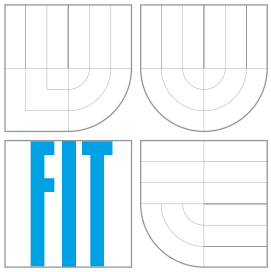
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PARALELNÍ SYNTAKTICKÁ ANALÝZA

PARALLEL SYNTAX ANALYSIS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ OTÁHAL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARTIN ČERMÁK

BRNO 2012

Abstrakt

Diplomová práce se zabývá moderními metodami pro popis jazyků. Představuje několik řízených gramatik, přičemž podrobněji se věnuje stromem řízené gramatice. Je založena na relativně nové technice syntaktické analýzy, která využívá právě stromem řízené gramatiky. V textu je blíže popsán postup této analýzy a následně návrh, jak jí zpracovat paralelně. Daný návrh se nám podařilo implementovat a syntaktickou analýzu tím posílit z hlediska rychlosti, čímž jsme dosáhli hlavního cíle této práce.

Abstract

This thesis focuses on modern methods of language description. It introduces several controlled grammars, describing in detail the tree controlled grammar. The thesis is based on relatively new technique of syntax analysis using tree controlled grammars. The process of this analysis is described in detail, followed by a design of parallel-processing of this analysis. We managed to succesfully implement this design, speed up the syntax analysis and therefore achieve the main goal of the thesis.

Klíčová slova

syntaktická analýza, paralelní programování, bezkontextové gramatiky, stromem řízené gramatiky, řízené gramatiky, řízené přepisování

Keywords

parser, parallel programming, context-free grammar, tree-controlled grammar, controlled grammar, controlled rewriting

Citace

Jiří Otáhal: Paralelní syntaktická analýza, diplomová práce, Brno, FIT VUT v Brně, 2012

Paralelní syntaktická analýza

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně pod vedením pana Ing. Martina Čermáka.

. . . . . .
Jiří Otáhal
22. května 2012

Poděkování

Rád bych poděkoval panu Ing. Martinu Čermákovi za odborné připomínky vedoucí k celkovému zkvalitnění tohoto text.

Obsah

Kapitola 1

Úvod

V dnešní době máme k dispozici obrovskou výpočetní sílu, kterou samozřejmě chceme naplno využít. Zřejmě již každý počítač obsahuje dva a více procesorů. Paralelní programování je možností, jak těchto procesorů využít. Otázkou však zůstává, jak využít paralelního programování tak, abychom zrychlili výpočet tzv. syntaktické analýzy. Dnes je již samozřejmostí, že překladače díky modularitě moderních programovacích jazyků pracují paralelně a tím dosahují lepších výsledků překladu z hlediska rychlosti, což je jedním z cílů paralelních výpočtů. Je velmi snadné zaměstnat více procesorů tak, že každý jeden procesor zpracuje právě jeden soubor. Tedy otázku, jak vylepšit syntaktickou analýzu z hlediska rychlosti, za nás vyřešila právě modularita moderních programovacích jazyků. Existuje však mnoho typů tzv. řízených gramatik, které zlepšují překlad z hlediska síly. V této práci se pokusíme navrhnout paralelní přístup, který provede syntaktickou analýzu využívající některý z typů řízených gramatik. Jelikož je práce založena pouze na několik měsíců starém článku [4], existuje zde velký prostor pro vytváření nových, či kombinování stávajících přístupů k syntaktické analýze. Ve výsledku tak získáme rychlejší a silnější syntaktickou analýzu.

Zpráva je rozčleněna do několika logických celků. Kapitola 2 je věnována definování základních pojmů teorie překladačů. Mimo jiného zde najdeme i klasifikaci základních gramatik, což nám pomůže lépe se zorientovat v moderních přístupech k popisu jazyků. Samotné gramatiky však pouze generují příslušné jazyky. K syntaktické analýze potřebujeme na základě těchto gramatik vytvořit různé automaty, které budou přijímat generované jazyky. Proto si zde definujeme jak samotné automaty, tak algoritmy, umožňující automatický převod z gramatik na tyto automaty. Nakonec si uvedeme jednotlivé fáze překladu, jehož součástí je i syntaktická analýza.

Následující kapitola (3) je věnována moderním metodám popisu jazyků. Je zde velmi obecně popsáno několik typů tzv. řízených gramatik, které omezením pravidel bezkontextových gramatik dokáží zvýšit jejich generativní sílu. Důležitou část tohoto dokumentu představuje tzv. stromem řízená gramatika, kterou se v dané kapitole zabýváme mnohem podrobněji. Vysvětlíme si zde na několika příkladech, jak tato metoda funguje.

Výsledkem této práce má být návrh metody pro paralelní syntaktickou analýzu. Dostáváme se ke kapitole 4, která se právě tímto návrhem zabývá. Představíme si zde dvě metody, kde jedna využívá přístupu shora dolů a druhá pak zdola nahoru. U obou si podrobně popíšeme postup, jakým syntaktickou analýzu provést a poté navržený postup předvedeme na konkrétním příkladě. Dále si předvedeme možnost paralelního zpracování navrženého algoritmu.

Implementaci navržených metod popisuje kapitola 5. Řekneme si, jaké prostředky jsme pro implementaci zvolili a jak výslednou aplikaci použít. Neberme však tento popis jako manuál k přiloženému programu (ten je součástí CD). Spíše budeme diskutovat jeho možnosti. Dále zde navrhneme postup testování navržených metod. Několik testů také provedeme a vyhodnotíme výsledky.

Kapitola 2

Teorie překladačů

Abychom mohli dále v tomto textu zkoumat moderní syntaktické analýzy a obecně možnosti překladu, musíme si nejdříve zavést základní definice prostředků. Základní pojmy a definice zmíněné v této kapitole jsou převzaty z [6]. Dále budeme definovat gramatiky, které využíváme jak pro generování jazyků, tak pro jejich kontrolu. Lépe řečeno, na základě gramatik dokážeme specifikovat nástroje, které jazyk generovaný danou gramatikou dokáží kontrolovat. Abychom uživateli co nejvíce zjednodušili použití aplikace přiložené k tomuto textu (viz. kapitola 5), budeme automaticky pro zadanou gramatiku specifikovat automat, který dokáže přijímat právě touto gramatikou generovaný jazyk. Proto si zde přiblížíme metody, které tento převod popisují. Součástí této kapitoly je také popis jednotlivých fází překladu a samozřejmě detailněji popsaný přístup k syntaktické analýze (2.4.2). Konkrétněji je to přístup shora dolů a přístup zdola nahoru.

2.1 Základní pojmy

Zde se seznámíme se základními prostředky používanými ve spojení s překladači a obzvlášť pak se syntaktickou analýzou.

2.1.1 Abeceda a symboly

Příkladem může být binární abeceda, kterou představuje množina $\{0, 1\}$. Symboly zde jsou 0 a 1. Formálněji bychom mohli abecedu definovat následovně.

Definice 2.1.1. Abeceda je konečná, neprázdná množina elementů, které nazýváme symboly.

2.1.2 Řetězec

V literatuře se někdy můžeme setkat s označením řetězce jako řetěz nebo také slovo. Než si uvedeme operace, které s řetězci můžeme provádět, musíme si řetězec nadefinovat. Prázdný řetězec budeme označovat jako ε.

