VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

Regulovaný syntaxí řízený překlad

Regulated Syntax-Directed Translation

DIPLOMOVÁ PRÁCE
MASTER´S THESIS

AUTOR PRÁCE — Bc. TOMÁŠ DVOŘÁK
AUTHOR

VEDOUCÍ PRÁCE — prof. RNDr. Alexander Meduna, CSc.
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Dvořák Tomáš, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Regulovaný syntaxí řízený překlad**
Regulated Syntax-Directed Translation
Kategorie: Překladače

Zadání:

1. Seznamte se s vybranými metodami syntaktické analýzy a syntaxí řízeného překladu dle pokynů vedoucího. Seznamte se regulovanymi automaty.
2. Zaveďte regulovane převodníky a syntaxí řízený překlad založený na nich.
3. Studujte vlastnosti překladu z bodu 2. Demonstrujte jeho přednosti oproti klasickému překladu.
4. Na základě konzultace s vedoucím uvažujte řadu syntaktických struktur. Zaměřte se na struktury, které nejsou bezkontextové. Popište jejich analýzu a překlad prostřednictvím metod z bodu 2.
5. Aplikujte regulovany syntaxí řízený překlad v kompilátorech. Aplikaci zaměřte na syntaktické struktury, které nejsou bezkontextové.
6. Implementujte aplikaci navrženou v bodě 5 a testujte ji.
7. Zhodnoťte dosažené výsledky. Diskutujte další vývoj projektu.

Literatura:

- Meduna Alexander: Deep Pushdown Automata, Acta Informatica, roč. 2006, č. 98, DE, pp. 114-124
- Rozenberg, G. and Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1 through 3, Springer, 1997, ISBN 3-540-60649-1
- Aho, A. V., Sethi, R., Ullman, J. D. : Compilers : principles, techniques, and tools, Addison-Wesley, 2nd ed., 2007, ISBN: 0321486811

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Meduna Alexander, prof. RNDr., CSc.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 23. října 2018

Regulovaný syntaxí řízený překlad

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. RNDr. Alexandera Meduny, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Dvořák
22. května 2019

Poděkování

Tímto bych rád poděkoval vedoucímu své diplomové práce prof. RNDr. Alexanderu Medunovi, CSc. za veškerou pomoc při tvorbě této práce a jeho cenné rady.

Abstrakt

Tato práce se zabývá formálním pohledem na regulovaný syntaxí řízený překlad. První část obsahuje formální základy teorie jazyků, jejich klasifikaci a analýzu. Jsou uvedeny příklady gramatik generující jazyky, které nejsou bezkontextové, především maticové gramatiky, gramatiky s nahodilým kontextem a programované gramatika. Jsou uvedeny konečné, zásobníkové, hluboké a regulované automaty. Formálně vymezuje převodníky a jejich roli v rámci formálního a syntaxí řízeného překladu. Zavádí regulované převodníky založené na regulovaných automatech. Jádrem práce je návrh algoritmů regulované syntaktické analýzy jako rozšíření tradičních algoritmů prediktivní syntaktické analýzy. Tyto algoritmy jsou navrženy pro všechny uvedené speciální typy gramatik. Závěr práce je věnován návrhu jazyka jako prostředku pro popis těchto gramatik a překladače těchto gramatik na kód syntaktického analyzátoru a jejich grafického analyzátoru.

Abstract

This thesis deals with formal and syntax directed translation. This thesis contains theoretical part, which defines regular, context free, context sensitive and recursively enumerable languages a grammar. There are given examples of grammars which are able to generate languages that are not context free. Covered by this thesis are matrix grammars, random context grammars and programed grammars. Researched are also finite, pushdown, deep and regular automata, transducers and their part within format syntax directed translation. This project also defines regular transducers based as regulated automata. Thesis defines regulated methods of syntax analysis based on predictive parsers. These methods cover analysis of studied regulated grammars. The final part of this thesis describes new language capable of effective description of these grammars and compiler producing parser code for these grammars written in this new language and their graphical analyzer.

Klíčová slova

překladač, gramatika, automat, převodník, překlad, syntaxí řízený překlad, regulované automaty, regulovaný převodník, syntaktická analýza, překladač

Keywords

compiler, grammar, automaton, transducer, syntax directed translation, regulated automata, regulated transducers, syntax analysis, compiler

Citace

DVOŘÁK Tomáš. Regulovaný syntaxí řízený překlad. Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. RNDr. Alexander Meduna, CSc.

Obsah

Kapitola 1

Úvod

Teorii formálních jazyků je věnována v posledních několika desítkách let stále větší pozornost a zavádějí se nové formální nástroje rozšiřující svou funkčností a vyjadřovací silou nástroje předešlé. Podobně tomu je i v oblasti syntaxí řízeného překladu, kterému se tato práce věnuje. Stávající metody založené na bezkontextových jazycích zdaleka nepokrývají některé potřeby moderních jazyků, na které se daný překlad aplikuje. Regulovaný syntaxí řízený překlad se tento problém snaží redukovat zaváděním speciálních typů gramatik, které jsou založeny na bezkontextových gramatikách, ale mají větší vyjadřovací sílu. Modely zmíněného regulovaného syntaxí řízeného překladu bývají často založeny na formalismech převodníků, nebo překladových gramatik, na které je potřeba formálně navrhnout metody syntaktické analýzy.

Práce se klade za cíl vymezit základní formální prostředky pro výstavbu aplikace zaměřené na regulovaný syntaxí řízený překlad a zavést metody regulované syntaktické analýzy a syntaxí řízeného překladu. V úvodních kapitolách jsou vymezeny základní typy jazyků a gramatik, které jsou klasifikovány dle Chomského hierarchie jazyků. Mezi popsané gramatiky patří především speciální třídy gramatik, které nejsou triviálním způsobem popsatelné pomocí uvedené hierarchie.

První z takových gramatik je maticová gramatika, která rozšiřuje třídu bezkontextových jazyků o jazyk nad prvky přechodové funkce, do kterého musí spadat posloupnost pravidel užitých pro kladné rozhodnutí členství jazyka v uvedené gramatice. Další uvedenou speciální třídou gramatik jsou gramatiky s nahodilým kontextem. Gramatiky z této třídy rozšiřují přepisovací pravidla bezkontextových gramatik o množinu symbolů, které se musejí (resp. nesmějí) vyskytnout ve větné formě, aby bylo pravidlo na danou větnou formu aplikovatelné. Poslední uvedenou speciální třídou gramatik je třída programovaných gramatiky. Programované gramatiky přepisovací pravidla bezkontextových gramatik rozšiřují o předepsání množin pravidel aplikovatelných bezprostředně po daném pravidle.

V práci jsou dále uvedeny formální definice konečných a zásobníkových automatů. Rovněž jsou zde věnovány kapitoly věnující se speciálním typům automatů. Prvním takovým automatem je regulovaný zásobníkový automat, jako rozšíření zásobníkového automatu o jazyk nad množinou prvků přechodové funkce. Je diskutována jeho vyjadřovací síla v porovnání se zásobníkovým automatem. Posledním uvedeným speciálním typem automatu je hluboký zásobníkový automat. Definice tohoto automatu je v práci dále užita jako formální základ některých dále navržených algoritmů syntaktické analýzy a regulované syntaktické analýzy.

Stěžejní částí práce je kapitola věnující se převodníkům. Tato kapitola uvádí formální definice konečných a zásobníkových převodníků jako rozšíření odpovídajících automatů. Zaveden je speciální regulovaný zásobníkový převodník, který odpovídajícím způsobem rozšiřuje regulovaný zásobníkový automat.

Přednosti nově zavedených formalismů (konkrétně zásobníkových převodníků a regulovaných zásobníkových převodníků) jsou patřičným způsobem demonstrovány příklady syntaxí řízeného překladu a regulovaného syntaxí řízeného překladu založených na zásobníkových a regulovaných zásobníkových převodnících. V kapitole věnující se syntaxí řízenému překladu je ukázáno, že regulovaný převodník dosahuje lepších výsledků, neboť je schopen překládat i některé jazyky, které nejsou bezkontextové, což je předmětem jednoho ze zmíněných příkladů.

Práce dále poskytuje ucelený pohled na strukturu překladačů a jeho základních stavebních bloků. Jednotlivé stavební bloky jsou pečlivě zkoumány a popsány. Speciální pozornost byla věnována lexikálnímu a syntaktickému analyzátoru. Jako modelu lexikální analýzy je stručně zmíněn konečný automat, který však omezuje jazyk lexikálních jednotek na třídu regulárních jazyků. Proto je pro potřeby práce dále zaveden i speciální typ lexikálního analyzátoru založeného na exotickém typu automatu, jehož funkce je popsána a formálně vytyčena algoritmem v kapitole věnované implementaci systému.

Důkladně je rozebrán blok syntaktické analýzy překladače. Zmíněny jsou základní přístupy syntaktické analýzy vyrovnávající se s nejednoznačností gramatik a základní přístupy simulace konstrukce derivačního stromu a návaznosti tohoto procesu na proces překladu.

Samostatnou kapitolou jsou studovány a důkladně popsány jsou metody syntaktické analýzy založené na LL tabulkách, ke kterým neodmyslitelně patří algoritmy výpočtu pomocných množin a konstrukce zmíněné LL tabulky. Blíže zkoumán je nerekurzivní algoritmu prediktivní syntaktické analýzy, který pojme vstupní řetězec, a pokusí se rozhodnout členství tohoto řetězce ve vstupní gramatice. Sekundárním výstupem tohoto algoritmu je posloupnost přepisovacích pravidel (levý rozbor) popisující konstrukci derivačního stromu.

Na podobném principu je zaveden a představen algoritmus regulované syntaktické analýzy lišící se od uvedeného nerekurzivního algoritmu pouze abstrakcí výběru přepisovacích pravidel k dané aktuální konfiguraci a užitím struktury kontextu. Na uvedené speciální typy gramatik jsou vytyčeny speciální podmínky pro jejich deterministickou syntaktickou analýzu. Ke každému podporovanému typu gramatiky byl uveden algoritmus výběru přepisovacího pravidla a příklad přehledně demonstrující tyto algoritmy.

Závěrečné kapitoly práce jsou věnovány návrhu a implementaci systému nástrojů, jejichž zaměřením je především aplikace regulované syntaktické analýzy a regulovaného syntaxí řízeného překladu v kompilátorech. Jedná se o nástroj *Parser*, který ověřuje příslušnost řetězce v jazyce dané gramatiky, nástroj *Generator*, jehož účelem je ověřit členství řetězce představující zápis gramatiky ve speciálním jazyce pro popis gramatik. Potvrdí-li se toto členství, je výstupem produkt generování výstupu. Tímto produktem může být zdrojový kód vstupní části syntaktického analyzátoru, nebo binární reprezentace vstupní gramatiky. Posledním navrženým nástrojem je *Editor*. Tento nástroj je aplikace s grafickým uživatelským rozhraním, která umožňuje pracovat

se zadanými gramatikami, provádět jejich analýzu a syntaktickou analýzu přijetí řetězce danou vstupní gramatikou. V kapitolách věnujících se popisem navržených nástrojů a jejich implementaci je, vzhledem k rozsáhlosti implementace, zvolen abstraktnější pohled s důrazem na důkladný popis principů, které implementace realizuje. Pro všechny uvedené nástroje je navržen nový typ lexikálního analyzátoru, který, mimo jiné, podporuje explicitní priority lexikálních jednotek. Primárním účelem implementace není efektivita výpočtu, proto jsou využity některé neefektivní konstrukce (např. cyklické rozpoznání regulárních výrazů) a vynechány optimalizace, které by běh nástrojů značným způsobem urychlily.

Pro popis vstupní gramatiky je zaveden speciální jazyk nazvaný *definiční metajazyk*. Tento jazyk umožňuje zápis bezkontextových a regulovaných gramatik v předem dané syntaxi. Vyjadřovací síla tohoto jazyka je demonstrována popisem syntaxe tohoto jazyka. Konkrétní přehled syntaxe je zařazen v přílohách této práce. Význačnou vlastností definičního metajazyka je napojení na kód uživatele (tvůrce nového překladače založeného na navržených nástrojích) mechanismy nezávislými na konkrétním programovacím jazyce. Rovněž jsou uvedeny příklady, které navržené principy názorně demonstrují.

Kapitola 2

Základní formální prostředky

Tato kapitola popisuje základní formální prostředky nutné pro definice pojmů kapitol následujících. Pro čtenáře témata znalého se může jednat o zopakování známého tématu, proto je možno tuto kapitolu přeskočit a pokračovat kapitolou následující. Definice 2.1 a 2.2 jsou převzány z [1].

2.1 Symboly a abeceda

Abecedou rozumíme neprázdnou množinu prvků, které nazýváme *symboly abecedy*. V rámci této práce je však účelné tuto definici abecedy omezit pouze na takové množiny prvků, které jsou konečné.

2.2 Řetězec

Řetězcem (také *slovem* nebo *větou*) nad danou abecedou rozumíme každou konečnou posloupnost symbolů abecedy. Prázdnou posloupnost symbolů, tj. posloupnost, která neobsahuje žádný symbol, nazýváme *prázdný řetězec*. Prázdný řetězec označujeme symbolem ε. Formálně lze definovat řetězec nad abecedou Σ takto:

1. prázdný řetězec ε je řetězec nad abecedou Σ,
2. je-li x řetězec nad Σ a $a \in \Sigma$, pak xa je řetězec nad Σ,
3. y je řetězec nad Σ, když a jen když lze y získat aplikací pravidel 1 a 2.

2.3 Operace nad řetězci

Nad řetězci definujeme několik základních operací. Definice následujících operací byly s drobnými úpravami převzaty z [1].

Konkatenace řetězců

Nechť x a y jsou dva řetězce nad abecedou Σ. Konkatenací řetězců x a y je řetězec xy. Jako operátor specifikující konkatenaci bývá někdy využit symbol tečky. V mnohých případech je však explicitní použití redundantní a nezapisuje se.

Mocnina řetězce

Nechť x je řetězec nad abecedou Σ, $i \geq 0$. Potom i-tou mocninu řetězce x označujeme x^i a definujeme ji následovně:

- $x^i = \varepsilon$ pro $i = 0$
- $x^i = xx^{(i-1)}$ pro každé $i \geq 1$

Funkce *alph*

Funkcí *alph* rozumíme funkci definovanou nad každým řetězcem nad abecedou Σ následovně:

- $alph(\varepsilon) = \emptyset$
- $alph(ax) = \{a\} \cup alph(x)$, $a \in \Sigma$, $x \in \Sigma^*$

Jedná se tedy o funkci rozkládající řetězec na množinu symbolů, ze kterých je složen.

Funkce *card*

Funkcí *card* rozumíme funkci definovanou pro každou množinu jako kardinalitu této množiny.

Funkce *occur*

Funkcí *occur* rozumíme funkci definovanou pro každý řetězec $x \in \Sigma^*$ a symbol $a \in \Sigma$ jako počet výskytů symbolu a v řetězci x. V rámci této práce je rovněž využit ekvivalentní zkrácený zápis $\#_a x$.

Funkce *reversal*

Funkcí *reversal* rozumíme funkci definovanou nad každým řetězcem nad abecedou Σ následovně:

- $reversal\ (\varepsilon) = \varepsilon$
- $reversal\ (ax) = reversal(x)a$, $a \in \Sigma$, $x \in \Sigma^*$

Jedná se tedy o funkci, vracející řetězec symbolů v opačném pořadí.

Kapitola 3

Jazyky

Cílem této kapitoly je seznámit čtenáře s pojmem *jazyk* z pohledu teoretické informatiky, jejich klasifikaci dle Chomského hierarchie jazyků [2] a základními třídami jazyků. Jsou zde formálně definovány základní pojmy a operace nad jazyky, a příklady užití vybraných tříd jazyků. Informace uvedené v této kapitole byly čerpány z [8], [1] a [14].

Pod pojmem jazyk intuitivně uvažujeme podmnožinu všech řetězců nad danou abecedou. Speciálními případy jazyků jsou jazyky $\{\varepsilon\}$ a $\emptyset$, které jsou definovány nad každou abecedou. Přitom platí, že $\{\varepsilon\} \neq \emptyset$, neboť první z uvedených obsahuje prázdný řetězec, zatímco druhý neobsahuje žádný řetězec.

Pro efektivní zkoumání jazyků využíváme klasifikace jazyků do tříd, které sdružují jazyky s podobnými, případně stejnými vlastnostmi. Příkladem takové třídy jazyků je třída *konečných jazyků*, která obsahuje jazyky (množiny řetězců), mající konečný počet prvků. Přestože jsou jazyky chápány jako množiny řetězců, aplikace standardních množinových operací, jakými jsou například *průnik*, *sjednocení* nebo *doplněk* jsou zpravidla doprovázené řadou omezení. Důvodem těchto omezení je situace, při které aplikací uvedených operací nad dvojicemi jazyků ze stejných tříd vznikají jazyky, které nepatří do stejné třídy jazyků a záleží tedy na konkrétních jazycích, na které je daný operátor aplikován. Platí-li, že aplikací operátoru P na libovolný jazyk z třídy jazyků C_1 vnikne jazyk patřící do třídy jazyků C_2, je třída C_1 uzavřena na operátor P (viz [1] kapitoly o uzávěrových vlastnostech). Definice 3.1 byla převzata z [1].

3.1 Definice

Nechť Σ je abeceda. Označme symbolem Σ^* množinu všech řetězců nad abecedou Σ včetně řetězce prázdného, symbolem Σ^+ množinu všech řetězců nad abecedou Σ vyjma řetězce prázdného, tj. $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. Množinu L, pro níž platí $L \subseteq \Sigma^*$ (případně $L \subseteq \Sigma^+$, pokud $\varepsilon \notin L$) nazýváme *jazykem* L nad abecedou Σ. Jazykem tedy může být libovolná podmnožina řetězců nad danou abecedou. Řetězec $x \in L$ nazýváme *větou* (nebo také *řetězcem*) jazyka L.

3.1.1 Konkatenace jazyků

Konkatenace jazyků L_1 a L_2 je formálně definována následovně:

- $L_1.L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}$

3.1.2 Mocnina jazyka

Mocnina jazyka L^i je definována následovně:

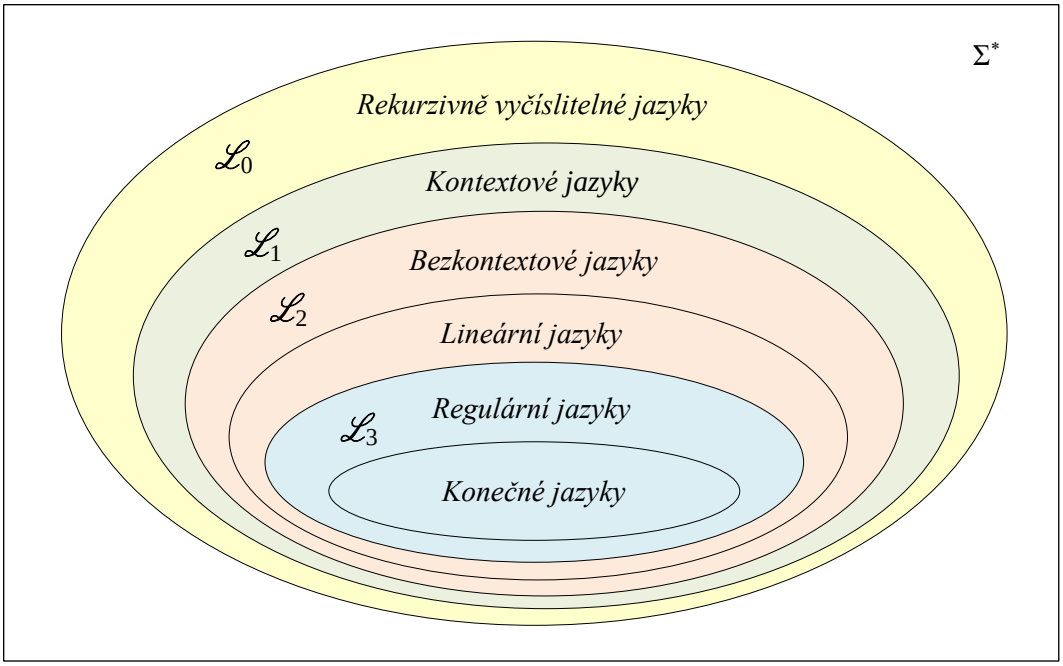
- $L^i = \{\varepsilon\}$ pro $i = 0$
- $L^i = L.L^{i-1}$ pro $i \geq 1$

3.1.3 Iterace jazyka

Iterací L^* jazyka L rozumíme sjednocení všech kladných mocnin jazyka. Speciálním případem iterace jazyka L označované jako *kladná iterace* L^+ je sjednocení všech kladných nenulových mocnin jazyka. Produktem kladné iterace bývá obvykle množina řetězců neobsahující prázdný řetězec.

3.2 Chomského hierarchie jazyků

Jak bylo v úvodu kapitoly naznačeno, je vhodné jazyky sdružovat do tříd podle jejich vlastností. Existuje mnoho způsobů klasifikace jednotlivých jazyků. Chomského hierarchie [2] byla zavedena Noamem Chomskym v roce 1956. Jazyky (a gramatiky) klasifikuje do 4 základních tříd podle obsahu množiny přepisovacích pravidel gramatik generujících tyto jazyky (viz 4). Uspořádání těchto tříd barevně znázorňuje obrázek 3.1, na kterém jsou mimo 4 základních tříd patrné i podtřídy bezkontextových a regulárních jazyků, jejichž význam bude využit v následujících kapitolách. Protože jsou jazyky množinami řetězců, platí v Chomského hierarchii jejich vzájemné obsažení dle výše zmíněného obrázku. To kupříkladu znamená, že všechny jazyky z třídy *regulárních jazyků* patří i do třídy *bezkontextových jazyků* atp. Za podotknutí stojí i skutečnost, že z pohledu Chomského hierarchie jazyků nad abecedou Σ existují i jazyky mimo třídu *rekurzivně*



Obrázek 3.1: Diagram tříd jazyků nad abecedou Σ

vyčíslitelných jazyků. Chomského hierarchie jazyků je jedním z mnoha způsobů, jak klasifikovat jazyky a zdaleka nevystihuje potřeby klasifikace jazyků generovaných regulovanými gramatikami (viz 4). V rámci tohoto textu jsou představeny gramatiky, které nespadají do výše uvedeného modelu. Takové gramatiky mohou například generovat pouze některé bezkontextové jazyky, a některé rekurzivně vyčíslitelné jazyky. Z pohledu Chomského hierarchie by jednoznačné určení třídy takových jazyků nebylo jednoduché (nebo vůbec možné), proto pro klasifikaci takových jazyků je využito množinové srovnání dle jejich obsahu s jinými jazyky.

3.3 Regulární jazyky

Třída regulárních jazyků představuje dle Chomského hierarchie jazyků nejméně obsáhlou třídu jazyků. Patří mezi ně všechny jazyky, pro které existuje *regulární výraz* (viz níže), který je značí. Důsledkem této skutečnosti mezi regulární jazyky řadíme i všechny *konečné jazyky*. Jako formální aparát pro rozpoznávání řetězců regulárních jazyků využíváme *konečné automaty*, kterým je věnována kapitola 5.1. Typickým zástupcem regulárních jazyků je dle [4] jazyk $L = \{a^n, n \geq 1\}$. Definice 3.3.1 byla převzána z [1].

3.3.1 Regulární výraz

Nechť Σ je abeceda. Regulární výrazy a jazyky, které značí, jsou definovány následovně:

- $\emptyset$ je *regulární výraz* značí prázdnou množinu (prázdný jazyk)
- ε je *regulární výraz* značící jazyk $\{\varepsilon\}$
- a pro všechna $a \in \Sigma$ je *regulární výraz* značící jazyk $\{a\}$
- Nechť r a s jsou regulárními výrazy značící po řadě jazyky L_r a L_s, potom:
  - $(r.s)$ je *regulární výraz* značící jazyk $L = L_r.L_s$
  - $(r + s)$ je *regulární výraz* značící jazyk $L = L_r \cup L_s$
  - (r^*) je *regulární výraz* značící jazyk $L = L_r^*$

3.4 Bezkontextové jazyky

Třída bezkontextových jazyků dle Chomského hierarchie obsahuje i všechny regulární jazyky. Pro rozpoznávání řetězců bezkontextových jazyků využíváme *zásobníkové automaty*, kterým je věnována kapitola 5.2. Typickým využitím bezkontextových jazyků je popis syntaktických struktur. Typickým zástupcem bezkontextových jazyků je dle [8] jazyk $L = \{a^n b^n, n \geq 1\}$. Bezkontextové jazyky jsou jednou z nejčastěji využívaných tříd jazyků při tvorbě překladačů, protože umožňují jednoduchý popis syntaxe některých moderních programovacích jazyků.

3.5 Kontextové jazyky

Jazyky z třídy kontextových jazyků dokážeme rozpoznat pomocí *lineárně omezeného turingova stroje*. Tento stroj je definován jako turingův stroj, který nikdy nepřekročí hranici pásky, na které je zapsán jeho vstup (viz [1]). Z pohledu této práce se nejedná o nijak zvlášť významnou třídu jazyků, proto ji nebude věnována další pozornost.

3.6 Rekurzivně vyčíslitelné jazyky

Rekurzivně vyčíslitelné jazyky jsou dle Chomského hierarchie nejobsáhlejší třídou jazyků. Každý takový jazyk dokážeme rozpoznat *turingovým strojem* [1]. Z pohledu této práce jsou však málo významné a nebude jim věnována další pozornost. Tyto jazyky (někdy označované jako *neomezené jazyky*) jsou generovány třídou *neomezených gramatik*. Tato třída jazyků se vyznačuje především tím, že problém příslušnosti řetězce do této třídy jazyků je pouze částečně rozhodnutelný [1]. Tato skutečnost má dopad na analýzu gramatik, které jsou svou vyjadřovací silou ekvivalentní neomezeným gramatikám.

Kapitola 4

Gramatiky

Tato kapitola si klade za cíl seznámit čtenáře se základními typy gramatik, jejich formální definicí včetně definic přímé i nepřímé derivace a všech formálních náležitostí nezbytných k jejich využití. Kapitola čerpá informace převážně z [1] a [11].

Gramatiky jsou jedním z formálních aparátů pro specifikaci a konečný popis jazyků. O jazyku $L(G)$, který je popsán gramatikou G, říkáme, že jej daná gramatika *generuje*. Základní funkcí gramatik je rozhodování členství řetězců v jazycích, který generují. V úvodu kapitoly jsou popsány základní třídy gramatik. Podobně jako základní třídy jazyků z předchozí kapitoly klasifikujeme i gramatiky do tříd podle jazyků, které generují. V druhé části kapitoly jsou rozebrány speciální třídy gramatik označované jako *regulované gramatiky*. Tyto gramatiky se vyznačují schopností generovat jazyky, které nelze jednoznačně klasifikovat dle Chomského hierarchie, proto jsou pro jejich srovnání využity vyjadřovací síly a množinové porovnávací operátory.

Rozlišení základních gramatik lze provádět i na základě její struktury, konkrétně podle podoby prvků *množiny přepisovacích pravidel*. Tohle rozdělení má základ v Chomského hierarchii jazyků uvedené v kapitole 3.2, přičemž první 4 uvedené třídy gramatik jsou tímto způsobem klasifikovány. Všechny definice uvedené v rámci této kapitoly byly inspirovány [1] a [11].

4.1 Definice

Gramatikou G obecně rozumíme čtveřici $G = (N, \Sigma, P, S)$, kde:

- N je *konečná množina neterminálních symbolů*
- Σ je *konečná množina terminálních symbolů*, přičemž $N \cap \Sigma = \emptyset$
- P je podmnožina kartézského součinu $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- $S \in N$ je *počáteční neterminální symbol*

Množinu P budeme dále označovat jako *množinu přepisovacích pravidel*, přičemž prvky (a, b) této množiny budeme označovat jako *přepisovací pravidla*. Řetězec a (resp. řetězec b) budeme nazývat *levou* (resp. *pravou*) *stranou přepisovacího pravidla*.

4.1.1 Větná forma

Větnou formou označujeme řetězec V složený z terminálních a neterminálních symbolů. Jedná se tedy o prvek z množiny všech řetězců nad množinou (abecedou) terminálních a neterminálních symbolů, formálně tedy $V \in (N \cup \Sigma)^*$.

4.1.2 Věta

V rámci této práce *větou* označujeme libovolný (tedy i prázdný) řetězec nad množinou terminálních symbolů.

4.1.3 Derivační krok

Derivačním krokem obecně v oblasti gramatik rozumíme nahrazení řetězce v aktuální větné formě dle některého z přepisovacích pravidel. Formálně je derivace relace mezi dvěma větnými formami V_1 a V_2. Někdy se používá označení *přímá derivace* (V_1 přímo derivuje V_2). Tato relace je v dalším textu zapisována ve tvaru $V_1 \Rightarrow V_2$ [p], nebo zkráceně $V_1 \Rightarrow V_2$. Derivační krok je formálně definován následovně:

$$uXv \Rightarrow uYv\ [p],\ u, v \in (N \cup \Sigma)^*,\ p = (X, Y) \in P.$$

Uvedená definice derivačního kroku je společná pro většinu gramatik uvedených v následujících kapitolách.

