Pro každou gramatiku počínaje SLL(1), ale i pro obecnější typy LL gramatik, je možné sestrojit rozkladovou tabulku, která pro každý neterminál určuje, podle kterého pravidla provést expanzi.

Proces tvorby tabulky se dá jednodušše zalgoritmizovat a pracuje podle následujícího principu (Habiballa, 2013): Na příslušný řádek (odpovídající neterminálu) a příslušný sloupec (odpovídající vstupnímu symbolu na pravé straně pravidla) se vloží řetězec, který je u zkoumaného pravidla na pravé straně. Pro uvednou gramatiku by rozkladová tabulka vypadala následovně (Habiballa, 2013):

$$G = (\{S, A\}, \{a, b, c\}, S, P)$$
$$P : S \rightarrow_1 aASc,\ S \rightarrow_2 b,\ A \rightarrow_3 a,\ A \rightarrow_2 cSAb$$

Tabulka 3.2 Rozkladová tabulka gramatiky G

	a	b	c
S	aASc, 1	b, 2	
A	a, 3		cSaB, 4

Pomocí rozkladové tabulky již lze pomocí konkrétních pravidel provést rozpoznání vstupního slova. Algoritmus pracuje s pamětí typu zásobník a provádí čtyři různé operace (Habiballa, 2013):

- expanze — rozšíření na další neterminál;
- porovnání — porovnání terminálního symbolu;
- přijetí — úspěšný konec analýzy;
- chyba — chybný konec analýzy;

V počáteční konfiguraci se v zásobníku vždy nachází startovací symbol gramatiky a symbol # a na vstupní pásce je analyzovaná věta. Koncová konfigurace pak předpokládá v zásobníku jediný speciální symbol # a na vstupní pásce prázdný řetězec.

3.6.2.4 SLL(1) analýza a Q-analýza

Pokud budeme více zkoumat analýzy typu „shora dolů“, přijdeme na to, že klíčovým problémem je rozhodnutí, jaké pravidlo použít, pokud máme u jednoho neterminálu více možností expandování. Proto existují typy gramatik s omezením, které zabraňuje vzniku nejednoznačnostem.

Pokud bude vstupní gramatika splňovat podmínku, která spočívá v tom, že vždy přepisuje neterminál na řetězec začínající terminálem a dvě pravidla pro jeden neterminál musí začínat různými terminály, pak se jedná o takzvanou jednoduchou LL(1) gramatiku. Hlavní omezení této gramatiky pro praxi tkví v nemožnosti použít epsilon pravidlo a tím pádem se distancuje od popisu složitějších syntaktických struktur.

Přesně tento problém řeší tzv. Q-gramatika. Tedy, jedná se praticky o jednoduchou LL(1) gramatiku, které je umožněno použití epsilon pravidel. Má však stále jednu zásadní nevýhodu a tou je nutná podmínka, aby každé pravidlo začínalo terminálem. V praxi to vede k nepřehlednosti a nesystematičnosti gramatiky (Habiballa, 2013).

3.6.2.5 LL(1) analýza

LL(1) gramatika, která se používá k LL(1) analýze již netrpí neduhy svých předchůdkyň, a proto je v praxi jednou z nejpoužívanějších metod syntaktické analýzy. Stále je ale třeba si dávat pozor na vznikající kolize v rozkladové tabulce, které mohou být dvojího druhu (Habiballa, 2013):

- FIRST-FIRST kolize — pokud existují dvě pravidla pro jeden neterminál, řetězce na pravé straně musí začínat různými terminálními symboly.
- FIRST-FOLLOW kolize — pokud se nějaký neterminál přepisuje na epsilon,

pak všechna pravidla pro tento neterminál musí začínat jiným symbolem než který za neterminálem může následovat.

V případě obecné bezkontextové gramatiky se můžeme pokusit o její převod na gramatiku LL(1). K tomu se využívají různé postupy, přičemž každý z nich odstraňuje jeden konflikt v rozkladové tabulce gramatiky, tedy vytváří ze vstupní gramatiky takovou gramatiku, která je deterministická a platí pro ni skutečnost, že na základě jednoho prohlíženého symbolu a daného neterminálu na vrcholu zásobníku existuje pouze jedno možné pravidlo pro expanzi. Nutno podotknout, že ani po provedení všech transformací není zaručeno, že bude výsledná gramatika opravdu ve formě LL(1).

- Odstranění levé rekurze — algoritmus nahrazuje levou rekurzi za pravou a odstraňuje levokurzivní smyčky.
- Levá faktorizace — algoritmus, který přímo řeší FIRST-FIRST konflikt v rozkladové tabulce.
- Levá rohová substituce — jedná se o zjednodušený převod gramatiky do GNF. Substituuje se neterminál na začátku pravidla za řetězce, na které lze tento terminál přepsat.
- Pohlcení terminálu — algoritmus. který odstraňuje FIRST-FOLLOW konflikt v rozkladové tabulce.
- Extrakce pravého kontextu — lze využít pouze tehdy, vykytuje-li se přímo za problematickým neterminálem.

V dnešní době existuje i několik generátorů LL(1) analyzázorů. Nejznámější jsou například yacc, bison nebo ANTLR (Habiballa, 2013).

3.6.2.6 LL(k) analýza

Gramatiky, které jsme si doposud představili dokáží jednodušše provádět syntaktickou analýzu pouze s informací, jaký následuje v analyzovaném slově první symbol. Pro většinu praktických problémů tento postup postačuje. Přesto ale existují LL gramatiky, které pracují s informací o k symbolech následujících ve slově. Dokonce existují takové gramatiky, kterým ani toto k rozhodování nestačí a musí se rozhodovat také na základě dosavadního průběhu analýzy.

LL(k) gramatiky mají upravené funkce FIRST a FOLLOW. Funkce FIRST_k(α) obsahuje jednak všechny řetězce o velikosti k, které se mohou vyskytovat na začátku řetězce αa jednak všechny řetězce o velikosti menší než k, pokud se takový řetězec vyskytuje na konci slova. Obdobným zesložitěněním prochází i funkce FOLLOW_k(α). Nalezení všech podmnožin je proto v tomto případě velmi náročné.

V syntaktické analýze se využívá podobný algoritmus na vytvoření tabulky jako u LL(1) gramatik s rozdílem rošíření na k-symbolů (Habiballa, 2013).

3.6.2.7 Pomocné množiny pro syntaktickou analýzu

Existují dvě množiny, které mohou uživateli celkem významně usnadnit práci při syntaktické analýze. V případě LL(k) analýzy je jejich sestavení navíc naprostá nutnost (Habiballa, 2013).

- Množina FIRST(α) je definovaná pro libovolný řetězec α. Jedná se o množinu obsahující všechny terminály, kterými může začínat některý řetězec odvoditelný z α v dané gramatice.
- Množina FOLLOW(A) je definována pro libovolný neterminál A. Jedná se o množinu obsahující všechny terminály, které mohou v některé větné formě derivace následovat za A, případně i konec vstupu.

Uveďme si příklad pro výpočet obou množin na gramatice s následujícími pravidly:

$$S \rightarrow Aa|bS$$
$$A \rightarrow cAd|B$$
$$B \rightarrow fS|\varepsilon$$

Množiny FIRST pravých stran pravidel vypadají takto:

$$FIRST(Aa) = \{c, f, a\}$$
$$FIRST(bs) = \{b\}$$
$$FIRST(cAd) = \{c\}$$
$$FIRST(B) = \{f, \varepsilon\}$$
$$FIRST(f) = \{f\}$$
$$FIRST(\varepsilon) = \{\varepsilon\}$$

Množiny FOLLOW levých stran pravidel jsou následující:

$$FOLLOW(S) = \{\varepsilon, a, d\}$$
$$FOLLOW(A) = \{a, d\}$$
$$FOLLOW(B) = \{a, d\}$$

3.6.2.8 Derivace slova

Druhá písmena u zkratek LR a LL analýzy vyjadřují typ derivace slova. Rozklad (derivace) slova může být buď levý nebo pravý (Habiballa, 2005).

- Pravá derivace přepisuje v každém kroku neterminál nejvíce vpravo, tedy vždy ten poslední.
- Levá derivace v každém kroku přepisuje první neterminál zleva.

Představme si gramatiku definovanou takto (Kot, 2015):

$$\begin{aligned} &\mathrm{G} = (\\ &\Pi = \{E\} \\ &\Sigma = \{a, +, *\} \\ &P = \{E \rightarrow E + E | E * E | a\}) \end{aligned}$$

Levá derivace této gramatiky probíhá tímto způsobem:

$$E => E + E => a + E => A + E * E => a + a * E => a + a * a$$

Pravá derivace takto:

$$E => E + E => E + E * E => E + E * a => E + a * a => a + a * a$$

Derivaci můžeme samozřejmě provádět také propojením obou pravidel. Pak výsledná derivace není ani pravá, ani levá.

$$E => E + E => E + E * E => E + a * E => a + a * E => a + a * a$$

Je důležité si uvědomit, že pro všechna provedená odvození je ve výsledku vždy stejný derivační strom. Pro určení jednoznačnosti gramtiky se ale obvykle využívá levé odvození a jeho derivační strom. Tedy musí existovat právě jedno takové odvození a právě jeden derivační strom. Na první pohled je zřejmé, že pro tento příklad existuje i jiná levá derivace a jiný derivační strom, tím pádem můžeme tvrdit, že uvedená gramatika je nejednoznačná.

3.6.3 Sémantická analýza

Sémanatická analýza je poslední částí pomyslné analytické části překladače. Každá skupina symbolů, vytvořená při syntaktické analýze dostane svůj výnam. Jsou zde prováděny kontroly deklarací, typová kontrola, atd.

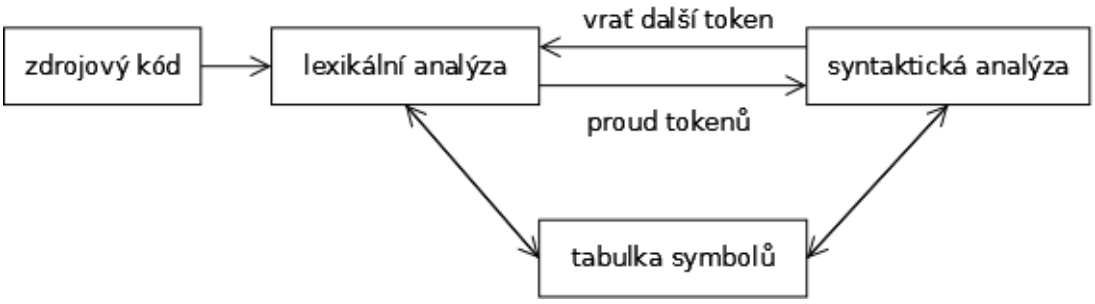
Následují syntetické kroky analýzy, tedy vygenerování cílového programu a optimalizace kódu, která je výrazně důležitá ve věci zvýšení efektivity vytvořeného programu. Pro tuto činnost se obvykle program převádí do tzv. intermediárního kódu, což je taková betaverze hotového programu. Po občas značně složité optimalizaci se kód převede do cílového programu. Jedná-li se o čistě interpretační překladač, je možné začít s intepretací ihned po skončení sémantické analýzy. Všechny části překladače spolupracují s pracovními tabulkami překladače. Zakladní tabulka kompilátoru i interpretu se nazývá tabulka symbolů (Wirth, 2005).

3.6.4 Tabulka symbolů

Tabulka symbolů je hlavním úložištěm dat víceméně pro všechny části překladače. V námi probírané analytické části slouží hlavně k uložení jednotlivých proměnných, jejich typu a výchozích hodnot. Implementačně je třeba řešit tabulku takovým způsobem, aby nad ni bylo možné provést zejména tyto dvě základní operace:

- Vkládání — operace, které vloží nový symbol do tabulky. Je nutné kontrolovat případně duplicity.
- Vyhledávání — operace, která musí nalézt daný identifikátor a vracet jeho funkční hodnoty.

Tyto dvě operace samozřejmě velmi ovlivňuje jazyk, v kterém je provedena implementace. Jiné postupy budeme aplikovat na jazyky bez blokové struktury (například Basic) a jiné na jazyky s blokovou strukturou (například Pascal). Víceméně lze říci, že situace je příznivější pro jazyky bez blokové struktury. Pro jazyky s blokovou strukturou totiž musíme zajistit lokální chování takové struktury. K tomu se využívají stejné postupy jako pro první skupinu jazyků s rozdílem přidání zásobníku (Habiballa, 2005, s.18).



Obrázek 3.5 Princip analýzy ve spojení s tabulkou symbolů

3.6.5 Reakce na chybu

V každé z jednotlivých částech překladače může dojít k chybě. Nejvíce nás ale obvykle zajímají chyby z analytické fáze. Překladač či analyzátor dokáže reagovat na chybu obecně dvěma způsoby.

- Při prvním výskytu chyby se překladač zastaví, chybu diagnostikuje, informuje uživatele a čeká na opravu (takto pracuje například Turbo Pascal).

- Překladač se pokusí najít co nejvíce chyb najednou a ty po ukončení průchodu zobrazí uživateli (například C++).

Chyby můžeme obecně rozlišit na syntaktické a sémantické (Poole, 2002). Syntaktické chyby jsou chyby zanesené do programového kódu, dále je dělíme na lexikální a gramatické.

- Lexikální chyby jsou často překlepy v lexémech, například „ofr“ místo „for“.
- Gramatická chyba souvisí s porušením nějakých pravidel gramatiky, která definuje jazyk. Například `if něco něco` (chybí zde vyhrazené slovo `then`).

Sémantické chyby se týkají významu kontrukce programu a jeho logiky. Zahrnují logické chyby, typové chyby a běhové chyby.