Definice 2.1.2. Nechť Σ je abeceda.

- ε je řetězec nad abecedou Σ
- pokud x je řetězec nad Σ a $a \in \Sigma$, potom xa je řetězec nad abecedou Σ

Nyní si formálně zavedeme pojmy, které nám dovolí s řetězci pracovat.

Definice 2.1.3. Nechť x je řetězec nad abecedou Σ. *Délka řetězce* x, kterou značíme $|x|$, je definována takto:

- pokud $x = \epsilon$, pak $|x| = 0$
- pokud $x = a_1 \dots a_n$, pak $|x| = n$ pro $n \geq 1$ a $a_i \in \Sigma$ pro všecha $i = 1, \dots, n$

Definice 2.1.4. Nechť x a y jsou dva řetězce nad abecedou Σ. *Konkatenace* x a y je řetězec xy.

Definice 2.1.5. Nechť x je řetězec nad abecedou Σ. Pro $i \geq 0$, i-tá *mocnina řetězce* x, x^i, je definována: $x^0 = \varepsilon$ a pro $i \geq 1 : x^i = xx^{i-1}$

Definice 2.1.6. Nechť x je řetězec nad abecedou Σ. *Reverzace* řetězce x, kterou značíme jako $reversal(x)$, je definována:

- pokud $x = \epsilon$ pak $reversal(\epsilon) = \epsilon$
- pokud $x = a_1 \dots a_n$, pak $reversal(a_1 \dots a_n) = a_n \dots a_1$ pro $n \geq 1$ a $a_i \in \Sigma$ pro všechna $i = 1, \dots, n$

Definice 2.1.7. Nechť x a y jsou dva řetězce nad abecedou Σ. Potom x je *prefixem* y, pokud existuje řetězec z nad abecedou Σ, přičemž platí $xz = y$.

Definice 2.1.8. Nechť x a y jsou dva řetězce nad abecedou Σ. Potom x je *sufixem* y, pokud existuje řetězec z nad abecedou Σ, přičemž platí $zx = y$.

Definice 2.1.9. Nechť x a y jsou dva řetězce nad abecedou Σ. Potom x je *podřetězcem* y, pokud existuje řetězec z, z' nad abecedou Σ, přičemž platí $zxz' = y$.

2.1.3 Jazyk

Jazyk může být konečný nebo nekonečný. Jazyk L je konečný, pokud L obsahuje konečný počet řetězců, jinak je nekonečný. Nad jazyky můžeme provádět hned několik operací. Než si je popíšeme, uvedu definici samotného jazyka.

Definice 2.1.10. Nechť Σ^* značí množinu všech řetězců nad Σ. Každá podmnožina $L \subseteq \Sigma^*$ je jazyk nad Σ.

Definice 2.1.11. Nechť L_1 a L_2 jsou dva jazyky nad Σ. *Sjednocení* jazyků L_1 a L_2, $L_1 \cup L_2$ je definováno: $L_1 \cup L_2 = \{x : x \in L_1 \text{ nebo } x \in L_2\}$

V této práci budeme často využívat tzv. reverzaci jazyka. Definujme si tedy tento pojem. Neformálně můžeme říct, že reverzace je otočení všech řetězců patřící do daného jazyka.

Definice 2.1.12. Nechť L je jazyk nad Σ. *Reverzace* jazyka L, které značíme jako $reversal(L)$, je definována následovně.

$$reversal(L) = \{reversal(x) \mid x \in L\}$$

2.2 Reprezentace jazyků

Konečné jazyky můžeme jednoduše specifikovat výčtem jejich slov. Nekonečné jazyky (např. programovací jazyky) takto specifikovat nemůžeme. Pro jejich popis si zavedeme následující gramatiky a automaty. Tyto prostředky totiž představují konečnou reprezentaci nekonečných, ale i konečných jazyků.

Každá gramatika obsahuje konečnou množinu pravidel, pomocí kterých generujeme daný jazyk. Stejně tak i automaty definují jazyk pomocí konečných prostředků, na jejichž základě algoritmus rozhodne, zda daný řetězec patří do jazyka či nikoliv. Následuje Chomského klasifikace gramatik, jejich definice a další pojmy, které budeme potřebovat dále v tomto textu.

2.2.1 Neomezená gramatika

Základní gramatika, ze které vychází všechny dále uvedené gramatiky. To znamená, že ostatní gramatiky jsou pouze podmnožinou této.

Definice 2.2.1. *Gramatika* je čtveřice $G = (N, T, P, S)$, kde

- N je abeceda neterminálů
- T je abeceda terminálů, přičemž $N \cap T = \varnothing$
- P je konečná množina pravidel a $P \subseteq (N \cup T)^* N (N \cup T) \times (N \cup T)^*$
- $S \in N$ je počáteční neterminál

Definice 2.2.2. Nechť $G = (N, T, P, S)$ je gramatika a u, v jsou řetězce z $(N \cup T)^*$. Mezi řetězci u a v platí relace $u \Rightarrow_G v$, nazývána *přímá derivace*, jestliže můžeme řetězce u, v vyjádřit ve tvaru

$$u = \gamma\alpha\delta$$

$$v = \gamma\beta\delta$$

kde $\gamma, \delta \in (N \cup T)^*$ a $\alpha \to \beta$ je nějaké přepisovací pravidlo z P.

Definice 2.2.3. Nechť $G = (N, T, P, S)$ je gramatika a u, v jsou řetězce z $(N \cup T)^*$. Mezi řetězci u a v platí relace $u \Rightarrow^+ v$, nazývána *derivace*, jestliže existuje posloupnost přímých derivací $w_{i-1} \Rightarrow w_i$, kde $w_i \in (N \cup T)^*$ a $1 \leq i \leq n$, pro nějaké $n \geq 1$ taková, že platí

$$u = w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_{n-1} \Rightarrow w_n = v$$

Tuto posloupnost nazýváme derivací délky n a značíme jí $\Rightarrow^n$.

Definice 2.2.4. Nechť $G = (N, T, P, S)$ je gramatika a u, v jsou řetězce z $(N \cup T)^*$. Jestliže platí relace $u \Rightarrow^+ v$ nebo identita $u = v$, pak píšeme $u \Rightarrow^* v$.

2.2.2 Kontextová gramatika

Definice 2.2.5. *Kontextová gramatika* je čtveřice $G = (N, T, P, S)$, kde

- N je abeceda neterminálů
- T je abeceda terminálů, přičemž $N \cap T = \varnothing$
- P je konečná množina pravidel a $\alpha A\, \beta \to \alpha\gamma\beta$, kde $A \in N, \alpha, \beta \in (N \cup T)^*, \gamma \in (N \cup T)^+$
- $S \in N$ je počáteční neterminál

2.2.3 Bezkontextová gramatika

Bezkontextová gramatika (2.2.6) nejčastěji popisuje syntaxi programovacích jazyků. Pro názorné zobrazení derivace (struktury věty) v bezkontextové gramatice slouží *derivační strom* (2.2.8). Nejlepší ukázkou zřejmě bude příklad takového stromu. Nejdříve si však uvedeme formální definice bezkontextové gramatiky a derivačního stromu.