4.1.4 Nepřímá derivace

Nepřímou derivací označujeme relaci mezi větnými formami V_a a V_b takovou, že existuje taková posloupnost větných forem $V_1, V_2, \ldots, V_i$ pro $i \geq 2$, že $V_a = V_1$, $V_1 \Rightarrow V_2$, $V_2 \Rightarrow V_3$, …, $V_{i-1} \Rightarrow V_i$, $V_i = V_b$. Jako nepřímá derivace je označován i případ, kdy $V_a = V_b$ (neprovedení žádného derivačního kroku). Speciálním případem nepřímé derivace je i *přímá derivace*. Nepřímá derivace větných forem V_a a V_b je v dalším textu zapisována jako $V_a \Rightarrow^* V_b$. Říkáme, že větná forma V_b je *derivovatelná* z větné formy V_a.

4.1.5 Generovaný jazyk

Říkáme, že gramatika G generuje jazyk $L(G)$, pokud $L(G) = \{w \mid S \Rightarrow^* w, w \in \Sigma^*\}$, tedy pokud jazyk obsahuje všechny a pouze takové věty, které jsou derivovatelné z počátečního neterminálního symbolu.

4.1.6 Nejednoznačnost gramatik

Gramatika G je nejednoznačná, pokud jazyk $L(G)$ generovaný touto gramatikou obsahuje alespoň jeden řetězec, pro který existuje více než jedna nepřímá derivace z počátečního neterminálního symbolu a tímto řetězcem. Jinak je gramatika G jednoznačná. Nejednoznačnost gramatik je obecně problém snižující efektivitu algoritmů rozhodujících členství řetězce v jazyce generovaném gramatikou. Z tohoto důvodu se v řadě případů využívá jako vstup efektivních algoritmů podmnožina gramatik neobsahující nejednoznačné gramatiky. Typickým zástupcem je třída bezkontextových LL gramatik (viz 9), která je podmnožinou bezkontextových gramatik.

4.2 Regulární gramatiky

Regulární gramatiky mají z pohledu Chomského hierarchie nejslabší vyjadřovací sílu, což je dáno velmi striktním omezením tvaru prvků množiny přepisovacích pravidel.

4.2.1 Definice

Regulární gramatikou G rozumíme čtveřici $G = (N, \Sigma, P, S)$, kde:

- N, Σ a S mají stejný význam jako v definici 4.1
- P představuje podmnožinu kartézského součinu $(N \times \Sigma^* \cup \Sigma^* N)$, tedy přepisovacích pravidel tvaru (A, xB) nebo (A, x), $A, B \in N, x \in \Sigma^*$

Z výše uvedené definice je patrné, že přepisovací pravidla regulárních gramatik mají na své levé straně pouze neterminální symboly. Tato vlastnost je činí jednoduše zpracovatelnými a jejich implementace je velmi jednoduchá, což je ukázáno v [7] v kapitole 7 věnující se hledání vzorů.

4.3 Lineární gramatiky

Lineární gramatiky jsou podmnožinou bezkontextových gramatik. Oproti nim však na pravé straně přepisovacích pravidel mohou mít nejvíce jeden neterminální symbol. Oproti regulárním gramatikám však mohou mít na pravé straně neterminální symbol i před libovolným počtem symbolů terminálních.

4.3.1 Definice

Lineární gramatikou G rozumíme čtveřici $G = (N, \Sigma, P, S)$, kde:

- N, Σ a S mají stejný význam jako v definici 4.1
- P představuje podmnožinu kartézského součinu $(N \times \Sigma^* \cup \Sigma^* N \Sigma^*)$, tedy přepisovacích pravidel tvaru (A, xBy) nebo (A, x), $A, B \in N, x, y \in \Sigma^*$

4.4 Bezkontextové gramatiky

Bezkontextové gramatiky přináší oproti regulárním gramatikám silnější vyjadřovací sílu. Je to dáno tím, že mohou větnou formu rozšiřovat přímou derivací o libovolný počet neterminálních symbolů, zatímco regulární gramatiky větnou formu o více neterminálních symbolů rozšířit nemohou. Za důkaz lze považovat například jazyk, jež je generován gramatikou z příkladu 4.4.2, který regulární gramatika vygenerovat nedokáže z důvodu plynoucích z konečnosti množiny neterminální symbolů a nekonečné spočetné množiny vět generovaných uvedenou gramatikou (formální důkaz viz [5]).

4.4.1 Definice

Bezkontextovou gramatikou G rozumíme čtveřici $G = (N, \Sigma, P, S)$, kde:

- N, Σ a S mají stejný význam jako v definici 4.1
- P představuje podmnožinu kartézského součinu $N \times (N \cup \Sigma)^*$, tedy přepisovacích pravidel tvaru (A, x), $A \in N, x \in (N \cup \Sigma)^*$

4.4.2 Příklad

Mějme gramatiku $G = (\{S\}, \{a, b\}, \{1{:}(S, aSa), 2{:}\ (S, b)\}, S)$

$$\begin{aligned} \underline{S} &\Rightarrow \underline{aSa} && [1] \\ a\underline{S}a &\Rightarrow a\underline{aSa}a && [1] \\ &\ldots \\ a^{i-1}\underline{S}a^{\,i-1} &\Rightarrow a^{i}\underline{S}a^{\,i} && [1] \\ a^{i}\underline{S}a^{i} &\Rightarrow a^{i}\underline{b}a^{i} && [2] \end{aligned}$$

Je patrné, že k větě a^iba vedla posloupnost přepisovacích pravidel $1^i\,2$. Gramatika G tedy zjevně generuje jazyk $L(G) = \{a^iba^i,\ i \geq 0\}$.

4.5 Kontextové gramatiky

Kontextové gramatiky přináší oproti bezkontextovým gramatikám rozšíření o kontext prvků množiny přepisovacích pravidel, které se mohou (resp. nemohou) užít v aktuální větné formě.

4.5.1 Definice

Kontextovou gramatikou G rozumíme čtveřici $G = (N, \Sigma, P, S)$, kde:

- N, Σ a S mají stejný význam jako v definici 4.1
- P představuje podmnožinu kartézského součinu $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^* N (N \cup \Sigma)^*$, tedy pravidel tvaru $(\alpha A\beta, \alpha B\beta)$, $A \in N$, $B \in (N \cup \Sigma)^+$, $\alpha, \beta \in (N \cup \Sigma)^*$

Dle výše uvedené definice lze α a β chápat jako kontext větné formy v rámci kterého lze A přepsat na B.

4.6 Maticové gramatiky

Maticové gramatiky jsou první z uvedených regulovaných gramatik. Jsou rozšířením bezkontextových gramatik o konečný jazyk nad množinou přepisovacích pravidel. Tento jazyk je zapojen do procesu výběru přepisovacího pravidla, které se má aplikovat v rámci nepřímé derivace. Tato podkapitola čerpá informace z [12].

4.6.1 Definice

Maticovou gramatikou H rozumíme dvojici $H = (G, M)$, kde:

- G je bezkontextová gramatika, $G = (N, \Sigma, P, S)$
- M je konečný jazyk nad abecedou $P \times \{+, -\}$

Nechť Ψ je abeceda, jejímiž prvky jsou pouze dvojice sestávající se z jednoznačného identifikátoru prvku množiny přepisovacích pravidel P gramatiky G a symbolu z množiny $\{+, -\}$. Potom M je jazykem nad abecedou Ψ. Prvky $(p, +) \in \Psi$ označujeme jako *pravidla s kontrolou na výskyt*. Řetězce jazyka M budeme označovat jako *řádky matice*. Pro zjednodušení zápisu budeme v rámci

tohoto textu zapisovat jako dvojice pouze taková pravidla, která mají kontrolu na výskyt. Ostatní pravidla budeme značit jako elementární prvky. Každé pravidlo $p = (X, \varepsilon) \in P$, $X \in N$ označujeme jako *vymazávací pravidlo*.

4.6.2 Nepřímá derivace

Z hlediska dalších definic je účelné zavést speciální variantu nepřímé derivace. Nechť $A_1, A_2, ..., A_i$ pro $i \geq 2$ jsou větnými formami gramatiky G takovými, že $A_{j-1} \Rightarrow A_j\ [p_j]$, nebo $A_{j-1} = A_j \wedge p_j$ není aplikovatelné na $A_{j-1} \wedge q_j = +$ pro všechna $j \in <1, i>$. Potom $A_1 \Rightarrow^* A_i$, pouze pokud $(p_2, q_2)(p_3, q_3)...(p_{i-1}, q_{i-1}) \in M$. Mezi nepřímou derivaci řadíme i speciální případ, kdy $i = 1$, tedy neprovedení žádného derivačního kroku.

4.6.3 Vyjadřovací síla

Díky možnosti explicitně určovat pořadí použití některých přepisovacích pravidel v nepřímé derivaci, je vyjadřovací síla maticových gramatik větší, než v případě gramatik bezkontextových. Z hlediska vlivu tvaru přepisovacích pravidel a dané matice rozlišujeme několik tříd maticových gramatik. Označme $M, M_\varepsilon, M^{ac}, M_\varepsilon^{ac}$ po řadě maticové gramatiky bez vymazávacích pravidel a bez pravidel s kontrolou na výskyt, maticové gramatiky s vymazávacími pravidly bez pravidel s kontrolou na výskyt, maticové gramatiky bez vymazávacích pravidel s pravidly s kontrolou na výskyt a maticové gramatiky s vymazávacími pravidly a pravidly s kontrolou na výskyt. Dle [12] platí následující hierarchie:

$$\mathscr{L}_2 \subseteq L(M) \subseteq L(M_\varepsilon) \subseteq L(M^{ac}) \subseteq L(M_\varepsilon^{ac}) = \mathscr{L}_0$$

4.6.4 Příklad

Uvažujme maticovou gramatiku $H = (G, M)$, kde $G = (N, \Sigma, P, S)$ je bezkontextová gramatika, a jednotlivé množiny jsou definovány:

$N = \{S, A, B\}$

$\Sigma = \{a, b, c\}$

$P = \{$ 1: (S, AB),
2: (A, aA),
3: (B, bBc),
4: (A, a),
5: (B, bc) $\}$

$M = \{1, 23, 45\}$

Potom posloupností derivačních kroků z počátečního neterminálního symbol S je možno dokázat, že gramatika generuje jazyk $L(H) = \{a^i b^i c^i, i \geq 1\}$, a to následujícím způsobem:

$$\begin{aligned}
\underline{S} &\Rightarrow \underline{AB} && [1]\\
\underline{A}B &\Rightarrow \underline{aA}B && [2]\\
aA\underline{B} &\Rightarrow aA\underline{bBc} && [3]\\
a\underline{A}bBc &\Rightarrow a\underline{aA}bBc && [2]\\
aaAb\underline{B}c &\Rightarrow aaAb\underline{bBc}c && [3]\\
&\ldots\\
a^{i-1}Ab^{\,i-2}\underline{B}\,c^{\,i-2} &\Rightarrow a^{i-1}Ab^{\,i-2}\underline{bBc}c^{\,i-2} && [3]
\end{aligned}$$

$$a^{i-1}\underline{A}b^{\,i-1}Bc^{\,i-1} \Rightarrow a^{i-1}\underline{a}b^{\,i-1}Bc^{\,i-1} \quad [4]$$
$$a^{i}b^{\,i-1}\underline{B}c^{\,i-1} \Rightarrow a^{i}b^{i}c^{i} \quad [5]$$

Je patrné, že k výsledné větě $a^ib^ic^i$ vedla posloupnost derivačních kroků využitím přepisovacích pravidel $w = 1(23)^{i-1}45 \in M^*$, tedy nepřímá derivace z počátečního neterminálního symbolu v souladu s definicí 4.6.2.

4.7 Gramatiky s nahodilým kontextem

Gramatiky s nahodilým kontextem jsou rozšířením bezkontextových gramatik o konečnou relaci mezi množinou přepisovacích pravidel a množinou neterminálních symbolů. Tato relace předepisuje každému přepisovacímu pravidlu, za jakých okolností může být aplikováno v závislosti na aktuální větné formě. Tato podkapitola čerpá informace z [12].

4.7.1 Definice

Gramatikou s nahodilým kontextem H rozumíme trojici $H = (G, R, F)$, kde:

- G je bezkontextová gramatika, $G = (N, \Sigma, P, S)$
- R je množina *povolujících kontextů* přepisovacích pravidel P
- F je množina *omezujících kontextů* přepisovacích pravidel P

Nechť Ψ je abeceda, jejímiž prvky jsou pouze jednoznačné identifikátory prvků množiny přepisovacích pravidel P gramatiky G. Potom R a F jsou podmnožinami kartézského součinu $\Psi \times N^*$. Pokud množina omezujících (resp. povolujících) kontextů každého pravidla dané gramatiky je prázdná, jedná se o povolující (resp. omezující) gramatiku.

4.7.2 Derivační krok

Formálně je derivační krok definován následovně: $\forall p = (X, b) \in P$, $u, v \in (N \cup \Sigma)^*$, $(p, x) \in R$, $(p, y) \in F$: $uXv \Rightarrow ubv\,[p]$ pouze pokud $x \subseteq alph(uXv) \wedge y \cap alph(uXv) = \emptyset$. Je patrné, že derivační krok je závislý na aktuální větné formě, což neodpovídá definici tvaru přepisovacích pravidel bezkontextových gramatik. Uvedená definice neformálně vyjadřuje myšlenku, že derivační krok lze uskutečnit pouze tehdy, je-li v aktuální větné formě zastoupen každý symbol daný relací R a žádný symbol daný relací F pro dané přepisovací pravidlo.

4.7.3 Vyjadřovací síla

Vyjadřovací síla je opět větší než u bezkontextových gramatik. Z hlediska vlivu tvaru přepisovacích pravidel a množin povolovacích a omezujících kontextů rozlišujeme několik tříd maticových gramatik. Označme $RC, RC_\varepsilon, RC^{ac}, RC_\varepsilon^{ac}$ po řadě povolující gramatiky s nahodilým kontextem bez vymazávacích pravidel, povolující gramatiky s nahodilým kontextem s vymazávacími pravidly, gramatiky s nahodilým kontextem bez vymazávacích pravidel a gramatiky s nahodilým kontextem s vymazávacími pravidly.

Dle [12] platí následující hierarchie:

$$\begin{aligned} \mathscr{L}_2 &\subseteq L(RC) \\ L(RC) &\subseteq L(RC^{ac}) \\ L(RC) &\subseteq L(RC_\varepsilon) \\ L(RC^{ac}) &\subseteq L(RC_\varepsilon^{ac}) \\ L(RC_\varepsilon) &\subseteq L(RC_\varepsilon^{ac}) \\ L(RC_\varepsilon^{ac}) &= \mathscr{L}_0 \end{aligned}$$

4.7.4 Příklad

Uvažujme gramatiku s nahodilým kontextem $H = (G, R, F)$, kde $G = (N, \Sigma, P, S)$ je bezkontextová gramatika. Obsah jednotlivých množin je následující, přičemž pro zjednodušení a větší přehlednost zápisu jsou obsahy množin P, R a F zapsány intuitivně vedle sebe:

N = $\{S, A, B, C, A', B', C'\}$

$\Sigma = \{a, b, c\}$

P, R, F = {
1: $(S, ABC), \emptyset, \emptyset$
2: $(A, aA'), \{B\}, \emptyset$
3: $(B, bB'), \{C\}, \emptyset$
4: $(C, cC'), \{A'\}, \emptyset$
5: $(A', A), \{B'\}, \emptyset$
6: $(B', B), \{C'\}, \emptyset$
7: $(C', C), \{A\}, \emptyset$
8: $(A, a), \{B\}, \emptyset$
9: $(B, b), \{C\}, \emptyset$
10: $(C, c), \emptyset, \emptyset$ }

Potom posloupností derivačních kroků z počátečního neterminálního symbolu S je možno dokázat, že gramatika generuje jazyk $L(H) = \{a^i b^i c^i, i \geq 1\}$, a to následujícím způsobem:

$$\begin{aligned} \underline{S} &\Rightarrow \underline{AB}C && [1] \\ \underline{A}BC &\Rightarrow \underline{aA'}BC && [2] \\ aA'\underline{B}C &\Rightarrow aA'\underline{bB'}C && [3] \\ aA'bB'\underline{C} &\Rightarrow aA'bB'\underline{cC'} && [4] \\ aA'bB'cC' &\Rightarrow a\underline{A}bB'cC' && [5] \\ aAbB'cC' &\Rightarrow aAb\underline{B}cC' && [6] \\ aAbBcC' &\Rightarrow aAbBc\underline{C} && [7] \\ &\dots \\ \underline{A}a^{i-1}Bb^{i-1}Cc^{i-1} &\Rightarrow \underline{a}a^{i-1}Bb^{i-1}Cc^{i-1} && [8] \\ a^i\underline{B}b^{i-1}Cc^{i-1} &\Rightarrow a^i\underline{b}b^{i-1}Cc^{i-1} && [9] \\ a^ib^i\underline{C}c^{i-1} &\Rightarrow a^ib^i\underline{c}c^{i-1} && [10] \end{aligned}$$

Barevně jsou vyznačeny vyžadované symboly (*kontexty*) umožňující použití daného přepisovacího pravidla. Podtržení vyjadřuje výběr konkrétního neterminálního symbolu pro využití jako levé strany daného přepisovacího pravidla. Je patrné, že k výsledné větě $a^ib^ic^i$ vedla posloupnost

derivačních kroků v souladu s definicí 4.7.2. V příkladu bylo v několika krocích možno vybrat několik přepisovacích pravidel najednou. Pořadí jejich výběru však v tomto případě nehrálo roli.

4.8 Programované gramatiky

Programované gramatiky jsou rozšířením bezkontextových gramatik o konečnou relaci definovanou nad dvojicemi prvků z množiny přepisovacích pravidel. Tato relace vymezuje každému pravidlu množinu přepisovacích pravidel, ve které se nachází pravidlo aplikované bezprostředně po aplikaci daného přepisovacího pravidla. Lze tedy předepisovat sekvence přepisovacích pravidel, díky čemuž je vyjadřovací síla opět vyšší než v případě bezkontextových gramatik, jak dokazuje příklad 4.8.4. Tato podkapitola čerpá informace z [12].

4.8.1 Definice

Programovanou gramatikou H rozumíme trojici $H = (G, R, F)$, kde:

- G je bezkontextová gramatika, $G = (N, \Sigma, P, S)$
- R a F jsou konečné podmnožiny kartézského součinu $P \times P^*$

Sémantiky množin R a F jsou dány definicí nepřímé derivace 4.8.2. Množinu R označujeme jako množinu *pravidel úspěchu*, množinu F množinu *pravidel neúspěchu*.

4.8.2 Nepřímá derivace

Nechť $A_1, A_2, ..., A_n$, $n \geq 3$ jsou větnými formami gramatiky G takovými, že $A_1 \Rightarrow A_2\ [p_2]$, $A_2 \Rightarrow A_3\ [p_3]$, ..., $A_{n-1} \Rightarrow A_n\ [p_n]$ pokud $p_i \in R(p_{i-1})$ nebo $A_{i-1} = A_i \wedge p_i \in F(p_{i-1}) \wedge p_i$ není aplikovatelné na A_{i-1} pro všechna $i \in \langle 3, n \rangle$. Potom $A_1 \Rightarrow^* A_n$. Mezi nepřímou derivaci řadíme i speciální případy pro $n = 0$ a $n = 1$, tedy provedení nejvýše jednoho derivačního kroku. Na tyto speciální případy se z výše uvedené definice nevztahují speciální omezující podmínky specifické pro programované gramatiky, ale jen pravidla přímé derivace bezkontextových gramatik.

4.8.3 Vyjadřovací síla

Vyjadřovací síla je opět větší než u bezkontextových gramatik. Z hlediska vlivu tvaru přepisovacích pravidel a množin R a F rozlišujeme několik tříd programovaných gramatik. Označme $P, P_\varepsilon, P^{ac}, P_\varepsilon^{ac}$ po řadě programované gramatiky bez vymazávacích pravidel a bez pravidel s kontrolou na výskyt, programované gramatiky s vymazávacími pravidly a bez pravidel s kontrolou na výskyt, programované gramatiky bez vymazávacích pravidel s pravidly s kontrolou na výskyt a programované gramatiky s vymazávacími pravidly a s pravidly s kontrolou na výskyt. Dle [12] platí následující hierarchie:

$$\mathscr{L}_2 \subseteq L(P) \subseteq L(P_\varepsilon) \subseteq L(P^{ac}) \subseteq L(P_\varepsilon^{ac}) = \mathscr{L}_0$$

4.8.4 Příklad

Uvažujme programovanou gramatiku $H = (G, R, F)$, kde $G = (N, \Sigma, P, S)$ je bezkontextová gramatika. Obsah jednotlivých množin je následující, přičemž pro zjednodušení a větší přehlednost zápisu jsou obsahy množin P, R a F zapsány intuitivně vedle sebe:

N = {*S*, *A*, *B*, *C*}
Σ = {*a*, *b*, *c*}
P, *R*, *F* = { 1*:* (*S*, *ABC*), {2, 5}, ∅
2*:* (*A*, *aA*), {3}, ∅
3*:* (*B*, *bB*), {4}, ∅
4*:* (*C*, *cC*), {2, 5}, ∅
5*:* (*A*, *a*), {6}, ∅
6*:* (*B*, *b*), {7}, ∅
7*:* (*C*, *c*), {7}, ∅ }

Potom posloupností derivačních kroků z počátečního neterminálního symbol S je možno dokázat, že gramatika generuje jazyk $L(H) = \{a^i b^i c^i, i \geq 1\}$, a to následujícím způsobem:

$$
\begin{aligned}
\underline{S} &\Rightarrow \underline{ABC} && [1]\\
\underline{A}BC &\Rightarrow \underline{aA}BC && [2]\\
aA\underline{B}C &\Rightarrow aA\underline{bB}C && [3]\\
aAbB\underline{C} &\Rightarrow aAbB\underline{cC} && [4]\\
&\ldots\\
a^{i-1}\underline{A}b^{i-1}Bc^{i-1}C &\Rightarrow a^{i-1}\underline{a}b^{i-1}Bc^{i-1}C && [4]\\
a^i b^{i-1}\underline{B}c^{i-1}C &\Rightarrow a^i b^{i-1}\underline{b}c^{i-1}C && [6]\\
a^i b^i c^{i-1}\underline{C} &\Rightarrow a^i b^i c^{i-1}\underline{c} && [7]
\end{aligned}
$$

Je zřejmé, že k výsledné větě $a^i b^i c^i$ vedla nepřímá derivace v souladu s definicí 4.8.2.

Kapitola 5

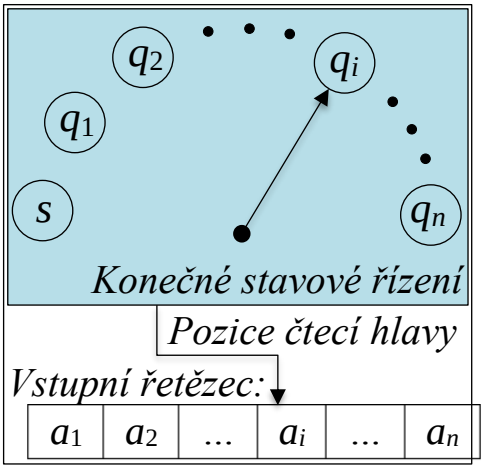
Automaty

Tato kapitola se věnuje základním typům automatů, jakými jsou konečné a zásobníkové automaty. Dále jsou představeny regulované automaty, jako rozšíření zásobníkových automatů. Závěr kapitoly je věnován speciálnímu typu automatu, označovaného jako hluboký zásobníkový automat, který se oproti zásobníkovému automatu odlišuje možností provádět některé zásobníkové operací i mimo vrcholu zásobníku. Kapitola poskytuje formální definici zmíněných automatů a obsahuje ucelenou myšlenku jejich funkcionality. Využití zde formálně definovaných pojmů nachází uplatnění v následujících kapitolách, věnovaných překladačům, převodníkům, (regulovanému) syntaxí řízenému překladu a regulovaným metodám syntaktické analýzy. Tato kapitola je inspirována a čerpá informace z [8], [9] a [11].

Automaty jsou dalšími z formálních aparátů, kterými můžeme ověřovat, zdali je daný řetězec obsažen v nějakém jazyce. Automaty reflektují funkcionalitu gramatik uvedených v předchozí kapitole. Místo derivačních kroků však automaty provádějí přechody. Základní myšlenkou automatů je stavové řízení reagující na vstupní řetězec. Posloupností takových reakcí (přechodů mezi stavy) se simuluje postup přijetí (resp. odmítnutí) daného vstupního řetězce.

5.1 Konečné automaty

Konečné automaty jsou strukturou s vnitřním stavovým řízením a konečnou vstupní páskou, na které je zapsán vstupní řetězec. Vnitřní stavové řízení je složeno pouze z konečné množiny stavů s vyznačeným aktuálním stavem. Mimo této množiny konečný automat nedisponuje žádnou jinou formou paměti. Na vstupní pásce je vyznačena pozice čtecí hlavy, jak je patrné na obrázku 5.1. Uvedené definice 5.1.1 až 5.1.5 byly do velké míry inspirovány definicemi uvedenými v [8] a [11].



Obrázek 5.1: Vnitřní reprezentace konečného automatu

5.1.1 Definice

Konečným automatem M rozumíme pětici $M = (Q, \Sigma, P, s, F)$, kde:

- Q je *konečná množina stavů*
- Σ je *konečná abeceda vstupních symbolů*
- P je podmnožina kartézského součinu $Q \times \Sigma^* \times Q$ označovaná jako *přechodová funkce*
- $s \in Q$ je *počáteční stav*
- $F \subseteq Q$ je *konečná množina koncových stavů*

Prvky přechodové funkce budeme označovat, stejně jako v případě prvků množiny přepisovacích pravidel u gramatik, jako *přepisovací pravidla*.

Konečné automaty lze dále klasifikovat dle definice konkrétní přechodové funkce na:

- Deterministické konečné automaty
- Nedeterministické konečné automaty

Pro účely této práce uvažujeme deterministickým konečným automatem každý automat, pro nějž platí:

$$\forall (q_{11}, x_1, q_{12}), (q_{21}, x_2, q_{22}) \in P, (q_{11} = q_{21} \wedge x_1 = x_2) \Rightarrow q_{12} = q_{22}$$

a neobsahuje žádné ε-přechody (viz 5.1.3).

5.1.2 Konfigurace

Konfigurace konečného automatu je reprezentace aktuálního stavu daného automatu a musí pokrývat aktuální stav stavového řízení i aktuálně doposud nezpracovanou část vstupního řetězce. Ve shodě s definicí 5.1.1 ji lze vyjádřit jako prvek kartézského součinu uložených prvků, tedy $\chi \in Q \times \Sigma^*$. V některých literaturách se konfigurace automatů zapisuje jako řetězec, což je možné jen tehdy, jsou-li množiny sousedících komponentních prvků disjunktní. Zápis n-ticemi byl zvolen z důvodu jednotnosti zápisu v rámci této kapitoly, neboť některé sousedící komponentní prvky konfigurací některých dále uvedených typů automatů nejsou disjunktní a nebylo by možno tyto prvky v řetězci jednoznačně implicitně rozlišit.

5.1.3 Přechod mezi konfiguracemi

Základní funkcí konečného automatu je přechod mezi konfiguracemi. Automat přejde z konfigurace $\chi_1 = (q_1, xy)$, do konfigurace $\chi_2 = (q_2, y)$, $x \in \Sigma \cup \{\varepsilon\}$, $y \in \Sigma^*$, $q_1, q_2 \in Q$, pokud $p = (q_1, x, q_2) \in P$. Tuto skutečnost zapisujeme $\chi_1 \Rightarrow \chi_2 [p]$, případně pouze $\chi_1 \Rightarrow \chi_2$. Je-li $x = \varepsilon$, není ze vstupní pásky přečten žádný symbol a takový přechod označujeme jako *ε-přechod*. V opačném případě dojde k přečtení symbolu x ze vstupní pásky, posunutí čtecí hlavy na symbol následující a stavové řízení se po vykonání přechodu nachází ve stavu q_2.

5.1.4 Posloupnost přechodů

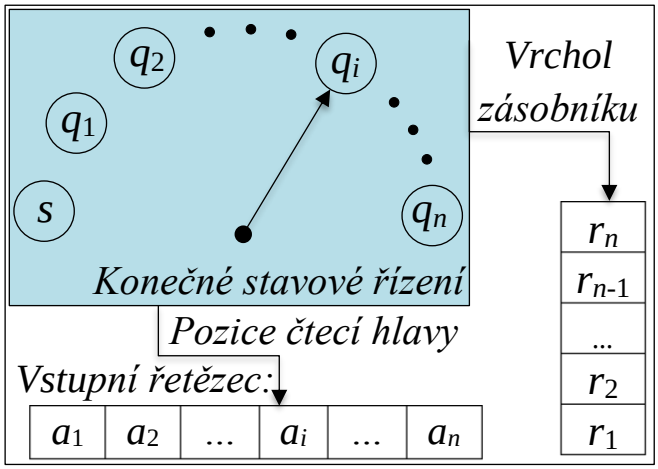
Posloupností přechodů rozumíme takovou posloupnost konfigurací $\chi_1, \chi_2, ..., \chi_i$, $i \geq 2$, že $\chi_1 \Rightarrow \chi_2$, $\chi_2 \Rightarrow \chi_3$, ..., $\chi_{i-1} \Rightarrow \chi_i$ a $\chi_1 = (q_1, x)$, $\chi_i = (q_i, y)$, $q_1, q_i \in Q$, $x, y \in \Sigma^*$. Tuto skutečnost zapisujeme $\chi_1 \Rightarrow^* \chi_i$. Speciální posloupností přechodů je i případ pro $i = 1$, tedy neprovedení žádného přechodu. Neprovedení žádného přechodu má smysl mimo jiné tehdy, pokud je počáteční stav zároveň stavem koncovým a jazyk automatu obsahuje prázdný řetězec.