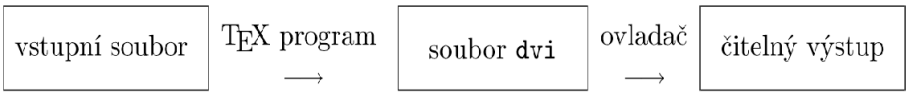
- Logické chyby se týkají špatně koncipovaného programu nebo algoritmu. Může jít například o zápis `while x=y do`, přičemž `x` a `y` nemohou nikdy nabývat rozdílných hodnot, tudíž se nikdy neprovedou příkazy vně této této konstrukce.
- Typové chyby nastávají, pokud je operátor aplikován na argument špatného typu.
- Běhové chyby mohou nastat pouze při spuštění programu. Jedná se většinou o chyby způsobené interakcí uživatele s počítačem, uživatel má zadat číslo, vybere si nulu a tou se poté dělí jiné číslo, což je nepovolená operace.

4 Použité technologie, programy a formáty

4.1 TEX

TEX je volně šiřitelný počítačový program k pořizování vysoce kvalitní elektronické sazby. Patří do rodiny značkovacích jazyků (markup languages). Za jeho vznikem stojí Stanfordský profosor Donald Ervin Knuth, který již v 70. letech dvacátého století položil základní kámen tomuto nyní velmi populárnímu systému. Mezi jeho nesporné výhody patří celková otevřenost a právě proto již za několik desítek let stihlo vzniknout velké množství uživatelských nadstaveb. TEX je dodnes obecně považován za nejlepší nástroj pro sázení matematických vzorců, ale hojně se využívá i v běžné sazbě.

Nutno podotknout, že se již málokdy využívá čistý TEX, ale z důvodu zjednodušení a zpříjemnění práce spíše některý z jeho nadstaveb. Z nejznámějších to může být například LATEX, XETEX, ConTEXt nebo MikTEX. K použití TEX-u potřebujeme zpravidla jeho odpovídající implementaci, která zahrnuje kromě samotného překladače a sad řídicích sekvencí pro různá použití také prohlížeče, konvertory a jiné doplňky. K fungování programu TEX lze zjednodušeně říci, že TEX je překladač, který obdrží na vstup soubor s textem (včetně texových příkazů pro sazbu), rozměry znaků a z těchto údajů vygeneruje soubor, který obsahuje rozmístění znaků na každé stránce. Ten se poté zpracuje dle libosti uživatele (například tisk). Celý proces můžeme dobře chápat pomocí tohoto obrázku:



Obrázek 4.1 Schéma práce TEX-u

4.1.1 Hlubší souvislosti práce TEX-u

TEX-u se při zpracování dokumentu vloží na vstup textový soubor, který kromě samotného textu obsahuje řídicí sekvence nebo také povely (některé z nich si rozeberem v následujících kapitolách). TEX zdrojový kód překládá tak, že rozmístí jednotlivé znaky do sazebního zrcadla. Ty jsou pak chápány jako obdélníky, jejichž rozměry jsou již uloženy do souboru typu tfm. Průběh překladu obvykle doprovází důležitá varování a chybová hlášení, která se zapisují jak na standardní výstup, tak také do logového souboru, aby byly připraveny k pozdějšímu nahlédnutí. TEX na konci své překladové fáze vytvoří soubor s informacemi o rozmístění jednotlivých znaků dokumentu. Výstupový soubor má formát .dvi (tzn. DeVice Independent, neboli nezávislý na zařízení). Pak už jen záleží jak a v jaké kvalitě chceme výstup prohlédnout, tzn. záleží už jen na možnostech uživatele, zda mu postačí výstup prohlédnout na obrazovce nebo jej chce poslat k tisku, či se jej rozhodne zpracovat úplně jinak. (Rybička, 2003)

Práce s TEX-em je, jak uvádí Rybička (2003), velmi podobná programování a probíhá v těchto krocích:

- příprava zdrojového textu;
- překlad (vysázení);
- prohlížení;

Tyto kroky se opakují tak dlouho, dokud není uživatel spokojený s výsledným vysázeným dokumentem. Velkou nevýhodou této metodiky je absence výsledného obrazu v průběhu práce se zdrojovým kódem. Zatímco komerční typgrafické systémy, obvykle s WYSIWYG uživatelským rozhraním, předpokládají piplavou ruční práci myší, systém TEX od uživatele očekává zápis efektivních příkazů, které vedou k preciznímu a rychlému zpracování. Mezi další výhody TEX-u patří (Rybička, 2003):

- použití libovolného textového editoru;
- použití libovolných textových filtrů a jiným programů pro úpravu zdrojového kódu;
- přizpůsobivost změnám prostředí;
- snadná přenositelnost, archivace a bezpečnost zdrjových textů;
- využití možností operačního systému ke zdokonalení systému TEX;

4.1.2 Možnost učení se

TEX při svém běhu nenačítá pouze vstupní příkaz s textem a příkazy. Při startu nejdříve přečte rozsáhlý soubor definic příkazů, které pak může autor v textu použít.

Kdyby tento soubor definic TEX nenačetl, tak je v jeho výbavě v základu k dispozici pouze něco kolem tří set primitivních příkazů, které tvoří jakési stavební kameny pro odvození nových příkazů načtených z definic. Těmto primitivním příkazům říkáme příkazy nižší úrovně, zatímco ty příkazy, které uživatel používá v textu jsou známe jako příkazy vyšší úrovně (někdy též označovány jako makra). Tento soubor maker je TEX schopen uložit a následně obrazu struktur z paměti vytvořit binární soubor. Tomuto procesu se říká inicializace souboru maker. V praxi TEX přečte tento inicializovaný binární soubor a tím se poměrně rychle naučí všechna obsažená makra. Načítanému binárnímu souboru se odborně říká formát. Tato jedinečná možnost učení dává TEX-u veliké možnosti. Ve většině TEX-ových implementacích má proces učení trochu jiný technický ráz, totiž učení je dotaženo k dokonalosti tím, že při inicializaci je TEX schopen vytvořenou binární strukturu formátu zahrnout do nového spustitelného kódu a sestavit tak novou verzi programu TEX, která toho umí o něco více, nežli jeho předchůci. Obyčejnému uživateli je sice ve výsledku jedno, jak celý proces proběhne, jestli se spustí základní TEX, jemuž se předloží formát nebo jestli se použije nově vytvořený chytřejší program (Doob, 1993, s. 3).

4.2 ConTEXt

ConTEXt je program pro vysoce kvalitní sazbu dokumentů, přesněji řečeno se jedná o balík maker a nových funkčností pro systém TEX. Narozdíl od jiných světoznámých textových procesorů, kde je konečný dokument formátován současně s vytvářením dokumentu, ConTEXt-ové dokumenty jsou psané bez jakéhokoliv formátování s použitím jazyku maker. To znamená, že dochází ke změnám obsahu dokumentu nezávisle na jeho formátování. Tato skutečnost platí i opačně.

ConTEXt, jakožto jeden z mladších balíku maker, byl vyvinut teprve v roce 1990 Hansem Hagenem z nizozemské společnosti PRAGMA Advanced Document Engineering (Context Garden, 2015). Jak píše sám tvůrce (Hagen, 2000), vývoj ConTEXt-u začal během produkce velkého množství učebních materiálů, manuálů a učebnic. V roce 1994 se stala distribuce natolik stabilní, že si dokázala ospravedlnit svůj první nizozemský manuál. Během dalších let se ConTEXt zlepšoval z pohledu jak funkčního, tak také jazykového, když byly přidány interface v anglickém a německém jazyce.

4.2.1 Základní vlastnosti ConTEXt-u

ConTEXt má několik významných vlastností, pro které může být upřednostňován

před jinými balíky.

- Použití grafiky — ConTEXt v sobě integruje grafický doplněk MetaFun napsaný v jazyce MetaPost, díky kterému uživatel získá novou množinu grafických prvků, přidává možnost kreslit do dokumentu nebo upravovat pozadí dokumentu. ConTEXt je systém, která přímo spolupracuje s MetaPostem a všechny jeho funkce jsou uživateli k dispozici přímo při tvorbě dokumentu. O problematice použití MetaPostu v ConTEXt-u pojednává rozsáhlý manuál (Hagen, 2010).
- Kompatibilita — ConTEXt je podporován texovými enginy jako pdfTEX nebo XeTEX. Uživatel nemusí měnit uživatelské prostředí.
- Jazyková různorodost — ConTEXt nabízí uživatelské prostředí s podporou Angličtiny, Němčiny, Italštiny, Francouzštiny, Holandštiny, Rumunštiny a Češtiny. Výstup je podporován jazyky celého světa (Arabština, Čínština, atd.)
- Licence — ConTEXt je dostupný zcela zdarma.
- Systém poskytuje vynikající kvalitu výstupu matematického textu.
- Neustálý vývoj — zatímco vývoj plain TEX-u byl zastaven již v roce 1982, ConTEXt je neustále zlepšován a vznikají k němu nová makra.

4.2.2 ConTEXt vs LATEX

Vzhledem k nemalé pozornosti, kterou na sebe ConTEXt v posledních letech strhává (zejména u fanoušků TEX-u), jistě stojí za pozornost srovnání se známějším rozšířením LATEX-em. Tato kapitola pojednává o hlavních rozdílech mezi těmito systémy z pohledu funkcionalit, rychlosti a podpory.

4.2.2.1 Funkcionalita

Hlavní rozdíl mezi oběma systémy je již v samotné struktuře. Zatímco LATEX je poskládán z mnoha oddělených souborů-balíčků (packages), které jsou při započetí překladu načítány, ConTEXt je koncipován jako monolitický systém, se všemi funkcionalitami koncentrovanými do vlastního formátu. Drobná LATEX-ová rozšíření jsou relativně snadno implementovatelná a běžnému uživateli stačí jenoduchý manuál s vysvětlením, co který balíček přesně dělá, nikoliv nějaká hlubší znalost systému. To je nesporná výhoda balíčkového konceptu, ale je tu druhá strana věci. Rozšiřitelnost bývá problémem pro správce LATEX-ového jádra (kernel) a existuje spousta balíčků, které by mohly mít lepší funkcionalitu, pokud by byly více podporovány jádrem sys-

tému. U ConTEXt-u je situace vcelku odlišná. Přidání nové funkcionality není tak jednoduché a každá změna musí být schválena vývojářemi systému. Na druhou stranu ConTEXtnemusí zůstat kompatibilní s jeho staršímu verzemi a nové změny bývají zapracovány obvykle v krátkém časovém horizontu. Obecně lze říci, že ConTEXt přejímá téměř veškerou funkcionalitu LATEX-ového jádra, možná s jedním rozdílem, a sice, že nemá obrázkové prostředí, které ovšem nahrazuje podporou formátu METAPOST.

4.2.2.2 Rychlost

Měření lze provést na několik různých úrovních. Nejdříve lze porovnat paměťové kapacity potřebně k vykonání triviálního programu „HelloWorld“ v obou systémech. ConteXt ve výsledku spotřebuje o hodně více prostoru v paměti. Typický contextový soubor o kapacitě 2,3 Megabajty je v porovnání s latexovými 550 Kilobajty a poměrem 5:1 opravdu velký rozdíl hovořící za vše. Podrobnější srovnání je k dispozici v odborném článku (Hoekwater, 2015, s.282).

Pokud si oba systémy rozebereme z pohledu rychlosti překladu, je docela zřejmé, že musí být o hodně pomalejší ConTEXt protože v základu nabízí o mnoho více funkcionalit, nežli LATEX. Byly provedeny testy na jednoduchém vstupu, který tvoří 200 stran obyčejného textu.

Tabulka 4.1 Srovnání ConTEXt vs. LATEX v rychlosti, zdroj: Hoekwater (2015)

System	Pointsize	Pages	Time
plain	10pt	181	21s
LATEX	10pt (default)	244	27s
ConTEXt	12pt (default)	259	2m59s
ConTEXt	10pt	185	2m17s
ConTEXt	6pt	83	1m22s

Jak vidno, předpoklady se naplnily a ConTEXt je opravdu pomalejší nežli LATEX. Nutno ale podotknout, že pokud spustíme ConTEXt-ový překlad vícekrát, čas se po každé iteraci zkracuje.

4.2.2.3 Podpora

Uživatelská podpora obou balíků je vcelku rozsáhlá. Z pohledu LaTeX-u je složená typicky z několika uživatelských spolků a kromě toho existuje také speciální adresa k posílání nalezených bugů (v každé distribuci je přiložen soubor bugs.txt s více informacemi). Všechna podpora pro ConTeXt je soustředěna na skupinu „Context Task Force“, která slouží jako spojka mezi uživateli a společností Pragma. Obě TeX-ové distribuce jsou také součástí archivní TeX-ové sítě CTAN, která sdružuje veškeré materiály související s TeX-em.

4.2.3 Příkazy

Vstupní soubory by měly obsahovat sazební materiál, který chceme zpracovat pomocí příkazů ConTeXt-u, na které se později blíže podíváme.

Všechny ConTeXt-ové příkazy začínají znakem obrácene lomítko \(backslash). Ve většině případech takový příkaz nějak zpracovává text následovaný za ním. například příkazem `\starttext` a `\stoptext` označujeme začátek a konec celé textové části dokumentu. Místo nad prvním příkazem (někdy označováno jako preambule) je vyhrazeno pro deklaraci a definování nových příkazů a nastavení vzhledu dokumentu (Otten a Hagen, 2006, s. 6).