Definice 2.2.6. *Bezkontextová* gramatika je čtveřice $G = (N, T, P, S)$, kde

- N je abeceda neterminálů
- T je abeceda terminálů, přičemž $N \cap T = \varnothing$
- P je konečná množina pravidel ve tvaru: $A \to x$, kde $A \in N, x \in (N \cup T)^*$
- $S \in N$ je počáteční neterminál

Definice 2.2.7. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. Jazyk generovaný gramatikou G, který značíme $L(G)$, je definován jako

$$L(G) = \{w \mid w \in T^*, S \Rightarrow^* w\}$$

Definice 2.2.8. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. Strom nazýváme *derivační strom* v G, jestliže.

- Každý uzel je ohodnocen symbolem z $N \cup T$.
- Kořen je ohodnocen symbolem S.
- Jestliže uzel má nejméně jednoho následovníka, pak je ohodnocen symbolem z N.
- Jestliže $b_1, b_2, \ldots, b_k$ jsou přímí následovníci uzlu a, jenž je ohodnocen symbolem A, v pořadí zleva doprava s ohodnocením $B_1, B_2, \ldots, B_k$, pak $A \to B_1 B_2 \ldots B_k \in P$.

Derivační strom

Derivační strom je graf, který reprezentuje syntaktickou strukturu řetězce a slouží jak pro názorné zobrazení takové struktury, tak i přímo pro analýzu jednotlivých derivačních kroků a tedy i celého řetězce.

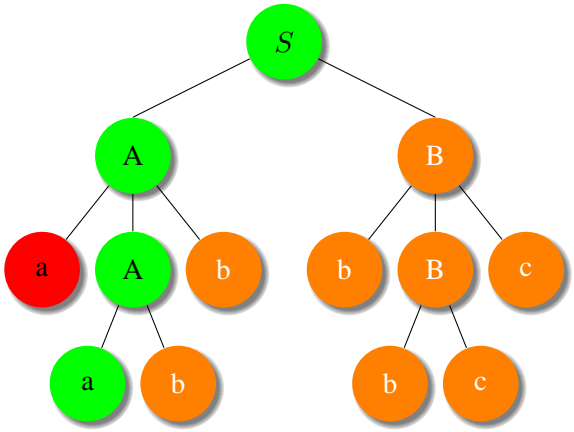
Jak je uvedeno v definici, kořenovým uzlem je vždy startující symbol S. Takovýto startující uzel je spojen s dalšími uzly, kterými mohou být buďto uzly označeny neterminálními nebo terminálními symboly. Cílem derivačního stromu je dostat se od startujícího uzlu S s využitím pravidel z gramatiky G do uzlů označenými terminálními symboly (ty nazýváme listy stromu). Když potom spojíme označení všech listů zleva doprava, tak dostáváme řetězec, který derivační strom reprezentuje.

Jak uvidíme v kapitole 3.2, existují gramatiky, které přímo využívají těchto stromů pro kontrolu syntaktické správnosti určitého řetězce. My se zaměříme na typ, kdy se v derivačním stromě hledají cesty, které spadají do řídícího jazyka dané gramatiky. Co je to řídící jazyk si vysvětlíme v dalších kapitolách, nyní si však zaveďme pojem cesta v derivačním stromě.

Definice 2.2.9. Mějme bezkontextovou gramatiku $G = (N, T, P, S)$ a $x \in T^*$. Dále mějme ${}_G\Delta(x)$, značící množinu všech derivačních stromů, které reprezentují řetězec x v G a $t \in {}_G\Delta(x)$. Cestou (path) v t označíme nějakou sekvenci uzlů, kde prvním uzlem je kořenový (root) uzel stromu t a posledním uzlem je list v daném stromu t, přičemž musí platit, že každé dva po sobě jdoucí uzly jsou spojeny hranou. Řekněme, že s je nějaká sekvence uzlů z t. Potom $word(s)$ značí řetězec získaný spojením všech symbolů uzlů v s.

Příklad 2.2.10. Mějme gramatiku $G = (\{S, A, B\}, \{a, b, b\}, P, S)$, kde P obsahuje pravidla $S \to AB$, $A \to aAb \mid ab$, $B \to bBc \mid bc$. Pak v této gramatice můžeme generovat řetězec $aabbbbcc$ například takto:

$$S \Rightarrow AB \Rightarrow aAbB \Rightarrow aAbbBc \Rightarrow aAbbbcc \Rightarrow aabbbbcc$$



Obrázek 2.1: Derivační strom pro příklad 2.2.10

Nyní si popišme, jak budeme v tomto textu chápat grafickou podobu jednotlivých uzlů stromu.

- **Cesta** – V derivačním stromě označené jako zelené uzly. V tomto konkrétním případě vede cesta od kořenového uzlu S do listu a. V některých příkladech může být označena i nekompletní cesta. Chápejme tedy uzel označený zelenou barvou, jako uzel patřící do potenciální cesty.
- **Nepřijatá cesta** – Takový uzel označme červenou barvou. Znamená to tedy, že daný uzel již prošel analýzou hledající potenciální cesty a byl odmítnutý.
- **Uzel** – Pokud uzel nepatří do žádné z výše uvedených kategorií, pak je označený oranžovou barvou.
- **List** – List neodlišujeme od ostatních uzlů jinou barvou. Chápejme jej však jako poslední, do kterého pouze vstupuje nějaká hrana a žádná z něj již nevystupuje.

Definice 2.2.11. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. Pokud existuje řetězec $x \in L(G)$ s více jak jedním derivačním stromem, potom G je *nejednoznačná*. Jinak je G jednoznačná.

Definice 2.2.12. Bezkontextový jazyk L je *vnitřně nejezdnoznačný*, pokud L nelze generovat žádnou jednoznačnou bezkontextovou gramatikou.

Zásobníkový automat

Definice 2.2.13. *Zásobníkový automat* (ZA) je sedmice $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, kde

- Q je konečná množina stavových symbolů reprezentujících vnitřní řídící jednotky
- Σ je vstupní abeceda
- Γ je konečná abeceda zásobníkových symbolů
- δ je zobrazení z množiny $Q \times (\Sigma \in \{\epsilon\}) \times \Gamma$ do konečné množiny podmnožin $Q \times \Gamma^*$ popisující funkci přechodů
- q_0 je počáteční stav řídící jednotky
- Z_0 je symbol, který je na počátku uložen do zásobníku
- $Z \subseteq Q$ je množina koncových stavů

Definice 2.2.14. *Konfigurace zásobníkového* automatu P je trojice $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$, kde

- q je přítomný stav řídící jednotky
- w je doposud nepřečtená část vstupního řetězce. První symbol řetězce je pod čtecí hlavou. Je-li $w = \epsilon$, pak byly všechny symboly ze vstupní pásky přečteny.
- α je obsah zásobníku. Je-li $\alpha = \epsilon$, pak je zásobník prázdný.

Definice 2.2.15. *Přechod zásobníkového automatu* P budeme reprezentovat binární relací $\vdash_p$ (nebo jednodušeji $\vdash$ bude-li zřejmé, že jde o automat P), která je definována na množině konfigurací zásobníkového automatu P. Relace

$$(q, aw, Z\alpha) \vdash (q', w, \gamma\alpha)$$

platí, jestliže $\delta(q, a, Z)$ obsahuje prvek (q', γ) pro nějaké $q \in Q, a \in (\Sigma \cup \{\epsilon\}), w \in \Sigma^*, Z \in \Gamma$ a $\alpha, \gamma \in \Gamma^*$.