5.1.5 Přijímaný jazyk

Konečný automat M přijme vstupní řetězec, pokud existuje posloupnost přechodů $(s, x) \Rightarrow^* (q_f, \varepsilon)$, $x \in \Sigma^*$, $q_f \in F$. Neformálně řečeno, k přijetí řetězce dojde tehdy, pokud aktuální stav automatu po přečtení posledního symbolu vstupního řetězce bude patřit mezi koncové stavy, nebo do takového stavu bude moci vykonat přechod nějakou posloupností ε-přechodů. *Přijímaným jazykem* $L(M)$ je označována množina všech řetězců, které daný automat přijímá.

5.2 Zásobníkové automaty

Zásobníkové automaty se strukturálně podobají konečným automatům z předchozí kapitoly. Jejich stavové řízení je však rozšířeno o strukturu nazývanou *zásobník*. Celá vnitřní struktura zásobníkového automatu je znázorněna na obrázku 5.2. Uvedené definice 5.2.1 až 5.2.5 byly do velké míry inspirovány definicemi uvedenými v [8] a [11].



Obrázek 5.2: Vnitřní reprezentace zásobníkového automatu

5.2.1 Definice

Zásobníkový automat M je sedmice $M = (Q, \Sigma, \Gamma, R, s, S, F)$, kde:

- Q je *konečná množina stavů*
- Σ je *konečná abeceda vstupních symbolů*
- Γ je *konečná zásobníková abeceda*
- R je konečná podmnožina kartézského součinu $\Gamma \times Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^* \times Q$ označovaná jako *přechodová funkce*
- $s \in Q$ je *počáteční stav*
- $S \in \Gamma$ je *počáteční symbol na zásobníku*
- $F \subseteq Q$ je *konečná množina koncových stavů*

V literatuře se často zavádí speciální symbol *$* (někdy označován také symbolem #) označující dno zásobníku. Tento symbol bývá implicitně obsažen v zásobníkové abecedě, ale zpravidla nebývá obsažen v abecedě vstupních symbolů. V následujícím textu je symbol *$* využit pro jednoznačnou specifikaci dna zásobníku.

Podobně, jako v případě konečných automatů rozlišujeme zásobníkové automaty na deterministické a nedeterministické. Deterministickým zásobníkovým automatem je takový zásobníkový automat, který splňuje následující podmínky:

- $\forall q, q_1, q_2 \in Q, a \in \Sigma \cup \{\varepsilon\}, x \in \Gamma, x_1, x_2 \in \Gamma^*, ((x, q, a, q_1, x_1) \in P \wedge (x, q, a, q_2, x_2) \in P) \Rightarrow q_1 = q_2 \wedge x_1 = x_2$
- $\forall q, q_1, q_2 \in Q, a \in \Sigma, x \in \Gamma, x_1, x_2 \in \Gamma^*, (x, q, \varepsilon, q_1, x_1) \in P \Rightarrow (x, q, a, q_2, x_2) \notin P$

První z uvedených podmínek zajišťuje, že pro každou kombinaci stavu a vstupního symbolu existuje nejvýše jeden možný přechod. Smysl druhé podmínky spočívá v zabránění přechodu automatu přečtením symbolu ze vstupní pásky, je-li možno provést přechod bez přečtení symbolu ze vstupní pásky.

5.2.2 Konfigurace

Konfigurace musí obsáhnout všechny informace o aktuálním stavu automatu. Konkrétně tedy stavové řízení (aktuální stav), doposud nepřečtenou část vstupního řetězce a obsah zásobníku. Formálně jde tedy o prvek kartézského součinu $\chi \in \Gamma^* \times Q \times \Sigma^*$.

5.2.3 Přechod mezi konfiguracemi

Zásobníkový automat vykoná přechod z konfigurace $\chi_1 = (yw\$, q_1, x_1x)$ do konfigurace $\chi_2 = (zw\$, q_2, x)$, $x_1 \in (\Sigma \cup \{\varepsilon\})$, $x \in \Sigma^*$, $y \in \Gamma$, $q_1, q_2 \in Q$, $w, z \in \Gamma^*$, pokud $p = (y, q_1, x, z, q_2) \in R$. Tuto skutečnost zapisujeme $\chi_1 \Rightarrow \chi_2 [p]$, případně pouze $\chi_1 \Rightarrow \chi_2$. Podobně jako u konečného automatu, pokud $x_1 = \varepsilon$, není ze vstupní pásky přečten žádný symbol. V opačném případě je symbol x_1 ze vstupní pásky přečten a čtecí hlava se přesune na symbol následující. Dále je ze zásobníku vyjmut symbol y, na zásobník zapsán řetězec z a stavové řízení se po vykonání přechodu nachází ve stavu q_2.

5.2.4 Posloupnost přechodů

Posloupnost přechodů, podobně jako u konečného automatu, je jistá sekvence po sobě následujících konfigurací v souladu s definicí 5.2.3. Formálně jde tedy o takovou posloupnost konfigurací $\chi_1, \chi_2, ..., \chi_i\ [p_1, p_2, \dots p_{i-1}]$, $i \geq 2$, že $\chi_1 \Rightarrow \chi_2\ [p_1], \chi_2 \Rightarrow \chi_3\ [p_2], ..., \chi_{i-1} \Rightarrow \chi_i\ [p_{i-1}], p_1, p_2, \dots p_{i-1}, \in R$. Tuto posloupnost přechodů označujeme $\chi_1 \Rightarrow^* \chi_i\ [p_1, p_2, \dots p_{i-1}]$, případně pouze $\chi_1 \Rightarrow^* \chi_i$. Speciální posloupností přechodů je i případ pro $i = 1$, tedy neprovedení žádného přechodu.

5.2.5 Přijímaný jazyk

Oproti konečnému automatu je zde možno definovat více způsobů, kterými automat vstupní řetězec přijme, a to:

1: *Přijetí vyprázdněním zásobníku*, tj. pokud automat přejde z konfigurace $\chi_1 = (S\$, s, w)$, $w \in \Sigma^*$ posloupností přechodů do konfigurace $\chi_2 = (\$, q, \varepsilon)$, $q \in Q$.
2: *Přijetím přechodem do koncového* stavu, tj. pokud automat přejde z konfigurace $\chi_1 = (S\$, s, w)$, $w \in \Sigma^*$ posloupností přechodů do konfigurace $\chi_2 = (x\$, q_f, \varepsilon)$, $q_f \in F$, $x \in \Gamma^*$.
3: *Přijetí vyprázdněním zásobníku a přechodem do koncového stavu*, tj. pokud automat přejde z konfigurace $\chi_1 = (S\$, s, w)$, $w \in \Sigma^*$ posloupností přechodů do konfigurace $\chi_2 = (\$, q_f, \varepsilon)$, $q_f \in F$.

Přijímaný jazyk $L(M, i)$ je množina všech řetězců, které automat odpovídajícím způsobem přijme. Existuje tudíž stejný počet typů přijímaných jazyků zásobníkovým automatem, jako je počet typů přijímaných řetězců týmž automatem. Třída jazyků přijímaných zásobníkovými automaty se liší v závislosti na tom, jestli se jedná o deterministický nebo nedeterministický automat. Lze však předpokládat, že vyjadřovací síla zásobníkových automatů je vyšší, než v případě regulárních automatů (formální důkaz viz [4]).

5.3 Regulované automaty

Regulovanými automaty rozumíme takové automaty, jejichž přechody mezi konfiguracemi jsou regulovány jiným aparátem, podobně jako některé gramatiky v předchozích kapitolách. Řídicím aparátem tedy může být například konečná relace, jazyk nad množinou prvků přechodové funkce. V této kapitole uvažujeme řídicím aparátem jazyk nad množinou prvků přechodové funkce, tedy jistou obdobu maticové gramatiky. Definice 5.3.1 až 5.3.5 byly s drobnými úpravami převzaty z [13] a [5].

5.3.1 Definice

Regulovaný automat H je dvojice $H = (M, \Xi)$, kde:

- $M = (Q, \Sigma, \Gamma, R, s, S, F)$ je zásobníkový automat definován stejným způsobem, jako v definici 5.2.1
- Ξ je jazyk nad množinou prvků přechodové funkce

Nechť Ψ je abeceda, jejímiž symboly jsou jednoznačné identifikátory prvků přechodové funkce R zásobníkového automatu M. Potom $\Xi \subseteq \Psi^*$, kde Ψ^* je množina všech řetězců nad abecedou Ψ. V závislosti na třídě jazyka Ξ lze dále rozlišovat vyjadřovací sílu regulovaného zásobníku.

Determinismus regulovaných automatů je v době psaní této práce jedním z otevřených problémů (viz [5]). Dalším otevřeným problémem týkající se regulovaných automatů je zjištění jejich vyjadřovací síly. Předpokládá se, že jejich vyjadřovací síla je větší než v případě tradičních zásobníkových automatů. Důkazem je příklad 7.3.1, který regulovaným převodníkem ukazuje překlad jazyka, který není bezkontextový. Odebráním výstupní části tohoto převodníku se převodník stává zde uvedeným regulovaným zásobníkovým automatem.

5.3.2 Konfigurace

Konfigurací regulovaného automatu χ rozumíme konfiguraci χ zásobníkového automatu M. Jelikož, jak je patrné z následující kapitoly, řídicí jazyk neobsahuje žádný dodatečný údaj popisující aktuální stavu automatu, není nutno do konfigurace zahrnout jakékoliv další údaje.

5.3.3 Přechod mezi konfiguracemi

Přechodem mezi konfiguracemi rozumíme stejný přechod $\chi_1 \Rightarrow \chi_2 [p]$, jako je definován v 5.2.3.

5.3.4 Posloupnost přechodů

Posloupností přechodů regulovaného automatu rozumíme takovou posloupnost konfigurací $X = \chi_1, \chi_2, \ldots, \chi_i [p_1, p_2, \ldots p_{i-1}]$, že X je posloupností konfigurací zásobníkového automatu M v souladu s definici 5.2.4 a $p_1p_2\ldots p_{i-1} \in \Xi$. Neformálně řečeno, jedná se o takovou posloupnost přechodů automatu M, která je dána užitím takové posloupnosti prvků přechodové funkce, že konkatenací jednoznačných identifikátorů těchto prvků je řetězec z řídicího jazyka Ξ.

5.3.5 Přijímaný jazyk

Podobně, jako i zásobníkový automat, regulovaný zásobníkový automat definuje 3 způsoby přijímání řetězců stejným způsobem, jako v 5.2.5. Přijímaný jazyk $L(M, \Xi, i)$ je potom množina všech řetězců, které regulovaný zásobníkový automat odpovídajícím způsobem přijme.

5.4 Hluboké zásobníkové automaty

Pro analýzu některých gramatik (viz např. 4.7) je potřeba ve větné formě expandovat i některé neterminální symboly, kterým v dané větné formě mohou zleva předcházet i jiné neterminální symboly. Tato myšlenka dala vzniknout speciálnímu typu zásobníkového automatu, který se označuje jako *hluboký zásobníkový automat*. Tento typ zásobníkového automatu byl představen v [9], odkud tato kapitola čerpá všechny zde uvedené definice a informace. Hluboký zásobníkový automat nachází uplatnění mimo jiné i v regulovaných metodách syntaktické analýzy, jak je ukázáno v kapitole 10 této práce.

5.4.1 Definice

Hluboký zásobníkový automat M je sedmice $M = (Q, \Sigma, \Gamma, R, s, S, F)$, kde:

- Q, s, S, F je stejného významu, jako v definici 5.2.1
- Σ je *konečná abeceda vstupních symbolů*, $\Sigma \subseteq \Gamma$
- Γ je *konečná zásobníková abeceda*, $\Gamma - \Sigma = \{\$\}$
- R je konečná podmnožina kartézského součinu $\mathbb{N} \times \Gamma \times Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^* \times Q$ označovaná jako *přechodová funkce*

První prvky přepisovacích pravidel nazýváme *hloubkou přepisovacího pravidla*. Dále platí, že $\Sigma \subseteq \Gamma$, tedy každý symbol vstupní abecedy může být uložen na zásobník. Přechody mezi jednotlivými konfiguracemi hlubokého zásobníkového automatu mohou být dvojího typu, a to buď *expanze*, nebo *redukce*. Definice obou těchto typů přechodů jsou definovány níže. Posloupnost přechodů je definována stejným způsobem, jako v definici 5.2.3, jen s využitím redukcí a expanzí místo přechodů mezi konfiguracemi. Podobně, jako tomu bylo u zásobníkových automatů, rozlišujeme stejné typy přijímaných jazyků hlubokým zásobníkových automatů, jako v definici přijímaného jazyka 5.2.5.

5.4.2 Redukce

Pokud hluboký zásobníkový automat M přejde z konfigurace $\chi_1 = (az\$, q, aw)$ do konfigurace $\chi_2 = (z\$, q, w)$, kde $q \in Q$, $a \in \Sigma$, $w \in \Sigma^*$, $z \in \Gamma^*$, označujeme tento přechod jako *redukci* a zapisujeme jako $\chi_1 \ {}_r\Rightarrow \chi_2$. Redukce neformálně označuje odebrání symbolu ze vstupu, pokud se tento symbol nachází rovněž na vrcholu zásobníku.

5.4.3 Expanze

Pokud hluboký zásobníkový automat M přejde z konfigurace $\chi_1 = (uXy\$, q_1, w)$ do konfigurace $\chi_2 = (uYy\$, q_2, w)$, kde $q_1, q_2 \in Q$, $w \in \Sigma^*$, $X \in N$, $u, Y, y \in \Gamma^*$, $(m, X, Y) \in R \wedge (\#_X u = m\text{-}1 \vee \#_X y = 0)$, značíme takový přechod $\chi_1 \ {}_e\Rightarrow \chi_2$ a označujeme jej jako *expanze*. Myšlenka tohoto přechodu spočívá v nahrazení výskytu levé strany přepisovacího pravidla jeho pravou stranou na pozici m-tého výskytu (číslovaného od 1) levé strany od vrcholu zásobníku. Pokud takových výskytů je méně než m, nahradí se poslední takový výskyt. Argument přechodu m označuje hloubku daného přechodu.

Kapitola 6

Převodníky

Cílem této kapitoly je seznámit čtenáře se základními typy převodníků a formálně zavést regulované převodníky založené na kombinaci regulovaných automatů a zásobníkových převodníků. Tato kapitola čerpá informace z [8] a [5].

Funkce převodníků by se dala obecně charakterizovat jako zobrazení řetězce z jazyka nad nějakou abecedou na řetězec z jazyka nad jinou abecedou. Všechny zde uvedené typy převodníků jsou založené na automatech uvedených v předchozí kapitole. Intuitivně se jedná o obohacení přechodové funkce o výstupní část. Lze tedy uvažovat vyjadřovací sílu a problémy determinismu uvedených typů převodníků jako ekvivalentní s problémy automatů, na kterých byly založeny. Příklady využití některých zde uvedených převodníků jsou uvedeny v kapitole 7.

6.1 Konečné převodníky

Konečné převodníky jsou úzce spjaty s konečnými automaty. Rozdílným definiční prvkem je výstupní abeceda, a tedy schopnost v rámci přechodů mezi konfiguracemi generovat výstupní řetězce. Definice 6.1.1 až 6.1.5 byly s drobnými úpravami převzaty z [8].

6.1.1 Definice

Konečným převodníkem M rozumíme pětici $M = (Q, \Sigma, P, s, F)$, kde:

- Q je *konečná množina stavů*
- Σ je *konečná abeceda symbolů*, $\Sigma \cap Q = \emptyset$, $\Sigma \cap \{|\} = \emptyset$, $\Sigma = I \cup O$, kde I je konečná *vstupní abeceda* a O je konečná *výstupní abeceda*
- P je konečná podmnožina kartézského součinu $Q\,(I \cup \{\varepsilon\}) \times Q\,O^*$ označovaná jako *přechodová funkce*
- $s \in Q$ je *počáteční stav*
- $F \subseteq Q$ je *konečná množina koncových stavů*

6.1.2 Vstupní konečný automat

Nechť M je konečný převodník $M = (Q, \Sigma, P, s, F)$, potom $M_I = (Q, I, P_I, s, F)$ je *vstupní konečný automat*, kde:

- I je *konečná vstupní abeceda převodníku M*
- P_I je konečná podmnožina kartézského součinu $Q\,(I \cup \{\varepsilon\}) \times Q$ označovaná jako *vstupní přechodová funkce*

Pro každý prvek přechodové funkce $(q_1a, q_2) \in P_I$, platí, že $\exists x \in O^*$: $(q_1a, q_2x) \in P$, tedy že existuje odpovídající výstupní řetězec převodníku M.

6.1.3 Konfigurace

Pro vyjádření konfigurací konečného převodníku, oproti vyjádření konfigurací n-ticemi u konečných automatů v kapitole 5.3, volíme řetězec obsahující aktuální stav daného převodníku. Formálně jde tedy o řetězec $\chi \in QI^*\{|\}O^*$. Je zde zaveden speciální symbol | vyjadřující oddělovač mezi vstupním a výstupním řetězcem pro lepší názornost překladu.

6.1.4 Přechod mezi konfiguracemi

Převodník přejde z konfigurace $\chi_1 = q_1aw|y$, do konfigurace $\chi_2 = q_2w|yz$, $a \in I \cup \{\varepsilon\}$, $z \in O$, $w \in I^*$, $y \in O^*$, $q_1, q_2 \in Q$, pokud $p = (q_1\ a, q_2\ z) \in P$. Tuto skutečnost zapisujeme $\chi_1 \Rightarrow \chi_2\ [p]$, případně pouze $\chi_1 \Rightarrow \chi_2$. Přechod mezi konfiguracemi je možno chápat jako binární relaci definovanou nad dvojicemi konfigurací. Symbolem $\Rightarrow^*$ budeme označovat reflexivní a tranzitivní uzávěr této relace (podobně jako posloupnost přechodů automatů). Přechod mezi konfiguracemi vyjadřuje přechod konečného automatu, který na výstup zapíše symbol z výstupní abecedy O.

6.1.5 Překlad

Překladem konečného převodníku intuitivně rozumíme přijetí vstupního řetězce vstupním konečným automatem, a tudíž vygenerování řetězce nad výstupní abecedou. Jedná se tedy o množinu dvojic vstupních a k nim odpovídajících výstupních řetězců. Formálně je překlad $T(M)$ konečného převodníku M definován následovně:

$$T(M) = \{(x, y) \in I \times O : sx| \Rightarrow^* f|y, f \in F\}$$

6.2 Zásobníkové převodníky

Zásobníkové převodníky oproti konečným převodníkům, podobně jako zásobníkové automaty vůči konečným automatům, je rozšiřují o strukturu zásobníku, což se korespondujícím způsobem odráží v jejich definici. Definice 6.2.1 až 6.2.5 byly s drobnými úpravami převzaty z [8].

6.2.1 Definice

Zásobníkovým převodníkem M typu i rozumíme šestici $M = (Q, \Sigma, P, s, S, F)$, kde:

- Q je *konečná množina stavů*
- Σ je *konečná abeceda symbolů*, $\Sigma \cap Q = \emptyset$, $\Sigma \cap \{|\} = \emptyset$, $\Sigma = I \cup O \cup \Gamma$, kde I je konečná *vstupní abeceda*, O je konečná *výstupní abeceda*, Γ je *zásobníková abeceda*
- P je konečná podmnožina kartézského součinu $\Gamma\ Q\ (I \cup \{\varepsilon\}) \times \Gamma^*\ Q\ O^*$ označovaná jako *přechodová funkce*
- $s \in Q$ je *počáteční stav*
- $S \in \Gamma$ je *počáteční symbol na zásobníku*
- $F \subseteq Q$ je *konečná množina koncových stavů*

6.2.2 Vstupní zásobníkový automat

Nechť $M = (Q, \Sigma, P, s, F)$ je zásobníkový převodník typu i, potom $M_I = (Q, I, \Gamma, P_I, s, S, F)$ je vstupní zásobníkový automat typu i kde:

- $I \cup \Gamma$ je *konečná vstupní abeceda převodníku M*
- S je *počáteční symbol na zásobníku*
- P_I je konečná podmnožina kartézského součinu $\Gamma\, Q\, (I \cup \{\, \varepsilon \,\}) \times \Gamma^*\, Q$ označovaná jako *vstupní přechodová funkce*

Pro všechny prvky vstupní přechodové funkce $(rq_1a, wq_2) \in P_I$ platí:

$$\exists x \in O^* : (rq_1a, wq_2x) \in P$$

6.2.3 Konfigurace

Podobně jako u zásobníkových automatů v kapitole 5.2 konfigurací zásobníkového převodníku rozumíme řetězec obsahující aktuální stav daného převodníku. Formálně jde tedy o řetězec $\chi \in \Gamma^* Q I^* \{|\} O^*$. Oproti konfiguraci zásobníkových automatů je zde ale zaveden speciální symbol | vyjadřující oddělovač mezi vstupním a výstupním řetězcem.

6.2.4 Přechod mezi konfiguracemi

Převodník přejde z konfigurace $\chi_1 = xu\$q_1aw|y$, do konfigurace $\chi_2 = vu\$q_2w|yz$, $a \in I \cup \{\varepsilon\}$, $u, v \in \Gamma^*$, $x \in \Gamma$, $z \in O$, $w \in I^*$, $y \in O^*$, $q_1, q_2 \in Q$, pokud $p = (xq_1\, a, vq_2\, z) \in P$. Tuto skutečnost zapisujeme $\chi_1 \Rightarrow \chi_2\, [p]$, případně pouze $\chi_1 \Rightarrow \chi_2$. Přechod mezi konfiguracemi má stejnou funkci jako v definici 6.1.4.

6.2.5 Překlad

Překladem zásobníkového převodníku, podobně jako v kapitole 6.1.5 rozumíme množinu dvojic vstupních a k nim odpovídajících výstupních řetězců. Shodně se zásobníkovými automaty definují zásobníkové převodníky více možných překladů, v závislosti na typu vstupního zásobníkového automatu. Formálně je překlad $T(M, i)$ zásobníkového převodníku M typu i definován následovně:

- $T(M, 1) = \{(x, y) \in I \times O : S\$sx| \Rightarrow^* \$q|y, q \in Q\}$
- $T(M, 2) = \{(x, y) \in I \times O : S\$sx| \Rightarrow^* z\$f|y, z \in \Gamma^*, f \in F\}$
- $T(M, 3) = \{(x, y) \in I \times O : S\$sx| \Rightarrow^* \$f|y, f \in F\}$

6.3 Regulované převodníky

Regulované převodníky je možno definovat mnoha způsoby, které nemusí být vždy ekvivalentní [12]. Podobně, jako regulované automaty jsou v tomto textu chápány jako rozšíření zásobníkových automatů, regulované převodníky jsou v rámci tohoto textu chápány jako rozšíření zásobníkových převodníků. Regulovaný převodník je tedy takový zásobníkový převodník, jehož přechody mezi konfiguracemi jsou řízeny nějakým dalším formálním aparátem. V rámci této práce

byl jako řídicí aparát vybrán řídicí jazyk nad prvky přechodové funkce. Jedná se tedy o jistou kombinaci zásobníkových převodníků a regulovaných automatů uvedených v předchozích kapitolách. Podobným způsobem by se však dala definovat celá řada dalších regulovaných převodníků.

6.3.1 Definice

Zásobníkovým převodníkem H typu i rozumíme dvojici $H = (M, \Xi)$, kde:

- $M = (Q, \Sigma, P, s, S, F)$ je zásobníkový převodník typu i dle definice z kapitoly 6.2.1.
- Ξ je jazyk nad množinou prvků přechodové funkce

Nechť Ψ je abeceda, jejímiž symboly jsou jednoznačné identifikátory prvků přechodové funkce P zásobníkového převodníku M. Potom $\Xi \subseteq \Psi^*$, kde Ψ^* je množina všech řetězců nad abecedou Ψ.

6.3.2 Posloupnost přechodů

Je účelné zavést pojem posloupnost přechodů tak, aby pokrýval potřeby řídicího aparátu. Posloupností přechodů regulovaného převodníku H rozumíme posloupnost konfigurací zásobníkového převodníku M $\chi_1, \chi_2, \ldots, \chi_i$ $[p_1, p_2, \ldots p_{i-1}]$ pro $i \geq 2$, že $\chi_1 \Rightarrow \chi_2$ $[p_1]$, $\chi_2 \Rightarrow \chi_3$ $[p_2]$, ..., $\chi_{i-1} \Rightarrow \chi_i$ $[p_{i-1}]$ a $p_1 p_2 \ldots p_{i-1} \in \Xi$. Speciálním případem je i situace pro $i = 1$, tedy neprovedení žádného přechodu. Posloupnost přechodů značíme $\chi_1 \Rightarrow^* \chi_i$. Myšlenkou této definice je vymezit definici přechodu regulovaného převodníku tak, aby pokrývala posloupnost přechodů zásobníkového převodníku.

6.3.3 Překlad

Překladem regulovaného převodníku, podobně jako v kapitole 6.2.5, rozumíme množinu dvojic vstupních a k nim odpovídajících výstupních řetězců. Analogicky definujeme překlad $T(H, i)$ regulovaného převodníku typu i jako překlad $T(M, i)$ zásobníkového převodníku z kapitoly 6.2, a to následovně:

- $T(H, 1) = \{(x, y) \in I \times O : S\$sx| \Rightarrow^* \$q|y, q \in Q\}$
- $T(H, 2) = \{(x, y) \in I \times O : S\$sx| \Rightarrow^* z\$f|y, z \in \Gamma^*, f \in F\}$
- $T(H, 3) = \{(x, y) \in I \times O : S\$sx| \Rightarrow^* \$f|y, f \in F\}$

Za podotknutí stojí zdůraznění, že na rozdíl od definice překladu 6.2.5 zde symbol $\Rightarrow^*$ neoznačuje uzávěr relace, nýbrž posloupnost přechodů z kapitoly 6.3.2.

Kapitola 7

Syntaxí řízený překlad

Tato kapitola zavádí formální prostředky pro syntaxí řízený překlad a uvádí dva ilustrační příklady konkrétního překladu založeném na převodnících. Závěr kapitoly je věnován regulovanému syntaxí řízenému překladu za využití regulovaných převodníků zavedených v předchozích kapitolách. Syntaxí řízený překlad lze realizovat speciálními překladovými gramatikami, nebo převodníky, přičemž překladovým gramatikám zde není věnována žádná další pozornost. Informace uvedené v rámci této kapitoly jsou čerpány z [10].

7.1 Formální překlad

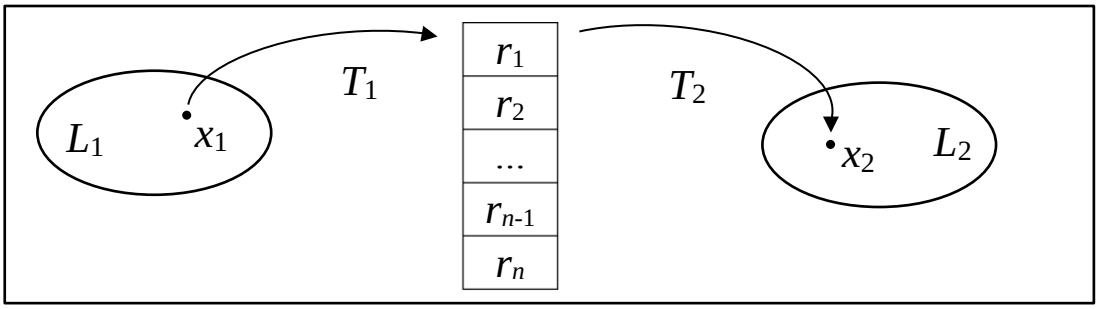
Formálním překladem T z jazyka L_1 do jazyka L_2 rozumíme podmnožinu kartézského součinu $T \subseteq L_1 \times L_2$, tedy binární relaci (w_1, w_2) takovou, že $w_1 \in L_1$ a $w_2 \in L_2$.

Přestože podle výše uvedené definice může existovat řetězec v jazyce L_2, který je v relaci formálního překladu s řetězcem z jazyka L_1, jeho přímé hledání je z praktického hlediska příliš náročné. Vzhledem k tomu, že však dokážeme pro každý řetězec patřící do jazyka daného bezkontextovou gramatikou (případně nějakou její omezenou verzí dle konkrétní metody syntaktické analýzy, viz kapitola 9) nelézt posloupnost pravidel vedoucí k úspěšné simulaci konstrukce derivačního stromu (nebo některého jeho popisu), dokážeme realizovat překlad z tohoto řetězce na tento derivační strom a následně z tohoto derivačního stromu do cílového jazyka.

7.2 Definice

Syntaxí řízený překlad T lze tedy formálně zapsat jako složením formálních překladů následujícím způsobem:

- $T = T_1 \cdot T_2$, kde T_1 je formální překlad řetězce ze zdrojového jazyka na derivační strom (resp. nějaký popis jeho konstrukce) a T_2 je formální překlad z derivační stromu na řetězec z cílového jazyka.