Uživateli LaTeX-u se moc nebude líbit použití výrazu příkaz, protože se v LaTeX-u ve stejném významu používá spíše pojem makro. Makro je obvykle malý program, vytvořený schopnějším uživatelem programátorem. Pojmu příkaz budeme rozumět výhradně jako makru, obvykle používanému s parametry, psanými mezi hranatými závorkami. Příkaz je často následován zmíněnými parametry nebo prostým textem. Parametry jsou obsaženy v hranatých závorkách (`[]`). Text se zapisuje do složených závorek. Pro představu zde uvádím příklad.

```
\framed[width=2cm,height=1cm]{můj příklad}
```

Po překladu dostaneme na vstupu toto: 

4.2.4 Soubory

ConTEXt očekává na vstupu soubor v kódování ASCII nebo UTF8, proto je nutné využívat takový textový editor, který dokáže editovat a vytvářet soubory s tímto kódováním (Hagen, 2000). ASCII soubor s příponou .tex je zpracováván TEX-em, tedy TEX-ovým kompilátorem. Během procesu je vytvořen soubor s připonou dvi s grafickými příkazy, které určují rozmístění textu na stránce. Tento soubor je poté transformován do formátu (například PDF) akceptovaného výstupními zařízeními (obvykle tiskárna). ConTEXt vychází ze základního plain TEX-u (jedná se o základní TEX-ový dialekt, z něhož vychází většina ostatních TEX-ových distribucí).

4.2.5 Text

TEX obsahuje znaky z ASCII tabulky. Některé znaky mají v TEX-u zvláštní význam a ke správnému vysázení je zapotřebí před ně vložit zpětné lomítko. Např znak procento je nutno zapsat jako %. Kdyby uživatel tuto skutečnost ignoroval, program by znak vyhodnotil jako začátek komentáře a všechen text napsaný za procentem by se na výstupu nevysázel. Zde uvádím seznam všech rezervovaných znaků (Hagen, 2000): # $ % ^ & _ { } ~ \

Znak zpětného lomítka je zapotřebí ošetřit jinak než pouhým přidáním dalšího zpětného lomítka \\ . Tato kombinace znaků se totiž používá k zalamování řádku. Správně se musí použít povelu `\backslash`. Výše vypsané znaky se správně zapíší takto (Hagen, 2000):

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

Všechny znaky v TEX-u a ConteXtu mají také přesně definované rozměry udávané v různých jednotkách, jejichž seznam je k náhledu v tabulce.

Tabulka 4.2 Rozměry v TEX-u, zdroj: Hagen (2000, s. 18)

dimension	meaning	equivalent
pt	pint	181 72.27pt = 1in
pc	pica	1pc = 12pt
in	inch	1in = 2.54cm
bp	big point	72bp = 1in
cm	centimeter	2.54cm = 1in
mm	millimeter	10mm = 1cm
dd	didot point	1157dd = 1238pt
cc	cicero	1cc = 12dd
sp	scaled point	65536sp = 1pt

4.2.6 Vymezení dokumentu

Dokument psaný systémem context lze rozdělit na dvě části. V hlavičce dokumentu, v tzv. deklarační oblasti či preambuli se vyskytuje nastavení parametrů dokumentu. Poté následuje blok ohraničený příkazy `\starttext` a `\stoptext` a právě vše umístěné mezi těmito příkazy bude systémem ConTEXt zpracováno. Hlavní textový blok můžeme rozdělit na čtyři části (Otten a Hagen, 2006):

- úvodní část;
- hlavní část;
- závěrečná část;
- přílohy;

Všechny čtyři části jsou odděleny podobně jako celý blok párovými příkazy. Systém ConTEXt využívá veliké množství příkazů. Některé příkazy se zapisují bez parametrů (`\starttext`), ale obvykle následuje za příkazem dvojice hranatých závorek, případně i dvojice složených závorek. Vezěme si například příkaz `\chapter[a]{b}`. Tento povel po ConTEXt-u požaduje, aby vykonal rovnou několik činností, které se týkají úpravy, typografie a struktury. Těmito činnostmi může být (Otten a Hagen, 2006):

- začátek nové stránky;
- zvětšení počítadla kapitol o jednotku;
- umístění číla kapitoly před název kapitoly;
- vynechání určitého prostoru za nadpisem;
- použití většího písma;

– uložení kapitoly (jejího názvu a čísla stránky) do obsahu;

Všechny tyto činnosti se vykonají s argumentem zadaným mezi složenými závorkami, v našem případě `{b}`. Parametr v hranatých závorkách `[a]` obsahuje referenci s návěštím, které lze použít jako odkaz na odpovídající kapitolu. V dokumentu potom stačí zadat povel `\in{kapitola}[a]` a ConTEXt vysází číslo kapitoly, zatímco povel `\about[a]` vrátí její název.

4.3 TEX-ové distribuce

Z historického hlediska již bylo vytvořeno velké množství TEX-ových distribucí. Postupem času ale zůstalo aktivních jen několik vývojových větví. Nás budou zajímat primárně takové distribuce, které jsou nabízeny zdarma, obsahují ConTEXt a hlavně jsou do dnešního dne stále ve vývoji a mají podporu.

– TEX Live Tato distribuce je v dnešní době asi nejrozšířenější díky podpoře všech možných operačních systémů. Zahrnuje hlavní programy TEX-u a vše s nimi spojené, balíčky maker, fonty, včetně podpory mnoha jazyků z celého světa. TEX Live byl vyvinut v roce 1996 spolkem lidé, točících se kolem diskuzních skupin uživatelů TEX-u, proto nelze vyjmenovat konkrétní členy, kteří se podíleli na vývoji (TeXLive, 2015).
– MikTEX V tomto případě se jedná o jednu z distribucí, která je vyvíjena pouze pro operační systém Windows. Nabízí velice jednoduchou intuitivní instalaci, což rozhodně nebývá u TEX-ových produktů zvykem. Integruje v sobě modul na stahování chybějících komponentů a automatických aktualizací. V současné době je kromě klasické verze ke stažení i verze přenosná bez nutnosti instalace a bez zbytečného zasahování do registrů PC, obě tyto verze nabízí grafický interface (MikTeX, 2015).

Dále existují fungující distribuce jako proTeXt nebo MacTeX, které ale primárně využívají základu a funkcí distribucí výše popsaných.

4.4 Programovací jazyk Lua

Syntaktický analyzátor, tedy hlavní cíl této diplomové práce, je naprogramován v jazyce Lua. Programovací jazyk Lua je bezplatný, velice rychlý skriptovací jazyk, v dnešní době hojně využívaný například při programovaní umělé inteligence a scénářů v počítačových hrách.

4.4.1 Stručná historie

Jazyk se začal vyvíjet v roce 1993, kdy skupina inženýrů z brazilské ropné společnosti PETROBRAS potřebovala zjednodušit svoji práci při ručním zadáváním údajů pro aplikaci provádějící různé simulace. Postupně se produkt začal rozšiřovat po celé firmě a přicházelo čím dál více požadavků na obohacení jazyka o další funkcionality. Během roku vývoje byla představena první verze plnohodnotného programovacího jazyka, který dostal název Lua (v portugalštině měsíc).

První licence na použití jazyka byla bezplatná pouze pro akademické účely, proto došlo k většímu rozšíření až při nasazení nové verze Lua 2.1, která již byla představena jako freeware a dokonce také jako opensource (ovšem s vlatní licencí).

Nyní je Lua nabízena ke stažení již v páte verzi s dokumentací přeloženou do nejrůznějších světových jazyků. Její využití je k vidění v mnoha známých komerčních aplikacích (například Adobe Photoshop, Lightroom), embedded systémech (Ginga) nebo počítačových hrách (World of Warcraft, Angry Birds). O jazyce Lua již vyšlo i několik knih (Tišnovský, 2009). Přes všechny výhody a kvality jde jazyk Lua spíše tou alternativní cestou a většina běžných programátorů raději volí nějaký obecně rozšířenější jazyk. Výjimkou je mezi známými českými prográmatory například František Fuka, známý jako návrhář interaktivních scénářů a herních algoritmů , který dělá, jak sám tvrdí, v jazyce Lua 95% své práce a pohlíží na Luu jako na metajazyk, vhodný pro rychlý prototyping a pro implementaci složitějších věcí (Hassman, 2009).

4.4.2 Lua z pohledu programátora

Jazyk Lua byl již od svého počátku koncipován jako integrovatelná součást konvenčních jazyků, jako je například C. Lua se nesnaží být funkčním jazykem v oblastech, které bez problému zvládne jazyk C. Právě proto, že se v mnoha věcech spoléhá na jiný jazyk, je Lua velmi jednoduchý a hlavně prostorově nenáročný jazyk. Co ale vlastně Lua nabízí za benefit tak rozsáhlým jazykům jako je C? Perfektně zvládá práci s dynamickými proměnnými, není závislá na hardware a nabízí jednoduché a rychlé testování a debugging. Má navíc ještě automatickou správu paměti, bezpečné prostředí a další výhody. Jazyk Lua bychom mohli zařadit mezi další skriptovací jazyky, jakými jsou například Perl, Ruby, Python a jiné. I když je Lua s těmito jazyky v základu dosti podobná, uvádím zde profilové vlastnosti, které ji činí jedinečnou:

- Rozšiřitelnost — Lua řeší většinu z její funkcí skrze externí knihovny. Velmi jednoduše jde Lua propojit s jinými jazyky.
- Přenositelnost — spuštění Lua skriptu je možné na všech možných platfor-

mách. Lua nevyužívá podmíněné kompilace k přizpůsobení kódu různým strojům. Místo toho se striktně drží standaru ANSI C. Tudíž Lua skript je spustitelný všude, kde je přítomen kompilátor ANSI C.
- Jednoduchost — Lua je jednoduchý a malý jazyk. Jeho kompletní distribuce včetně manuálů se vejde na jednu disketu.
- Efektivita — Lua je ná základě provedených benchmarků jedním z nejrychlejších na poli skriptovacích jazyků.

4.4.3 Lua z pohledu ConTEXt-u

Již zde abylo avizováno, že je sytém ConTEXt úzce spjat s programovacím jazykem Lua. ConTEXt již při svém spuštění načítá množství Lua knihoven. Nejen, že je tedy celý ConTEXt v Lue implementován, ale uživatel také může použít skripty v jazyce Lua v contextovém dokumentu. K tomu je potřeba dát vědět TEX-ovému procesoru, že začíná vyhodnocovat lua kód, přičemž existují dva způsoby, jak to provést. Prvním je použití celého lua prostředí, ohraničeného povely `\startluacode` a `\stopluacode` a druhým jednotlivý příkaz `\ctxlua`. V praxi si uživatel definuje funkci uvnitř lua prostředí a poté definuje texový příkaz, který ji využívá. S použitím lua kódu uvnitř texového dokumentu vyvstává zásadní problém. Všechen lua kód (například i komentáře) musí být zapsán v adekvátní contextové syntaxi. Například řetězec `"\undefined"` způsobí při kompilaci okamžité selhání (Context Garden, 2015a).

```
\ctxlua
  {-- Lua komentář
   tex.print("toto se nevypíše")}
\ctxlua
  {% TeX komentář
   tex.print("toto ano")}
```

4.4.4 LuaTEX

Jedná se o další z TEX-ových rozšíření. Jedná se o nadstavbu známějšího PDFTEX-u, která v sobě má zahrnut integrovaný skriptovací engine pro jazyk Lua. Hlavním cílem projektu bylo poskytnout takovou nadstavbu TEX-u, která bude interně zvládat zpracovávat Lua skripty. LuaTEX nabízí nativní podporu pro OpenType fonty. Fonty se

dají stahovat kromě knihoven operačního systému (jako je to například u nadstavby XeTeX), také skrze knihovny editoru FontForge (FontForge, 2015).

Vývoj projektu začal teprve v roce 2005 a první beta verze byla představena na konferenci TUG2007 v San Diegu. Poslední stabilní verze je součásti distribuce TeX Live 2013.

4.4.5 Vzájemná interakce obou jazyků

Pokud se budeme chtít více přiblížit vzájemné interakci obou jazyků, zjistím, že vše probíhá velmi intuitivně a jednodušše.

TeX načte vstupní soubor do paměti, pracuje nad jednotlivými tokeny, dokud nenarazí na klíčové slovo `\directlua` (to je ve skutečnosti primitivum LuaTeX-u a v praxi píšeme zmíněné `\startluacode` a `\ctxlua`). V tuto chvíli přestává zpracovávat vstupní soubor a přenechává plnou kontrolu Lua instanci. Instance Lua, běžící obvykle s několika načtenými knihovnami, vyhodnotí celý Lua blok kódu a výsledek pošle na speciální výstupní proud, k jehož obsahu se uživatel může dostat použitím příkazu `tex.print()`. Tato funkce funguje na stejném principu jako obyčejná funkce `print()` s jediným rozdílem a sice, že TeX-ový print posílý výstup TeX-ovému proudu místo standardnímu výstupu. Když Lua instance ukončí vyhodnocování skriptu, předá obsah texového proudu zpět do TeX-u a skript je použit na místě v dokumentu, kde byl zavolán. (Context Garden, 2015a).

4.4.6 Kontejnerový datový typ

Jazyk Lua má jediný kontejnerový datový typ pro ukládání dat a tím je tabulka. Víceméně se jedná o asociativní pole, které v sobě sdružuje páry klíčů a hodnot. V takovém páru lze ukládat hodnotu libovolného datového typu a později ji v případě potřeby získat pomocí unikátního klíče, která má také libovolný datový typ s výjimkou booleovských hodnot (Ierusalimschy, 2013).