Definice 2.2.16. Nechť $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ je zásobníkový automat. Jazyk přijímaný tímto automatem je definován jako

$$L(M) = \{w \mid w \in \Sigma^*, Z_0 q_0 w \vdash^* f, f \in Q\}$$

.

2.2.4 Lineární gramatika

Tato gramatika přímo nepatří mezi chomského klasifikaci gramatik, avšak dále v textu jí budeme často využívat. Jedná se o gramatiku, která je vlastní podmnožinou bezkontextových gramatik a níže definovaná regulární gramatika je vlastní podmnožinou právě lineární gramatiky.

Definice 2.2.17. *Lineární gramatika* je čtveřice $G = (N, T, P, S)$, kde

- N je abeceda neterminálů
- T je abeceda terminálů, přičemž $N \cap T = \varnothing$

- P je konečná množina pravidel ve tvaru: $A \to \alpha B \beta$ nebo $A \to \alpha$, kde $A, B \in N$ a $\alpha, \beta \in T^*$
- $S \in N$ je počáteční neterminál

Tedy neformálně řečeno - je to gramatika, jejíž každé pravidlo má maximálně jeden neterminál na pravé straně.

2.2.5 Regulární gramatika

Definice 2.2.18. *Regulární gramatika* je čtveřice $G = (N, T, P, S)$, kde

- N je abeceda neterminálů
- T je abeceda terminálů, přičemž $N \cap T = \varnothing$
- P je konečná množina pravidel ve tvaru: $A \to xB$ nebo $A \to x$, kde $A, B \in N, x \in T^*$
- $S \in N$ je počáteční neterminál

2.3 Algoritmy

Popíšeme si algoritmy, které budeme později používat obzvláště v aplikaci přiložené k této práci. Nejdříve si však zaveďme dvě potřebné množiny $First$ a $Follow$.

2.3.1 First množina

$First(x)$ je množina všech terminálů, kterými může začínat větná forma derivovatelná z x.

Definice 2.3.1. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. Pro každé $x \in (N \cup T)^*$ je definována $First(x)$ jako

$$First(x) = \{a \mid a \in T, x \Rightarrow^* ay, y \in (N \cup T)^*\}$$

Pro LL gramatiku musí platit, že pro neterminál A nesmí existovat více jako jedno pravidlo, které začíná stejným terminálem (případně neterminálem, který se dá na takovýto terminál přepsat). Jinými slovy, pokud je na vstupu terminál a, pak musíme být jednoznačně schopni určit, jaké pravidlo použijeme. Zaveďme si LL gramatiku formálně.

Definice 2.3.2. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika bez ε-pravidel. G je LL gramatika, pokud pro každé $a \in T$ a $A \in N$ existuje maximálně jedno pravidlo $A \to X_1 X_2 \dots X_n \in P$ takové, že $a \in First(X_1 X_2 \dots X_n)$.

Výpočet First množiny

Algoritmus pro výpočet $First$ množiny je rekurzivní, jak uvidíme níže. Uvažujme gramatiku bez ε-pravidel.

1. Pro každé $a \in T$ je $First(a) = \{a\}$
2. Pokud $A \to X_1 X_2 \dots X_n \in P$, pak přidej $First(X_1)$ do $First(A)$. Tento bod opakuj tak dlouho, dokud bude možné měnit některou z $First$ množin.

Příklad 2.3.3. Mějme gramatiku $G = (N, T, P, S)$, kde

- $N = \{S, A, B\}$
- $T = \{a, b, c, d, e, k\}$ a množina pravidel P (čísla představují jedinečné označení pravidel):

$$\begin{array}{lrcl} \mathbf{1:} & S & \rightarrow & AA \\ \mathbf{2:} & A & \rightarrow & aAd \\ \mathbf{3:} & A & \rightarrow & bBc \\ \mathbf{4:} & A & \rightarrow & e \\ \mathbf{5:} & B & \rightarrow & bBc \\ \mathbf{6:} & B & \rightarrow & k \end{array}$$

Příklad 2.3.4. Pro výše uvedenou gramatiku z příkladu 2.3.3 jsou množiny $First$ uvedeny v tabulce 2.1.

x	$First(x)$
a	$\{a\}$
b	$\{b\}$
c	$\{c\}$
d	$\{d\}$
e	$\{e\}$
S	$\{a, b, e\}$
A	$\{a, b, e\}$
B	$\{a, b, e\}$

Tabulka 2.1: Množiny $First$

2.3.2 Follow množina

Množina $Follow(A)$ je množina všech terminálů, které se mohou vyskytovat vpravo od A ve větné formě.

Definice 2.3.5. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. Pro všechna $A \in N$ definujeme množinu $Follow(A)$ takto

$$Follow(A) = \{a \mid a \in T, S \Rightarrow^* xAay, x, y \in (N \cup T)^*\} \cup \{\$ \mid S \Rightarrow^* xA, x \in (N \cup T)^*\}$$

Výpočet Follow množiny

I výpočet množiny $Follow$ probíhá rekurzivně. Ukážeme si algoritmus, kterým je možné jí získat a poté ukážeme výsledné množiny pro danou gramatiku.

1. $Follow(S) = \{\$\}$, kde \$ je symbol, který značí konec vstupu.
2. Pokud $A \rightarrow xBy \in P$, potom:

- Pokud $y \neq \varepsilon$, pak přidej všechny symboly z $First(y)$ do $Follow(B)$
- Pokud $Empty(y) = \{\varepsilon\}$, pak přidej všechny symboly z $Follow(A)$ do $Follow(B)$.

Druhý krok opakuj tak dlouho, dokud se mění některá z $Follow$ množin.

Uveďme si význam množiny $Empty(x)$. Jde o množinu, která obsahuje jediný prvek ε, pokud x derivuje ε, jinak je prázdná.

Definice 2.3.6. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika.

- $Empty(x) = \{\varepsilon\}$, pokud $x \Rightarrow^* \varepsilon$, jinak
- $Empty(x) = \emptyset$, kde $x \in (N \cup T)^*$

Příklad 2.3.7. Pro výše uvedenou gramatiku (viz. 2.3.3) jsou množiny uvedeny v tabulce 2.2.

x	$Follow(x)$
S	$\{\$\}$
A	$\{a, b, e, d, \$\}$
B	$\{c\}$

Tabulka 2.2: Množiny $Follow$

2.3.3 Převod gramatiky na zásobníkový automat

Dalším algoritmem je převod gramatiky na zásobníkový automat. Do aplikace přiložené k této práci bude uživatel zadávat gramatiky pro generování nějakého jazyka. K přijmutí tohoto jazyka pak budeme potřebovat právě zásobníkový automat. Převod mezi zadanou gramatikou a zásobníkovým automatem bude probíhat automaticky podle následujícího algoritmu.

Příklad 2.3.8. Mějme bezkontextovou gramatiku $G = (N, T, P, S)$. Gramatiku G převedeme na rozšířený zásobníkový automat $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \emptyset)$ takto:

- $Q = \{q\}$
- $\Sigma = T$
- $\Gamma = N \cup T$
- $q_0 = q$
- $Z_0 = S$
- Konstrukce množiny δ:
  - Pro každé $a \in \Sigma$ bude $\delta(q, a, a) = \delta(q, \varepsilon)$.
  - Pro každé $A \to x \in P$ bude $\delta(q, \varepsilon, A) = \delta(q, x)$.