Obrázek 7.1: Syntaxí řízený překlad

Myšlenka uvedená v předchozí definici je zobrazena i na obrázku 7.1, kde jako reprezentace derivačního stromu slouží jeho popis ve formě posloupnosti přepisovacích pravidel, jakou může být například levý rozbor (viz 8.3). Samotný překlad T_1 může být realizován algoritmy syntaktické analýzy, a překlad T_2 obohacením přepisovacích pravidel gramatiky o výstupní část. Základem algoritmu syntaktické analýzy uvedeného v této práci je automat. Obohacením prvků přechodové funkce tohoto automatu o výstupní část vzniká převodník (viz kapitola 6), který realizuje jak překlad T_1, tak překlad T_2.

Základ syntaxí řízeného překladu je založen na myšlence nalezení překladu T_1. K tomuto účelu je možno využít automaty popsané v kapitole 5. Na těchto automatech se v praxi navrhují algoritmy hledání překladu T_1, přičemž proces samotného hledání se označuje jako *syntaktická analýza*. V následujících kapitolách jsou rozebrány existující algoritmy syntaktické analýzy a návrh nových metod regulované syntaktické analýzy jako modelu regulovaného syntaxí řízeného překladu z kapitoly 7.3.

7.2.1 Příklad

Jako realizaci syntaxí řízeného překladu uvažujme zásobníkový převodník z kapitoly 6.2. Uvažujme dále bezkontextový jazyk L nad abecedou $\Sigma = \{a, b\}$ obsahující pouze neprázdné řetězce a formální překlad T definovaný následujícím způsobem:

$$T = \{(w_1, w2): w_1, w_2 \in L, w_2 = reversal(w_1)\}$$

tedy překlad řetězců na řetězce v obráceném pořadí a vstupní řetězec $x = abbab$. Dále uvažujme zásobníkový převodník $M = (Q, \Sigma, P, \$, A, F)$ typu 3, který překlad T realizuje. Obsah jednotlivých množin je následující:

$Q = \{A, B, C\}$
$\Sigma = \{a, b\}$
$P = \{$ 1: ($\$Aa$, $\$aA$),
2: ($\$Ab$, $\$bA$),
3: (aAa, aaA),
4: (bAa, baA),
5: (aAb, abA),
6: (bAb, bbA),
7: (bA, bB),
8: (aA, aB),
9: (aB, Ba),
10: (bB, Bb),
11: ($\$B$, $\$C$), $\}$
$F = \{C\}$
Posloupnost konfigurací vedoucí k úspěšnému překladu:

$\$\underline{Aa}bbab\,\vert$	$\Rightarrow$	$\$\underline{aA}bbab\,\vert$	[1]
$\$\underline{aAb}bab\,\vert$	$\Rightarrow$	$\$\underline{abA}bab\,\vert$	[5]
$\$ab\underline{Abab}\,\vert$	$\Rightarrow$	$\$ab\underline{bbA}ab\,\vert$	[6]

$$
\begin{array}{rcll}
\$abb\underline{Aa}b\,| & \Rightarrow & \$abb\underline{aA}b\,| & [4] \\
\$abb\underline{aA}b\,| & \Rightarrow & \$abb\underline{abA}\,| & [5] \\
\$abba\underline{bA}\,| & \Rightarrow & \$abba\underline{bB}\,| & [7] \\
\$abba\underline{bB}\,| & \Rightarrow & \$abba\underline{B}\,|\underline{b} & [10] \\
\$abb\underline{aB}\,|b & \Rightarrow & \$abba\underline{B}\,|b\underline{a} & [9] \\
\$abb\underline{B}\,|ba & \Rightarrow & \$ab\underline{B}\,|ba\underline{b} & [10] \\
\$a\underline{bB}\,|bab & \Rightarrow & \$a\underline{B}\,|babb & [10] \\
\$\underline{aB}\,|babb & \Rightarrow & \$\underline{B}\,|babb\underline{a} & [9] \\
\$\underline{B}\,|babba & \Rightarrow & \$\underline{C}\,|babba & [11]
\end{array}
$$

Z výše uvedené posloupnosti přechodů je patrné, že převodník přeložil vstupní řetězec $x = abbab$ na řetězec $x' = babba$, přičemž $x' = reversal(x)$. Jedná se tudíž o platný překlad, lze psát $(x, y) \in T$.

7.3 Regulovaný syntaxí řízený překlad

Regulovaný syntaxí řízený překlad je v rámci této kapitoly chápan jako překlad realizován regulovaným převodníkem, kterému je věnována kapitola 6.3. Ve své podstatě se jedná pouze o jistou variantu klasického syntaxí řízeného překladu popsaného v předchozí kapitole. Oproti němu však dosahuje lepších výsledků, neboť jeho vstupní automat dokáže přijímat i některé jazyky, které nejsou bezkontextové. Převodník založen na takovém automatu je tudíž dokáže překládat, jak dokazuje příklad 7.3.1, který překládá jazyk L_1, který dle [1] není bezkontextový.

7.3.1 Příklad

Jako realizace regulovaného syntaxí řízeného překladu uvažujeme regulovaný převodník typu 3 z kapitoly 6.3. Uvažujme jazyky L_1 a L_2 nad abecedou $\Sigma = \{a, b, c\}$, $L_1 = \{a^i b^i c^i,\ i \geq 1\}$, $L_2 = \{(abc)^j, j \geq 1\}$, a překlad T definovaný následovně:

$$T = \{(w_1, w_2): w_1 \in L_1, w_2 \in L_2, w_1 = a^i b^i c^i, w_2 = (abc)^i, i \geq 1\}$$

Dále uvažujme regulovaný převodník $H = (M, \Xi)$ typu 3, který převod T realizuje, přičemž $M = (Q, \Sigma, P, A, \$, F)$, a $\Xi = \{12^m34^n5^n6, m, n \geq 0\}$ je lineární jazyk. Obsah uvedených množin je následující:

$Q = \{A, B, C\}$
$P = \{$ 1: ($\$Aa$, $\$aA$),
2: (aAa, aaA),
3: (aAb, B),
4: (aBb, B),
5: ($\$Bc$, $\$Babc$),
6: ($\$Bc$, $\$Cabc$) $\}$
$F = \{C\}$

Jako vstupní řetězec zvolíme libovolný řetězec z jazyka L_1, například *aaabbbccc*. Posloupnost konfigurací vedoucí k úspěšnému překladu je následující:

$\underline{\$Aa}aabbbccc\|$	⇒	$\$\underline{aA}aabbbccc\|$	[1]
$\$\underline{aAa}abbbccc\|$	⇒	$\$\underline{aaA}abbbccc\|$	[2]
$\$a\underline{aAa}bbbccc\|$	⇒	$\$a\underline{aaA}bbbccc\|$	[2]
$\$aa\underline{aAb}bbccc\|$	⇒	$\$aa\underline{B}bbccc\|$	[3]
$\$a\underline{aBb}bccc\|$	⇒	$\$a\underline{B}bccc\|$	[4]
$\$\underline{aBb}ccc\|$	⇒	$\$\underline{B}ccc\|$	[4]
$\underline{\$Bc}cc\|$	⇒	$\underline{\$B}cc\|\underline{abc}$	[5]
$\underline{\$Bc}c\|abc$	⇒	$\underline{\$B}c\|abc\underline{abc}$	[5]
$\underline{\$Bc}\|abcabc$	⇒	$\underline{\$C}\|abcabc\underline{abc}$	[6]

Z výše uvedené posloupnosti přechodů mezi konfiguracemi je patrné, že převodník přeložil vstupní řetězec $x = aaabbbccc$ na výstupní řetězec $x' = abcabcabc$ posloupností přechodů $122344556 \in \Xi$ a jedná se tedy o platný překlad. Lze psát $(x, x') \in T$.

Rovnost počtů symbolů *a* a *b* jsou porovnány mechanismem zásobníkového automatu. Rovnost počtu symbolů *b* a *c* (první symbol *b* a poslední symbol *c* jsou zpracovány jinak) je dána definicí jazyka Ξ, konkrétně částí udávající stejný počet aplikací pravidel 4 a 5. Je tedy zaručeno, že se přeloží pouze řetězce z jazyka L_1. Za povšimnutí stojí skutečnost, že levé strany pravidel 5 a 6 jsou totožné a vstupní automat tohoto převodníku tudíž není deterministický.

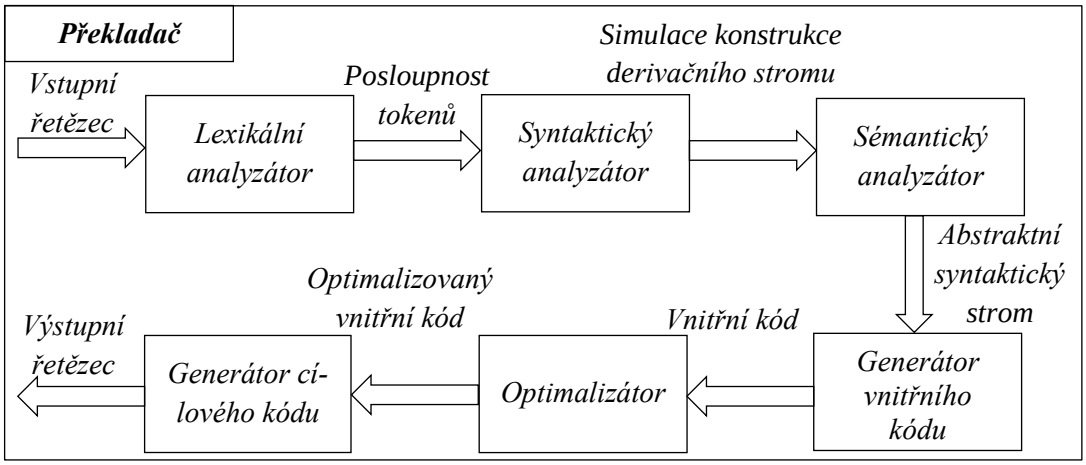
Kapitola 8

Překladače

V rámci této práce překladačem rozumíme algoritmus, který na svém vstupu pojme řetězec reprezentující program napsaný ve zdrojovém jazyce a jeho výstupem je buď řetězec obsahující programu v cílovém jazyce, nebo chyba. Z teoretického hlediska se tedy jedná o praktickou realizaci syntaxí řízeného překladu z předchozí kapitoly. V praxi se zpravidla jedná o program, který překládá zdrojové texty jednoho jazyka, na zdrojové texty obecně jiného jazyka. Typickým užitím moderních překladačů bývá překlad z nějakého vyššího programovacího jazyka do nižšího, ale výpočetně zaměřeného jazyka, jakým je například strojový kód. Úvod kapitoly se zaměřuje na základní stavební bloky překladače a stručný popis analytických funkcí. Cílem kapitoly je uvedení významu syntaktické analýzy do kontextu překladačů a představení dvou základních přístupů algoritmů syntaktických analyzátorů. Na informace uvedené v této kapitole navazuje kapitoly zaměřené na praktický návrh a implementaci navrženého systému. Tato kapitola čerpá informace z [10] a [11].

8.1 Struktura překladače

Struktura překladače nemusí být vždy pevně daná, nebo ani podobná struktuře užité v rámci této práce. Základní bloky překladače zde popisovaného jsou znázorněny na obrázku 8.1. Vstupem překladače bývá obvykle vstupní řetězec. Tento řetězec je předán lexikálnímu analyzátoru, který



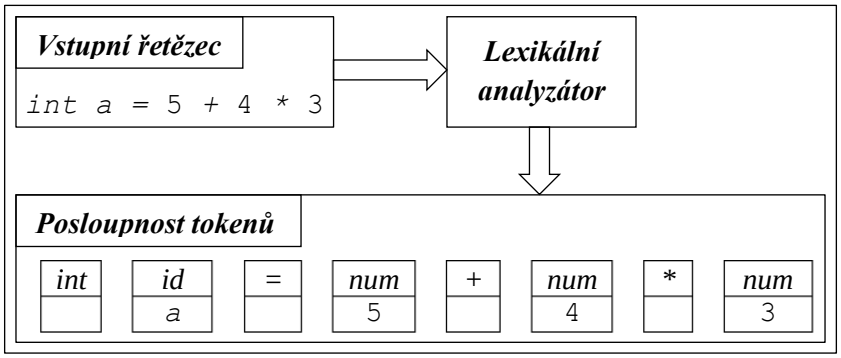
Obrázek 8.1: Struktura překladače

jej přeloží na posloupnost lexikálních jednotek zvaných *tokeny*. Posloupnost těchto tokenů tvoří vstup syntaktického analyzátoru, jehož úkolem je provést simulaci konstrukce derivačního stromu, a tedy ověřit, že takový strom existuje. Vedlejším produktem této analýzy je popis nalezeného derivačního stromu představující vstup sémantického analyzátoru, který jej přeloží na

abstraktní syntaktický strom. Tento strom je následně přeložen na vnitřní kód, který je optimalizován a převeden na výstupní řetězec.

8.2 Lexikální analýza

Lexikální analyzátor, někdy označovaný jako *Scanner*, je základním stavebním blokem moderních překladačů. Úkolem lexikálního analyzátoru je převést vstupní řetězec na posloupnost lexikálních jednotek nazvaných *tokeny*. Jeho funkcionalitu znázorňuje obrázek 8.2, na kterém analyzátor převádí vstupní řetězec na posloupnost tokenů. Ve spoustě programovacích jazyků jsou lexikální jednotky definovatelné právě pomocí regulárních výrazů. Proto u takových jazyků jako



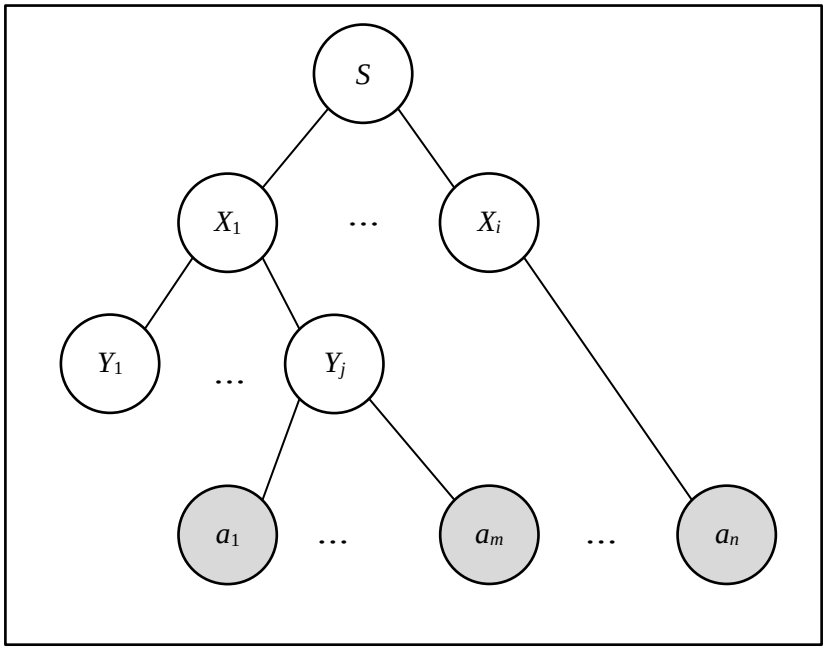
Obrázek 8.2: Ukázka principu funkce lexikálního analyzátoru

základ lexikálních analyzátorů je zpravidla využíván aparát pro rozpoznávání regulárních výrazů, jakým je například konečný automat popsaný v kapitole 5.1. Z tohoto důvodu může nastat i situace odmítnutí vstupního řetězce automatem, která samotný překlad ukončí lexikální chybou.

8.3 Syntaktická analýza

Syntaktický analyzátor, někdy nazýván jako *Parser*, je jádrem překladače, neboť jeho úkolem je provést simulaci sestavení derivačního stromu na základě zadané gramatiky. Derivačním stromem označujeme stromovou strukturu, jejímž kořenem je počáteční neterminální symbol dané gramatiky a listy tvoří vstupní řetězec, přičemž větvemi tohoto stromu jsou přepisovací pravidla užitá nepřímou derivací mezi kořenem tohoto stromu a jeho listy. Pokud se simulace konstrukce tohoto stromu zdaří, je vstupní řetězec (program) zapsán ze syntaktického hlediska korektně (patří do jazyka zadané gramatiky) Ukázka konkrétního derivačního stromu je např. v příkladu 9.7.1.

Formálně se syntaktické analyzátory snaží nalézt nepřímou derivaci mezi počátečním neterminálním symbolem a vstupním řetězcem. Situaci znázorňuje obrázek 8.3, který zachycuje zjednodušenou strukturu derivačního stromu. V kořeni tohoto stromu je počáteční neterminální symbol S, v uzlech neterminální symboly označující použití pravidel (například úroveň X_1 ... X_i vznikla aplikací pravidla (S, $X_1 \ldots X_i$) na větnou formu S atp.). V listech jsou terminální symboly, které obsahují vstupní řetězec $x = a_1 \ldots a_m \ldots a_n$.



Obrázek 8.3: Příklad derivačního stromu

Z důvodů pramenících z nejednoznačnosti některých bezkontextových gramatik však může existovat více možných nepřímých derivací vět přijímaného jazyka, a tedy i více možných derivačních stromů. Proto je vhodné zavést *nejpravější* a *nejlevější derivace*, které v každém derivačním kroku předepisují aplikaci právě jedné přímé derivaci ze všech možných. Z praktického hlediska byly zavedeny dva přístupy, jak tuto analýzu provádět a jakých principů výběru přepisovacího pravidla v rámci přímé derivace při tom využívat. Těmto přístupům jsou věnovány následující podkapitoly.

8.3.1 Nejlevější derivace

Nejlevější derivace je nepřímá derivace taková, která v každé větné formě vždy přepisuje neterminální symbol na nejlevější pozici. Posloupnost pravidel vedoucí k nejlevější derivaci je označována jako *levý rozbor*. Na obrázku 8.3 by se jednalo o pre-order (viz [3]) průchod daným stromem, tedy první by se zpracoval počáteční neterminální symbol S, poté X_1, Y_1 atd. Výhodou tohoto přístupu je nutnost hledání nepřímé derivace mezi nejlevějším doposud nezpracovaným symbolem na vstupu a aktuálně zpracovávaným neterminálním symbolem, což lze realizovat velmi efektivně. Konkrétní algoritmy hledání této derivace jsou obsaženy v kapitole 9 této práce.

8.3.2 Nejpravější derivace

Nejpravější derivace je nepřímá derivace taková, která v každé větné formě vždy přepisuje neterminální symbol na nejpravější pozici. Posloupnost pravidel vedoucí k nejpravější derivaci je označována jako *pravý rozbor*. Na obrázku 8.3 by se jednalo o průchod daným stromem v pořadí zpracování S, X_i, X_1 atd. Nevýhodou tohoto přístupu je fakt, že z praktického hlediska je nutné buď zpracovávat vstupní řetězec zprava, nebo ho zpracovávat zleva a ukládat v pomocné paměti pro pozdější zpracování.

8.3.3 Syntaktická analýza shora dolů

Metodika syntaktické analýzy shora dolů přistupuje ke konstrukci derivačního stromu postupnou expanzí počátečního neterminálního symbolu. Konstrukce derivačního stromu tak probíhá od jeho kořene, což přináší některé nevýhody, které jsou detailněji zkoumány kapitolou věnovanou vybraným metodám syntaktické analýzy. Metody založené na tomto přístupu typicky využívají levého rozboru, tedy k expanzi neterminálních symbolů využívají pouze přečtenou část vstupního řetězce (obvykle symbol na počátku doposud nezpracované části vstupního řetězce). Typickým zástupcem metody syntaktické analýzy shora dolů je prediktivní syntaktická analýza, které je věnována pozornost v kapitole 9.

8.3.4 Syntaktická analýza zdola nahoru

Syntaktická analýza zdola nahoru označuje algoritmy provádějící konstrukci derivačního stromu od jeho listů (věty) postupnou redukcí vstupního řetězce na počáteční neterminální symbol. Metody založené na tomto principu mohou využívat i tu část vstupního řetězce, která nebyla doposud zpracovaná, což přináší některé výhody oproti opačnému přístupu, například při výběru přepisovacího pravidla, které povede ke správné nepřímé derivaci. Metody syntaktické analýzy pracující zdola nahoru obvykle pracují se zásobníkem, do kterého ukládají větnou formu, kterou dále analyzují a snaží se ji redukovat na počáteční neterminální symbol.

8.4 Sémantická analýza

Sémantická analýza bývá řazena za syntaktickou analýzu. Úkolem sémantického analyzátoru je ověření vstupního řetězce z hlediska sémantické korektnosti. Formálně se tedy jedná o algoritmus, jehož vstupem je simulace konstrukce derivačního stromu a výstupem buď abstraktní syntaktický strom, nebo sémantická chyba. V praxi se zde typicky realizuje kontrola deklarací proměnných, kontrola datových typu, jejich inference a jiné sémantické kontroly. Sémantickou analýzu lze realizovat v některých případech již při samotné simulaci konstrukce derivačního stromu syntaktickým analyzátorem. Například při vytváření levého rozboru lze přistupovat k sémantickým kontrolám po zpracování určitého neterminálního symbolu, který reprezentuje popis sémantické konstrukce. Po dokončení zpracování tohoto neterminálního symbolu (redukci celé pravé strany přepisovacího pravidla, které bylo pro jeho zpracování aplikováno) je tedy zajištěna existence konkrétních dat v rámci derivačního stromu. Tohoto principu využívá navržený a implementovaný systém popsaný v kapitolách 11 a 12 této práce.

8.5 Generátor vnitřního kódu

Generování vnitřního kódu převádí stromovou strukturu tokenů z výstupu sémantické analýzy do lépe reprezentovatelné podoby. Touto podobou může být například tří adresný kód, nebo různé hybridní datové struktury. Důležitým požadavkem na vnitřní kód je jeho jednoduchá reprezentace v rámci implementačního jazyka, se kterým překladač dokáže jednoduše pracovat. Důvodem tohoto požadavku je možnost jednoduché aplikace optimalizačních transformací a také jednoduchý překlad do cílového kódu.

8.6 Optimalizace

Moderní překladače dokáží vykonávat celou řadu optimalizací výsledných programů. Cílem optimalizací bývá obvykle úspora některého z prostředků, jako je například velikost cílového kódu, doba zpracování a/nebo vykonání výsledného programu, nebo, v některých případech, čitelnost cílového kódu. Ačkoliv je optimalizace klíčovou činností moderních překladačů, překladač ji nemusí vůbec provádět.

8.7 Generátor cílového kódu

Generátor cílového kódu je závěrečným stavebním blokem zde uvedené struktury překladače. Jeho vstupem je vnitřní kód (ať už optimalizovaný nebo neoptimalizovaný), který převádí do kódu cílového. Způsob a náročnost tohoto převodu se může lišit v závislosti na architektuře obou kódů. Řešení využito v implementační části této práce jako generátor cílového kódu využívá strategii předepisující výstupní formát.

Kapitola 9

Prediktivní syntaktická analýza

Prediktivní syntaktická analýza je algoritmus využívající tabulky pro deterministický výběr přepisovacího pravidla pro použití na aktuální větnou formu. Tento algoritmus je založený na tzv. *LL tabulce*. Název LL je odvozen od skutečnosti, že vstupní řetězec se zpracovává zleva (**L**eft to right) a hledá se nejlevější derivace (**L**eftmost derivation). Algoritmy prediktivní syntaktická analýzy přistupují k simulaci konstrukce derivačního stromu přístupem shora dolů. Jádro kapitoly tvoří analýza těchto algoritmů, algoritmy výpočtu pomocných rozhodovacích množin a tabulek.

Algoritmy založené na LL tabulkách provádí vytváření levého rozboru, což odpovídá nejlevější derivaci. Obecně tyto algoritmy pracují s LL tabulkou, jejímž obsahem je informace vedoucí ke zvolení správného přepisovacího pravidla gramatiky za využití aktuálních symbolu na vstupu a nejlevějšího neterminálního symbolu aktuální větné formy. Počet symbolů na vstupu, na jejichž základě se vybírá přepisovací pravidlo, ovlivňuje sílu analýzy, a podle tohoto počtu rozlišujeme *LL*(*n*) *tabulky* a na nich založené algoritmy. V rámci této práce uvažujeme *LL*(1) *tabulku*, tedy rozhodování, které přepisovací pravidlo se má použít, na základě pouze jednoho následujícího vstupního symbolu. Je zde zaveden speciální symbol *$* označující dno zásobníku a současně konec vstupního řetězce. Tento symbol se implicitně předpokládá jako součást vstupního řetězce.

V této kapitole je implicitně uvažována bezkontextová gramatika G = (*N*, *Σ*, *P*, *S*) označovaná jako *LL Gramatika*, která navíc splňuje následující podmínku:

$$\forall (X, x), (X, y) \in P, a \in (\Sigma \cup \{\$\}): (X, x) \in LL[X, a] \Rightarrow x = y$$

Neformálně řečeno, existuje maximálně jedno přepisovací pravidlo s levou stranu *X*, pro nějž platí, že symbol *a* vede k výběru tohoto pravidla syntaktickým analyzátorem. Gramatiky splňující tuto podmínku jsou označovány jako *LL*(1) *gramatiky*. Obsah *LL* tabulky je vypočten na základě algoritmů uvedených níže.

Při konstrukci LL tabulky vycházíme z několika vypočtených množin, kterým se věnují následující kapitoly. Definice algoritmů popsaných v podkapitolách 9.1 až 9.7 byly s drobnými úpravami převzaty z [11]. Informace o algoritmech rekurzivního sestupu a nerekurzivního algoritmu prediktivní syntaktické analýzy byly čerpány z [10].

9.1 Množina *Empty*

Množina Empty je pomocná množina, která je definována pro každý terminální a neterminální symbol gramatiky G algoritmem 9.1, a pro každý neprázdnou větnou formu gramatiky G algoritmem 9.2. Obsah množiny vyjadřuje myšlenku, zda-li je možno daný symbol z větné formy vymazat (přepsat na ε), či nikoliv.

Vstup: Bezkontextová gramatika $G = (N, \Sigma, P, S)$
Výstup: Množina *Empty* pro každý terminální a neterminální symbol gramatiky G
Metoda:

```
(1)   for each A in N do
(2)       if (A, ε) ∈ P do
(3)           Empty(A) ← {ε}
(4)       else do
(5)           Empty(A) ← ∅
(6)   for each a in Σ do Empty(a) ← ∅
(7)   Empty(ε) ← {ε}
(8)   repeat
(9)       for each A in N do Empty'(A) ← Empty(A)
(10)      for each (A, x1x2...xn) in P do
(11)          if Empty(xi) = {ε} for each i in <1, n> do
(12)              Empty(A) ← {ε}
(13)  until Empty(A) = Empty'(A) for each A in N
```

Algoritmus 9.1: Algoritmus výpočtu množiny *Empty* pro terminální a neterminální symboly

---

Vstup: Bezkontextová gramatika $G = (N, \Sigma, P, S)$
Množina *Empty* pro každý terminální a neterminální symbol gramatiky G
Větná forma $x = x_1x_2...x_n \in (N \cup \Sigma)^+$
Výstup: Množina *Empty* pro zadanou větnou formu x gramatiky G
Metoda:

```
(1)   if Empty(xi) = {ε} for each i in <1, n> do
(2)       Empty(x) ← {ε}
(3)   else do
(4)       Empty(x) ← ∅
```

Algoritmus 9.2: Algoritmus výpočtu množiny *Empty* pro libovolnou neprázdnou větnou formu

9.2 Množina *First*

Množina *First* je definována pro každý terminální a neterminální symbol gramatiky G algoritmem 9.3 a pro každou neprázdnou větnou formu gramatiky G algoritmem 9.4. Obsahem množin *First* jsou pro každou větnou formu všechny terminální symboly, kterými může začínat větná forma derivovatelné z předchozí větné formy.

Vstup: Bezkontextová gramatika $G = (N, \Sigma, P, S)$
Množina *Empty* pro každý terminální a neterminální symbol gramatiky G

Výstup: Množina *First* pro každý terminální a neterminální symbol gramatiky G

Metoda:

(1) **for each** a **in** Σ **do** $First(a) \leftarrow \{a\}$
(2) **for each** A **in** N **do** $First(A) \leftarrow \emptyset$
(3) **repeat**
(4) **for each** A **in** N **do** $First'(A) \leftarrow First(A)$
(5) **for each** $(A, x_1x_2...x_{k-1}x_k...x_n)$ **in** P **do**
(6) $First(A) \leftarrow First(A) \cup First(x_1)$
(7) **if** $Empty(x_i) = \{\varepsilon\}$ **for each** i **in** $<1, k\text{-}1>$, $k \leq n$ **do**
(8) $First(A) \leftarrow First(A) \cup First(x_k)$
(9) **until** $First(A) = First'(A)$ **for each** A **in** N

Algoritmus 9.3: Algoritmus výpočtu množiny *First* pro terminální a neterminální symboly

---

Vstup: Bezkontextová gramatika $G = (N, \Sigma, P, S)$
Množina *Empty* pro každou větnou formu gramatiky G
Větná forma $x = x_1x_2...x_n \in (N \cup \Sigma)^+$

Výstup: Množina *First* pro zadanou větnou formu x gramatiky G

Metoda:

(1) $First(x_1x_2...x_n) \leftarrow First(x_1)$
(2) **repeat**
(3) $First'(x_1x_2...x_n) \leftarrow First(x_1x_2...x_n)$
(4) **if** $Empty(x_i) = \{\varepsilon\}$ **for each** i **in** $<1, k\text{-}1>$, $k \leq n$ **do**
(5) $First(x_1x_2...x_n) \leftarrow First(x_1x_2...x_n) \cup First(x_k)$
(6) **until** $First(x_1x_2...x_n) = First'(x_1x_2...x_n)$

Algoritmus 9.4: Algoritmus výpočtu množiny *First* pro libovolnou neprázdnou větnou formu

9.3 Množina *Follow*

Množina *Follow* je definována pro každý neterminální symbol gramatiky G algoritmem 9.5. Smyslem této množiny je zjistit, kterými terminálními symboly může pokračovat větná forma po vymazání symbolu, pro který je množina *Follow* definována.