Inicializace tabulky je prostá:

```
tabulka={}
```

Naplnění tabulky vypadá takto:

```
tabulka["klíč"] = 123
```

Klíčovým indexem je v tomto případě datový typ string a hodnota je obyčejné číslo, tedy integer. Pokud chceme zobrazit hodnotu v tabulce, jenodušše se do ní podíváme pomocí klíče.

```
print (tabulka["klíč"])
```

Tabulky jsou z optimalizačních důvodů vnitřně rozděleny na část hashovací a část datovou. Další optimalizací je správa paměti, Lua se v tabulkách snaží vyhnout častému alokování paměti.

4.4.7 Regulární výrazy

V programovacím jazyce Lua se nesetkáme s termínem regulární výraz (angl. regular expression) pro přiřazení vzorů. Hlavním důvodem je velikost implementace klasické POSIX regular expressions (jak je známe z jiných programovacích jazyků), která zabere skoro 4000 řádků kódu. To je více, nežli mají všechny Lua knihovny dohromady. Jazyk Lua má svoji obdobu, která se značně liší od způsobů, které jsou všeobecně známé u jiných jazyků. V konečném důsledku možná nemá tolik možností jako klasické regexp implementace, ale jednak je imlementačně zhruba o tři čtvrtiny menší a také si umí leckdy poradit s problémy, které jsou pomocí regexp neřešitelné.

Pro všechny situace, které jiný programový jazyk řeší regulárním výrazem, používá Lua velice mocné funkce z knihovny řetězců (zejména `find()`, `gfind()`, `gsub()` nebo `match()`) ve spojení s třídami znaků (angl. character classes) (Ierusalimschy, 2013). Jako názorný příklad uvádím použití funkce `sub()`, která v tomto případě vyměňuje všechna slova o délce jednoho písmena a více za slovo „něco“. Funkce vypadá takto:

```
print(string.gsub("jedna, a dva; a tři", "%a+", "něco"))
```

Výstupem je textový řetězec: `něco, něco něco; něco něco`

5 Syntaktický analyzátor

5.1 Současný stav

Výsledkem této diplomové práce je především programové dílo, řešící syntaktickou analýzu zdrojových textů systému ConTEXt a s ní spojené následné hledání běžných chyb. Protože je tato problematika, týkající se parsování TEX-ových zdrojových textů a kontroly syntaktických chyb, zřídkakdy řešená (souvísí to nejspíše s obecnou nepopularitou sázecích programům na bázi TEX-u), je k nalezení velice málo informací poskytujících správný teoretický základ této problematiky. I přes to a možná právě proto bylo nasnadě, pokusit se řešit daný probém v rámci diplomové práce. Problém tedy tkví v nemožnosti zkontrolovat chyby v ConteXtovém dokumentu před překladem. Zatím neexistuje nebo alespoň není veřejně dostupný nástroj, který by byl něčeho takového schopný.

5.2 Syntaktické analyzátory pro zdrojové texty na bázi TEX-u

Současný stav v oblasti syntaktické analýzy zdrojových textů na bázi TEX-u je jednodušše nijaký. V TEX-ovém archivu CTAN je k nalezení pouze jeden slušně použitelný nástroj pracující s dokumenty napsanými v LATEX-u. Jedná se o nástroj Lacheck. Lacheck je napsaný v programovacím jazyce C a výhodně tak využívá pro lexikální analýzu nástroj flex. Bohužel k němu neexistuje žádný manuál či alespoň informace k instalaci.

5.3 Návrh řešení

Předpokladem pro implementaci syntaktického analyzátoru je provedení tokenizace zdrojového textu, tedy lexikální analýzy, a na jejím základě vystavět syntaktickou analýzu. Z pohledu syntaktické analýzy se jako nejschůdnější jeví metoda „shora dolů“ díky její přehlednosti. Následně přichází na řadu výběr co nejjednodušší ana-

lýzy v ohledu k vytvářené gramatice jazyka ConTEXt. Jednoduché LL gramatiky, ani rozšířené Q-gramatiky neumožňují i přes jejich expresivitu pojmout podmínky zadání. Ukázalo se, že ideálním prostředkem pro provedení syntaktické analýzy zdrojového souboru ConTEXt je z důvodu svých vlastností a jednoduché implementace LL(1) analýza. S pužitím této analýzy souvisí výpočet funkcí FIRST a FOLLOW pro všechny neterminály gramatiky, vytvoření rozkladové tabulky a nutností je také zajištění bezkonfliktnosti v rozkladové tabulce použité gramatiky.

5.4 Implementace

Pro implementaci analyzátoru byl zvolen programovací jazyk LUA. Protože je celý ConTEXt napsán v tomto programovacím jazyce a LUA je také nedílnou součástí jeho instalace, byl tento výběr logický. Důležitou výhodou je, že se uživatel nemusí zabývat instalací dalšího softwaru a analyzátor je tak spustitelný na všech strojích, kde je nainstalovaný systém ConTEXt (nebo některá z distribucí, která ho obsahuje).

Během vývojové analýzy bylo bohužel zjištěno, že programovací jazyk Lua není ideálním nástrojem pro implementaci parserů a podobných záležitostí. Nedává totiž možnost použít již hotová řešení, sloužící k lexikální analýze. Jedním ze skvěle zvládnutých lexikálních analyzátorů je program flex. Flex je, přesněji řečeno, nástroj pro tvorbu tokenizérů, neboli scannerů. Bohužel generuje pouze kompilovatelný soubor v jazyce C. Dalším velmi dobře použitelným nástrojem je program Bison. Jedná se o přímý generátor parserů, který převede vstupní gramatiku na syntaktický analyzátor, který přijímá tokeny této gramatiky. Dohromady s programem flex vytváří prakticky již hotová řešení. Bohužel spolupracuje opět jen s jazykem C nebo C++.

Jazyk Lua má na na druhou stranu řadu vlastností, které implementaci parseru zpříjemňují. Disponuje například funkčně vcelku obsáhlou knihovnu pro práci s řetězci. Načítaní jednotlivých znaků ze vstupního řetězce se dá řešit efektivně například takto:

```
t = io.read("*all")
for i=0,string.len(t)-1 do
   i=i+1
   io.write(i,":",string.sub(t,i,i),"\n")
end
```

Tímto způsobem se načte celý vstupní soubor do jednoho řetězce a jednotlivé znaky se zpracovávají jednou z řetězcových funkcí.

5.4.1 Lexikální analýza

Prvním krokem k vytvoření parseru je provedení lexikální analýzy. ConTEXt-ový vstup je rozdělen na několik druhů lexikálních elementů, s nimiž se dále pracuje. Rozdělením, tedy tokenizací, se celý proces analýzy dosti zjednodušší tak, že z abstraktní syntaxe - z příkazů, závorek, atd. vytvoříme jeden token, který dále zpracujeme podle konkrétní gramatiky. Pro specifikaci vzorů jednotlivých symbolů se obecně výborně hodí regulární výrazy. Těmi bohužel programovací jazyk Lua nedisponuje, ale naštěstí je plně nahrazuje šikovnými funkcemi z třídy řetězců. Funkce, která načítá jednotlivé znaky k dalšímu zpracování vypadá takto:

```
function loadChar(howMany)
  positionInFile=positionInFile+1
  local nextChar=string.sub(t,positionInFile,positionInFile)
  lineColInfo = "("..lineOfInput..","..columnOfLine..")"
       if nextChar=="\n" then --konec radku
                     lineOfInput=lineOfInput+1 --pridej radek
                     columnOfLine=0 --vynuluj sloupce
       end
  if positionInFile~=lengthOfFile+1 then
    if nextChar=="\n" then
      return (nextChar)
    else
      columnOfLine=columnOfLine+1;
      return (nextChar)
    end
  else -- konec vstupu
    theEnd=true
  end
end
```

Na úrovni lexikální analýzy se rozlišuje, které znaky jsou pro analýzu důležité a které nikoliv. Obecně pro jakýkoliv (s výjimkou jazyka Whitespace) programovací jazyk (TEX-em a ConTEXt-em nevyjímaje) nemají podstatný význam bílé znaky, tedy tabelátory a mezery. V případě tohoto analyzátoru, jak si ukážeme později, to jsou také některé příkazy, jejichž obsah nás z pohledu kontroly nebude zajímat. Pokud lexikální analyzátor narazí na znak komentáře (%), který také patří mezi bezvýznamové znaky pro pozdější analýzu, je použita tato jednoduchá procedura.

```
function loadComment()
       while pchar ~="\n" and theEnd==false do
              pchar=loadChar(1)--nacita znak do konce radku
       end
end
```

Procedura načítá znaky do konce řádku. Podmínka `theEnd=false` zajišťuje případný konec souboru. Pokud se vysktytne případ, kdy je třeba načíst například mezera, je problém vyřešen přímým načtením znaku ze vstupního řetězce tímto způsobem: `string.sub(retezec,poziceVRetezci,poziceVRetezci)`. Tato funkce vrátí znak, který se nachází na určené pozici v řetězci.

Množina Σ pro bezkontextovou gramatiku vypadá takto:

- IDENTIFICATOR — jakýkoliv příkaz kromě níže specifikovaných
- ACCENT — akcentové příkazy
- KEYWORD_PRIMITIVE — TeXová primitiva
- KEYWORD_INPUT — příkaz input
- KEYWORD_DEF — příkaz def
- KEYWORD_DEFINE — příkaz define
- SQUAREBRACKETL — levá hranatá závorka
- SQUAREBRACKETR — pravá hranatá závork
- CURLYBRACKETL — levá složená závorka
- CURLYBRACKETR — pravá složená závorka
- MATHBLOCK — dvojitý znak dolaru
- MATHLINE — znak dolaru
- LETTER — jakékoliv písmeno
- DIGIT — číslice
- SPECIALCHAR — jakýkoliv speciální znak, vyjma bílých znaků a komentářů

Jednotlivé symboly charakterizují abecedu všech rozpoznávaných syntaktických struktur ConTEXt-u. Důležité je zdůraznit, že rozpoznání těchto symbolů postačuje k zachycení jen určité množiny problémů. Pokud bychom potřebovali obsáhnout větší část problému ConTEXt-u, množina symbolů by se musela výrazně rošířit. Lexikálním symbolem `IDENTIFICATOR` se genericky označuje jakýkoliv klíčový element, tedy znak či množina znaků začínající obráceným lomítkem. Pro některá konkrétní makra (input, def, define,...) analyzátor očekává netypickou akci, proto je pro ně vytvořen speciální lexikální symbol. Všechna klíčová slova by šla samozřejmě zachytit jen pomocí lex. elementu `IDENTIFICATOR` a řešení provádět na implementační úrovni (jak to také v některých případech je), ale tento postup vede k celkovému zpřehlednění gramatiky a napsaného kódu.

Kořenem lexikální analýzy je funkce `getToken()`. Pracuje tak, že postupně načítá znak po znaku ze vstupního řetězce (vstupní proud je jako celek uložen do jednoho řetězce) a vybírá pro něho příslušný lexikální symbol. Načítání znaku provádí další definovavá funkce `loadChar()`, v níž se také kontroluje, zda nedošlo ke čtení konce

souboru a také se zde aktualizují proměnné pro uchování aktuálního čteného řádku a sloupce vstupu. Část funkce, která ukazuje podmínku pro nalezení lexikálního symbolu IDENTIFICATOR vypadá takto:

```
function getToken()
       pchar=loadChar(1) --nacteni jednoho znaku
       while theEnd == false do -- true kdyz je konec souboru
    if pchar == [[\]] then --identifikator
      pchar=loadChar(1) --nacte dalsi znak nasledujici za lomitkem
                    loadIdent() -- nacti identifikator
      if lexIdent == "def" then return "KEYWORD_DEF" end
      if lexIdent == "define" then return "KEYWORD_DEFINE" end
      if lexIdent == "input" then return "KEYWORD_INPUT" end
      return "IDENTIFICATOR"
```

Protože za lexikálním elementem IDENTIFICATOR se skrývá celý název příkazu následujícím za lomítkem, je potřeba tento příkaz postupně načíst. O to se stará funkce `loadIdent()`. Funkce načítá jednotlivé znaky do té doby, než narazí na některý z ukončovacích symbolů. Zde je velice důležitý krok zpětného vrácení znaku při načítání vstupu. Funkce je totiž postavená na cyklu while, který probíhá až do nalezení konkrétního znaku. Problémem je tedy právě ten ukončovací znak. Kdyby se tento krok neprovedl, program by nebyl schopen zpracovat například dva po sobě jdoucí příkazy (například `\startttext\stoptext`). Zdrojový kód této funkce je následující:

```
function loadIdent()
  lexIdent="" -- inicializace (toto provadim uz ve funkci init)
  zavorkaVID = 0
  if string.match(pchar,'%A') and string.match(pchar,'%D') then
    lexIdent=lexIdent .. pchar return --vse osatni nez pismena a cislice
  end
  --znaky, kterymi muze koncit identifikator
  while pchar ~="[" and pchar ~="{" and pchar ~="}" and pchar ~="]"
  and pchar ~=" " and pchar ~=[[\]] and pchar ~=[[%]] and pchar ~=[[$]]
  and pchar ~=[[#]] and pchar ~="\n" and theEnd == false do --VRATIT ZNAK!
    if pchar == "(" then --kulatou zavorku beru jako soucast identifikatoru
      zavorkaVID = 1
    end
    if zavorkaVID==0 then
      lexIdent=lexIdent .. pchar --spojeni stringu
    end
    pchar=loadChar(1)
       end
  positionInFile=positionInFile-1 --vratim pozici v~souboru o~1 znak zpet
  return lexIdent --vracim poskladany identifikator
end
```

Podmínka uvedená hned z počátku funkce vrací jediný speciální znak. V implementaci jsem tyto speciální znaky zahrnul do množiny příkazů. Stejně tak by se na některé z nich dalo dívat jako na textové tokeny, prakticky na tom nezáleží. Výjimkou jsou řídicí znaky, které zpětné lomítko pro správné vysázení vyžadují. Jsou to například akcentové příkazy, jejichž syntaktická správnost je analyzátorem kontrolována. Správně provedená lexikální analýza dává předpoklad pro správné sestavení syntaktické analýzy a pro konstrukci syntaktického analyzátoru.