Příklad 2.3.9. Ukažme si převod dané gramatiky (viz. 2.3.3) na rozšířený zásobníkový automat.

- $Q = \{q\}$

- $\Sigma = \{a, b, c, d, e, k\}$
- $\Gamma = \{S, A, B, a, b, c, d, e, k\}$
- $q_0 = q$
- $Z_0 = S$
- Množina δ:
  - $\delta(q, \varepsilon, S) = \{(q, AA)\}$
  - $\delta(q, \varepsilon, A) = \{(q, aAd), (q, bBc), (q, e)\}$
  - $\delta(q, \varepsilon, B) = \{(q, bBc), (q, k)\}$
  - $\delta(q, a, a) = \{(q, \varepsilon)\}$
  - $\delta(q, b, b) = \{(q, \varepsilon)\}$
  - $\delta(q, c, c) = \{(q, \varepsilon)\}$
  - $\delta(q, d, d) = \{(q, \varepsilon)\}$
  - $\delta(q, e, e) = \{(q, \varepsilon)\}$
  - $\delta(q, k, k) = \{(q, \varepsilon)\}$

2.4 Fáze překladu

Samotná kompilace je rozčleněna do několika fází. Těmito částmi musí zdrojový program zapsaný ve zdrojovém jazyce projít, aby se z něj stal cílový program v cílovém jazyce.

2.4.1 Lexikální analyzátor

První fází překladače je lexikální analyzátor (lexical analysis, scanning). Vstupem lexikální analýzy je soubor se zdrojovým kódem a výstupem jsou tokeny. Funkcí lexikální analýzy je číst znaky ze vstupního souboru a tvořit z nich lexikální symboly. Každý takový symbol představuje logickou posloupnost znaků a nazývá se lexém. Výstupem lexikální analýzy má být však token. Ten je složen z daného lexému a jeho hodnoty. Token je tedy dvojice lexém:hodnota (např. int:20). Pokud daný lexém nemá hodnotu, můžeme jí vynechat (např. rovnítko). Z následující posloupnosti znaků budou vytvořeny tyto tokeny:

$$promenna = 20 * 4.00;$$

Pořadí	Typ	Hodnota
1	identifikátor	promenna
2	rovnítko	
3	int	20
4	operátor	*
5	double	4.00
6	středník	

Tabulka 2.3: Fronta lexikálních symbolů

V tabulce 2.3 můžeme vidět výstup lexikální analýzy daného výrazu. Každý řádek představuje jeden token (pro lepší přehlednost je v tabulce navíc sloupeček pořadí). Syntaktický analyzátor (viz. 2.4.2) by požadoval tokeny postupně v pořadí od 1 (v našem příkladu tedy od identifikátoru).

V praxi se pro implementaci lexikální analýzy využívá tzv. konečný automat. Symboly nepotřebné pro syntaktickou analýzu („bílé znaky“, komentáře, . . .) jsou konečným automatem vynechány a lexikální analyzátor je tedy na svém výstupu vůbec neuvádí.

2.4.2 Syntaktický analyzátor

Výstupem syntaktické analýzy je derivační strom nebo určitá posloupnost akcí, které uchovávají vnitřní reprezentaci struktury zdrojového textu a sémantiky těchto struktur. Pro vytvoření takové struktury (příp. derivačního stromu) syntaktický analyzátor potřebuje tokeny. Ty dostává od lexikálního analyzátoru popsaného v předchozí sekci. Během syntaktické analýzy překladač kontroluje správné pořadí lexémů. Pokud zdrojový text obsahuje chyby, může se využít techniky pro zotavení se z chyb a analýza i po chybě může pokračovat dále. Pokud se této techniky nevyužije, tak analyzátor skončí s kontrolou hned na první chybě. Vhodné jsou i detailnější výpisy o nalezené chybě, protože v praxi nám většinou nestačí pouze informace o správném či špatném zápisu programu – chceme například vědět i pozici chyby v souboru, abychom ji mohli pohodlně opravit.

Pro implementaci této fáze jsou nejčastěji používané principy shora dolů nebo zdola nahoru. Tyto názvy odpovídají směru vytváření derivačního stromu. Vzhledem k tomu, že se tento text zabývá paralelní syntaktickou analýzou, řekněme si více o těchto dvou přístupech. Pro oba přístupy předpokládejme, že máme k dispozici nějaký zásobníkový automat založený na gramatice 2.3.3. Dále si můžeme o syntaktické analýze přečíst v [6] nebo [2].

Přístup shora dolů

Jak již název napovídá, tato metoda konstruuje derivační strom od nejvyšší úrovně po tu nejnižší. Začíná tedy se startovacím symbolem dané gramatiky, jako s kořenem derivačního stromu. Následně podle pravidel gramatiky postupuje směrem dolů a zleva doprava, dokud nenarazí na listový uzel (nejnižší úroveň stromu). Jakmile získá derivační strom, který reprezentuje derivaci $S \Rightarrow^* w$, pro nějaký řetězec z gramatiky, pak končí úspěchem. Syntaktická analýza založená na přístupu shora dolů, využívá tzv. LL tabulku. My si nyní ukážeme, jak takovou tabulku vytvořit.

Se znalostí množiny $First$ (viz. 2.3) je to celkem jednoduché. Mějme záznam v tabulce $\alpha(A, a)$, kde $A \in N$ (v tabulce označuje řádek) a $a \in T$ (v tabulce označuje sloupec). Dále mějme nějakou gramatiku $G = (N, T, P, S)$ a nějaká pravidla z této gramatiky $r_1, r_2, \ldots r_j$.

- Pokud existuje pravidlo $r_i : A \rightarrow X_1 X_2 \ldots X_n \in P$ a pokud $a \in First(X_1)$, pak $\alpha(A, a) = r_i$, kde $1 \leq i \leq j$.
- Pokud takové pravidlo neexistuje, pak je $\alpha(A, a)$ prázdné, což vede k chybě při syntaktické analýze.

Příklad 2.4.1. Vypočítejme nyní LL tabulku pro gramatiku uvedenou v 2.3.3. Výsledek představuje tabulka 2.4.