Vstup: Bezkontextová gramatika $G = (N, \Sigma, P, S)$
Množina *First* pro každou větnou formu gramatiky G

Výstup: Množina *Follow* pro každý terminální a neterminální symbol gramatiky G

Metoda:

(1) $Follow(S) \leftarrow \{\$\}$
(2) **repeat**
(3) **for each** A **in** N **do** $Follow'(A) \leftarrow Follow(A)$
(4) **for each** (A, xBy), $B \in N$ **in** P **do**
(5) **if** $y \neq \varepsilon$ **do**
(6) $Follow(B) \leftarrow Follow(B) \cup First(y)$
(7) **if** $Empty(y) = \{\varepsilon\}$ **do**
(8) $Follow(B) \leftarrow Follow(B) \cup Follow(A)$
(9) **until** $Follow(A) = Follow'(A)$ **for each** A **in** N

Algoritmus 9.5: Algoritmus výpočtu množiny *Follow*

9.4 Množina *Predict*

Množina *Predict* je definována pro každý prvek množiny přepisovacích pravidel gramatiky G následujícím algoritmem:

Vstup: Bezkontextová gramatika $G = (N, \Sigma, P, S)$
Množina *Empty* pro každou větnou formu gramatiky G
Množina *Follow* pro každý neterminální symbol gramatiky G

Výstup: Množina *Predict* pro každé přepisovací pravidlo gramatiky G

Metoda:

(1) **for each** $p = (A, x)$ **in** P **do**
(2) **if** $Empty(x) = \{\varepsilon\}$ **do**
(3) $Predict(p) \leftarrow First(y) \cup Follow(A)$
(4) **else if** $Empty(y) = \emptyset$ **do**
(5) $Predict(p) \leftarrow First(y)$

Algoritmus 9.6: Algoritmus výpočtu množiny *Predict*

9.5 Konstrukce LL tabulky

Pro konstrukci této tabulky je nezbytné vypočíst nejdříve množinu Predict z předcházející podkapitoly. Obsah tabulky je vypočten pomocí algoritmu 9.7.

Vstup: Bezkontextová gramatika $G = (N, \Sigma, P, S)$
Množina *Predict* pro každé přepisovací pravidlo gramatiky G

Výstup: LL tabulka pro zadanou gramatiku G

Metoda:

(1) **for each** a **in** Σ **do**
(2) **for each** $p = (A, x)$ **in** P **do**
(3) **if** $a \in Predict(p)$ **do**
(4) $LL[A, a] \leftarrow \{ p \}$
(5) **else do**
(6) $LL[A, a] \leftarrow \emptyset$

Algoritmus 9.7: Algoritmus konstrukce LL tabulky

Pokud se po konstrukci v libovolném políčku tabulky objeví množina obsahující více než 1 pravidlo, nejedná se o LL gramatiku.

9.6 Algoritmus rekurzivního sestupu

Myšlenka algoritmu rekurzivního sestupu je založena na principu reprezentace neterminálních symbolů prostředkem implementačního jazyka, v němž je analyzátor vytvořen. Typicky se využívá funkce nebo procedura. Tento algoritmus přistupuje k simulaci konstrukce derivačního stromu přístupem shora dolů, tedy postupnou expanzí počátečního neterminálního symbolu. Princip tohoto algoritmu spočívá v expanzi neterminálních symbolů voláním funkcí (resp. procedur), které tyto neterminální symboly expandují rekurzivním voláním dalších funkcí, případně čtením vstupního řetězce a porovnávání terminálních symbolů obsažených ve vstupní gramatice. Výhodou algoritmu je jeho rychlost. Značnou nevýhodu představuje nemožnost (resp. složité modifikace) expandovat neterminální symbol, kterému ve větné formě předchází jiné neterminální symboly. Tato nevýhoda znemožňuje jeho využití pro regulovanou syntaktickou analýzu, a právě kvůli této nevýhodě nebude algoritmu rekurzivního sestupu věnována další pozornost.

9.7 Nerekurzivní algoritmus prediktivní syntaktické analýzy

Myšlenka tohoto přístupu spočívá v odstranění rekurze algoritmu rekurzivního sestupu. K ověřování zde slouží zásobník, jehož funkce je srovnatelná se zásobníkem zásobníkového automatu. Algoritmus je závislý na aktuálním symbolu na vstupu, a na symbolu na vrcholu zásobníku, což odpovídá výpočetnímu kroku zásobníkového automatu. Od tohoto automatu se odlišuje svou funkcí, neboť může provádět akce na základě vstupního symbolu, a tento symbol na vstupu zanechat pro následující akce.

Vstup: Bezkontextová gramatika $G = (N, T, P, S)$, vstupní řetězec
Výstup: Posloupnost pravidel vedoucí k přijetí vstupního řetězce nebo chyba
Metoda:

(1) **push**($)
(2) **push**(S)
(3) **repeat**
(4) $X \leftarrow$ symbol na vrcholu zásobníku
(5) $a \leftarrow$ aktuální symbol na vstupu
(6) **pop**(X)
(7) **if** $X = \$$ **and** $a = \$$ **do**
(8) *Úspěch*
(9) **else if** $X \in \Sigma$ **and** $X = a$ **do**
(10) Přečti další symbol a ze vstupu
(11) **else if** $X \in N$ **and** $r = (X, x) \in LL\_Tabulka[X, a]$ **do**
(12) **push**(**reversal**(x))
(13) **output** r
(14) **else do**
(15) *Chyba*
(16) **until** *Úspěch* **or** *Chyba*

Algoritmus 9.8: Algoritmus prediktivní syntaktické analýzy

9.7.1 Příklad

Uvažujme zadanou LL gramatiku $G = (N, \Sigma, P, S)$. Obsah jednotlivých množin je následující:

N = {*S*, *F*, *T*, *V*, *R*}
Σ = {*i*, +, *}
P, = { 1: (*S*, *TF*),
2: (*F*, +*TF*),
3: (*F*, ε),
4: (*T*, *VR*),
5: (*R*, **VR*),
6: (*R*, ε),
7: (*V*, *i*) }

Je dán vstupní řetězec $x = i + i * i$, u kterého je nutno ověřit příslušnost do jazyka gramatiky *G*. Sestavením LL tabulky dle algoritmu 9.7 a využitím algoritmu 9.8 je možno dokázat, že vstupní řetězec *x* patří do jazyka generovaného gramatikou *G*, a to následujícím postupem:

Tabulka 9.1: Prediktivní syntaktická analýza

LL tabulka

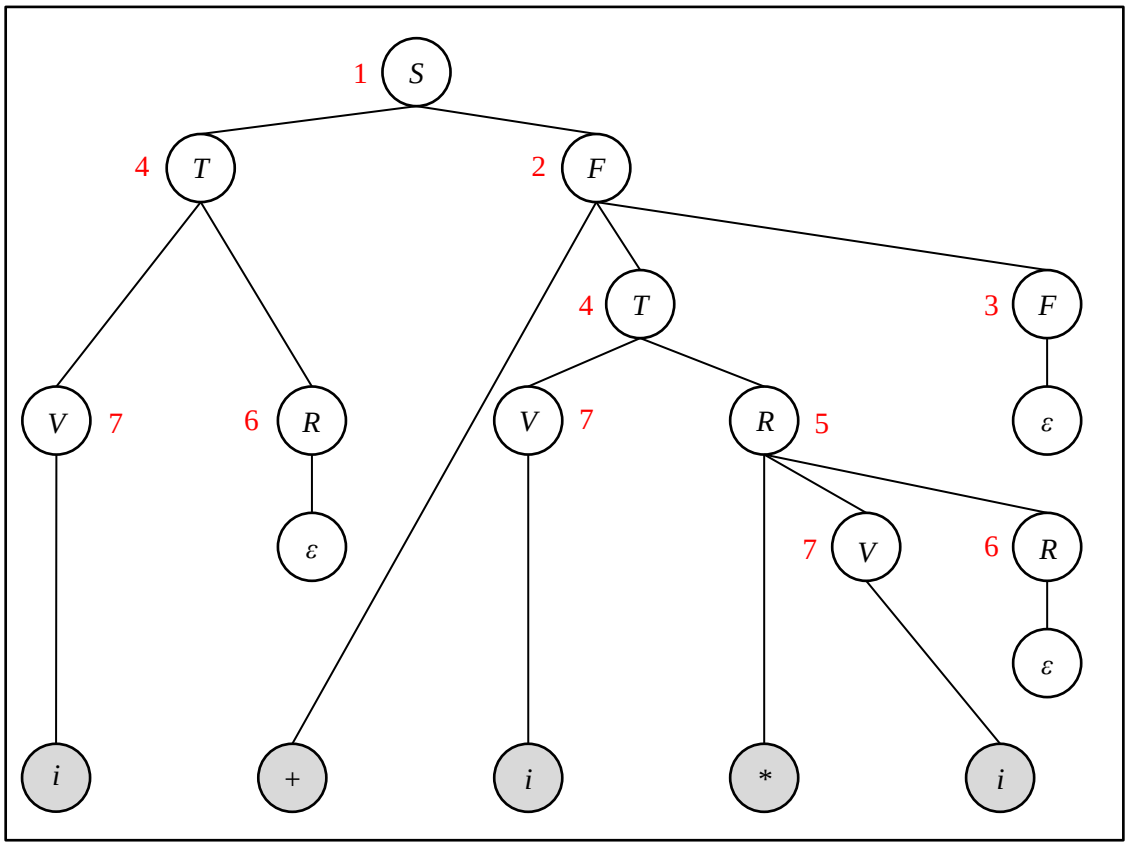
	i	**+**	*****	**$**
S	1			
F		2		3
T	4			
V	7			
R		6	5	6

Vstupní řetězec: *i* + *i* * *i*

Zásobník	**Vstup**	**Tabulka [*X*, *a*]**	**Derivace**
$S	*i* + *i* * *i* $	1: (*S*, *TF*)	*S* ⇒ *TF*
$FT	*i* + *i* * *i* $	4: (*T*, *VR*)	⇒ *VRF*
$FRV	*i* + *i* * *i* $	7: (*V*, *i*)	⇒ *iRF*
$FRi	*i* + *i* * *i* $		
$FR	+ *i* * *i* $	6: (*R*, ε)	⇒ *iF*
$F	+ *i* * *i* $	2: (*F*, +*TF*)	⇒ *i*+*TF*
$FT+	+ *i* * *i* $		
$FT	*i* * *i* $	4: (*T*, *VR*)	⇒ *i*+*VRF*
$FRV	*i* * *i* $	7: (*V*, *i*)	⇒ *i*+*iRF*
$FRi	*i* * *i* $		
$FR	* *i* $	5: (*R*, **VR*)	⇒ *i*+*i***VRF*
$FRV*	* *i* $		
$FRV	*i* $	7: (*V*, *i*)	⇒ *i*+*i***iRF*
$FRi	*i* $		
$FR	$	6: (*R*, ε)	⇒ *i*+*i***iF*
$F	$	3: (*F*, ε)	⇒ *i*+*i***i*
$	$	*Úspěch*	

Vstupní řetězec byl přijat posloupností přepisovacích pravidel 14762475763.

Na obrázku 9.1 je znázorněn derivační strom k předešlému příkladu. V obrázku jsou u uzlů, reprezentujících neterminální symboly, červeně vyznačeny identifikátory přepisovacích pravidel užitých k expanzi daného neterminálního symbolu. Princip konstrukce probíhá průchodem tímto stromem od jeho vrcholu. Pořadí průchodu jednotlivými uzly odpovídá levému rozboru, který je výstupem této analýzy. Patrně se jedná o pre-order průchod (viz [3]). Jak již bylo v předchozím textu zmíněno, výhoda tohoto průchodu spočívá v tom, že při zpracování uzlů každého neterminálního symbolu v rámci derivačního stromu jsou zpracovány všechny terminální symboly derivovatelné z předcházejících neterminálních symbolů. Při průchodu výše uvedeného příkladu (14762475763) se jedná například o situaci, kdy byla použita pravidla 147624 a aktuálním symbolem na vrcholu zásobníku je neterminální symbol *V*. Jsou tedy zpracovány terminální symboly *i* a +, které již mohou hrát roli v bloku překladače, který následuje syntaktické analýze (např. zde může probíhat kontrola deklarací proměnné *i*). Následující kapitola se zabývá návrhem algoritmů regulované syntaktické analýzy, které tuto vlastnost (výstup ve formě levého rozboru) zachovávají.



Obrázek 9.1: Derivační strom z příkladu 9.7.1

Kapitola 10

Regulované metody syntaktické analýzy

Úvodní kapitoly definovaly některé specifické typy gramatik, které svou vyjadřovací silou překonaly gramatiky bezkontextové. Jejich společnou vlastností je především zaměření na omezení výběrového mechanismu přepisovacích pravidel mezi jednotlivými derivacemi větných forem. Tato omezení však mají za následek snížení efektivity některých z uvedených metod syntaktické analýzy, především z důvodu neefektivní možnosti kontroly dodržování těchto omezení. Mezi nejméně zasažené metody z výše uvedených patří prediktivní syntaktická analýza s využitím zásobníku z předchozí kapitoly.

Navržená metoda regulované syntaktické analýzy je velmi podobná metodě uvedené v předchozí kapitole. Jedním z nutných rozšíření je však použití hlubokého zásobníkového automatu pro možnost expanze neterminálních symbolů i pod vrcholem zásobníku. Důležitou vlastností zde navržené metody je zachování levého rozboru jako výstupu syntaktické analýzy. Tato vlastnost je dána ukládáním přepisovacích pravidel (nebo jejich vhodně zvolených identifikátorů) na zásobník a jejich následný výstup, jakmile se stanou vrcholem zásobníku. Nahrazením kroku 16 indexací LL tabulky algoritmu 10.1 se metoda stává prediktivní syntaktickou analýzou pro třídu LL bezkontextových jazyků popsanou v předchozí kapitole. Algoritmy regulované syntaktické analýzy se pokusí rozhodnout členství vstupního řetězce v jazyce generovaném gramatikou. Formální důkaz klasifikace problému členství řetězců v uvedených regulovaných gramatikách (resp. jejich níže uvedených podtřídách) a způsob jeho rozhodování je předmětem dalšího zkoumání. Protože jsou však uvedené regulované gramatiky v některých variantách (bez kontroly na výskyt s vymazávacími pravidly) svou vyjadřovací silou srovnatelné s třídou gramatik neomezených, tak se pro účely této práce se nepředpokládá rozhodnutelnost tohoto problému žádným z navržených algoritmů.

Vstup: Regulovaná gramatika $G = (N, \Sigma, P, S, R, F)$
vstupní řetězec w

Výstup: Posloupnost pravidel vedoucí k přijetí vstupního řetězce nebo chyba

Metoda:

```
(1)   push($)
(2)   push(S)
(3)   C ← ∅
(4)   repeat
(5)       X ← symbol na vrcholu zásobníku
(6)       a ← aktuální symbol na vstupu
(7)       if X ∈ P do
(8)           output X
(9)           pop(X)
(10)      else if X = $ and a = $ do
(11)          Úspěch
(12)      else if X ∈ Σ and X = a do
(13)          Přečti další symbol a ze vstupu
(14)          pop(X)
(15)      else
(16)          Urči množinu M dovolených přepisovacích pravidel a uprav kontext C
(17)          if X ∈ N and r = (Y, x) ∈ M and card(M) = 1 do
(18)              replace(Y, reversal(x) . r)
(19)          else do
(20)              Chyba
(21)  until Úspěch or Chyba
```

Algoritmus 10.1: Algoritmus regulované prediktivní syntaktické analýzy

Na pozadí algoritmu figuruje hluboký zásobníkový automat rozšířený o strukturu kontextu, která obsahuje dynamickou tabulku počítadel neterminálních symbolů, které se na zásobníku nachází (aktuální větnou formu). Součástí kontextu může být například i informace o předchozím použitém pravidle, čehož některé v níže uvedených metod využívají. Operace uvedená v kroku 16 výše uvedeného algoritmu zaštiťuje deterministický výběr množiny použitelných přepisovacích pravidel na základě dané regulované gramatiky a aktuálního stavu analýzy (kontextu). Užité operace **push**, **pop** a **replace** jsou pro hluboký zásobníkový automat definovány po řadě jako vložení na vrchol zásobníku, vyjmutí symbolu z vrcholu zásobníku a nahrazení prvního výskytu symbolu nejblíže vrcholu zásobníku daným řetězcem. Jedná se tedy o variantu hlubokého zásobníkového automatu s hloubkou 1 (viz 5.4). Uvedená definice stanovuje další implicitní omezení na vstupní gramatiky, která jsou do větších detailů popsána v následujících kapitolách pro každou analyzovanou gramatiku samostatně.

10.1 Syntaktická analýza maticových gramatik

Jak bylo zmíněno v úvodu této kapitoly, syntaktická analýza regulovaných gramatik je realizovatelná algoritmem z předchozí kapitoly nahrazením metody výpočtu množiny aplikovatelných pravidel a úpravy kontextu.

Sestrojit deterministickou variantu výběrové funkce pro libovolnou maticovou gramatiku je úkol složitý a pro většinu případů zbytečný. Proto se zaměříme pouze na podmnožinu maticových gramatik, do které patří pouze takové maticové gramatiky, které splňují následující dodatečné požadavky:

- $\forall R_1, R_2 \in M: first(R_1) = first(R_2) \Rightarrow R_1 = R_2$
- $\forall R \in M: first(R) = (r, -)$
- $\forall R \in M: first(R) = ((X, x), -) \Rightarrow empty(x) = \emptyset$

Užitá funkce *first* zde, oproti předcházejícím kapitolám, vyjadřuje výběr prvního prvku struktury dané argumentem. První požadavek vyjadřuje myšlenku deterministického výběru řádku matice, protože žádné dva řádky matice nezačínají stejným pravidlem, což je v případě jednoznačné vstupní gramatiky zaručeno. Druhý požadavek zajistí, že každé první pravidlo každého řádku matice nemá kontrolu na výskyt. Z praktického hlediska to značným způsobem zjednoduší níže uvedený algoritmus. Význam posledního pravidla spočívá v zamezení možného vypuštění symbolu, na jehož základě má proběhnout výběr řádku matice obsahující přepisovací pravidla, která se následně aplikují.

Vstup: Maticová gramatika $H = (G, M)$, $G = (N, \Sigma, P, S)$
Aktuální symbol na vstupu $a \in \Sigma$
Neterminální symbol na vrcholu zásobníku $X \in N$
Kontext $C = \{r: r \in P\}$
Aktuální větná forma w

Výstup: Množina aplikovatelných pravidel
Kontext

Metoda:

```
(1)   if C = ∅ do
(2)       Firsts ← ∅
(3)       for each R in M do
(4)           r = (p, f) ← first(R)
(5)           p = (A, x)
(6)           if A ∈ alph(w) do
(7)               C ← C ∪ { R }
(8)               if A = X do
(9)                   Firsts ← Firsts ∪ { p }
(10)      if card(C) = 1 do
(11)          goto (1)
(12)      else if card (C) > 1 do
(13)          LL ← predict(X, a) ∩ Firsts
(14)          if card(LL) = 1 do
(15)              LL ← first(LL)
(16)              for each R in M do
(17)                  if (LL, f) = first(R) pro nějaké f ∈ {+, -} do
(18)                      C ← { R }
(19)                      goto (1)
(20)  else if C = { R } do
(21)      r = (p, f) ← first(R)
(22)      if R = ( r ) do
(23)          C ← ∅
(24)      else do
(25)          C ← { next(R) }
(26)      p = (A, x)
(27)      if f = + and A ∉ alph(w) do
(28)          goto (1)
(29)      output ({ p }, C)
(30)  output (∅, ∅)
```

Algoritmus 10.2: Algoritmus výběru množiny přepisovacích pravidel maticové gramatiky

Algoritmus je možno rozdělit na 2 základní části. Cílem první části (řádky 1-19) je provést výběr samotného řádku matice, který se má použít v případě, kdy takový výběr nebyl doposud

učiněn. V principu se provádí podobný výběr, jako v případě LL prediktivního analyzátoru s tím rozdílem, že jsou do výběru zahrnuta pouze pravidla, která jsou na první pozici některého řádku matice a jsou na danou aktuální větnou formu aplikovatelná. V případě více možných aplikovatelných pravidel je konzultována LL tabulka. Řádek 13 výše uvedeného algoritmu popisuje případ, kdy je nutno kombinovat výběr množiny pravidel na základě dat z LL tabulky a množiny všech úvodních pravidel řádků matice dané maticové gramatiky. Jelikož je množina pravidel i matice dané gramatiky konečná, je možno modifikovat samotnou konstrukci LL tabulky na míru maticové gramatiky za účelem optimalizace této části algoritmu.

Druhá část (řádky 20-29) popisuje postup výběru následujícího pravidla z aktuálně vybraného řádku matice. V případě, že je dané pravidlo poslední v daném kontextu, je kontext vyprázdněn, a následující invokace způsobí výběr nového řádku z matice.

10.1.1 Příklad

Uvažujme maticovou gramatiku $H = (G, M)$, kde $G = (N, \Sigma, P, S)$ je bezkontextová gramatika, a jednotlivé množiny jsou definovány:
$N = \{S, A, B\}$
$\Sigma = \{a, b, c\}$
$P = \{$ 1: $(S, aAbBc)$,
2: (A, aA),
3: (B, bBc),
4: (A, ε),
5: (B, ε) }
$M = \{1, 23, 45\}$
Dále uvažujme vstupní řetězec *aaabbbccc*, který je analyzován algoritmem 10.1 s využitím algoritmu výběru přepisovacího pravidla 10.2. Postup analýzy shrnuje tabulka 10.2. Operace expanze (replace) jsou zapsány zkrácenou formou pouze identifikátorem pravidla, jehož pravá strana se spolu s identifikátorem vloží do zásobníku na místo daného neterminálního symbolu. Symboly, na jejichž základě se provede samotný výběr přepisovacího pravidla, jsou v tabulce barevně zvýrazněny.

Tabulka 10.2: Regulovaná syntaktická analýza maticové gramatiky

LL tabulka

	a	***b***	***c***	***$***
S	1			
A	2	4		
B		3	5	

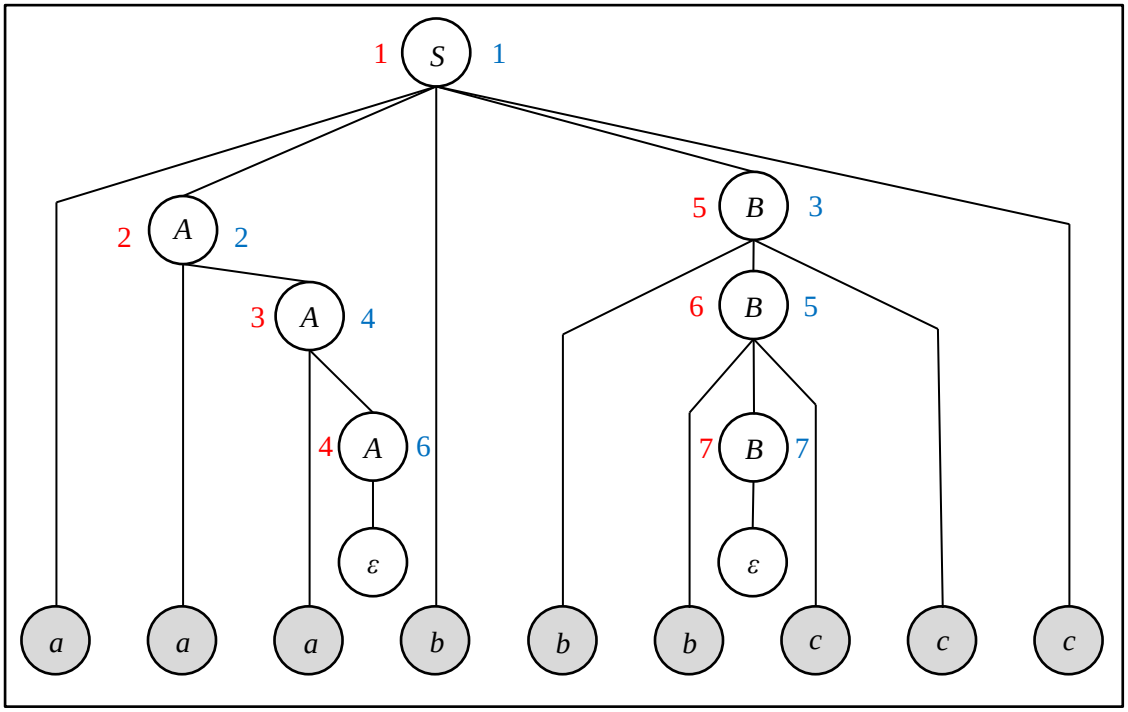
Vstupní řetězec: *aaabbbccc*

Zásobník	Kontext	Vstup	Operace
$S	∅	*aaabbbccc$*	**replace(***S***,** 1**)**
*$cBbAa*1	∅	*aaabbbccc$*	**output** 1
$cBbAa	∅	*aaabbbccc$*	**reduce(***a***)**
$cBbA	∅	*aabbbccc$*	**replace(***A***,** 2**)**
*$cBbAa*2	{3}	*abbbccc$*	**output** 2
$cBbAa	{3}	*aabbbccc$*	**reduce(***a***)**
$cBbA	{3}	*abbbccc$*	**replace(***B***,** 3**)**
$ccBb3bA	∅	*abbbccc$*	**replace(***A***,** 2**)**
*$ccBb3bAa*2	{3}	*abbbccc$*	**output** 2
$ccBb3bAa	{3}	*abbbccc$*	**reduce(***a***)**
$ccBb3bA	{3}	*bbbccc$*	**replace(***B***,** 3**)**
$cccBb3b3bA	∅	*bbbccc$*	**replace(***A***,** 4**)**
*$cccBb3b3b*4	{5}	*bbbccc$*	**output** 4
$cccBb3b3b	{5}	*bbbccc$*	**reduce(***b***)**
*$cccBb3b*3	{5}	*bbccc$*	**output** 3
$cccBb3b	{5}	*bbccc$*	**reduce(***b***)**
*$cccBb*3	{5}	*bbccc$*	**output** 3
$cccBb	{5}	*bccc$*	**reduce(***b***)**
$cccB	{5}	*ccc$*	**replace(***B***,** 5**)**
*$ccc*5	∅	*ccc$*	**output** 5
$ccc	∅	*ccc$*	**reduce(***c***)**
$cc	∅	*cc$*	**reduce(***c***)**
$c	∅	*c$*	**reduce(***c***)**
$	∅	*$*	**úspěch**

Vstupní řetězec byl přijat posloupností přepisovacích pravidel 1232345, přičemž výstupní levý rozbor je 1224335.

Princip konstrukce derivačního stromu k danému příkladu shrnuje obrázek 10.1, kde je postup jeho konstrukce algoritmem regulované syntaktické analýzy vyznačen modře jako pořadí expanzí neterminálních symbolů. Červeně je vyznačen levý rozbor, který je vedlejším produktem algoritmu. Z obrázku je patrný význam navržené metody regulované syntaktické analýzy a jejího rozdílného přístupu ke konstrukci derivačního stromu. Modré pořadí expanzí vyjadřuje průchod derivačním stromem po jednotlivých úrovních, zatímco červené pořadí vyjadřuje pre-order průchod.

Jak již bylo v předchozím textu uvedeno, levý rozbor přináší nespornou výhodu v rozvíjení větné formy postupně zleva, a tedy možného zpracovávání úvodní část větné formy již při expanzi neterminálních symbolů. Bez vedlejšího produktu výše zmíněného algoritmu by to nebylo možné, neboť například druhý výskyt terminálního symbolu b vstupního řetězce nelze zpracovat při expanzi prvního výskytu neterminálního symbolu B zleva, protože ve větné formě po této expanzi uvedenému terminálnímu symbolu předchází ještě neterminální symbol A, který nebyl doposud zpracován.



Obrázek 10.1: Derivační strom z příkladu 10.1.1

10.2 Syntaktická analýza gramatik s nahodilým kontextem

Podobně jako tomu bylo v předcházející kapitole, i analýza gramatik s nahodilým kontextem využívá obecný algoritmus 10.1 s doplněním výběrové funkce. Dále zavádíme dodatečný požadavek na vstupní gramatiky:

- $\forall\, p_1, p_2 \in P, x \in (N \cup \Sigma)^*$: $(R[p_1] \subseteq alph(x) \wedge R[p_2] \subseteq alph(x) \wedge F[p_1] \cap alph(x) = F[p_2] \cap alph(x) = \emptyset \wedge S \Rightarrow^* x) \underline{\Rightarrow} p_1 = p_2$

Přičemž $\Rightarrow^*$ označuje nepřímou derivaci a $\underline{\Rightarrow}$ označuje implikaci. Význam uvedeného požadavku spočívá v možnosti deterministického výběru pravidla pro všechny větné formy derivovatelné z počátečního neterminálního symbolu. Omezení přináší nutnost striktnější definice jednotlivých pravidel, zejména pak množin R a F.