Pokud bychom vytvářeli lex. analýzu pro prakticky jakýkoliv moderní programovací jazyk a chtěli bychom obsáhnout jeho důležité prvky pro běžné programátorské operace, určitě bychom se neobešli bez velmi široké abecedy a také bez lineárních gramatik. V tomto případě se lineární gramatika jeví jako velmi jednoduchá a její konstrukce je pro budoucí fungování syntaktického analyzátoru zbytečná. Uvádím zde jen pro představu gramatiku pro jazyk Identifikátorů, jejíž sestrojení má smysl. Ta říká, že identifikátory mohou být složeny pouze z písmenných a číselných znaků, vyjma těch jednoznakových, ty mohou mít podobu také speciálního znaku.

$$G_I = \{N_I, \Sigma_I, P_I, S_I\}$$
$$\Sigma_I = \{letter, digit, specialchar\}$$
$$P_I : S_I \rightarrow letterE|letter|specialchar, E \rightarrow letterE|digitE|digit|letter$$
$$N_I = \{S_I, E\}$$

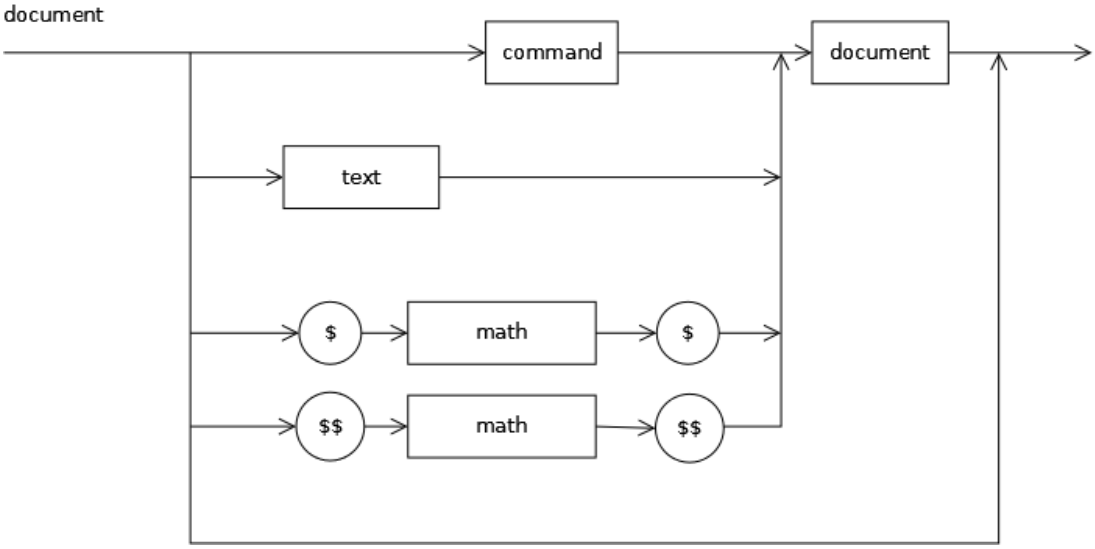
5.4.2 Syntaktická analýza

V syntaktické analýze jde o vytvoření derivačního stromu. Jak již bylo avizováno, v tomto případě se bude jednat o analýzu bez vracení do výchozí konfigurace, pro kterou je typické, že je v každý moment rozkladu věty známý jeden terminální symbol. Půjde tedy o analýzu LL(1).

Pokud bychom uvažovali o sepsání přesné a smysluplné gramatiky pro celý TEX, potažmo ConTEXt, dost tvrdě bychom narazili. TEX obsahuje jednak obrovské množství pravidel a definic, ale hlavním problémem je, že dokáže předefinovávat svoje makra, čímž se stává čitelným pouze pro kompletní Turingův stroj. Pro potřeby této diplomové práce byl problém velice zjednodušen a navržená gramatika zachycuje pouze probírané jevy, které je nutné zkontrolovat analyzátorem (seznam jevů je představen v kapitole 5.5). Množina terminálních symbolů tvořené gramatiky je rovna množině lexikálních symbolů (tak, jak byla představena v minulé kapitole). Množina neterminálních symbolů se skládá z těchto symbolů:

```
document (startovací symbol)
math
brackets
squareBrackets
curlyBrackets
squareBracketsOPT
squareBracketsContent
curlyBracketsContent
command
text
primitivesParam
inputParam
accentParam
letters
digits
repeatText
repeatDigits
```

K velice přehledné demonstraci syntaktických pravidel rozkladu věty poslouží diagramy syntaxe. Zde uvádím jako příklad diagram pro zpracování pravidel neterminálního symbolu `documet`, což je v tomto případě i startovací symbol gramatiky. Ostatní diagramy jsou uvedeny v příloze této práce. Jak lze vidět, analyzátor může dostat na vstup libovolný text, příkaz nebo matematickou rovnici. Tyto neterminální symboly se mohou nekonečně opakovat.



Obrázek 5.1 Diagram syntaxe pro pravidla neterminálního symbolu document

Intuitivním přepsáním grafu syntaxe vznikne bezkontextová gramatika. Název grafu

je levá část přepisovacího pravidla a pravou část tvoří jednotlivé terminální a neterminální symboly, které zapisujeme přesně tak, jak jimi procházíme pomocí šipek. Gramatika je zjednodušena přidáním symbolu roury pro variantu. Značení gramatiky je opačné ve srovnání s konvenčním značením. Neterminální symboly jsou označeny malými pismeny a terminální symboly velkými. Důvodem takového počínání je zavedená osobní konvence na implementační úrovni, kdy jsem zvyklý na značení funkcí malými písmeny a tokenů velkými. Gramatika byla optimalizována tak, aby byly splněny podmínky pro konstrukci LL(1) analyzátoru. To se víceméně podařilo, až na výjimky v podobě například načítání hranatých závorek. Vzniklé nedeterminismy ale byly beze zbytku úspěšně vyřešeny na programové implementační úrovni. Nežádoucí levá rekurze byla z gramatiky odstraněna už na úrovni tvorby diagramů syntaxe. Je důležité upozornit, že uvedená gramatika má smysl jakési kostry celé aplikace. Některé funkcionality jsou z jejího pohledu skryty a jsou řešeny až v implementovaných procedurách. Přepisovací pravidla pro neterminální symbol `document` se přepíší podle výše uvedeného grafu syntaxe takto:

```
document ->
document -> command document
document -> text document
document -> MATHLINE math MATHLINE document
document -> MATHBLOCK math MATHBLOCK document
```

Celá bezkontextotová gramatika je uvedena v příloze práce.

Pro vytvořenou gramatiku je také nutné vyjádřit množiny FIRST a FOLLOW, které dokazujín zda je možné procházet mezi konfiguracemi pod určitým tokenem pouze jednou cestou. V druhé řadě také výpočet těchto množin stanoví pro každé pravidlo konečnou množinu zásobníkových symbolů. Na základě získaných informací je možné zkonstruovat zásobníkový automat v podobě rozkladové tabulky. Cennými pomocníky se v této věci staly aplikace Desátomat (Dusíková, 2010) a aplikace RAB-JII (Habiballa, 2013). Výpočet obou množin spolu s rozkladovou tabulkou je uveden v příloze práce.

Všechny neterminální symboly jsou v programu implementované jako procedury, které jsou součástí LL(1) analýzy. Obsah procedury se rovná pravé straně pravidla gramatiky. V úvodu syntaktické analýzy jsou deklarovány potřebné proměnné a také je načten první token, který vstupuje do startovacího neterminálu gramatiky, v tomto případě do procedury `document`. Jako příklad zde uvádím funkci, která řeší korektní použití příkazů - TEX-ových primitivů. Ty mají povoleno zapsání v různých formách. Buď je můžeme zapsat takto:

```
\hbox {}
```

nebo takto:

```
\hbox to 10mm {}
```

klidně i tímto způsobem:

```
\hbox to .5\identifikator {}
```

a nebo ještě jinak:

```
\hbox to 5.\identifikator {}
```

Na ukázce lze vidět velkou různorodost zápisů jednoho příkazu. Takovou variabilitou jsou TEX-ové nástroje známé a někdy je velmi obtížné zachytit a přenést všechna pravidla do gramatiky nebo na implementační úroveň. Postup analyzátoru je následující, pokud v průběhu analýzy program narazí na lexikální element KEYWORD_PRIMITIVE, zavolá se funkce `command()` a následně funkce `primitivesParam()`, která řeší syntaktickou korektnost zápisu. Část funkce vypadá následovně:

```
function primitivesParam()--
  if currentToken == "CURLYBRACKETL" then
    currentToken=getToken()
    curlyBracketsContent()
    if currentToken == "CURLYBRACKETR" then
      currentToken=getToken()
    else
      if errorEnd==0 then
      print ("ERROR!",lineColInfo," :Missing } bracket")
      errorEnd=1
      end
    end
  elseif currentToken == "LETTER" then
    currentToken=getToken()
    if currentToken == "LETTER" then
      currentToken=getToken() --znak po "to"
      if currentToken == "DIGIT" then
        digits()
      elseif currentToken == "IDENTIFICATOR" then --muze byt take hned prikaz
        command()
        return
      else
        if errorEnd==0 then
          print ("ERROR!",lineColInfo," :Bad params of primitive command,
          expected ident, or {} or to digit dimensions{}")
              errorEnd=1
        end
      end
      if currentToken == "LETTER" then
        currentToken=getToken()
        if currentToken == "LETTER" then
          currentToken=getToken() --curlybracketl
          if currentToken == "CURLYBRACKETL" then
            currentToken=getToken()
            curlyBracketsContent()
            if currentToken == "CURLYBRACKETR" then
              currentToken=getToken()
            else
              print ("ERROR!",lineColInfo," :Missing } bracket")
              errorEnd=1
            end
```

Uvedená funkce je implementována pomocí rozkladové tabulky a postupuje přesně dle vytvořené gramatiky. Na nepřípustné hodnoty reaguje chybovým hlášením. V proměnné `lineColInfo` je uchována informace s aktuálním řádkem a sloupcem právě načítaného symbolu.

5.4.3 Reakce na chybu

K testovacím účelům byl připraven dokument ve formátu ConTEXt, který má za snahu obsáhnout všechny příkazy postihnutelné analyzátorem. Testování bylo vedeno stylem postupného zanesení chyby do jednotlivých příkazů dokumentu a následného vyhodnocení. Jak bylo vysvětleno v teoretické části, analyzátor může zpracovávat chyby dvěma různými způsoby. Implementovaný analyzátor reaguje na syntaktickou chybu tak, že při první nalezené chybě předá upozornění na standardní výstup. Protože je program implementován rekurzivně, často dochází po oznámení chyby k dalšímu průchodu neterminálem. Proto je někdy nahlášena i chyba související s první nalezenou. V některých situacích tento fakt zkonkretizuje danou chybu a pomůže tak k jejímu nalezení a odstranění.

Ukázalo se, že je tento způsob implementačně jednodušší a hlavně nemůže docházet k situacím, kdy jedna chyba přímo ovlivňuje několik dalších chyb. Nedochází tak k matoucím situacím, kdy uživatel vidí velké množství chyb, jednu opraví a rázem je vše v pořádku.

Po diagnostikování je chyba vypsána na standardní výstup následujícím způsobem:

```
ERROR! (řádek,sloupec) :popis chyby
```

5.5 Jevy postihnutelné kontrolorem syntaxe

Tato kapitola se zabývá všemi jevy, které jsou lokalizovány v implementovaném předkontroloru syntaxe. Uživatel je na všechny chyby upozorněn skrze textové chybové hlášení na standardním výstupu.