Jakmile máme LL tabulku a nějaký zásobníkový automat, pak je samotná syntaktická analýza jednoduchá. Jestliže je na vrcholu zásobníku neterminál A, první vstupní symbol je a a existuje pravidlo $A \rightarrow a$ na pozici $[A, a]$ v LL tabulce, pak automat přepíše A na zásobníku obráceným řetězcem x – tzn. $reversal(x)$. Jestliže je na vrcholu zásobníku a a na vstupu terminál a, pak automat odstraní a ze zásobníku a přečte jej ze vstupu.

	a	b	c	d	e	k	**$**
S	1	1			1		
A	2	3			4		
B		5				6	

Tabulka 2.4: LL tabulka

Přístup zdola nahoru

Opakem předchozího způsobu je přístup zdola nahoru. Tato metoda začíná tam, kde předchozí končí, tedy v listových uzlech stromu. Postupuje tedy zleva doprava a pokud syntaktická analýza proběhne v pořádku, tak končí kořenem stromu. Činnost této metody je založena na tzv. LR tabulce, která je složena ze dvou částí:

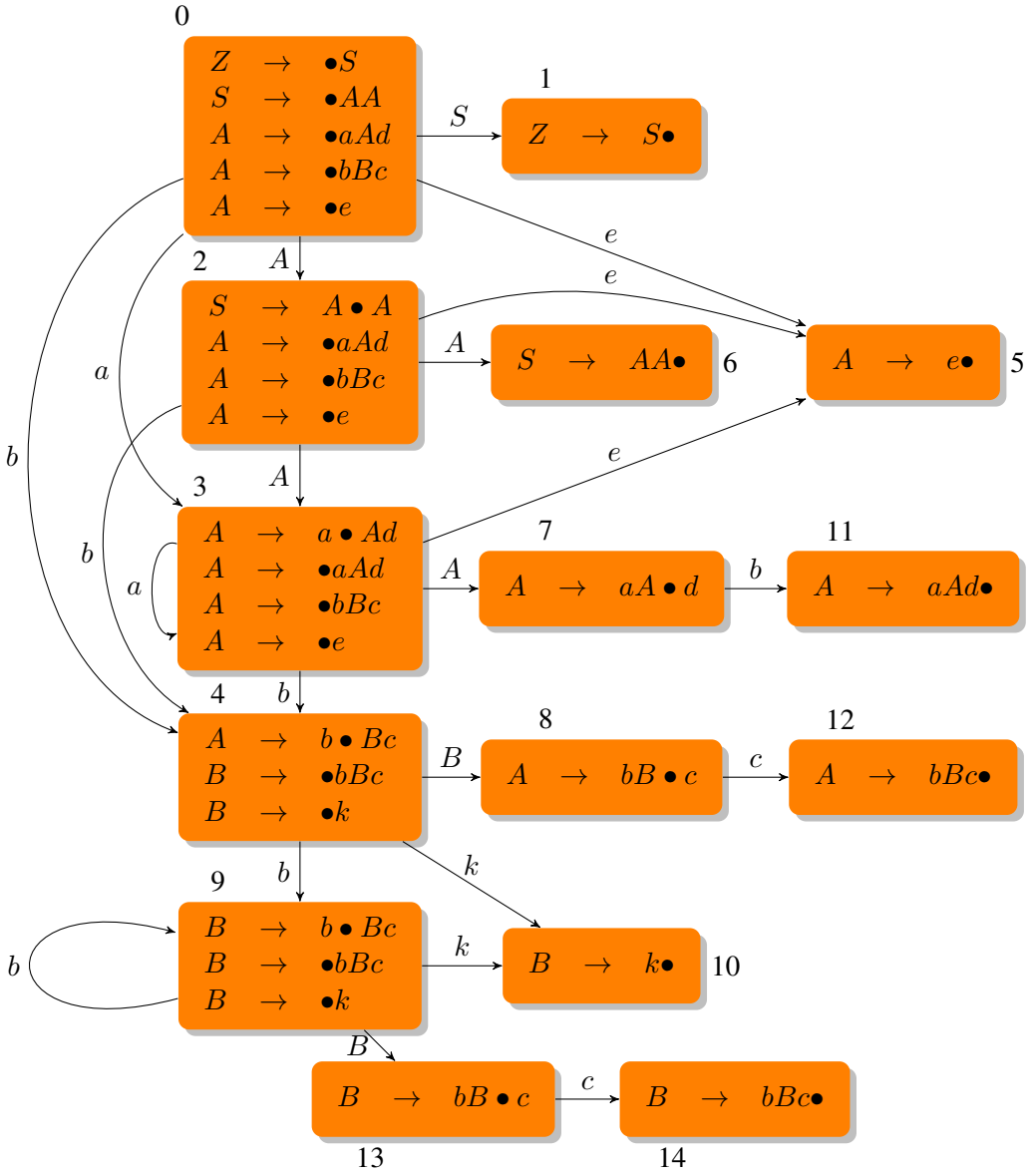
- akční část – označme jako tabulka α,
- přechodová část – označme jako tabulka β.

Uvažujme nyní opět gramatiku uvedenou v příkladě 2.3.3. Ukažme si, jak lze pro tuto gramatiku vytvořit LR tabulku. Uvedeme zde pouze neformální postup, protože konstrukce LR tabulky není hlavním cílem této práce, je však použita v přiložené aplikaci a uživatel musí zadávat gramatiku, ze které lze tuto tabulku vytvořit. Je tedy vhodné alespoň naznačit, jak se LR tabulka vytváří. Formálněji je algoritmus vytvoření LR tabulky popsaný v [2].

Nejdříve si do gramatiky zavedeme pomocné pravidlo $Z \to S$, kde S reprezentuje startující symbol v gramatice. S tímto pravidlem zároveň změníme startující symbol na Z. Nyní budeme konstruovat tzv. LR(0) automat, ze kterého později získáme právě dvě požadované části LR tabulky. Vytvoříme nový stav obsahující námi zavedené pravidlo $Z \to S$. Navíc si však toto pravidlo označíme jako počáteční položku značkou $\bullet$ (tedy $Z \to \bullet S$). Pro lepší přehlednost si stav označme jako S_0. Do tohoto stavu budeme přidávat postupně další pravidla takto.

1. Pokud je $A \to \alpha \bullet B\beta \in S_0$, pak všechny pravidla z dané gramatiky $B \to \gamma$, které označíme takto $B \to \bullet\gamma$, jsou také v S_0. Dodejme, že $\alpha, \beta \in (N \cup T)^*$ a $\gamma \in (N \cup T)^+$. Tento krok opakujeme tak dlouho, dokud lze přidávat nějaká pravidla.
2. Jakmile dokončíme přidávání, tak vytváříme nové stavy pro každé pravidlo z aktuálního stavu následujícím způsobem. Mějme pravidlo $A \to \alpha \bullet B\gamma$.
   - Pokud $B \in (N \cup T)$, pak posuneme značení za B a vytvoříme nový stav S_1 s pravidlem $A \to \alpha B \bullet \gamma$. Pokud ve stavu S_0 existuje více pravidel, které mají značku právě před B, pak i těmto pravidlům posuneme značku a vložíme je do vytvořeného stavu S_1. Nakonec povedeme hranu ze stavu S_0 do S_1 a označíme ji jako B.
   - Pokud $B\gamma = \varepsilon$, pak toto pravidlo pro tuto chvíli ignorujeme. (Značka se nachází na konci pravé strany pravidla.)
   - Pro každý takto vytvořený stav pokračujeme znovu od prvního kroku. Tento postup opakujeme tak dlouho, dokud se některý ze stavů mění.

Po získání automatu na obrázku 2.4.2 musíme ještě spočítat množinu $Follow$. Pro tento příklad je uvedena v tabulce 2.2. Nyní tedy vytvoříme tabulku, kde na řádcích mějme označení jednotlivých stavů konečného automatu a ve sloupcích jednotlivé terminální a neterminální symboly gramatiky. Pro každý stav v konečném automatu postupujeme následovně.



Obrázek 2.2: LR(0) automat pro generování LR tabulky.