Vstup: Gramatika s nahodilým kontextem H = (G, R, F), $G = (N, \Sigma, P, S)$
Aktuální větná forma w

Výstup: Množina aplikovatelných pravidel

Metoda:

(1) $Rules \leftarrow \emptyset$
(2) **for each** $r = (A, x)$ **in** P **do**
(3) **if** $A \in$ **alph**(w) **and** $R[r] \subseteq$ **alph**(w) **and** $F[r] \cap$ **alph**(w) $= \emptyset$ **do**
(4) $Rules \leftarrow Rules \cup \{ r \}$
(5) **output** *Rules*

Algoritmus 10.3: Algoritmus výběru množiny přepisovacích pravidel gramatik s nahodilým kontextem

Navržený algoritmus popisuje deterministický výběr množiny pravidel jen na základě dané množiny pravidel vstupní gramatiky a aktuální větné formy bez jakékoliv další predikce.

10.2.1 Příklad

Uvažujme gramatiku s nahodilým kontextem $H = (G, R, F)$, kde $G = (N, \Sigma, P, S)$ je bezkontextová gramatika, a jednotlivé množiny jsou definovány:
$N = \{S, A, B, C, A', B', C'\}$
$\Sigma = \{a, b, c\}$
$P, R, F = \{$ 1: $(S, aAbBcC), \emptyset, \emptyset$
2: $(A, aA'), \{B, C\}, \emptyset$
3: $(B, bB'), \{C\}, \{A, C'\}$
4: $(C, cC'), \{A'\}, \{B\}$
5: $(A', A), \{B', C'\}, \emptyset$
6: $(B', B), \{C'\}, \{A'\}$
7: $(C', C), \{A\}, \{B'\}$
8: $(A, \varepsilon), \{B, C\}, \emptyset$
9: $(B, \varepsilon), \{C\}, \{A, A'\}$
10: $(C, \varepsilon), \{C\}, \{B, A'\}$ $\}$

Dále uvažujme vstupní řetězec *aaabbbccc*, který je analyzován algoritmem 10.1 s využitím algoritmu výběru přepisovacího pravidla 10.3. Postup analýzy shrnuje tabulka 10.3. Operace expanze (replace) jsou zapsány zkrácenou formou pouze identifikátorem pravidla, jehož pravá strana se spolu s identifikátorem vloží do zásobníku na místo daného neterminálního symbolu. Symboly, na jejichž základě se provede samotný výběr přepisovacího pravidla, jsou v tabulce zvýrazněny.

Tabulka 10.3: Regulovaná syntaktická analýza gramatiky s nahodilým kontextem

LL tabulka

	a	***b***	***c***	***$***
S	1			
A	2	8		
B		3	9	
C			4	10
A'	5	5		
B'		6	6	
C'			7	7

Vstupní řetězec: *aaabbbccc*

Zásobník	**Vstup**	**Operace**
$S	*aaabbbccc$*	**replace**(*S*, 1)
*$CcBbAa*1	*aaabbbccc$*	**output** 1
$CcBbAa	*aaabbbccc$*	**reduce**(*a*)
$CcBbA	*aabbbccc$*	**replace**(*A*, 2)
*$CcBbA'a*2	*aabbbccc$*	**output** 2
$CcBbA'a	*aabbbccc$*	**reduce**(*a*)
$CcBbA'	*abbbccc$*	**replace**(*B*, 3)
$CcB'b3bA'	*abbbccc$*	**replace**(*C*, 4)
$C'c4cB'b3bA'	*abbbccc$*	**replace**(*A'*, 5)
*$C'c4cB'b3bA*5	*abbbccc$*	**output** 5
$C'c4cB'b3bA	*abbbccc$*	**replace**(*B'*, 6)
$C'c4cB6b3bA	*abbbccc$*	**replace**(*C'*, 7)
$C7c4cB6b3bA	*abbbccc$*	**replace**(*A*, 2)
*$C7c4cB6b3bA'a*2	*abbbccc$*	**output** 2
$C7c4cB6b3bA'a	*abbbccc$*	**reduce**(*a*)
$C7c4cB6b3bA'	*bbbccc$*	**replace**(*B*, 3)
$C7c4cB'b36b3bA'	*bbbccc$*	**replace**(*C*, 4)
$C'c47c4cB'b36b3bA'	*bbbccc$*	**replace**(*A'*, 5)
*$C'c47c4cB'b36b3bA*5	*bbbccc$*	**output** 5
$C'c47c4cB'b36b3bA	*bbbccc$*	**replace**(*B'*, 6)
$C'c47c4cB6b36b3bA	*bbbccc$*	**replace**(*C'*, 7)
$C7c47c4cB6b36b3bA	*bbbccc$*	**replace**(*A*, 8)
*$C7c47c4cB6b36b3b*8	*bbbccc$*	**output** 8
$C7c47c4cB6b36b3b	*bbbccc$*	**reduce**(*b*)
*$C7c47c4cB6b36b*3	*bbccc$*	**output** 3
$C7c47c4cB6b36b	*bbccc$*	**reduce**(*b*)
*$C7c47c4cB6b3*6	*bccc$*	**output** 6
*$C7c47c4cB6b*3	*bccc$*	**output** 3
$C7c47c4cB6b	*bccc$*	**reduce**(*b*)
*$C7c47c4cB*6	*ccc$*	**output** 6
$C7c47c4cB	*ccc$*	**replace**(*B*, 9)
*$C7c47c4c*9	*ccc$*	**output** 9
$C7c47c4c	*ccc$*	**reduce**(*c*)
$C7c47c4	*cc$*	**output** 4

$C7c47c	cc$	**reduce(***c***)**
$C7c47	c$	**output** 7
$C7c4	c$	**output** 4
$C7c	c$	**reduce(***c***)**
$C7	$	**output** 7
$C	$	**replace(***C***,** 10**)**
$10	$	**output** 10
$	$	**úspěch**

Vstupní řetězec byl přijat posloupností přepisovacích pravidel 12345672345678910, přičemž výstupní levý rozbor je 12525836369474710.

10.3 Syntaktická analýza programovaných gramatik

Při syntaktické analýze programovaných gramatik postupujeme podobně, jako v kapitolách přecházejících, a to specifikací funkce výběru použitelných přepisovacích pravidel. Je rovněž účelné vymezit podtřídu programovaných gramatik, u kterých je možno na základě prediktivních tabulek deterministicky rozhodnout, které pravidlo se má dále použít. Vzhledem k tomu, že derivace může probíhat prakticky na libovolném místě v rámci větné formy, je nutno vymezit pravidla pro výběr z více možných aplikovatelných pravidel. Intuitivně se nabízí vybírat takové pravidlo, které má na své levé straně neterminální symbol, jehož výskyt ve větné formě je na nejlevější pozici v rámci daných aplikovatelných pravidel. Taková třída jazyků by však vyžadovala striktnější definici všech pravidel. Proto při nutnosti výběru z několika aplikovatelných pravidel vybíráme to, jehož levá strana je nejlevější neterminální symbol v rámci dané větné formy, a zároveň je obsaženo v prediktivních tabulkách pro daný neterminální symbol a aktuální symbol na vstupu.

Algoritmus se dá opět rozdělit do dvou částí. V první části (řádky 1-10) se určuje pravidlo k použití při prázdném kontextu, tedy při expanzi úvodního neterminálního symbolu, nebo po použití pravidla, které nevymezovalo žádné další použití nějakého pravidla. Druhá část (řádky 11-27) určují množinu aplikovatelných a povolených pravidel (množina *Allowed* v řádcích 12-15), výběr z této množiny na základě prediktivní tabulky, případně výběr pravidla z množiny vymezených pravidel s kontrolou na výskyt (množina *F*) a opět případná konzultace prediktivní tabulky.

Vstup: Programovaná gramatika $H = (G, R, F)$, $G = (N, \Sigma, P, S)$
Aktuální symbol na vstupu $a \in \Sigma$
Neterminální symbol na vrcholu zásobníku $X \in N$
Kontext $C = \{r \in P^*\}$

Výstup: Množina aplikovatelných pravidel
Kontext C

Metoda:

```
(1)   If C = ∅ do
(2)       Firsts ← ∅
(3)       for each p = (A, x) in P do
(4)           if A ∈ alph(w) do
(5)               C ← C ∪ { p }
(6)               if A = X do
(7)                   Firsts ← Firsts ∪ { p }
(8)       if card (C) > 1 do
(9)           C ← predict(X, a) ∩ Firsts
(10)      output {C, C}
(11)  else if C = {p} do
(12)      Allowed ← ∅
(13)      for each p' = (A', x') in R[p] do
(14)          if A' ∈ alph(w) do
(15)              Allowed ← Allowed ∪ { p' }
(16)      if card(Allowed) > 1 do
(17)          Allowed ← predict(X, a) ∩ Allowed
(18)      else if Allowed = ∅ do
(19)          Firsts ← ∅
(20)          for each p' = (A', x') in F[p] do
(21)              if A' ∈ alph(w) do
(22)                  Allowed ← Allowed ∪ { p' }
(23)                  if A' = X do
(24)                      Firsts ← Firsts ∪ { p' }
(25)          if card(Allowed) > 1 do
(26)              Allowed ← predict(X, a) ∩ Allowed
(27)      C ← Allowed
(28)  output {C, C}
```

Algoritmus 10.4: Algoritmus výběru přepisovacího pravidla programovaných gramatik

10.3.1 Příklad

Uvažujme programovanou gramatiku $H = (G, R, F)$, kde $G = (N, \Sigma, P, S)$ je bezkontextová gramatika, a jednotlivé množiny jsou definovány:
N = {*S*, *A*, *B*, *C*}
Σ = {*a*, *b*, *c*}
P, *R*, *F* = { 1*:* (*S*, *aAbBcC*), {2, 5}, ∅
2*:* (*A*, *aA*), {3}, ∅
3*:* (*B*, *bB*), {4}, ∅
4*:* (*C*, *cC*), {2, 5}, ∅
5*:* (*A*, *ε*), {6}, ∅
6*:* (*B*, *ε*), {7}, ∅
7*:* (*C*, *ε*), {7}, ∅ }

Dále uvažujme vstupní řetězec *aaabbbccc*, který je analyzován algoritmem 10.1 s využitím algoritmu výběru přepisovacího pravidla 10.4. Postup analýzy shrnuje tabulka 10.4. Operace expanze (replace) jsou zapsány zkrácenou formou pouze identifikátorem pravidla, jehož pravá strana se spolu s identifikátorem vloží do zásobníku na místo daného neterminálního symbolu. Symboly, na jejichž základě se provede samotný výběr přepisovacího pravidla, jsou v tabulce zvýrazněny.

Tabulka 10.4: Regulovaná syntaktická analýza maticové gramatiky

LL tabulka

	a	***b***	***c***	***$***
S	1			
A	2	5		
B		3	6	
C			4	7

Vstupní řetězec: *aaabbbccc*

Zásobník	Kontext	Vstup	Operace
$S	∅	*aaabbbccc$*	**replace**(*S*, 1)
*$CcBbAa*1	{1}	*aaabbbccc$*	**output** 1
$CcBbAa	{1}	*aaabbbccc$*	**reduce**(*a*)
$CcBbA	{1}	*aabbbccc$*	**replace**(*A*, 2)
*$CcBbAa*2	{2}	*aabbbccc$*	**output** 2
$CcBbAa	{2}	*aabbbccc$*	**reduce**(*a*)
$CcBbA	{2}	*abbbccc$*	**replace**(*B*, 3)
*$CcBb*3*bA*	{3}	*abbbccc$*	**replace**(*C*, 4)
*$Cc*4*cBb*3*bA*	{4}	*abbbccc$*	**replace**(*A*, 2)
*$Cc*4*cBb*3*bAa*2	{2}	*abbbccc$*	**output** 2
*$Cc*4*cBb*3*bAa*	{2}	*abbbccc$*	**reduce**(*a*)
*$Cc*4*cBb*3*bA*	{2}	*bbbccc$*	**replace**(*B*, 3)
*$Cc*4*cBb*3*b*3*bA*	{3}	*bbbccc$*	**replace**(*C*, 4)
*$Cc*4*c*4*cBb*3*b*3*bA*	{4}	*bbbccc$*	**replace**(*A*, 5)
*$Cc*4*c*4*cBb*3*b*3*b*5	{5}	*bbbccc$*	**output** 5
*$Cc*4*c*4*cBb*3*b*3*b*	{5}	*bbbccc$*	**reduce**(*b*)
*$Cc*4*c*4*cBb*3*b*3	{5}	*bbccc$*	**output** 3
*$Cc*4*c*4*cBb*3*b*	{5}	*bbccc$*	**reduce**(*b*)
*$Cc*4*c*4*cBb*3	{5}	*bccc$*	**output** 3
*$Cc*4*c*4*cBb*	{5}	*bccc$*	**reduce**(*b*)
*$Cc*4*c*4*cB*	{5}	*ccc$*	**replace**(*B*, 6)
*$Cc*4*c*4*c*6	{6}	*ccc$*	**output** 6
*$Cc*4*c*4*c*	{6}	*ccc$*	**reduce**(*c*)
*$Cc*4*c*4	{6}	*cc$*	**output** 4
*$Cc*4*c*	{6}	*cc$*	**reduce**(*c*)
*$Cc*4	{6}	*c$*	**output** 4
$Cc	{6}	*c$*	**reduce**(*c*)
$C	{6}	*$*	**replace**(*C*, 7)
*$*7	{7}	*$*	**output** 7
$	{7}	*$*	**úspěch**

Vstupní řetězec byl přijat posloupností přepisovacích pravidel 1234234567, přičemž výstupní levý rozbor je 1225336447.

Kapitola 11

Navržený systém

Zbylá část této práce se zabývá praktickým využitím informací z předchozích kapitol, především pak regulovaných metod syntaktické analýzy. Cílem této kapitoly je představit systém nástrojů umožňující syntaktickou analýzu založenou na regulovaných gramatikách. Tyto nástroje byly dále rozšířeny na plnohodnotný překladač zápisu gramatiky na kód tohoto syntaktického analyzátoru. Význam tohoto rozšíření demonstruje i samotná implementace, která je na tomto principu postavena.

Při návrhu systému je třeba zohlednit některé zásadní vlastnosti, které jeho implementace musí splňovat. Mezi základní požadavky patří jednoduchost implementace, nezávislost na architektuře operačního systému a programovacího jazyka, možnost jednoduchého rozšíření jednotlivých modulů a jednoduché používání a integrace do existujících řešení. Z těchto důvodů byl pro popis návrhu a některých částí implementace zvolen více abstraktní přístup. Při praktickém návrhu je navíc potřeba zohlednit požadavky kladené na moderní překladače. Uvedené důvody vedly k prvotnímu rozdělení systému na moduly *Generátor*, *Parser* a *Editor*, které jsou v následujících kapitolách podrobně popsány.

Dále byl navržen definiční metajazyk pro zápis bezkontextových gramatik, který byl následně rozšířen o podporu zápisu několika typů regulovaných pravidel. Tato regulovaná pravidla vychází ze speciálních typů gramatik, které byly předmětem předcházejících kapitol (viz 4.6, 4.7 a 4.8). V úvodu kapitoly je detailně, včetně dvou příkladů, představen tento jazyk pro zápis bezkontextových a vybraných regulovaných gramatik. Závěr kapitoly je věnován architektuře navrženého systému, jeho komponent, modulů, a vzájemných vztahů mezi nimi.

11.1 Syntaxe definičního metajazyka

Pro důkladný popis vstupní gramatiky je nutno zavést speciální jazyk (*definiční metajazyk*). Při návrhu syntaktických struktur tohoto jazyka však musíme dbát na skutečnost, že pomocí něj bude muset existovat možnost popisovat jiné jazyky, včetně navrhovaného definičního metajazyka. Ukázka popisu definičního metajazyka tímto jazykem je zařazena v přílohách této práce. Z tohoto důvodu je jeho syntaxe velmi strohá a sémantika prostá. Elementární syntaktickou strukturou jsou zápisy terminálních a neterminálních symbolů. Terminálním symbolem se stane každá posloupnost symbolů, která odpovídá standardní definici proměnné pro jazyk C (viz [6]) libovolné kladné délky, nebo právě jeden tisknutelný znak. Každý neterminální symbol je terminálním symbolem uzavřeným v ostrých závorkách (například `<S>`). Pro zápis prázdného řetězce (ε) je použit symbol `e`, což však nijak nezabraňuje výskytu symbolu `e` i jako terminálního symbolu (jen se jeho zápis musí označit jinak). Každá metainformace popisující vlastnosti překladu se nachází na začátku řádku a je uvozena znakem `@`. Tyto metainformace označujeme jako *anotace*.

Přehled dostupných anotací a dalších jazykových konstrukcí je předmětem následujících podkapitol. Řádkové komentáře mají stejný zápis, jako v jazyce C (viz [6]).

Nejdůležitější částí navrženého jazyka a systému obecně je však způsob spolupráce přeložené vstupní gramatiky a kódu uživatele. Jak již bylo zmíněno v úvodu této kapitoly, tato spolupráce nesmí být závislá na konkrétním programovacím jazyce, nebo platformě operačního systému, na kterém je výsledný kód provozován. Ačkoliv se nabízí více možností, jak tuto spolupráci umožnit, vybrány byly operace vygenerovaného kódu nad uživatelskými objekty. Tyto objekty existují jak v zápisu gramatiky definičním metajazykem, tak ve vygenerovaném kódu. Konkrétně se jedná o uložení tokenu na zásobník a volání uživatelské funkce, neboť obě tyto operace jsou dostupné ve většině moderních programovacích jazyků. Konkrétní příklad demonstrující tohle propojení je řazen v příloze této práce věnující se možnosti výstavby nového překladače.

11.1.1 Zápis pravidel

Zápis pravidel představuje nejdůležitější syntaktickou konstrukci definičního metajazyka. Pro označení omezení na využití daného přepisovacího pravidla v rámci regulovaných gramatik (viz 4) je rámci tohoto textu využit pojem *regulace*. Každé pravidlo musí začínat na novém řádku, a to volitelnou regulací. Každá regulace končí dvojtečkou, za kterou následuje samotné pravidlo. Přehled podporovaných regulací je poskytuje tabulka 11.5. Podtržená část regulace označuje identifikátor pravidla pro jeho adresaci v rámci regulací. Detailní popisy syntaxe regulovaných pravidel jsou obsaženy v definičním souboru definičního metajazyka v příloze této práce.

Tabulka 11.5: Příklady zápisů regulovaných pravidel

Název regulace	Příklad zápisu	Odpovídající regulované pravidlo
Matice	`$1, 1, 2+: <A> -> a` `$2: <B> -> b`	$\{(A, a), (B, b)\} \subseteq P$ $(((A, a), -), ((B, b), +)) \in M$
Program	`$1, $1 | $2: <A> -> a` `$2: <B> -> b`	$\{(A, a), (B, b)\} \subseteq P$ $((A, a), \{(A, a)\}) \in R$ $((A, a), \{(B, a)\}) \in F$
Kontext	`$1, <B> | <C>: <A> -> a`	$\{(A, a), (B, b)\} \subseteq P$ $((A, a), \{<B>\} \in R$ $((A, a), \{<C>\} \in F$

Pro maticové gramatiky (viz 4.6) je regulací pravidla prvek matice, který daným pravidlem začíná. Tuto regulaci označujeme jako *matice*. Zápis matice musí začínat identifikátorem pravidla, které je v matici první (to, u kterého je matice zapsána). Pro gramatiky s nahodilým kontextem (viz 4.7) jsou regulací množiny povolených a omezujících kontextů. Tuto regulaci označujeme jako *kontext*. Pro programované gramatiky (viz 4.8) je regulací odpovídající prvek z množiny pravidel úspěchu a prvek z množiny neúspěchu. V případě, že takový prvek není v množinách obsažen, je jeho zápis prázdný. Regulaci těchto pravidel nazýváme *program*.

V syntaxi zápisu regulací se pro oddělení množin (a seznamů) využívá znak „|“. Při použití regulace pravidel je nutno zvolit pouze jeden z uvedených typů pro celou gramatiku. Nelze tedy pro některá pravidla využít regulaci maticí, a pro jiná regulaci programem. Lze však regulovat

jen určitá pravidla, což ale dle algoritmu výběru přepisovacích pravidel příslušné metody výběru přepisovacího pravidla (viz 10.1, 10.2 a 10.3) může zcela znemožnit výběr některých pravidel, která regulaci neobsahují. Tato skutečnost zásadním způsobem ovlivňuje důsledky využití kvantifikátorů generujících syntetické neterminální symboly (viz 11.1.3).

11.1.2 Anotace

Anotace obsahují metainformace upravující požadované chování systému. Jejich přehled je v tabulce 11.6. Tučně vyznačené anotace jsou pro každou definici gramatiky povinné. Anotace `@Name` umožňuje přejmenovat terminální symbol v rámci dané definice, a současně změnit i jeho název ve vygenerovaném kódu, což je užitečné pro pojmenování struktur, které nemohou být součástí syntaxe nebo sémantiky výstupního kódu (čárky, uvozovky, závorky atd.). Anotace `@Lex` umožňuje definovat lexikální jednotku regulárním výrazem, a tedy i například přejmenování terminálního symbolu `e`.

Tabulka 11.6: Tabulka anotací

Anotace	Parametry	Význam
`@Main`	Neterminální symbol	Označení vstupního neterminálního symbolu
`@Stack`	Terminální symbol	Deklarace uživatelského zásobníku
`@Func`	Terminální symbol	Deklarace uživatelské funkce
`@Name`	2 terminální symboly	Přejmenování terminálního symbolu
`@Package`	Terminální symbol	Výsledný kódový balíček
`@OutputFile`	Terminální symbol	Soubor pro uložení výstupu
`@ResultClassName`	Terminální symbol	Název třídy výsledného parseru
`@Binary`	-	Vygenerování binární reprezentace
`@Regulated`	-	Vygenerování kódu regulované syntaktické analýzy (výchozí)
`@NoAutoBuild`	-	Zabránění automatickému překladu (pro možnou budoucí integraci do existujících nástrojů)
`@Lex`	Terminální symbol, Textový řetězec	Definice lexému pomocí regulárního výrazu
`@Unaddressed`	-	Vygenerování i nepoužitých lexémů
`@Include`	Textový řetězec	Vložení definičního souboru
`@Include_once`	Textový řetězec	Unikátní vložení definičního souboru
`@Ignore`	Textový řetězec	Regulární výraz specifikující ignorované lexémy (bílé znaky a komentáře)
`@EndOfFile`	Terminální symbol	Specifikace terminálního symbolu pro označení konce vstupu

11.1.3 Užití kvantifikátorů

Definiční metajazyk dovoluje užití kvantifikátorů pro zkrácení zápisu pravidel. Užitím kvantifikátoru však vznikají další neterminální symboly a pravidla, což nemusí být vždy žádoucí s ohledem na vlastnosti regulovaného překladu. Přehled použitelných kvantifikátorů shrnuje tabulka

11.7. V uvedené tabulce je sice kvantifikátor aplikován na terminální symbol, ale obecně ho lze aplikovat i na neterminální symbol, nebo na sjednocení úseku pravé části pravidla. Kvantifikátory je možno téměř libovolně sdružovat, pokud výsledná gramatika zůstane LL(1) gramatikou, nebo bude splňovat podmínky kladené na deterministickou analýzu příslušné regulované gramatiky.

Význam posledního sloupce tabulky kvantifikátorů spočívá v praktickém znázornění důsledků užití daného kvantifikátoru. Podtržená část pravidla vyjadřuje zápis v definičním metajazyce, avšak místo této části se stane součástí tohoto pravidla neterminální symbol zastupující výskyt daného kvantifikátoru (dvojitě podtržená část pravidla). Kvantifikátor dále přidá do gramatiky nová pravidla, která ho realizují. Takto vzniklá pravidla označujeme jako *syntetická pravidla* a každý nově vzniklý neterminální symbol jako *syntetický neterminální symbol*. Syntetická pravidla však nelze dále regulovat (není definovaná implicitní regulace), proto jejich využití v rámci některých regulovaných gramatik může porušit pravidla kladená na danou gramatiku a znemožnit její analýzu. Možnosti implicitní nebo explicitní regulace syntetických pravidel jsou, podobně jako další regulované gramatiky, oblastí vyžadující další zkoumání.

Tabulka 11.7: Tabulka dostupných kvantifikátorů

Kvantifikátor	Význam	Překlad užití do pravidel

( )	Sjednocení úseku pravé části pravidla	`<A> -> (a b c)`
		`<A> -> <A_2>`
		`<A_2> -> a b c`

*	Libovolný počet výskytů	`<A> -> a*`
		`<A> -> <A_star>`
		`<A_star> -> e`
		`<A_star> -> a <A_star>`

+	Libovolný nenulový počet výskytů	`<A> -> a+`
		`<A> -> <A_plus>`
		`<A_plus> -> a <A_star>`

?	Volitelný výskyt	`<A> -> a?`
		`<A> -> <A_opt>`
		`<A_opt> -> e`
		`<A_opt> -> a`

|	Logický oddělovač více možností výskytů	`<A> -> a | b`
		`<A> -> <A_or>`
		`<A_or> -> a`
		`<A_or> -> b`

11.1.4 Práce s uživatelskými objekty

Nedílnou součástí syntaxe je podpora práce s definovanými uživatelskými objekty. Každému užití uživatelského objektu musí předcházet jeho deklarace. Pro každý typ objektu je definována pouze jediná operace. Pro uživatelské zásobníky je to uložení daného tokenu na zásobník (např. zápis `a => s1 => s2 => s3` způsobí uložení tokenu `a` do zásobníků `s1`, `s2` a `s3`). Zápisy uložení na zásobník mohou být pouze na pravé straně přepisovacích pravidel za terminálními

symboly. Pro deklarované uživatelské funkce je to volání dané funkce (např. zápis `<a> $f1 $f2 $f3` způsobí zavolání funkcí `f1`, `f2` a `f3` po kompletní redukci neterminálního symbolu `<a>`). Volání uživatelské funkce může být pouze na pravé straně přepisovacích pravidel, a to za terminálními i neterminálními symboly. Pořadí volání funkcí uvedených za sebou je nedefinované. V případě práce se zásobníky a volání funkcí v rámci stejného neterminálního symbolu se provedou nejdříve operace nad zásobníky, a následně volání funkcí bez ohledu na pořadí zápisu (operace se zásobníky se zapisují před voláními funkcí). Základní myšlenka tohoto přístupu je možnost přístupu k daným uživatelským objektům z kódu uživatele, který zahájil syntaktickou analýzu. Vstupní řetězec je zpracováván zleva, tokeny jsou postupně ukládány na uživatelské zásobníky a jsou volány uživatelské funkce, které mohou zajišťovat například sémantické kontroly, nebo například i generování výstupního kódu.

11.1.5 Příklad

Příklad 11.1 znázorňuje využití maticové gramatiky pro příjem vstupní syntaktické struktury $type^n\ name^n\ value^n$, která není bezkontextová. Metoda regulované syntaktické analýzy (viz algoritmus 10.1) navíc zajistí, že v době volání uživatelské funkce *HaveVar* obsahují dna uživatelských zásobníků *Type, Name, Value* po řadě typ, název a hodnotu proměnné, jako by se jednalo o sekvenční definici $(type\ name\ value)^n$.

```
 1: @Main <var_def>
 2: @EndOfFile <EOF>
 3: @Ignore "[ \t\n\r]" # This is ignored
 4: @Stack Type
 5: @Stack Name
 6: @Stack Value
 7: @Func HaveVar
 8: $1,1: <var_def> -> <type> <A> <name> <B> <value>
 9: $2,2,3: <A> -> <type> <A>
10: $3: <B> -> <name> <B> <value>
11: $4, 4,5: <A> -> e
12: $5: <B> -> e
13: $6, 6: <type> -> Name => Name
14: $7, 7: <name> -> Type => Type
15: $8, 8: <value> -> Value => Value $HaveVar
16: @Lex Name "[a-zA-Z_][a-zA-Z0-9_]*"
17: @Lex Type "(int|string)"
18: @Lex Value "(\"[a-zA-Z_0-9]*\")|([0-9]+)"
```

Příklad 11.1: Příklad zápisu syntaktické struktury maticovou gramatikou

11.2 Struktura systému

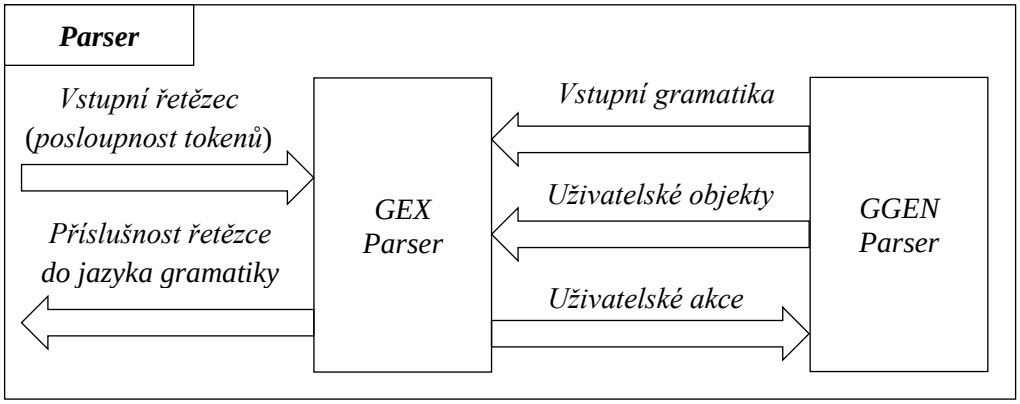
Navržený systém lze strukturovat více způsoby. Nejvhodnější způsob je rozdělit celý systém na moduly v závislosti na jejich účelu. Z důvodů přehlednosti byl systém rozdělen na moduly *Parser* a *Generátor*. První z uvedených reprezentuje myšlenku jednoduchého provedení regulované syntaktické analýzy. Druhý z uvedených představuje překladač zápisu gramatiky definičním metajazykem do ´výstupního formátu daného strategií výstupu (binární výstup, případně zdrojový kód

vstupní části syntaktického analyzátoru). Modul *Generátor* je speciální nástavbou modulu *Parser*, jehož funkci využívá pro syntaktickou analýzu vstupní gramatiky.