5.5.1 Korektní použití příkazů

Program využívá seznam příkazů, v kterém jsou uloženy veškeré informace o konkrétních povelech (počty parametrů, bloků, ukončení a podobně). Analyzátor je díky tomu schopen kontrolovat, zda uživatel zadal korektní počet složených a hranatých závorek k určitému klíčovému příkazu a také jeho párovou dvojici. Seznam povelů

je uložen v obyčejném textovém formátu (plain text) a tak je velice jednodušše rozšiřitelný o nové příkazy (zpravidla ty, které si uživatel sám dodefinuje). Data jsou zapsána v klasickém CSV formátu takto:

```
name;ending;curlyBrackets_b;squareBrackets_b;squareBrackets_opt;
squareBrackets_e;curlyBrackets_e
```

Jednotlivé proměnné mají následující význam:
- name — název příkazu;
- ending — určuje zda existuje i ukončovací příkaz. Proměnná může nabývat těchto hodnot: 0 - nemá ukončovací příkaz, 1 - ukončovací příkaz stop<něco>, 2 - ukončovací příkaz end<něco>, 3 - ukončovací příkaz e<něco>;
- ostatní proměnné označují počty složených a hranatých závorek;

Jedna konkrétní položka v diskutovaném seznamu vypadá takto:

```
starttext;1;0;0;0;0;0
```

5.5.1.1 Párovost příkazů

Analyzázor je schopný detektovat chybějící párový povel. U příkazu `\starttext` se kontroluje prezence `\stoptext`. Samozřejmostí je možnost přidání dalších uživatelsky určených dvojic ke kontrole (např `\bbib`, `\ebib`). V případě chybějícího párového povelu je vypsána chyba s názvem aktuálního povelu, jeho přibližným umístěním a názvem párového povelu. Problém vyvstane v případě, když potřebujeme kontrolovat obsah mezi párovými příkazy. Jde například o příkazy `\startlua` a `\startluacode`. Blok textu umístěný mezi těmito povely chceme ignorovat. Využijeme k tomu funkci, která funguje podobně jako funkce pro načítaní komentářů. Cyklus načítá nové tokeny, které zahazuje, dokud nenarazí na správný ukončovací příkaz. Funkce pro zpracování bloku `\startluacode` vypadá takto:

```
function loadLuaCode()
       while lexIdent ~="stopluacode" and theEnd==false do
              currentToken=getToken()
       end
end
```

5.5.2 Párovost závorek

V dokumentu je kontrolována párovost všech druhů závorek následujících za příkazy. Jedná se tedy o závorky kulaté, hranaté i složené. Hranaté závorky se používají na vymezení parametrů za konkrétními příkazy. například se podívejme na klíčové slovo `\chapter[]{}`. Jak vidno, klíčové slovo, sloužící pro vymezení kapitoly, využívá dvojí závorek. Hranaté závorky zde slouží k zapsání reference na danou kapitolu. Složené závorky v tomtéž slově vymezují prostor pro název kapitoly. Pro lepší představu uvedu povel využívající pouze složených závorek. Je jím například `\type{}`. Tento příkaz slouží k vysázení nějakého textu či znaku v uživateli přesně zapsané podobě. Například na zpětné lomítko už se nepohlíží jako na speciální znak.

Kulaté závorky se využívají skoro výhradně v povelu `\postition(a){b}`. Tento příkaz slouží k umístění textu na konkrétní pozici v dokumentu. Kulaté závorky určují souřadnice x,y a složené závorky obsahují text, který má být umístěn.

Pro lepší přehled a kontrolu nad zpracovávaným dokumentem nabídne předkontrolor v podobě parametru aktivování upozornění na vyskytující se závorky v textu. Důvodem je častá chybovost a selhání kompilace vycházející právě z důvodu špatného použití závorek. Kulaté i hranaté závorky lze v textu použít bez zpětného lomítka, ale je důležité kontrolovat jejich význam, typicky, jestli neleží za makrem a nejsou tak brány jako oddělovače parametrů nebo bloků. S hranatými závorkami se pojí jeden problém nerozhodnutelnosti analýzy, který nejde vyřešit na gramatické úrovni a je jedním z konfliktů v rozkladové tabulce. V případě, kdy se pravá hranatá závorka nachází za příkazem, který není v seznamu `keywords.txt` (tudíž analyzátor nezná jeho správné počty parametrů a bloků), automaticky se na něho pohlíží jako na začínající parametrový oddělovač. Analyzátor tudíž očekává i druhou uzavírací závorku. Ve skutečnosti ale může jít o bezparametrický příkaz a závorku v obyčejném textu. Situace kolem složených závorek je o poznání jednodušší, protože se v obyčejném textu značí výhradně s obráceným lomítkem. Složené závorky v následujícím textu se nevysází.

```
\CONTEXT\ je vynikající sázecí program, ale pozor na složené závorky {}
```

Tady již ano.

```
\CONTEXT\ je vynikající sázecí program, ale pozor na složené závorky \{\}.
```

5.5.3 Sazba matematiky

Systém TEX byl zprvopočátku určen právě pro sazbu matematických výrazů a i v dnešní době hledá těžko pro tuto disciplínu konkurenta. Matematický mód má v rodině TEX-u dvě alternativy. Vnitřní matematický mód se vyznačuje znakem dolaru `$...$` a display mode, který matematický výraz vysází na nový řádek označujeme takto: `$$...$$`. Případně lze pro display mód také použít povely `\startformulaa \stopformula`, které disponují drobnými odlišnostmi. Aby to nebylo tak jednoduché, tak také existuje příkaz `\placeformula`, který je umístěn před uvedenými příkazy. Tento příkaz zajišťuje mezery kolem matematického vzorce a také jeho očíslování. Nepovinně lze vzorci přiřadit také referenci pro budoucí odkazy. Analyzátor hledá zejména chyby v párovosti těchto symbolů. V matematickém režimu se můžeme setkat se situací, kdy jsou použity složené závorky ohraničující blok i jinde, nežli za příkazem (`\něco`). Jedná se o zápis horních a dolních indexů. Mocninu 100^2 lze korektně zapsat takto: `$100^{2}$`. V případě výskytu znaku dolaru ve vertikálním seznamu (tj. mimo text odstavce) bude uživatel upozorněn na použití dvojitého dolaru, případně povelu `\startformula`. Pro představu složitější matematické rovnice zde uvádím příklad. Zápis `$$1+\left(\frac{1}{1-x^{x-2}}\right)^3$$` bude vysázen takto:

$$1+\left(\frac{1}{1-x^{x-2}}\right)^3$$

5.5.4 Obrázky

Obrázky se do dokumentu ConTEXt-u vkládají pomocí příkazu `\placefigure[][]{}{}`. První hranaté závorky jsou nepovinné a starají se o umístění plovoucího objektu. ConTEXt zjistí, zda je na stránce pro obrázek dostatek místa a pokud ne, umístí jej na jiné vhodnější místo. V takovém případě pokračuje text a obrázek „proplouvá“, dokud pro něho není nalezeno místo. Parametr lze vybrat z několika přípustných variant, například `here` (pokud to jde, umístí obrázek přesně sem), `page` (umístí obrázek na vlastní stránku), atd. Druhý pár hranatých závorek určuje referenční jméno objektu. Na obrázek se tak lze v textu zpětně odkazovat. První složené závorky obsahují popisek obrázku. Lze použít libovolný text. Pokud nevyžadujeme žádný popisek, ani číslování, použijeme příkaz `none`. Nejdůležitější druhý pár závorek je určen pro vložení vlastního obrázku. Nejčastěji se používá příkaz pro vložení externího souboru

`\externalfigure`. Zde je příklad zápisu pro vložení obrázku.

```
\placefigure
[][obr:mujobrazek1]
{Toto je popisek obrázku}
{\externalfigure[presnynazevobrazku][width=.4\textwidth]}
```

Po zpracování příkazu se na dostupné místo vysází obrázek takto:

Obrázek 5.2 Toto je popisek obrázku

5.5.5 Tabulky

Tabulky jsou v ConTEXt-u vysázeny pomocí párových povelů `\startttable[]` a `\stoptable`. Před vymezením tabulky obvykle bývá příkaz `\placetable[a][b]{c}`, který se stará o vertikální zarovnání a číslování tabulky, stejně jako je tomu u příkazů, starající se o matematické vzorce nebo obrázky. Do hranatých závorek v příkazu `\starttable` se vkládají znaky pro formátování tabulky.

Tabulka 5.1 Značky pro formátování tabulky, zdroj: Otten a Hagen (2006, s.95)

Značka	Význam
|	oddělovač sloupců
c	sloupec se zarovnáním na střed
l	sloupec se zarovnáním vlevo
r	sloupec se zarovnáním vpravo
s<n>	mezisloupcová mezera o hodnotě n
w<rozměr>	minimální šířka sloupce o dané hodnotě

Dále existují formátovací příkazy přímo pro jednotlivé řádky, sloupce a buňky. Ty se zapisují přímo mezi povely, vymezující tabulku. Seznam alespoň těch základních je zde:

Tabulka 5.2 Značky pro formátování buňky, zdroj: Otten a Hagen (2006, s.96)

Příkaz	Význam
`\NR`	udělej řádku bez vertikálního vyrovnání (další řádka)
`\FR`	udělej řádku s horním vyrovnáním (první řádka)
`\LR`	udělej řádku s dolním vyrovnáním (poslední čárka)
`\MR`	udělej řádku s horním i dolním vyrovnáním (prostřední čárka)
`\SR`	udělej řádku s horním i dolním vyrovnáním (oddělovací čárka)
`\VL`	minimální šířka sloupce o dané hodnotě (svislá linka)

U povelů `\bTABLE`, `\starttable` a `\starttabulate` lze kontrolovat shodu počtu sloupců v datech tabulky a v definici tabulky.

Nakonec zde uvádím příklad tabulky vysázené s použitím právě popsaných pravidel.

```
\starttable[|c|c|]
\HL
\VL \bf Hráč \VL \bf Góly \VL\SR
\HL
\VL Jágr \VL 35 \VL\FR
\VL Eliáš \VL 20 \VL\MR
\VL Voráček \VL 32 \VL\MR
\VL Plekanec \VL 16 \VL\LR
\HL
\stoptable
```

Tabulka se vysází tímto způspobem:

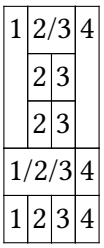
Hráč	**Góly**
Jágr	35
Eliáš	20
Voráček	32
Plekanec	16

5.5.5.1 Natural tables

Jedná se o alternativní sazbu tabulek, která je svou syntaxí velice podobná HTML. Tento druh sazby je velice vhodný pro tabulky s XML, s barevným pozadím nebo s nepravidelnou velikostí buněk. Princip vysázení tabulky spočívá v párových symbolech. Celá tabulka je vymezena povely `\btable` a `\etable`. Řádek se značí povely `\bTR` a `\eTR` a sloupec příkazy `\bTD` a `\eTD`.

```
\bTABLE
\bTR \bTD[nr=3] 1 \eTD \bTD[nc=2] 2/3     \eTD \bTD[nr=3] 4 \eTD \eTR
\bTR                   \bTD 2 \eTD \bTD 3 \eTD                   \eTR
\bTR                   \bTD 2 \eTD \bTD 3 \eTD                   \eTR
\bTR \bTD[nc=3] 1/2/3                     \eTD \bTD 4       \eTD \eTR
\bTR \bTD 1       \eTD \bTD 2 \eTD \bTD 3 \eTD \bTD 4       \eTD \eTR
\eTABLE
```

Tabulka vypadá po překladu takto:



5.5.6 TEX-ová primitiva

TEX-ová primitiva typu `\hbox`, `\vbox` a `\vtop` jsou následovány buď skupinou {}, nebo boxovou specifikací `to<číslo><jednotka>{}`. ConTEXt k těmto již zavedeným primitivům přidává další příkazy `\lbox`, `\cbox`, `\rbox`, `\sbox` a `\bbox` se stejnou syntaxí.

5.5.7 TEX-ové akcenty

Ve správném použití typografických akcentů tkví pravá výhoda TEX-ových programů. Dokáží totiž vysázet i taková znaménka, které bychom marně hledali na běžné klávesnici nebo by nám to při nejmenším trvalo dlouhou dobu. Pro připojení akcentu k písmennému znaku se jednodušše používá řídící sekvence.

Francouzské slovo Noël se korektně zapíše jako `No\"el` nebo `No\"{e}l`. Analyzátor kontroluje, zda se za akcentovými povely nachází přípustný písmenný znak nebo posloupnost písmenných znaků. Povolen je ještě blok složených závorek, na vše ostatní odpovídá analyzátor chybovým hlášením.

5.5.8 Vkládání externích zdrojů

Někdy je potřeba do zdrojového textu vložit jiný TEX-ový soubor a občas může být také velice výhodné rozdělit text psaný v ConTEXt-u do více souborů, které jsou zpracovány samostatně. Přesně k tomuto účelu slouží symbol `\inputsoubor`. Nutno podotknout, že vkládaný soubor nemusí být pouze s koncovkou `.tex`, ale může se jednat například o soubor definic (obyvykle s koncovkami `.mkii, mkiv`, atd.) nebo

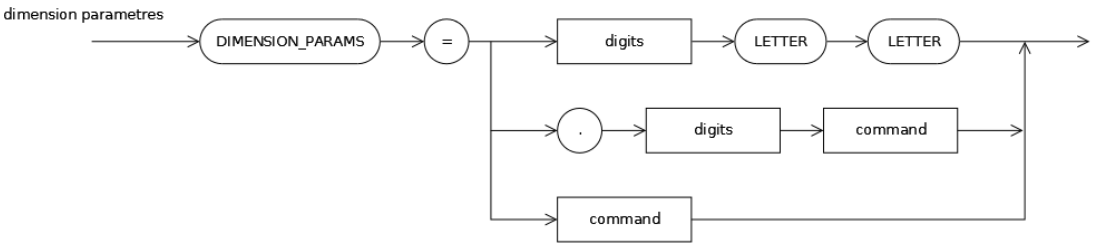
stylový soubor (`.sty`). Při kontrole se prověřuje, zda je externí soubor umístěn ve výchozí složce a zda je připraven ke čtení. Uživatel má také možnost výběru, jak se k externímu souboru zachovat. Soubor lze kontrolovat analyzátorem či ignorovat. Tato funkčnost je řešena spuštěním programu s přidaným argumentem. Aktuálně je program schopný vyřešit pouze jedno vnoření. Dokáže zpracovat pouze soubor vložený v aktuálním souboru. Provádí to takovým způsobem, že si v první řadě uloží všechny potřebné proměnné první analýzy, poté přesměruje standardní vstup na includovaný dokument, zavolá proceduru pro analýzu a na konec načte zpět proměnné a pokračuje v první analýze.

5.5.9 Formátování stránky a prostorové parametry

Jedná se o parametrické příkazy typu `margin`, `offset`, `width`, `height`, `distance`, `depth`, případně `dx` a `dy` u povelu `\setuplayout`. Tyto parametry jsou používané například v příkazu na vysázení obrázku či příkazu popisujícího celý formát stránky. Parametr může být zapsán několika způsoby.