- Pokud jde hrana ze stavu 0 přes symbol $a \in T$ do stavu označeného jako 3, pak do akční části tabulky zapíšeme $s3$ na pozici $(0, a)$.
- Pokud jde hrana ze stavu 0 přes symbol $A \in N$ do stavu označeného jako 3, pak do přechodové části tabulky zapíšeme 3 na pozici $(0, A)$.
- Pokud stav obsahuje nějaké pravidlo se značkou za posledním symbolem své pravé strany $(r_x : B \to \alpha\bullet)$, pak do akční části zapíšeme r_x na pozici všech symbolů, jež jsou ve $Follow$ množině neterminálu B.

Jakmile máme LR tabulku a nějaký zásobníkový automat, pak postupujeme následovně. Vložíme na zásobník $\langle \$, q_0 \rangle$ a nastavíme stav q_0. Předpokládejme vstupní symbol a.

	a	b	c	d	e	k	**$**	S	A	B
0	s3	s4			s5			1	2	B
1							$\surd$			
2	s3	s4			s5			6		
3	s3	s4			s5			7		
4		s9				s10			8	
5	r4	r4			r4					
6							r1			
7				s11						
8			s12							
9		s9				s10			13	
10			r6							
11	r2	r2		r2	r2		r2			
12	r3	r3		r3	r3		r3			
13			s14							
14			r5							

Tabulka 2.5: LR tabulka

- Pokud $\alpha[q_0, a] = sq$, pak vložíme na zásobník $\langle a, q\rangle$, přečteme další vstupní symbol a nastavíme stav na q.
- Pokud $\alpha[q_0, a] = rp$, kde $p : A \rightarrow X_1X_2 \ldots X_n$ je pravidlo dané gramatiky a na zásobníku je sekvence $\langle X_1, q_1\rangle, \langle X_2, q_2\rangle, \ldots, \langle X_n, q_3\rangle, \langle x, q\rangle$, pak vrchol zásobníku až po $\langle x, q\rangle$ zaměníme za $\langle A, \beta[q, A]\rangle$.
- Pokud $\alpha[q_0, a]$ označuje úspěch, pak končíme.
- Jinak jde o chybu.

2.4.3 Sémantický analyzátor

Vstupem tomuto analyzátoru je právě vnitřní reprezentace struktury zdrojového kódu vytvořená syntaktickým analyzátorem. Sémantický analyzátor provádí typovou kontrolu výrazu. Zjišťuje, zda všechny použité operátory ve výrazu mají operandem povolené specifikace. Například kontroluje, zda řetězcovou konstantu přiřazuje do proměnné, jejíž typ může řetězcovou konstantu uchovávat. Některé překladače však dovolují implicitní typovou konverzi. Sémantický analyzátor tedy provádí kontrolu, zda gramaticky správné fráze neporušují kontextová omezení (např., použité proměnné musí být deklarované).

2.4.4 Generování mezikódu

Po lexikální, syntaktické a sémantické kontrole následuje generování intermediární reprezentace zdrojového programu (mezikód). V praxi většinou nejde o samostatnou část překladu. Generování vnitřní formy programu se přímo spojuje se syntaktickou analýzou. Intermediární kód slouží jako podklad pro optimalizaci a generování cílového kódu.

Kapitola 3

Moderní metody popisu jazyků

Moderní programovací jazyky jsou uzpůsobeny tak, aby na jednom projektu mohlo nezávisle na sobě pracovat více lidí, což obstarává zejména jejich modularita. Tedy jeden projekt se skládá z více zdrojových souborů (neboli modulů). Tato vlastnost umožňuje překládat každý modul zvlášť a využít tak paralelního zpracování. V dnešní době je tak většina moderních překladačů posílena z hlediska rychlosti. V této kapitole se dovíme o možnostech, jak můžeme překlad posílit i z hlediska jeho vyjadřovací síly. Popíšeme tzv. řízené gramatiky (regulated grammars), které právě vyjadřovací sílu zvyšují. Dále se pak budeme zabývat nejdůležitějším bodem této práce – stromem řízenou gramatikou.

3.1 Řízené gramatiky

Základy těchto gramatik stojí na bezkontextových gramatikách, jejichž derivace jsou nějakým způsobem buďto omezené nebo řízené. Nyní se budeme blíže věnovat několika typům řízených gramatik. Zdůrazněme také, že všechny gramatiky uvedené v této sekci mají ekvivalentní generativní sílu.

Maticové gramatiky

Maticové gramatiky omezují bezkontextové gramatiky použitím množiny M, ve které jsou řetězce, určující možné derivační kroky. To znamená, že přímo udávají jaké pravidla a v jakém pořadí musejí být derivační kroky provedeny, aby daný řetězec patřil do jazyka generovaného touto gramatikou. Více se o této gramatice můžeme dozvědět v [1] a [3].

Definice 3.1.1. *Maticová gramatika* je dvojice: $H = (G, M)$, kde

- $G = (N, T, P, S)$ je bezkontextová gramatika
- $M \subseteq P^*$ je konečný jazyk nad množinou pravidel P

Derivační krok pro maticové gramatiky je definován následovně.

Definice 3.1.2. Pro $x, y \in (N \cup T)^*$ a $m \in M$ je $x \Rightarrow y[m]$ v H pokud existují $x_0, \ldots, x_n$ takové, že $x = x_0$, $x_n = y$ a platí $x_0 \Rightarrow x_1[p1] \Rightarrow \ldots \Rightarrow x_n[p_n]$ je v G a $m = p_1 \ldots p_n$.

Zjednodušeně bychom mohli říct, že derivace $x \Rightarrow y$ bude v maticové gramatice, jestliže v jazyce m existuje takový řetězec, který je totožný s posloupností pravidel použitých právě k této derivaci.

Příklad 3.1.3. Mějme maticovou gramatiku $H = (G, M)$.

- $G = (N, T, P, S)$, kde $N = \{S, A, B\}$, $T = \{a, b, c\}$ a množina pravidel P je následující.

$$\begin{array}{lccl} \mathbf{1:} & S & \rightarrow & AB \\ \mathbf{2:} & A & \rightarrow & aA \\ \mathbf{3:} & B & \rightarrow & bBc \\ \mathbf{4:} & A & \rightarrow & a \\ \mathbf{5:} & B & \rightarrow & bc \end{array}$$

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

Pak derivace $S \Rightarrow AB[1] \Rightarrow aAbBc[23] \Rightarrow aaAbbBcc[23] \Rightarrow aaabbbccc[45]$ je v H.

Gramatiky s nahodilým kontextem

Tyto gramatiky používají k omezení pravidel dvě množiny, tzv. povolující kontext (permitted context) a zakazující kontext (forbidding context). O této gramatice se můžete více dočíst v [3] a [9].

Definice 3.1.4. *Gramatika s nahodilým kontextem* s kontrolou výskytu je trojice: $H = (G, R, F)$, kde

- $G = (N, T, P, S)$ je bezkontextová gramatika.
- R, F jsou dvě konečné množiny binárních relací z P do N. Neboli $R \subseteq P \times N$ a $F \subseteq P \times N$.

Notace 3.1.5. Jestliže $p : A \rightarrow x \in P$, $R(p) = Q$, a $F(p) = K$, tak budeme psát $(p : A \rightarrow x, Q, K)$, kde množina neterminálů Q je povolující kontext a množina neterminálů K je zakazující kontext.