11.2.1 Parser

Základním modulem systému je modul *Parser*, který obsahuje kód parametrizovatelné syntaktické a lexikální analýzy. Jeho funkce je pouhé ověření příslušnosti vstupního řetězce do jazyka vstupní gramatiky a provedení příslušných uživatelských akcí. Na obrázku 11.1 je vyobrazena architektura tohoto modulu.

Vstup je tvořen dvěma částmi. První část (na obrázku vpravo) je kód vstupní gramatiky a realizace uživatelských objektů. Zmíněná realizace uživatelských objektů může být reprezentována například předáním objektu implementujícího vhodné rozhraní, nebo dědičností. Tuto vstupní část reprezentuje blok nazvaný *GGEN Parser*, který je výstupem modulu Generátor. Blok GGEN Parser je považován za součást modulu Parser.



Obrázek 11.1: Blokové schéma navrženého parseru

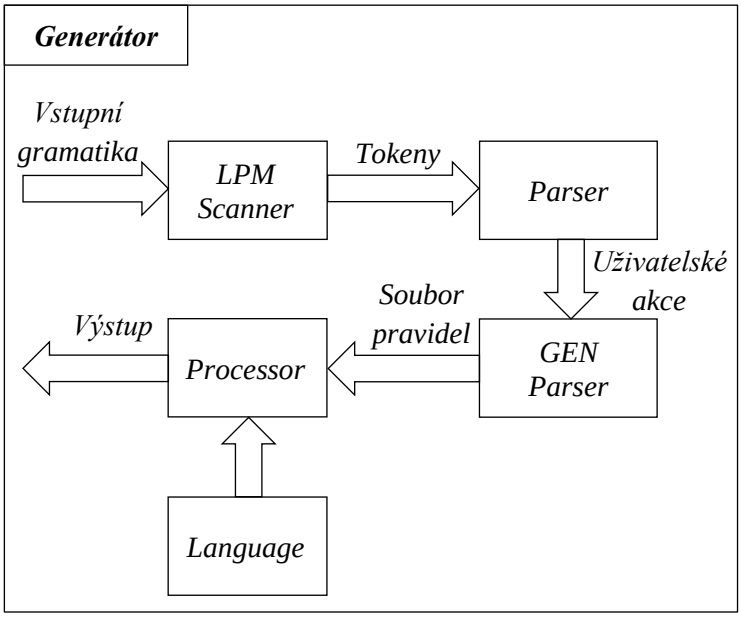
Druhá část vstupu (na obrázku vlevo) je vstupní řetězec, nad kterým se má provést syntaktická analýza uvnitř bloku *GEX Parser*. Jádrem tohoto bloku jsou neměnné algoritmy syntaktické analýzy popsané v kapitole 10. Pro podporu uživatelských akcí je však nutno doplnit množinu terminálních symbolů vstupní gramatiky o speciální symbol pro každou uživatelskou akci. Při redukci takových symbolů bude tyto akce algoritmus nejen odebírat ze zásobníku, ale také provádět.

11.2.2 Generátor

Modul generátoru je možno charakterizovat jako program, který na svém vstupu pojme definiční soubor gramatiky a výstupem je kód bloku *GGEN Parser* z obrázku 11.1. Struktura tohoto programu je znázorněna na obrázku 11.2. V závislosti na použitém bloku procesoru (na obrázku blok produkující výstup) je možno určovat formát výstupu, ať už se má jednat o kód části syntaktického analyzátoru, nebo třeba binárního produktu překladu. Binární produkt překladu může být využit například pro analýzu gramatik bez nutnosti generování jejich syntaktických analyzátorů. Využitím jazykového rozhraní (blok *Language* na obrázku 11.2) při procesu generování výstupu je splněn požadavek generování části syntaktického analyzátoru v libovolném programovacím

jazyce. Jazykové rozhraní obsahuje definici syntaktických struktur výstupního programovacího jazyka.

Je-li vstupem zápis gramatiky definičního metajazyka, je výstupem část programu (blok *GGEN Parser* na obrázku 11.1), který nad touto gramatikou provedl syntaktickou analýzu, čehož se dá elegantně využít při inkrementálním zanášení nových funkcí do celého systému.



Obrázek 11.2: Architektura navrženého generátoru

11.3 Formální pohled na navržený překladový model

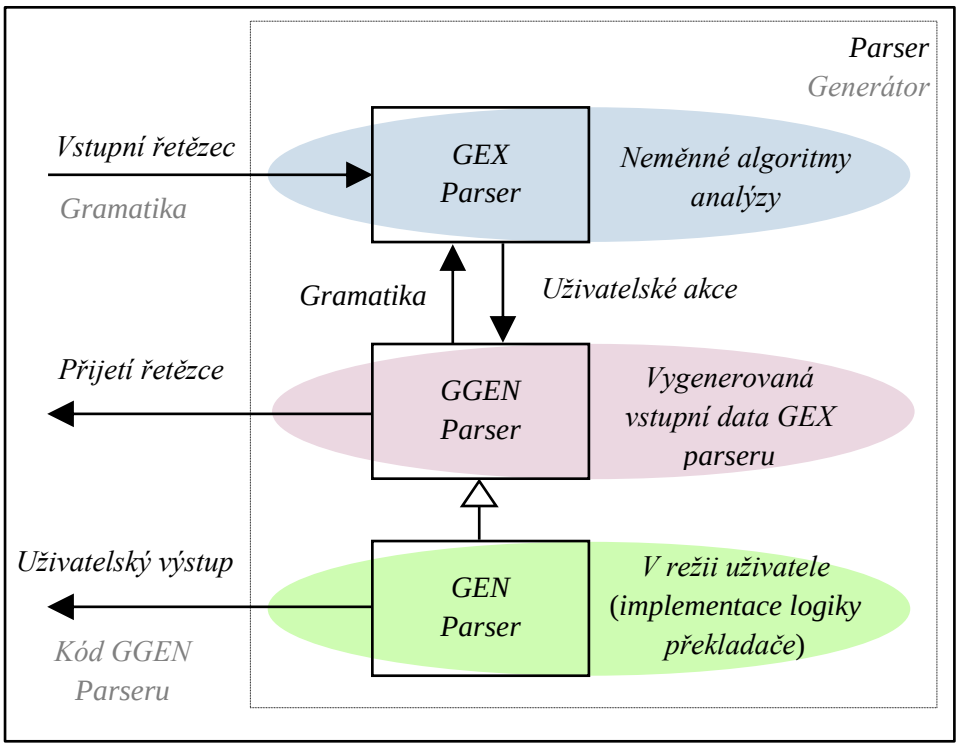
Z formálního hlediska modul *Parser* realizuje formální překlad z definičního metajazyka gramatiky do jazyka akcí nad uživatelskými objekty, jejímiž atributy jsou části vstupního řetězce. Při pohledu na modul navrženého generátoru se jedná o překlad z definičního metajazyka na akce nějakého programovacího jazyka. Při vhodném rozmístění uživatelských akcí v zápisu pravidel lze z výstupní posloupnosti aplikací uvedených uživatelských akcí odvodit posloupnost pravidel, která byla užita při syntaktické analýze navrženými algoritmy.

Kapitola 12

Implementace systému

Jak bylo v předchozí kapitole naznačeno, implementace systému spočívá v implementaci modulů *Generátor*, *Parser* a *Editor*. Dále je zapotřebí implementovat modul lexikálního analyzátoru označovaného jako *LPM scanner* (viz 12.1). Jako implementační jazyk byla vybrána Java, především díky možnosti jednoduché přenositelnosti spustitelných souborů. Moduly *Parser* a *Generator* jsou svou strukturou velmi podobné. Oba na svém vstupu pojmou vstupní řetězec, a na základě zabudované gramatiky ověřují příslušnost tohoto řetězce v jazyce své gramatiky. Pokud je vstupní řetězec v jazyce této gramatiky obsažen, je vyprodukován i uživatelský výstup. Pokud je vstupním řetězcem modulu parser zápis gramatiky, a uživatelským výstupem kód GGEN Parseru, jedná se o Generátor. Blokové schéma Parseru je na obrázku 12.1, který graficky znázorňuje rozdíl mezi oběma moduly. Obrázek byl s úpravou převzat z autorova posteru prezentovaného na konferenci Excel@FIT 2019.

Popis přeložení, detailní popis korektního využití a spuštění všech implementovaných modulů je zařazen v příloze této práce. Přiloženy jsou zdrojové texty všech navržených nástrojů, definiční soubor definičního metajazyka a soubor základních testů.



Obrázek 12.1: Schéma parseru a generátoru

12.1 LPM Scanner

LPM scanner (z anglického výrazu ***L**ongest **P**ossible **M**atch*) je lexikální analyzátor založený na speciálním typu zásobníkového automatu, který ukládá navštívené koncové stavy, a při nemožnosti vykonání dalšího přechodu se vrátí do konfigurace odpovídající naposled navštívenému koncovému stavu. Oproti modelu lexikálního analyzátoru založeném na konečných automatech popsanému v předchozích kapitolách dosahuje lepších výsledků. Implementace tohoto modulu v Javě využívá podpory vestavěných regulárních výrazů a jejich hledání v řetězci.

Je však nutno brát v potaz prioritu lexémů, neboť se jejich definice mohou překrývat. Určení priority v definičním metajazyce se provádí implicitním užitím lexému bez definice (nejvyšší priorita), nebo anotací `@Lex` (viz tabulka 11.6), přičemž dřívější výskyt anotace značí vyšší prioritu v rámci anotací. V rámci definičního metajazyka, na rozdíl od uživatelských objektů, není nutné uvádět deklarace lexémů před jejich využitím. Nejvyšší prioritu mají lexémy, které nejsou zpracovávány anotované anotací `@Ignore`.

Tento typ lexikálního analyzátoru je využit ve všech modulech navrženého systému, ve kterých probíhá lexikální analýza. Vstupem uvedeného *LPM scanneru* je vstupní řetězec, regulární výrazy popisující jednotlivé lexémy a regulární výrazy popisující struktury, které nejsou součástí vstupu (netisknutelné znaky, komentáře atp.). Do vstupu rovněž patří terminální symbol reprezentující konec vstupního řetězce. Výstupem je funkce, jejíž zavolání vrátí následující token ze vstupu. Při procesu syntaktické analýzy vstupního řetězce může docházet k inkluzi externích řetězců (anotace `@Include` a `@Include_once`), přičemž tyto externí řetězce jsou vloženy na pozice anotací, které jejich vložení předepsaly.

Celý postup lexikální analýzy shrnuje algoritmus 12.1, který vyjadřuje základní myšlenku rozlišovat struktury, které se mohou vyskytovat ve vstupním řetězci, ale nemají se brát v potaz. V úvodní části algoritmu (kroky 1 až 8) se tyto struktury vyhledají, a z počátku vstupního řetězce odeberou. Pokud je vstupní řetězec prázdný (krok 2), je výstupem algoritmu informace o tom, že vstupní řetězec již neobsahuje žádné znaky. Použitím cyklu repeat-until je navíc zajištěno, že tato kontrola proběhne vždy, a její výstup je korektní i v případě, že vstupní řetězec již obsahuje pouze takové struktury, které se mají ignorovat. Druhá část algoritmu (kroky 9 až 17) zajišťují výběr množiny takových tokenů z počátku vstupního řetězce, které se nachází na počátku vstupního řetězce a jejich délka je maximální. Pokud je tato množina prázdná, neleží na počátku vstupního řetězce žádný rozpoznatelný lexém, a jedná se o lexikální chybu (krok 18 a 19). V opačném případě se z této množiny vybere takový lexém, jehož priorita je nejvyšší, a tento se odebere z počátku vstupního řetězce a je výstupem algoritmu.

Vstup: Regulární výrazy popisující jednotlivé lexémy (*regexes*) s prioritami
Regulární výrazy popisující struktury, které se mají ignorovat (*comments*)
Terminální symbol (Token) označující konec vstupu (*EOF*)
Vstupní řetězec *w*

Výstup: Následující token ze vstupu
Výstupní řetězec *w*

Metoda:

```
(1)   repeat
(2)     if w is empty do
(3)       output(EOF, w)
(4)     w' ← w
(5)     for each comment in comments do
(6)       if comment matches w do
(7)         w ← removeMatch(w, comment)
(8)   until w' = w
(9)   lexems ← ∅
(10)  maxMatchSize ← 0
(11)  for each regex in regexes do
(12)    if regex matches w do
(13)      if maxMatchSize = matchSize(w, regex) do
(14)        lexems ← lexems ∪ {regex}
(15)      else if maxMatchSize > matchSize(w, regex) do
(16)        lexems ← {regex}
(17)        maxMatchSize ← matchSize(w, regex)
(18)  if lexems = ∅ do
(19)    Chyba
(20)  else do
(21)    sort lexems by priority
(22)    lexem ← first(lexems)
(23)    token ← new Token(lexem, match(w, lexem))
(24)    output(token, removeMatch(w, lexem))
```

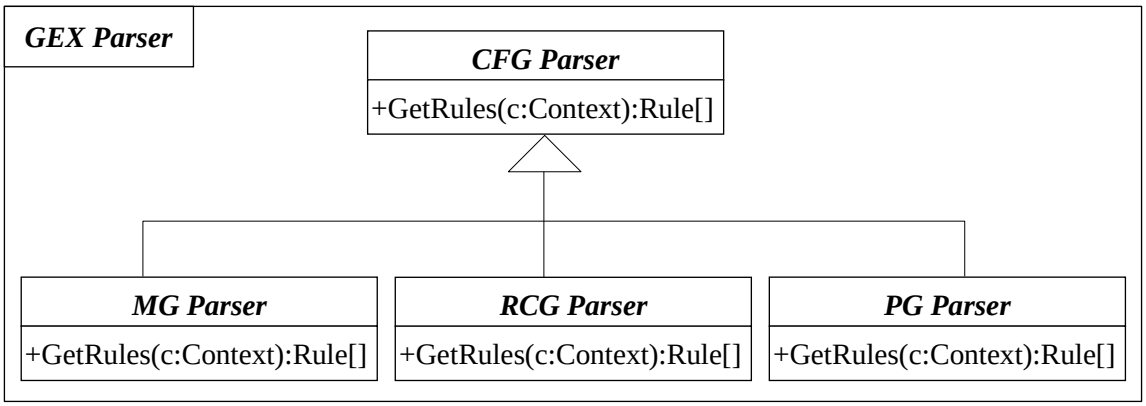
Algoritmus 12.1: Algoritmus lexikální analýzy LPM scanneru

12.2 Parser

Jádrem implementace je parser. Jak již bylo v předcházejícím textu uvedeno, primárním cílem parseru je ověření příslušnosti vstupního řetězce v jazyce dané gramatiky. Pokud se příslušnost vstupního řetězce v tomto jazyce potvrdí, produkuje parser ještě sekundární výstup. Určení obsahu a význam tohoto výstupu je zanecháno na uživateli. Modul je složen z 3 částí, a to *GEX Parseru, GGEN Parseru* a *GEN Parseru*. První z uvedených obsahuje neměnné algoritmy

syntaktické analýzy, přičemž konkrétní varianta využitá pro analýzu záleží na typu gramatiky v bloku GGEN Parser.

Základní myšlenka výběru odpovídajícího algoritmu spočívá již v samotném vytváření instance třídy *GEN Parser*. Situaci přehledně popisuje obrázek 12.2, na kterém je znázorněno přepisování metody *GetRules* v jednotlivých podtřídách třídy *CFG Parser*, což přesně odpovídá abstrahování metody výběru přepisovacích pravidel zmíněné v kapitole 10. Při vstupní bezkontextové gramatice se vytvoří instance třídy *CFG Parser*, při maticové gramatice instance třídy *MG Parser* atd. Třída *GGEN Parser* obsahuje deklarace uživatelských objektů a data vstupní gramatiky (instance objektů v implementačním jazyce, tedy pole terminálních a neterminálních symbolů, LL tabulku, pole přepisovacích pravidel atd.). Tato třída slouží jako vstup algoritmů syntaktické analýzy a je specifická pro každou definici gramatiky, pro kterou je vygenerována.

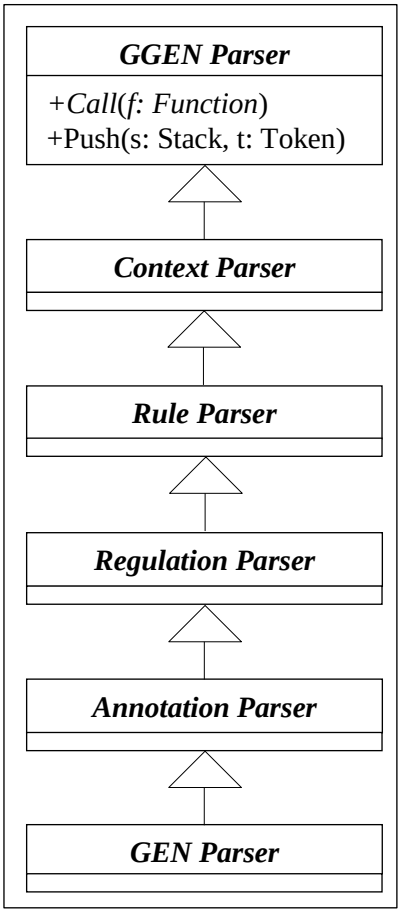


Obrázek 12.2: Diagram tříd navrženého parseru

Část modulu nazvaná *GEN Parser* označuje třídu, jejíž obsah tvoří logiku překladače. V rámci této třídy je na uživateli zanechána zodpovědnost za implementaci funkcí, které definoval ve vstupní gramatice. Při volání těchto funkcí lze využít obsah deklarovaných uživatelských zásobníků, což v principu představuje propojení analyzovaného vstupního řetězce a kódu uživatele.

12.3 Generátor

Modul navrženého systému nazvaný generátor označuje program s funkcemi podobnými tradičnímu překladači. Je založen na modulu předchozím, který dále rozšiřuje.



Obrázek 12.3: Hierarchie tříd *GEN Parseru*

Jádro tohoto modulu tvoří blok *GEN Parser*, který je realizován třídou delegující zodpovědnosti za zpracování jednotlivých syntaktických struktur hierarchickou dědičností. Situaci znázorňuje obrázek 12.3, ve kterém byly některé vztahy a obsahy tříd v rámci názornosti vypuštěny. V nejvyšší úrovni je třída *GGEN Parser* vygenerovaná právě pro gramatiku definičního metajazyka. Tato třída je složena z definic uživatelských zásobníků a deklarací abstraktních uživatelských funkcí. Jednotlivé nižší úrovně tyto uživatelské funkce definují a využívají dat úrovní vyšších (např. *Regulation Parser* umožňuje pracovat s daty třídy *Rule Parser* pro vkládání regulací atp.) a na nejnižší úrovni je bázová třída celého generátoru. Tato bázová třídy má přístup k datům všech hierarchicky nadřazených tříd, a celou analýzu inicializuje.

Uživatelským výstupem generátoru může být v závislosti na použité strategii kód GGEN Parseru, který nachází uplatnění v obdobné hierarchii a významu, jako na obrázku 12.3, tedy jako třída obsahující vstupní data a uživatelské objekty syntaktické analýzy nějakého dalšího syntaktického analyzátoru nebo překladače. Dalším možným výstupem je binární reprezentace vstupní gramatiky (anotace `@Binary`, viz tabulka 11.6), což je výhodné především pro nástroje pracující nad gramatikami zadávanými za běhu, jakým je například modul *Editor* popsaný v následující kapitole. Realizace daného typu výstupu se provádí přes strategii generátoru a implementaci jazykového rozhraní (viz obrázek 11.2).

Další možnou implementací strategie výstupu může být například metoda rekurzivního sestupu pro bezkontextové LL(1) gramatiky. Z praktických důvodů (nemožnost využití pro analýzu regulovaných gramatik) se však v implementaci nenachází. Implementace dále nabízí pouze implementaci jazykového rozhraní pro jazyk Java, a to pouze v omezeném rozsahu. Doplnění jazykového rozhraní o univerzální datové typy a operace nad nimi je jednou z možných cest pokračování projektu.

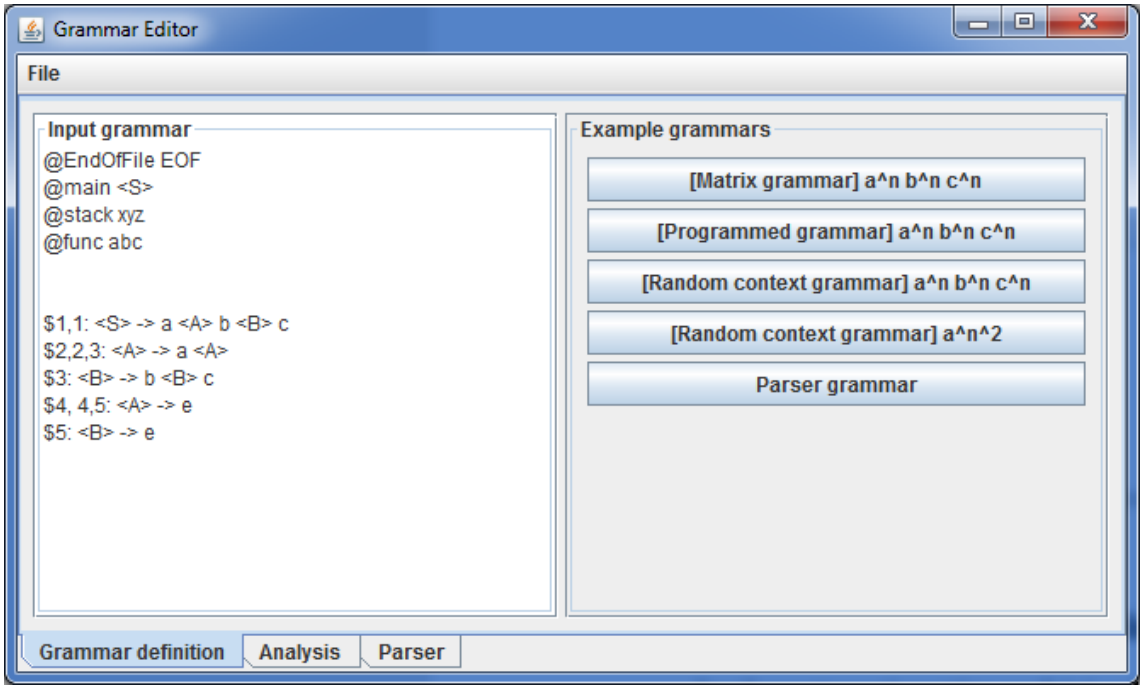
12.4 Editor

Poslední modul navrženého systému je editor. Tento modul představuje nástroj s grafickým uživatelským rozhraním, který umožňuje zapisovat a verifikovat gramatiku v definičním metajazyce, analyzovat využití uživatelských objektů a provádět vizuální syntaktickou analýzu zapsané

gramatiky a libovolného vstupního řetězce. Grafické uživatelské rozhraní je realizováno knihovnou uživatelských prvků Swing[1]. Význam tohoto nástroje spočívá především při zkoumání regulovaných gramatik a jejich deterministické syntaktické analýzy.

Uživatelské rozhraní je rozděleno podle účelu na 3 části, a sice definice gramatiky (*Grammar Definition*), analýza gramatiky (*Analysis*) a syntaktická analýza založená na této gramatice (*Parser*). Přepínání mezi okny je realizováno záložkami v dolní části aplikace. Celá aplikace je navržena s úmyslem nabídnout sadu intuitivních funkcí pro práci s gramatikami a jejich vizualizace. Velikost většiny prvků rozhraní je měnitelná a celá aplikace reaguje na změnu velikosti okna. Pro co nejlepší uživatelskou přívětivost aplikace nabízí ukládání sezení (vstupní gramatika a vstupní řetězec) a nahrávání sezení z existujících souborů, nebo z ukládané historie.

První částí uživatelského rozhraní je definice gramatiky. Toto okno obsahuje editovatelné textové pole, do kterého se vloží definice gramatiky v syntaxi dané definičním metajazykem. V pravé části okna je dále na výběr z několika příkladů definic gramatik, přičemž některé byly předmětem příkladů kapitoly 10. Po úspěšném zápisu je možno přepnout záložkou v dolní části okna do jiné části uživatelského rozhraní. Při chybném zápisu gramatiky je uživatel upozorněn odpovídající chybovou hláškou a kurzor editovatelné části okna je přesunut do místa výskytu dané chyby, je-li známo. Při využití anotací `@Include` nebo `@Include_once` je pozice kurzoru irelevantní. Uživatelské rozhraní této části je zobrazeno na obrázku 12.4.

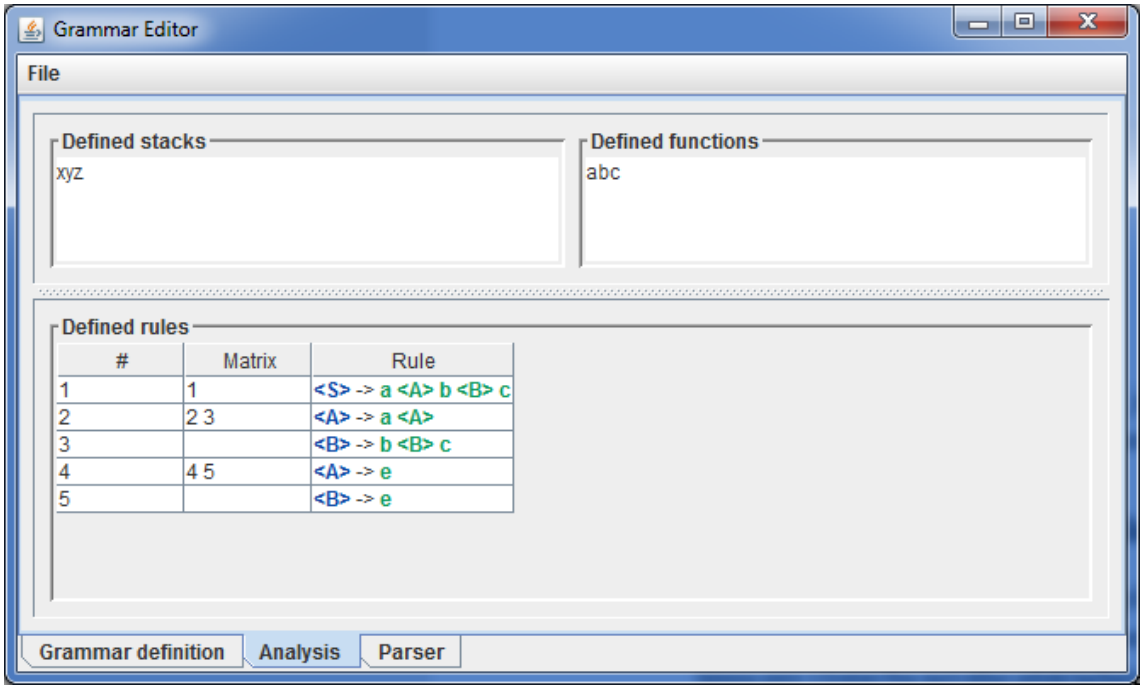


Obrázek 12.4: Uživatelské rozhraní definice gramatiky

Další částí uživatelského rozhraní se nachází okno analýzy gramatiky. Význam tohoto okna spočívá v kontrole zápisů deklarací a vygenerovaných syntetických neterminálních symbolů

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/swing/

a pravidel. Okno této části je na obrázku 12.5. V horní části jsou zobrazeny deklarace uživatelských objektů a v dolní části je seznam pravidel. Tento seznam podporuje adekvátní zobrazení všech podporovaných regulací pro snadné odhalení chybného zápisu. Identifikátory pravidel jsou přečíslovány od 1.