- délkovým údajem — `margin=5mm` (jednotky jsou vždy dvoupísmenné);
- délkovým údajem zapsaným poměrově — `width=.4\textwidth`;
- délkovým údajem a identifikátorem — `\textwidth`;

Gramatika pro tento typ problému nebyla zahrnuta do rámcové gramatiky. Parametry jsou kontrolovány v rámci netermiálu `squareBracketsContent`, který spravuje vnitřek hranatých závorek. Pravidla všech tří druhů zapsání parametru ukazuje tento tento syntaktický diagram:



Obrázek 5.3 Syntaktický diagram pro prostorové parametry

Speciální znaky „.“ a „=“ nemají svůj lexikální element. Existuje jen souhrnný element pro všechny speciální znaky, k jehož hodnotě se dá ovšem dostat skrze proměnnou `pchar`.

5.5.10 Prostředí typing

V ConTEXt-u existuje prostředí, kde je každý znak zpracováván ve své zapsané podobě. V LATEX-u se toto prostředí jmenuje Verbatim a také v ConTEXt-u se tento výraz občas používá.

Pokud chceme vysázet větší celek textu, je na místě použít povely `\starttyping` a `\stoptyping`, zatímco pro jednotlivý příkaz umístěný v textu se lépe hodí povel `\type{}`. Důležité je pečlivě kontrolovat obsah zapsaného textu a to zejména pokud text obsahuje složené závorky. Budeme-li mít například příkaz s lichým počtem složených závorek, tak se výsledek vysází špatně.

```
\type{text se závorkou {}.
```

V tomto případě se pro správný výstup musí jako oddělovač použít například znaky roury nebo lomítka.

```
\type|text se závorkou {|
```

Blok mezi příkazy jsem se rozhodl ignorovat stejně, jako je tomu u příkazů `\startlua` a `\startluacode`.

5.5.11 Uživatelské definice

Protože je systém ConTEXt typografickým systémem a zároveň také programovacím jazykem, uživateli ConTEXt-u je umožněno definování vlastních příkazů pro usnadnění práce a zvýšení flexibility dokumentu. Názvy příkazů mohou kolidovat s již existujícímu příkazy TEX-u a ConTEXt-u, proto je důrazně doporučováno používat v názvu velkých písmen. Zde je příklad uživateli nadefinovaného příkazu.

```
\def\MojeKapitola#1
{\kapitola{#1}\index{#1}}
```

Příkaz vytvoří novou kapitolu s indexovým vstupem se stejným jménem. V těle příkazu, tedy ve složených závorkách, může být až 9 dalších příkazů. Definovaný

příkaz může být také bezparametrický. V takovém případě se nemusí psát mřížka s počtem parametrů.

K definování nových funkcionalit se v ConTEXt-u využívá také povel `\define` s mírně odlišnou syntaxí.

```
\define[3]\mujobrazek
{\placefigure[here,force][fig:#1]{#2}{\externalfigure[#3][width=5cm]}}
```

V nepovinných hranatých závorkách je počet proměnných, používaných při volání příkazu. Následuje název definice a ve složených závorkách stanovujeme tělo definice. V tomto případě se jedná o obrázek s nastevenými parametry. Nadefinovaný příkaz se poté použije v dokumentu tímto způsobem:

`\mujobrazek{obrázek}{popisek}{umístění}`

Uživatelem definované povely se uloží do připravené tabulky klíčových příkazů a tím pádem se v případě jejich využití v dokumentu kontroluje správný počet parametrů, případně prezence párového povelu. Uživatel má možnost při spuštění programu aktivovat upozornění na právě přidané příkazy do tabulky.

5.6 Spuštění analyzátoru

Analyzátor byl zprvopočátku koncipován jako jednoduchý skript jazyka Lua, spustitelný z příkazové řádky. Uživateli stačí ke spuštění volně dostupná distribuce jazyka Lua (aktuálně dostupná ve verzi 5.3) nebo některá z distribucí sázecího programu ConTEXt, jehož je Lua součástí. Příkaz je spustitelný se třemi volitelnými parametry. Parametr `-i` umožňuje kontrolovat includované soubory uvnitř ConTEXt-ového souboru. Druhým parametrem `-b` lze aktivovat upozornění na vyskytující se hranaté a kulaté závorky v obyčejném textu. Třetím parametrem `-d` se deaktivují upozornění na uložení nadefinovaných příkazů do souboru `keywords.txt`. Program je i v případě nenalezených chyb ukončen upozorněním o době trvání analýzy. Volání příkazu se všemi parametry vypadá takto:

```
lua consyncheck.lua soubor.tex -b -i -d
```

Pokud jsou korekně nastavena práva, tak také takto:

```
./consyncheck.lua soubor.tex -b -i -d
```

Nutné je sjednotit umístění všech potřebných souborů ideálně do téhož adresáře. V jednom adresáři by měly být tyto soubory:

- consyncheck.lua — hlavní program - analyzátor;
- soubor.tex — kontrolovaný soubor;
- keywords.txt — textový soubor s příkazy;
- případně další volitelné soubory, které jsou includovány v soubor.tex;

Syntaktický analyzátor je včetně textového souboru s příkazy a testovacího souboru formátu TEX umístěn v příloze diplomové práce na optickém disku.

6 Diskuse

Pokud chceme zkvalitnit svůj textový výstup, vytvořený pomocí počítače nebo požadujeme přesně a precizně vysázet matematické rovnice, určitě se vyplatí prozkoumat možnosti programů pro počítačovou sazbu na bázi TEX-u. TEX-ové programy již dávno nejsou populární jen v akademických kruzích a zejména TEX-ové nadstavby či balíky maker se hodí i pro širokou veřejnost, nutno podotknout technicky gramotných. Jedním z rychle se rozvíjejících balíků maker je také systém ConTEXt, který ještě navíc nabízí několik výhod oproti konkurenci.

S datem dopsání této práce na Internetu neexistuje jediný nástroj, který by byl schopen na slušné úrovni kontrolovat syntaxi zdrojového textu formátu ConTEXt (vyjma oficiálního překladače). Uživatel má možnost některé vybrané jevy zkontrolovat programem lacheck, ale ten je primárně určen pro LATEX, takže logicky nepodporuje některé ConTEXt-ové příkazy nebo některé příkazy vůbec nediagnostikuje. Pro ConTEXt tedy není ideálním nástrojem.

Další možností, která se nabízí, je nechat zkontrolovat zdrojový text TEX-ovým překladačem, který samozřejmě objeví všechny syntaktické nesrovnalosti. Problémem je ale jeho vcelku vysoká časová náročnost, především při kontrole delších textů. Není neobvyklé čekat na překlad několik desítek sekund a nakonec zjistit, že v předposlední větě chybí jedna složená závorka. Častý uživatel ConTEX-u zkrátka postrádá nástroj, který by zvládl najít běžné chyby v krátkém časovém intervalu. Implementovaný analyzátor v rámci této práce by se měl pokusit tuto mezeru zaplnit a nabídnout rychlou kontrolu zdrojového textu před samotným překladem.

Cílem diplomové práce bylo popsat průběh sestavení takového syntaktického analyzátoru. Konstrukce je v první řadě spojena především s navržením lexikálního analyzátoru, který je schopen roztřídit a identifikovat jednotivé znaky a slova, přicházející ze vstupního proudu, tedy obvykle analyzovaného dokumentu. Vybrané lexémy tvoří abecedu jazyka, jejichž slova jsou vstupem pro syntaktickou analýzu. V průběhu lexikální analýzy program ignoruje bílé znaky a komentáře a také ukládá infomace o aktuálním procházeném řádku a sloupci.

S dalším postupem bylo třeba přijmout několik důležitých rozhodnutí. Syntaktická analýza se dá vytvořit dvěma způsoby. Metoda „zdola nahoru“ není tak intuitivní, proto byla vybrána k řešení daného problému metoda „shora dolů“, která je vhodnější pro ruční implementaci. Důvodem je také mimo jiné implementace pomocí programovacího jazyka Lua, který v podstatě nedává možnost použít již hotové nástroje jako yacc nebo bison. Derivační strom byl tedy rozvíjen od svého startovacího symbolu po poslední větve tvořené terminály. Správný průchod stromem, tedy syntaktickou korektnost, definuje navržená bezkontextová gramatika, která byla sestavena pomocí diagramů syntaxe. Ty tvoří ucelený a jasný pohled na průchod ana-

lyzátoru. Analýzu „shora dolů“ lze řešit metodou pokusu a omylu, kdy se v každém kroku analýzy aplikují pravidla gramatiky až do té doby, nežli je aplikace některého pravidla neúspěšná. V takovém případě se pak volí návrat v derivačním stromu do bodu, z kterého se pokračuje jinou cestou. Tento typ analýzy je v některých případech velmi pomalý, proto je lepší zajistit determinističnost gramatiky a použít nějaký jednoznačný postup.

V ohledu ke zjednodušení konstrukce parseru byla pro implementaci zvolena LL(1) analýza. Gramatika byla transformována tak, aby splňovala podmínky pro LL(1) analýzu a umožňovala tak výstavbu deterministické rozkladové tabulky. To se podařilo. Některé buňky, která hlásily kolizi, tak byly ponechány a problém se vyřešil až dodatečnými podmínkami přímo v implementovaných procedurách programu. V případě, že dojde k nesprávnému sestavení věty nebo načtené slovo není povoleno lex. analýzou, dochází k chybě. Uživatel je na tuto skutečnost upozorněn vypsaným chybovým stavem na obrazovce, doplněným o číslo řádku a sloupce a také popisem chyby. Reakce na chybu může být dvojího typu (jak je uvedeno v kapitole 3.6.5). Pro tento případ se ukazálo výhodnějším vypisovat chyby jednotlivě a čekat na jejich postupnou opravu. V některých případech je chyb nahlášeno více z důvodu lepší přehlednosti.

Zásadní zprávou je, že zhotovený analyzátor je velmi rychlý a běžný text dokáže zkontrolovat za několik setin či desetin sekundy, což je výrazný rozdíl oproti oficiálnímu překladači. Testovací soubor (všechny testovací soubory jsou uloženy na optickém disku) o tisíci řádcích bez syntaktických chyb je pomocí implementovaného analyzátoru zkontrolován za 0.450 sekundy. TEX-ový překladač tento stejný soubor zvládne zpracovat za více jak 5 sekund. Dvojnásobně velký soubor je analyzátorem zkontrolován za 0.91 sekundy, přičemž TEX-ový překladač soubor zpracuje za 10 sekund. Takových testů bylo provedeno ještě několik s různě velkými vstupními soubory. Ve výsledku lze konstatovat, že platí přibližně přímá úměra mezi velikostí souboru a stráveným časem a implementovaný analyzátor je zhruba desetkrát rychlejším nástrojem ve srovnání s TEX-ovým překladačem. Uživatel je na čas strávený analýzou upozorněn na konci běhu programu.

Implementaci analyzátoru se samozřejmě nevyhnuly ani některé problémy. První vcelku zásadní problém souvisí s nerozhodnutelností v rozkladové tabulce, související s lexikálními elementy SQUAREBRACKETL a SQUAREBRACKETR, jejichž obsahem jsou hranaté závorky. V ConTEXt-u jsou hranaté závorky používány jako oddělovače parametrů příkazů a nebo je lze brát jako součást obyčejného textu. Problém je, že pokud se hranaté závorky nacházejí přímo za příkazem, je těžké rozhodnout, jakou mají funkci. Tento problém je částečně vyřešen nasazením textového souboru, který uchovává informace o některých příkazech, ale bohužel není logicky schopen pojmout všechny TEX-ové a ConTEXt-ové příkazy.

Podobný problém je spjat také se znakem složené závorky. Její zpracování se ukázalo být ještě složitější. Nejdříve byl předpoklad takový, že se složená závorka může vyskytovat pouze za příkazem, jakožto oddělovač bloku. V normálním textu se dá zapisovat pomocí zpětného lomítka, umístěného před závorkou (viz. kapitola 4.2.5).

Jenže složená závorka se může nacházet také před příkazem. Jako příklad uvedu příkaz `\it`, jež vysázi text v bloku italikou (kurzívou). Problémem je, že se korektně zapisuje takto: `{\it\ Text napsaný kurzívou}`. Pokud by tedy měl analyzátor upozorňovat na špatně vloženou složenou závorku v obyčejném textu, logicky tak upozorní i na tu s blokovým významem, což samozřejmě není dobře. Tento typ problému by se dal výhodně řešit pomocí LL(k) analýzy (viz. kapitola 3.6.2.6). Počet dopředných symbolů by se pro tento konkrétní případ musel rozšířit o tři, aby došlo k přečtení příkazu následujícího za závorkou a ke správnému rozhodnutí, kam postupovat v syntaktickém stromu. Tím by se celá analýza stala o hodně složitější (je nutné jiným způsobem vyjádřit množiny FIRST a FOLLOW, jiným způsobem se tvoří rozkladová tabulka, …). Navíc by tento postup stále nezaručoval úplný úspěch. Za složenou závorkou by se totiž mohl nacházet takový příkaz, který tuto strukturu zápisu nepodporuje.

Jiným nedostatkem je provedená kontrola pouze pro prostorové parametry příkazů na vysázení obrázku, formátování stránky atp. Jedná se o parametry `width`, `height` a jim podobné. Množina těchto kontrolovaných parametrů by se dala jistě s výhodou rozšířit, aniž by to nějak výrazně poznamenalo rychlost analýzy.

Jako další příklad uvádím nemožnost zkontrolovat vložený soubor pomocí příkazu `include` v již vloženém jiném souboru. Analyzátor podporuje pouze jednoduché vložení, niloliv vícenásobné. S tímto typem chyby se ale uživatel v praxi skoro nesetká.