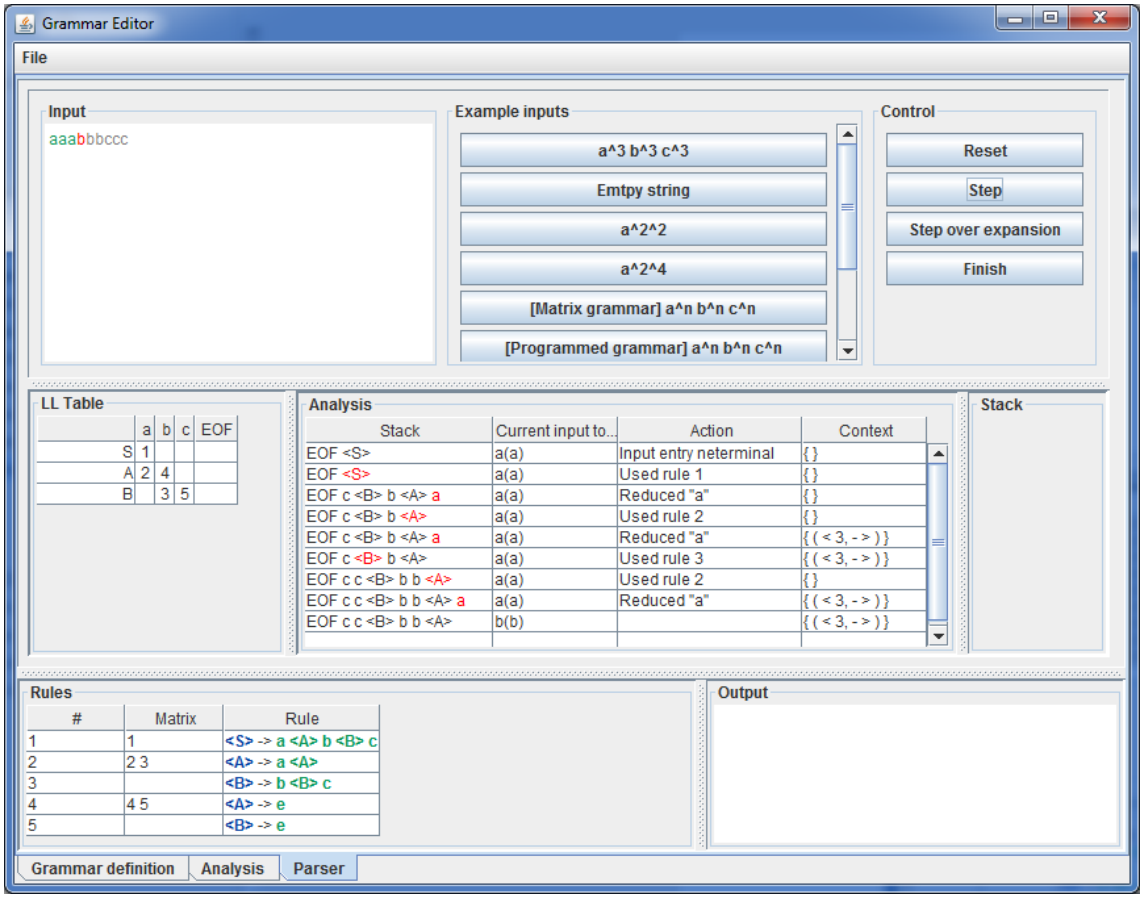


Obrázek 12.5: Uživatelské rozhraní analýzy gramatiky

Poslední a bezpochyby nejhodnotnější částí uživatelského rozhraní je okno syntaktické analýzy založené na gramatice v definiční části. Uživatelské rozhraní této části editoru je na obrázku 12.6. V levé horním rohu je editovatelná textová oblast umožňující zadání vstupního řetězce. Pro rychlejší práci je podporováno vkládání předdefinovaných vstupů do této oblasti. Mezi podporovanými vstupy patří ukázkové řetězce ukázkových gramatik, ale také gramatiky samotné. Zápis gramatik je možno analyzovat gramatikou definičního metajazyka, která je v seznamu na posledním místě označena jako *Parser Grammar*.

V pravém horním rohu je ovládací panel analýzy. Jednotlivá tlačítka umožňují vykonávat kroky algoritmů syntaktické analýzy s odpovídajícím projevem ve zbylých částech uživatelského rozhraní. Při krokování algoritmu se znemožní úprava vstupního řetězce a analýza započne. V textové oblasti, kde se nachází vstupní řetězec, se zeleně vyznačí již zpracovaná část vstupního řetězce, červeně aktuální symbol na vstupu a šedě doposud nezpracovaná část vstupního řetězce.

V dolní části je opět zobrazen seznam pravidel s podobným významem jako v případě předchozí části. V pravém dolním rohu je výstupní část. Do této části jsou v průběhu analýzy zapisovány uživatelské akce, které se provedly. Lze tak pozorovat chování vygenerované části překladače (*GEX Parser*, viz 12.2) a odhadovat chování uživatelské části překladače (*GEN Parser*, viz 12.2).



Obrázek 12.6: Uživatelské rozhraní syntaktické analýzy založené na vstupní gramatice

V prostřední části uživatelského rozhraní se nachází LL tabulka. Výpočet jejího obsahu je založen na algoritmech prediktivní syntaktické analýzy z kapitoly 9. Protože jsou však podporovány regulované metody syntaktické analýzy, není obsah této tabulky limitován na jediné pravidlo uvnitř jejích buněk. Vpravo od LL tabulky se nachází panel analýzy, do kterého se postupně doplňují řádkové informace o postupu algoritmu probíhající syntaktické analýzy. Jedná se o tabulku obsahující zásobník, aktuální symbol na vstupu, kontext a akci, kterou na základě těchto informací algoritmus vykonal. Při označení nějakého řádku je obsah zásobníku, z důvodu přehlednosti, vypsán do panelu v pravé části.

Při využití poslední kontrolní akce (tlačítko *Finish*) se provede asynchronní zpracování zbylé části vstupního řetězce až do ukončení běhu algoritmu syntaktické analýzy. Během tohoto procesu jsou deaktivována ostatní tlačítka, vyjma tlačítka *Stop* v dolní části uživatelského rozhraní. Toto tlačítko je viditelné pouze při tomto procesu a jeho stisknutí proces ukončuje. Alternativní možnost ukončení je přepnutí do jiné části editoru (např. přechod do definiční části). Toto přepnutí resetuje stav syntaktické analýzy prováděné v této části editoru.

12.5 Testování implementovaných nástrojů

Pro řádné testování jednotlivých nástrojů je zapotřebí specifikovat moduly a jejich části, které budou podrobeny testům. Pro potřeby řádné funkčnosti se lze omezit pouze na moduly provádějící syntaktickou analýzy a generování na základě výstupních strategií. Pro testování modulů provádějících syntaktickou analýzy byla vytvořena testovací sada obsahující řadu syntaktických struktur a regulovaných gramatik, které je popisují. Do této sady byly zahrnuty především všechny kombinace vstupních gramatik a vstupních řetězců, které nabízí modul Editor v grafickém uživatelském rozhraní.

Testování výstupů modulů produkujících výstup však není jednoduché provádět, a to z důvodů plynoucích z nerozhodnutelnosti problému ekvivalence bezkontextových gramatik (viz [1] věta 4.26). Intuitivně připadá v úvahu porovnávat zdrojový kód analyzátoru, což však rovněž nepřipadá v úvahu, neboť jsou ve zmíněných modulech operace nad neuspořádanými množinami (pořadí výskytu definice jednotlivých symbolů není pevně dáno, protože jsou využity vestavěné objekty s řadou interních optimalizací). Zvoleným přístupem testování tohoto modulu se stala přístup provádějící zpětný překlad na zdrojový text gramatiky a následným porovnáváním bez ohledu na pořadí řádků textů a ignorovaných struktur obou porovnávaných gramatik. Tímto způsobem je možné porovnávat pouze takové gramatiky, nad kterými se neprovádí žádné transformace, tedy takové, které neobsahují kvantifikátory. Porovnávají se očekávané návratové hodnoty modulů a obsah standardního výstupu. Konkrétní způsob spuštění testů je řazen v přílohách této práce.

Kapitola 13

Závěr

Vytyčeným cílem práce je zavedení metod regulované syntaktické analýzy a syntaxí řízeného překladu a implementace aplikace těchto metod v oblasti kompilátorů. Práce poskytla ucelený pohled na formalismy užité v metodách syntaxí řízeného překladu a zavedla regulovaný převodník jako model regulovaného syntaxí řízeného překladu. Výhody regulovaných převodníků vůči zásobníkovým převodníkům byly názorně demonstrovány příklady. Pro účely aplikace zaměřené na regulovaný syntaxí řízený překlad byly formálně zavedeny speciální regulované gramatiky. Vyjadřovací síly uvedených regulovaných gramatik byly demonstrovány příklady. Pro deterministickou analýzu uvedených regulovaných gramatik byl navržen algoritmus s abstrakcí metody výběru přepisovacího pravidla. Zkoumání tohoto algoritmu a formální důkaz jeho správnosti je jednou z oblastní vyžadující další výzkum. Implementovanou aplikací je systém nástrojů umožňující jednoduchou práci se zmíněnými regulovanými gramatikami.

Praktická část práce se zabývala návrhem a implementací systému nástrojů pro podporu práce se zmíněnými speciálními typy gramatik. Pro popis tohoto systému nástrojů byl zvolen více abstraktní pohled, přičemž konkrétní implementační detaily jsou obsaženy v programové dokumentaci. Implementovanými nástroji jsou *Parser* ověřující členství vstupního řetězce v jazyku gramatiky, *Generator* produkující zdrojový kód syntaktického analyzátoru vstupní gramatiky (nebo binární reprezentaci vstupní gramatiky) a *Editor* jako nástroj s grafickým uživatelským rozhraním sloužící primárně jako demonstrační a výzkumná aplikace.

Pro popis gramatik byl zaveden speciální definiční metajazyk vyznačující se svou jednoduchostí a možností popsat sama sebe, čehož je elegantně využito pro vytvoření syntaktického analyzátoru tohoto jazyka. Tento definiční metajazyk byl navržen s ohledem na nezávislost na výsledné implementaci, ať už platformní, či jazykovou. Propojení s kódem uživatele probíhá na základě operací s uživatelskými objekty, jejichž definice je obsažena jak v zápisu vstupní gramatiky tímto jazykem, tak ve zdrojovém kódu vygenerovaného analyzátoru. Tento princip eliminuje nevýhody některých běžně dostupných řešení, jakými jsou například Yacc[1] či Bison[2], které očekávají v definici gramatiky syntaktické konstrukce výstupního programovacího jazyka. Část této práce zabývající se metodami regulované syntaktické analýzy a výstavby nových překladačů na zavedených principech byla prezentována na konferenci Excel@FIT 2019.

Dalším omezením výše zmíněných existujících řešení, které implementace částečně eliminuje, je značná závislost na výstupním programovacím jazyce. Zatímco existující nástroje neu-

[1] https://www.javatpoint.com/yacc
[2] https://www.gnu.org/software/bison

možňují jednoduchým způsobem produkovat výstup v libovolném programovacím jazyce, implementovaný nástroj využívá jazykové rozhraní jako svůj výstup. Implementace tohoto rozhraní obsahuje konkrétní syntaktické konstrukce výstupního programovacího jazyka, čímž se docílí částečné nezávislosti výstupu generátoru na programovacím jazyce (některé abstraktní datové typy jsou pevně dány svými názvy, výstupní jazyk musí podporovat implicitní správu paměti).

Implementovaný systém nachází uplatnění v řadě odvětví. Především pak při výstavbě nových překladačů a analyzátorů strukturovaných dat nejen regulovaných gramatik. Praktický popis a ukázka využití navrženého systému pro výstavbu nového překladače je zařazen v příloze této práce a současně v implementaci generátoru, který byl na zmíněných principech vystavěn. Další možné využití spočívá například ve zkoumání vlastností regulovaných gramatik a metod syntaktické analýzy na nich založených. Možné pokračování projektu spočívá v zavedení dalších typů regulovaných gramatik a metod syntaktické analýzy na nich založených, případně dodání možnosti implicitně či explicitně regulovat syntetická pravidla, což povede k dalšímu snížení

Dalším, neméně významným využitím, které se do budoucna nabízí je implementace překladače překládajícího konkrétní implementaci neměnných algoritmů a použitých datových objektů syntaktické analýzy v jazyce Java do jazyka popisujícího syntaktické konstrukce v navrženém jazykovém rozhraní. Implementací takového překladače by bylo možno generovat kompletní zdrojové kódy syntaktických analyzátorů nejen regulovaných gramatik. Dodáním implementace použitého lexikálního analyzátoru využívajícího pouze syntaktické konstrukce popsatelné dodaným jazykovým rozhraním lze dále uvažovat o komplexním nástroji pro výstavbu překladačů, který, oproti existujícím řešením, bude umožňovat generování lexikálních i syntaktických analyzátorů v programovacím jazyce, jehož definici poskytne uživatel nástroje stejným způsobem, jako je dodána definice výstupního jazyka Java dodaná autorem tohoto nástroje. Protože pro implementaci generátoru strategie výstupního kódu (anotace `@Regulated`, viz 12.3) nebyly využity syntaktické konstrukce implementačního jazyka nepopsatelné dodaným jazykovým rozhraním (nebo jeho uvažovaným rozšířením), lze dále uvažovat o překladu samotného kódu generátoru implementující tuto strategii do jazyka popisujícího syntaktické konstrukce v navrženém jazykovém rozhraní. Doplněním jazykového rozhraní o užití abstraktních datových typů a operací nad nimi (množina, seznam, tabulka, průchod, indexace) a implementací překladače všech syntaktických konstrukcí užitých v rámci zdrojových kódů lze hovořit o nástroji, který dokáže své zdrojové kódy přeložit do jiného programovacího jazyka. Tento princip přináší řadu výhod, především pak jednoduchou portabilitu mezi různými architekturami operačních systémů, programovacích jazyků atd.

Literatura

[1] Češka M., Smrčka A., Vojnar T.: *Teoretická informatika TIN - studijní opora*

[2] Chomsky N., "Three models for the description of language," in IRE *Transactions on Information Theory*, vol. 2, no. 3, pp. 113-124, September 1956. doi: 10.1109/TIT.1956.1056813

[3] Drozdek, Adam. Data structures and algorithms in C++. Fourth edition. Boston, MA: Cengage Learning, [2013]. ISBN 978-1133608424

[4] Hopcroft J.E., Ullman J.D., *Formal Languages and Their Relation to Automata*, Addison-Wesley, (1969)

[5] Kolář, D.; Meduna, A.: *Regulated Pushdown Automata*. Acta Cybernetica, ročník 2000, č. 4, 2000: s. 653–664, ISSN 0324-721X.URL http://www.fit.vutbr.cz/research/view_pub.php?id=6020

[6] Kernighan, Brian W a Dennis M. RITCHIE. Programovací jazyk C. Brno: Computer Press, 2006, 286 s. ISBN 80-251-0897-X.

[7] Kozen D. C., *Automata and Computability*, Springer, New York (1997)

[8] Meduna, A.: *Automata and Languages: Theory and Applications*. Springer, 2000, ISBN 1-85233-074-0

[9] Meduna, A.: Deep pushdown automata. Acta Informatica 42, 541–552 (2006)

[10] Meduna, A.: *Elements of Compiler Design*. Auerbach Publications, 2008, ISBN 978-1-4200-6323-3

[11] Meduna, A., Lukáš R.: *Formální jazyky a překladače IFJ - studijní opora*

[12] Meduna, A.; Zemek, P.: *Regulated Grammars and Their Transformations*. Brno University of Technology, 2010, ISBN 978-80-214-4203-0

[13] Meduna, A.; Zemek, P.: *Regulated Grammars and Automata. Springer*, 2014, ISBN 978-1-4939-0368-9

[14] Rozenberg, G.; Salomaa, A.: *Handbook of Formal Languages*, vol. 1 through 3. Springer, 1997

Přílohy

Seznam příloh

Příloha A

Obsah CD

V kořenovém adresáři přiloženého CD se nachází tyto soubory a složky:

- **technicka_zprava.docx** – Zdrojový soubor této práce
- **technicka_zprava.pdf** – Tento soubor
- **bin/** – Adresář se spustitelnými soubory
- **src/** – Adresář obsahující implementaci
  - **build.xml** – Definiční soubor Apache ANT pro přeložení zdrojových textů
  - **grammar.cfg5** – Definiční soubor definičního metajazyka
  - **gex/** – Adresář se zdrojovými texty parseru
  - **gen/** – Adresář se zdrojovými texty generátoru
  - **gedit/** – Adresář se zdrojovými texty editoru
  - **lib/** – Adresář s využitými knihovnami
  - **test/** – Adresář se sadou testů

Příloha B

Přeložení a spuštění navržených nástrojů

Na přiloženém CD jsou ve složce `src` uloženy 3 adresáře se zdrojovými texty všech implementovaných nástrojů. Pro jejich přeložení je využit nástroj *Apache ant*[1]. Pro přeložení všech projektů stačí provést příkaz `ant` v adresáři `src` kořenového adresáře přiloženého CD. Vznikne nová složka `build` obsahující soubory `gex.jar`, `gen.jar` a `gedit.jar` odpovídající po řadě modulům parseru, generátoru a editoru. Pro vygenerování programové dokumentace stačí, ve stejném adresáři, provést příkaz `ant doc`. Programová dokumentace se po provedení zmíněného příkazu nachází v adresáři `doc`. Spuštění přiložené sady testů se provede příkazem `ant test`.

13.1 Parser

Modul parseru (soubor `gex.jar`) slouží výhradně pro ověření příslušnosti řetězce do jazyka daného gramatikou. Spuštění vyžaduje jediný parametr, a to název souboru vstupní gramatiky v binárním formátu (anotace `@Binary` a překlad pomocí generátoru). Vstupní řetězec se předává přes standardní vstup (`stdin`). Je-li vstupní řetězec obsažen v jazyce vstupní gramatiky, je návratová hodnota 0. Je-li vstupní gramatika neplatná, je návratová hodnota 1. Nepatří-li vstupní řetězec do jazyka vstupní gramatiky, je návratová hodnota 2. Jakákoliv jiná návratová hodnota značí selhání nástroje (chybějící název soubor v argumentech programu, nedostatek paměti, chyba načtení souboru a jiné).

Jsou-li součástí vstupní gramatiky uživatelské objekty, jsou operace nad těmito objekty ignorovány. Při vytváření nových analyzátorů založených na navrženém systému je nutno přidat k vygenerovaným souborům referenci na zdrojové kódy obsažené v souboru `gex.jar`.

13.2 Generátor

Modul generátoru (soubor `gen.jar`) slouží výhradně pro generování uživatelského výstupu použitím 2 strategií, a sice binárního výstupu gramatiky anotací `@Binary`, a kódem *GGEN Parseru* v jazyce Java anotací `@Regulated`. Spuštění vyžaduje jediný povinný a jeden volitelný argument. První a povinný argument je název souboru obsahující vstupní gramatiku zapsanou v definičním metajazyce. Druhý, volitelný argument, slouží pro vynucení strategie výstupu. Platné hodnoty tohoto argumentu jsou `Binary` a `Regulated` (nezáleží na velikosti písmen). Je-li vygenerován výstup, je návratová hodnota 0 a tento výstup vypsán na standardní výstup. V případě

[1] https://ant.apache.org/

neplatné gramatiky je návratová hodnota 1. Jakákoliv jiná návratová hodnota značí selhání nástroje (nedostatek paměti, chyba načtení souboru a jiné).

13.3 Editor

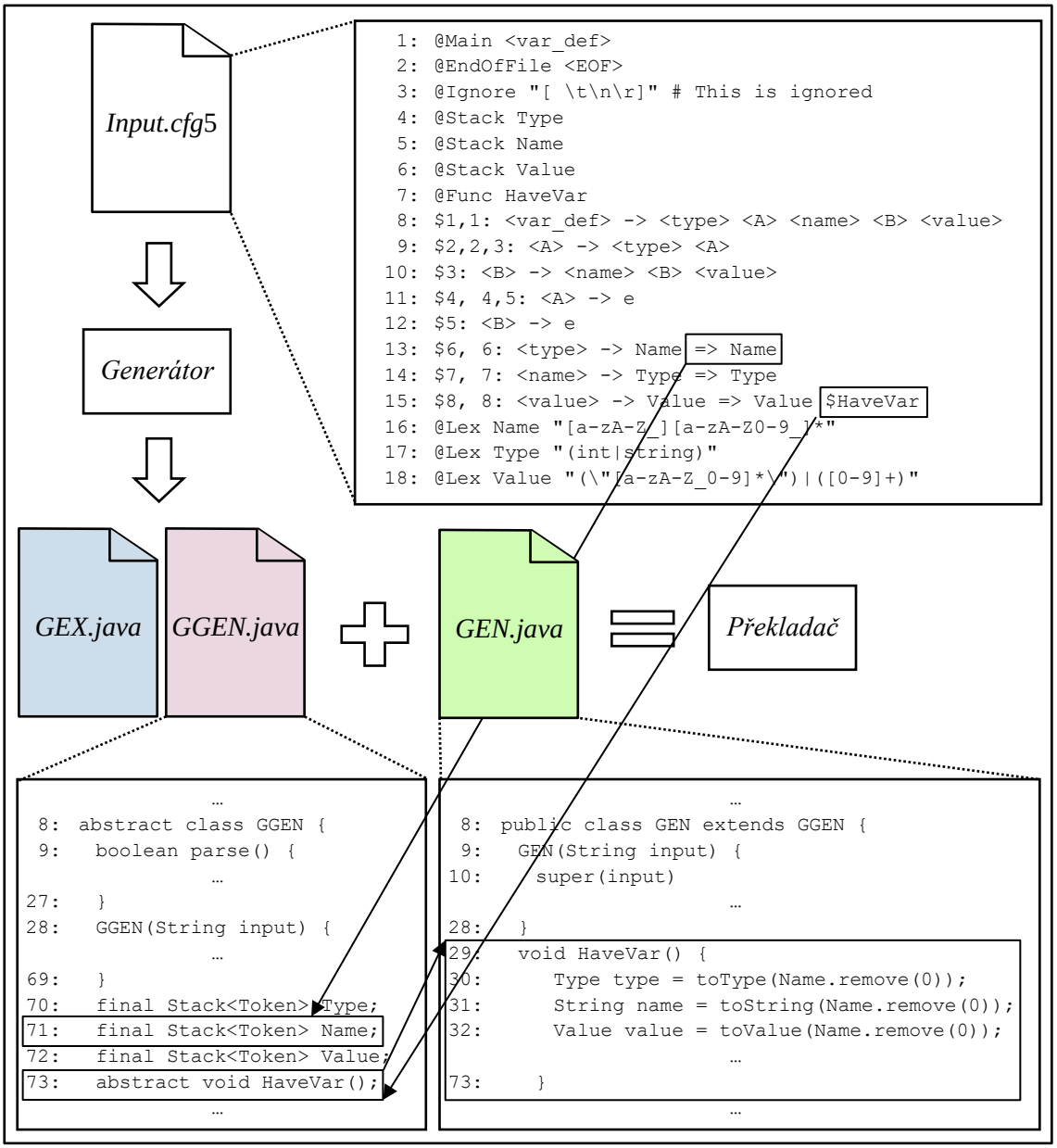
Modul editoru (soubor `gedit.jar`) obsahuje mimo jiné kódy jak parseru, tak generátoru. Jeho spuštění může mít až 2 volitelné parametry, a to název souboru analyzované gramatiky a název souboru analyzovaného řetězce. V případě předání pouze jednoho parametru je výchozí analyzovaný řetězec zvolen nástrojem automaticky. Při nezadání žádného parametru je nástrojem zvolena i výchozí gramatika. Pokud se některý ze vstupních souborů nepodaří načíst, předpokládá se, že jsou prázdné. Vygenerovaný soubor v sobě zahrnuje knihovnu MigLayout[1], která je využita pro elegantní rozmístění prvků uživatelského rozhraní s adaptivní změnou jejich velikost vzhledem k velikost okna aplikace.

[1] http://www.miglayout.com/

Příloha C

Výstavba nového překladače

Typickým využitím projektu je výstavba nového překladače. Tento proces by se dal shrnout příkladem na obrázku C.1, na kterém jako vstup figuruje soubor *Input.cfg*5, obsahující maticovou



Obrázek C.1: Způsob vytvoření nového překladače

gramatiku z příkladu 11.1 a soubor *GEN.java* s implementací uživatelských funkcí. Obrázek byl s úpravou převzat z autorova posteru prezentovaného na konferenci Excel@FIT 2019. Obsah souboru *Input.cfg*5 je generátorem přeložen na soubory *GEX.java* a *GGEN.java*, které se dále využijí v obsahu souboru *GEN.java*. Voláním funkce *parse* třídy *GEX* je provedena požadovaná syntaktická analýza, v rámci které jsou prováděny zapsané uživatelské operace. Po návratu z volání této funkce by měla, v závislosti na vůli uživatele, existovat vnitřní reprezentace vstupního řetězce dostupná z bázové třídy. Samotný překlad z této reprezentace do cílového jazyka je v celém rozsahu v režii uživatele (tvůrce překladače).

Při analýze existujícího překladače (modul *generator*, viz 12.3) je možno pozorovat existenci dvojí sady datových tříd (například *Token*, *Terminal*, atd.), lišící se především balíčkem, do kterého patří. V balíčku `cz.vutbr.fit.xznebe00.gex` jsou obsaženy datové třídy popisující vstupní struktury obsažené ve vstupním řetězci, který se překládá. Tyto struktury jsou popsány gramatikou, pro kterou byl daný generátor vygenerován. V balíčku `cz.vutbr.fit.xznebe00.gen` se potom nachází datové třídy odpovídající strukturám, které jsou popsány strukturami obsaženými ve zmíněném vstupním řetězci. Tyto třídy, ač sdílejí některé sémantické rysy, jsou velmi rozdílné ve způsobu přístupu a zpracování obsažených dat (první z uvedených mají obsah pevně daný, neboť byl předzpracován, druhé obsah získávají čtením vstupního řetězce).

Příloha D

Definice definičního metajazyka

Obrázky D.1 a D.2 obsahují popis definičního metajazyka v definičním metajazyce (viz 11.1).

```
 1: @main <program>
 2: @EndOfFile EOF
 3:
 4: <program> -> EOL* ( <decl> EOL+ )+ EOF
 5:
 6: <decl> -> Name Terminal Terminal
 7: <decl> -> Stack Terminal Terminal*
 8: <decl> -> Func Terminal Terminal*
 9: <decl> -> Main Neterminal
10: <decl> -> Package Terminal+
11: <decl> -> OutputFile Terminal
12: <decl> -> ResultClassName Terminal
13: <decl> -> NoAutoBuild
14: <decl> -> Lex Terminal String
15: <decl> -> Unaddressed
16: <decl> -> Binary
17: <decl> -> Regulated
18: <decl> -> Ignore String
19: <decl> -> Include String
20: <decl> -> Include_once String
21: <decl> -> <Regulation>? Neterminal Arrow <decls>
22: <decl> -> EndOfFileAnnotation Terminal
23:
24: <decls> -> <decl_first> <decls_more>
25: <decls_more> -> Or <decl_first> <decls_more>
26: <decls_more> -> <decl_first> <decls_more>
27: <decls_more> -> e
28:
29: <decl_first> -> Terminal <decl_repeat>? <decl_push>* <decl_call>*
30: <decl_first> -> Neterminal <decl_repeat>? <decl_call>*
31: <decl_first> -> LeftBracket <decls> RightBracket <decl_repeat>? <decl_call>*
32:
33: <decl_repeat> -> Plus | Star | QuestionMark
34:
35: <decl_push> -> PushArrow Terminal
36: <decl_call> -> FuncCall
37:
38: <Regulation> -> FuncCall (Comma <Regulation_spec>)? Colon
39: #RCG
40: <Regulation_spec> -> LeftBracket (Neterminal (Comma Neterminal)*)? RightBracket
    (Or LeftBracket Neterminal (Comma Neterminal)* RightBracket)?
41: #MG
42: <Regulation_spec> -> Terminal Plus? (Comma Terminal Plus?)*
43: #PG
44: <Regulation_spec> -> FuncCall (Comma FuncCall)* (Or FuncCall (Comma FuncCall)*)?
```

Obrázek D.1: Definice pravidel definičního metajazyka definičním metajazykem

```
 1:  @ignore "(#|\/\/)[^\\n]*"
 2:  @ignore "[ \t\x0b\r\f]+"
 3:
 4:  @lex EOL "\n"
 5:  @lex Name "@[Nn]ame"
 6:  @lex Stack "@[Ss]tack"
 7:  @lex Ignore "@[Ii]gnore"
 9:  @lex Func "@[Ff]unc"
10:  @lex Main "@[Mm]ain"
11:  @lex Unaddressed "@[Aa]ll[Ll]exems"
12:  @lex Package "@[Pp]ackage"
13:  @lex OutputFile "@[Oo]utputFile"
14:  @lex ResultClassName "@[Rr]esultClassName"
15:  @lex NoAutoBuild "@[Nn]oAutoBuild"
16:  @lex Lex "@[Ll]ex"
17:  @lex Iterative "@[Ii]terative"
18:  @lex Recursive "@[Rr]ecursive"
19:  @lex Regulated "@[Rr]egulated"
20:  @lex Binary "@[Bb]inary"
21:  @lex Include "@[Ii]nclude"
22:  @lex Include_once "@[Ii]nclude_once"
23:  @lex Arrow "->"
24:  @lex RightBracket "\)"
25:  @lex LeftBracket "\("
26:  @lex Plus "\+"
27:  @lex Minus "-"
28:  @lex PushArrow "=>"
29:  @lex Or "\|"
30:  @lex QuestionMark "\?"
31:  @lex Comma ","
32:  @lex Star "\*"
33:  @lex Colon "\:"
34:  @lex FuncCall "\$([0-9a-zA-Z_])+"
35:  @lex Neterminal "<([0-9a-zA-Z_\\.])+>"
36:  @lex String "\"(\\.|[^\"\\])*\""
37:  @lex Terminal "(([0-9a-zA-Z_\.])+|\\\S|\S)"
38:  @lex EndOfFileAnnotation "@[Ee]nd[Oo]f[Ff]ile"
```

Obrázek D.2: Definice lexémů definičního metajazyka definičním metajazykem