Kromě výše popsaných chyb již žádná skutečnost nevlivňuje negativním způsobem průběh analýzy. Analyzátor je tak pro běžného uživatele ConTEXt-u využitelný.

Vytvořená gramatika pokrývá jen vybrané jevy a syntaktické celky ConTEXt-u (viz. kapitola 5.5). Určitě by stálo za zvážení vylepšit analýzu ve smyslu kontroly zadávání vstupních parametrů u více příkazů. Nyní jsou kontrolovány jen počty parametrů a párovost jejich oddělovačů, kontrola obsahu je realizována jen u některých konkrétních povelů. Jistě by bylo možné celý proces analýzy výrazně zesložitit. Aktuální implementace je ale podle mého názoru rozumným kompromisem. Pokud by totiž byla analýza o hodně podrobnější, dalo by se polemizovat o tom, zda již není výhodnější použít obyčejný, byť o dost pomalejší, ConTEXt-ový překladač.

Budoucí možnosti vylepšení bych viděl zejména v opravení okomentovaných nedostatků, protože hlavně problémy závorek nyní velmi ovlivňují celou analýzu.

Syntaktický analyzátor by se dal spojit s dalšími užitečnými nástroji pro zpracování TEX-ových dokumentů. Například sloučením s nástrojem, jež detekuje chybné typografické jevy v textu, by vzniklo velmi užitečné a komplexní dílo pro běžného uživatele ConTEXt-u.

Obrovskou vizí do budoucna by mohlo být nasazení analyzátoru do nějakého, již fungujícího textového editoru. Před uložením ConTEXt-ového souboru by se nad editorem spustil analyzátor a informoval uživatele o chybách. Následující samotný překlad by pak probíhal s téměř stoprocentní jistotou úspěchu. Tato vize ovšem předpokládá naprosto bez výhrad optimalizovaný analyzátor bez nedostatků.

7 Závěr

Hned v úvodu, když jsem si zvolil téma této diplomové práce, jsem věděl, že to pro mě bude nelehká výzva. Z teorie formálních jazyků jsem sice měl některé teoretické základy díky předmětům vyučovaným na Mendelově univerzitě v Brně, svůj vlastní překladač nebo parser jsem však nikdy před tím neprogramoval. Navíc jsem do té doby vůbec neznal programovací jazyk Lua, s jehož syntaxí jsem se musel seznámit. Na druhou stranu se mi líbila myšlenka, že zkusím navrhnout a naprogramovat nástroj, o jehož vytvoření se zatím možná nikdo jiný nikdy nepokoušel. Typografický systém ConTEXt jsem již okrajově znal z mojí Bakalářské práce. I když nejsem velký fanoušek TEX-ové rodiny, zajímalo mě jak pracuje z trochu širšího úhlu pohledu.

Cílem diplomové práce bylo implementovat syntaktický analyzátor zdrojových textů ve formátu ConTEXt. K dosažení cíle bylo nutné objasnit základní termíny z teorie formálních jazyků, pochopit, jakým způsobem pracuje překladač a jaké různé postupy se používají k provedení syntaktické analýzy. Dále bylo nutné rozebrat fungování systému ConTEXt a to zejména z pohledu syntaxe jednotlivých příkazů a parametrů. Získané informace byly převedeny do diagramů syntaxe a následně do pravidel bezkontextové gramatiky. K úspěšnému sestavení gramatiky bylo potřeba objasnit, jakým způsobem se bude provádět syntaktická analýza. Zvolena byla analýza typu LL(1), kterou lze dobře implementovat metodou rekurzivního sestupu. Návrh gramatiky a diagramů, které musely splňovat určité podmínky pro fungování analyzátoru, byl pro mě velmi zajímavou a přínosnou zkušeností. Ikdyž jsem se ze začátku v návrhu dělal velké množství chyb, nakonec vznikla (i za pomocí aplikací programů RABJII a Desátomat) funkční gramatika, podle které jsem mohl implementovat analyzátor. Určitě stojí za zmínku, že zhotovený analyzátor má také možnost kontrolovat některé „nesyntaktické“ jevy, například upozorňuje na vyskytující se závorky v obyčejném textu, což může mít kladný dopad na vyhledávání chyb a celkově to může vést ke zpříjemnění práce.

Přiložený optický disk dává možnost ověřit si funkčnost syntaktického analyzátoru. Disk obsahuje, kromě samotného analyzátoru, pojmenovaného consyncheck.lua, také testovací soubory, soubor s příkazy ConTEXt-u, syntaktické diagramy a rozepsanou gramatiku se všemi dalšími náležitostmi.

Na závěr lze konstatovat, že cíl diplomové práce se podařilo splnit. Úspěšně byl implementován syntaktický analyzátor pro dokumenty napsané v jazyce ConTEXt. Nástroj disponuje textovým rozhraním a spouštěn je z příkazového řádku i s několika nepovinnými parametry. Protože se nejedná o kompletní rozebrání zdrojového textu, jak je tomu u TEX-ového překladače, analýza je z časového hlediska velmi nenáročná.

8 Literatura

Beneš, Miroslav; Kolář, Dušan. *Syntaktická analýza* [on-line]. 2015. [cit. 2015-02-18]. Dostupné na: www.cs.vsb.cz/behalek/vyuka/udp/prednasky/05-parser-4.pdf.

Compiler [on-line]. 2005. [cit. 2015-03-01]. Dostupné na: http://www.compilers.net/paedia/compiler/index.htm.

ConTeXt Garden. *Programming in LuaTeX* [on-line]. 2015a. [cit. 2015-04-15]. Dostupné na: http://wiki.contextgarden.net/Programming_in_LuaTeX.

ConTeXt Garden. *About LUA* [on-line]. 2015. [cit. 2015-04-15]. Dostupné na: http://wiki.contextgarden.net.

Černá, Ivana; Křetínský, Mojmír; Kučera, Antonín. *Automaty a formální jazyky I* [on-line]. 2002. [cit. 2015-03-11]. Dostupné na: http://is.muni.cz/el-portal/estud/fi/js06/ib005/Formalni_jazyky_a_automaty_I.pdf.

Češka, Milan. *Gramatiky a jazyky* [on-line]. 1992. [cit. 2015-04-10]. Dostupné na: http://kifri.fri.uniza.sk/ bene/vyuka/kompilatory/pomocne-materialy/Gramatiky%20a%20jazyky-Ceska.3.pdf.

Češka, Milan. *Teoretická informatika - přednášky k předmětu* [on-line]. 2011. [cit. 2015
-03-22]. Dostupné na: http://www.fit.vutbr.cz/study/courses/TIN/public/Prednasky/tin-pr04-bj1.pdf.

Češka, Milan; Hruška, Tomáš; Beneš, Miroslav. *Překladače* [on-line]. 2015. [cit. 2015
-04-22]. Dostupné na: http://www.fit.vutbr.cz/ meduna/fjp/skripta.pdf.

Doob, Michael. *Jemný úvod do TeXu* [on-line]. 1993. [cit. 2015-03-22]. Dostupné na: https://ksp.mff.cuni.cz/encyklopedie/jemny-uvod-do-TeXu.pdf.

Dusíková, Hana. *Desátomat.* [on-line]. (Diplomová práce.) Brno : Mendelova zemědělská a lesnická univerzita, 2011. [cit. 2015-03-21]. 58 s. Dostupné na: https://is.mendelu.cz/auth/lide/clovek.pl?id=24975;zalozka=7;studium=48281;zp=28980.

FontForge. *FontForge - oficiální stránky* [on-line]. 2015. [cit. 2015-03-22]. Dostupné na: http://fontforge.github.io/en-US/.

Habiballa, Hashim. *Teoretické základy informatiky II* [on-line]. 2003. [cit. 2015-03-16]. Dostupné na: http://www1.osu.cz/home/habibal/kurzy/ytzi2.pdf.

Habiballa, Hashim. *Regulární a bezkontextové jazyky II* [on-line]. 2005a. [cit. 2015-03-16]. Dostupné na: http://www1.osu.cz/home/habibal/publ/rabj2.pdf.

Habiballa, Hashim. *Překladače* [on-line]. 2005. [cit. 2015-03-16]. Dostupné na: www1.osu.cz/home/habibal/kurzy/xprek.pdf.

Habiballa, Hashim. *Gramatiky a jazyky* [on-line]. 2013. [cit. 2015-03-16]. Dostupné na: http://www1.osu.cz/home/habibal/kurzy/GRAJA.pdf.

Hagen, Hans. *ConTeXt the manual* [on-line]. 2000. [cit. 2015-03-21]. Dostupné na: www.ctex.org/documents/context/cont-enp.pdf.

Hagen, Hans. *METAFUN* [on-line]. 2010. [cit. 2015-03-21]. Dostupné na: http://pragma-ade.com/general/manuals/metafun-s.pdf.

Hassman, Martin. František Fuka: 95 procent všeho dělám v jazyce Lua [on-line]. *Zdroják*, 2009, 4. června 2009 [cit. 2015-04-15]. Dostupné na: http://www.zdrojak.cz/clanky/frantisek-fuka-95-procent-vseho-delam-v-jazyce-lua/.

Hoekwater, Taco. *Comparing ConTeXt and LaTeX* [on-line]. 2015. [cit. 2015-03-21]. Dostupné na: www.ntg.nl/maps/20/42.pdf.

Ierusalimschy, Roberto. *Programming in Lua.* 3. vyd. Rio de Janeiro : Lua.org, 2013. 348 s. ISBN 978-85-903798-5-0.

Jančar, Petr. *Studijní opora k předmětu Teoretická informatika* [on-line]. 2003. [cit. 2015-02-09]. Dostupné na: http://www.cs.vsb.cz/jancar/TJAA/tjaa_2p.pdf.

Kollmanová, Lada. *Využití lexikální analýzy pro interpretaci dokumentů LaTeX* [on-line]. (Diplomová práce.) Brno : Mendelova zemědělská a lesnická univerzita, 2013. [cit. 2015-04-06]. 88 s. Dostupné na: https://is.mendelu.cz/auth/lide/clovek.pl?id=6;zalozka=13;studium=41817.

Kot, Martin. *Animace k předmětům teoretické informatiky* [on-line]. 2015. [cit. 2015-03-20]. Dostupné na: http://www.cs.vsb.cz/kot/animace.php.

MiKTeX. *MiKTeX homepage* [on-line]. 2015. [cit. 2015-04-15]. Dostupné na: http://www.miktex.org/.

Natural tables in ConTeXt [on-line]. 2015. [cit. 2015-03-22]. Dostupné na: www.pragma-ade.com/general/manuals/enattab.pdf.

Otten, Ton; Hagen, Hans. *Exkurze do ConTeXtu* [on-line]. 2006. [cit. 2015-03-22]. Dostupné na: http://bulletin.cstug.cz/pdf/ma-cb-cz-print-bw.pdf.

Peterka, Jiří. *Compiler vs. interpreter* [on-line]. 2011. [cit. 2015-04-10]. Dostupné na: http://www.earchiv.cz/a95/a506c120.php3.

Poole, Matt. *Compilers - Course notes for module CS_218* [on-line]. 2002. [cit. 2015-03-02]. Dostupné na: http://www.win.tue.nl/ mvdbrand/courses/GLT/0910/papers/notes.pdf.

Rybička, Jiří. *Teorie programovacích jazyků - skripta k předmětu* [on-line]. 2015. [cit. 2015-03-22]. Dostupné na: nedostupné.

Rybička, Jiří. *LATEX pro začátečníky.* 3. vyd. Brno : Konvoj, 2003. 238 s. ISBN 80–7302–049–1.

Rybička, Jiří. *Úvod do teorie programovacích jazyků.* 1. vyd. Brno : Konvoj, 1999.

39 s. ISBN 80-85615-25-8.

TeX Live. *TeX Live homepage* [on-line]. 2015. [cit. 2015-04-15]. Dostupné na: `http://www.tug.org/texlive/`.

Tišnovský, Pavel. Seriál programovací jazyk Lua [on-line]. *Root.cz*, 2009, 10. března 2009 [cit. 2015-04-15]. (ISSN 1212-8309.) Dostupné na: `http://www.root.cz/clanky/programovaci-jazyk-lua/k02`.

Vavrečková, Šárka. *Syntaktická analýza* [on-line]. 2011. [cit. 2015-03-12]. Dostupné na: `http://vavreckova.zam.slu.cz/obsahy/prekl/prezentace/prekl__synt_zakladnimetody.pdf`.

Vavrečková, Šárka. *Tvorba Překladačů* [on-line]. 2008. [cit. 2015-03-12]. Dostupné na: `http://vavreckova.zam.slu.cz/prekl.html`.

Why was the first compiler written before the first interpreter? [on-line]. *Arstechnica*, 2014, 8. listopadu 2014 [cit. 2015-04-15]. (ISSN 0199-6649.) Dostupné na: `http://arstechnica.com/information-technology/2014/11/why-was-the-first-compiler-written-before-the-first-interpreter/`.

Wirth, Niklaus. *Compiler construction* [on-line]. 2005. [cit. 2015-04-20]. Dostupné na: `http://www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf`.

Seznam tabulek

Seznam obrázků

Přílohy

Všechny přílohy se nachází na přiloženém optickém disku. Seznam příloh:

- consyncheck.lua
- keywords.txt
- CFG.pdf
- FIRST.pdf
- FOLLOW.pdf
- ROZKLAD_TAB.pdf
- diplomaThesis.pdf
- syntaxDiagrams.png
- testFile1.tex
- testFile2.tex
- testFile3.tex
- testFile4.tex
- testFile5vybraneJevy.tex
- readme.txt