Definice 3.1.6. Jestliže pro každé pravidlo $(p : A \rightarrow x, Q, K)$ platí $Q = \emptyset$, pak je H *zakazující gramatika*.

Pokud jsou ve větné formě x všechny neterminály z množiny Q neboli $Q \subseteq alph(x)$ a zároveň v ní není žádný symbol z množiny K tedy $K \cap alph(x) = \emptyset$, pak je derivační krok povolen. Následující definice zavádí derivační krok této metody formálněji.

Definice 3.1.7. Pro $x, y \in (N \cup T)^*, p \in P$ je $x \Rightarrow y[p]$ v H pokud je $x \Rightarrow y[p]v\, G, R(p) \subseteq alph(x)$ a $F(p) \cap alph(x) = \emptyset$.

Příklad 3.1.8. Mějme gramatiku s nahodilým kontextem $H = (G, R, F)$, kde

- $G = (N, T, P, S)$, kde $N = \{S, A, B, D\}$, $T = \{a\}$, a množina pravidel jako trojice $(p, R(p), F(p))$:

Jedná se o zakazující gramatiku a jazyk generovaný touto gramatikou je $L(H) = \{a^{2^n} \mid n \geq 1\}$. Ukážeme vygenerování řetězce aa.

1: $(S \rightarrow AA, \emptyset, \{B, D\})$
2: $(A \rightarrow B, \emptyset, \{S, D\})$
3: $(B \rightarrow S, \emptyset, \{A, D\})$
4: $(A \rightarrow D, \emptyset, \{S, B\})$
5: $(D \rightarrow a, \emptyset, \{S, A, B\})$

$$
\begin{aligned}
S &\Rightarrow AA[1], \{2, 4\} && (3.1)\\
&\Rightarrow AD[4], \{4\} && (3.2)\\
&\Rightarrow DD[4], \{5\}\\
&\Rightarrow Da[5], \{5\}\\
&\Rightarrow aa[5]
\end{aligned}
$$

Za derivací je uvedena množina pravidel, které lze využít pro příští derivaci. Je odvozena z aktuální větné formy a povolujícího a zakazujícího kontextu.

Pro první derivaci (3.1) je aktuální větná forma AA, tedy uvažujme pravidla $2, 4$. Povolující kontext musí být podmnožinou dané větné formy, což v našem případě platí, protože prázdná množina je podmnožina množiny $\{A, A\}$. Dále průnik zakazujícího kontextu a množiny dané větné formy musí být prázdná množina. Pro pravidlo 2 je průnik $\{S, D\}$ a $\{A, A\}$ roven $\emptyset$, takže dané pravidlo můžeme použít. Pro pravidlo 4 s množinou zakazujícího kontextu $\{S, B\}$ podmínka platí taktéž - můžeme použít.

Nyní daný postup zopakujeme na derivaci druhou (3.2). Aktuální větná forma je AD. Uvažujme pravidla $2, 4, 5$. Podmínka pro povolující kontext je splněna pro všechna tři pravidla. Podmínka pro zakazující kontext však pouze pro 4. Pro pravidlo 2 je průnik zakazujícího kontextu s aktuální větnou formou $\{D\}$ a pro pravidlo 5 je to $\{A\}$. Protože výsledkem průniku není prázdná množina, jsou pro nás tyto dvě pravidla zakázána. Lze tedy použít pouze pravidlo 4.

Analogicky postupujeme i pro další derivační kroky.

Programové gramatiky

Více o poslední zde zmíněné gramatice najdete v [3] a [8].

Definice 3.1.9. *Programová gramatika* je dvojice: $H = (G, R)$, kde

- $G = (N, T, P, S)$ je bezkontextová gramatika.
- R je konečná množina binárních relací nad P.

Notace 3.1.10. Jestliže $p : A \rightarrow x \in P$, $R(p) = Q$, tak budeme psát $(p : A \rightarrow x, Q)$.

Derivační krok této gramatiky je formálně definován následovně.

Definice 3.1.11. Pro $(x, p), (y, q) \in (N \cup T)^* \times P$ je $(x, p) \Rightarrow (y, q)$ v H pokud je $x \Rightarrow y[p]v$ G, $q \in R(p)$.

Tato definice říká, že pokud bylo použito pravidlo p, pak příští použité pravidlo musí být z množiny $R(p)$.

$$
\begin{array}{lrl}
\textbf{1:} & (S & \rightarrow \quad ABC, \{2,5\}) \\
\textbf{2:} & (A & \rightarrow \quad aA, \{3\}) \\
\textbf{3:} & (B & \rightarrow \quad bB, \{4\}) \\
\textbf{4:} & (C & \rightarrow \quad cC, \{2,5\}) \\
\textbf{5:} & (A & \rightarrow \quad a, \{6\}) \\
\textbf{6:} & (B & \rightarrow \quad b, \{7\}) \\
\textbf{7:} & (C & \rightarrow \quad c, \{\})
\end{array}
$$

Příklad 3.1.12. Mějme gramatiku $H = (G, R)$, kde

- $G = (N, T, P, S)$, kde $N = \{S, A, B, C\}$, $T = \{a, b, c\}$, a množina pravidel:

Generovaný jazyk je $L(H) = \{a^n b^n c^n \mid n \geq 1\}$. Ukážeme vygenerování řetězce $aabbcc$.

$$
\begin{array}{rcl}
S & \Rightarrow & ABC[1], \{2,5\} \\
 & \Rightarrow & aABC[2], \{3\} \\
 & \Rightarrow & aAbBC[3], \{4\} \\
 & \Rightarrow & aAbBcC[4], \{2,5\} \\
 & \Rightarrow & aabBcC[5], \{6\} \\
 & \Rightarrow & aabbcC[6], \{7\} \\
 & \Rightarrow & aabbcc[7], \{\}
\end{array}
$$

Za derivací je uvedena množina pravidel, které lze využít pro příští derivaci. Je to stejná množina, která je uvedena u definice pravidel této konkrétní gramatiky.

3.2 Stromem řízené gramatiky

V této sekci se budeme zabývat stromem řízenou gramatikou (tree controlled grammar), což představuje velmi důležitý bod tohoto textu. Povíme si, jak tyto gramatiky pracují a později se v textu zaměříme na možnosti paralelní analýzy za použití právě tohoto typu gramatik.

Na rozdíl od výše popsaných typů se liší tím, že neomezuje derivační pravidla, ale zkoumá určité vlastnosti derivačního stromu. Jednou možností je horizontálně spojovat symboly uzlů na stejné úrovni derivačního stromu a zkoumat, zda vzniklý řetězec patří do řídícího jazyka, či nikoliv. Existuje však i vertikální přístup, kdy se nekontrolují symboly z uzlů na stejné úrovni, ale naopak symboly uzlů tvořící cestu od kořene derivačního stromu k jeho listům.

3.2.1 Gramatika řízená cestami

Tato metoda tedy využívá vertikální přístup ke kontrole derivačního stromu. Byla navržena a popsána v [5].

Definice 3.2.1. *Stromem řízená gramatika* (*tree controlled grammar*) je dvojice (G, R), kde

- $G = (N, T, P, S)$ je bezkontextová gramatika
- $R \subseteq (N \cup T)^*$ je řídící jazyk nad T

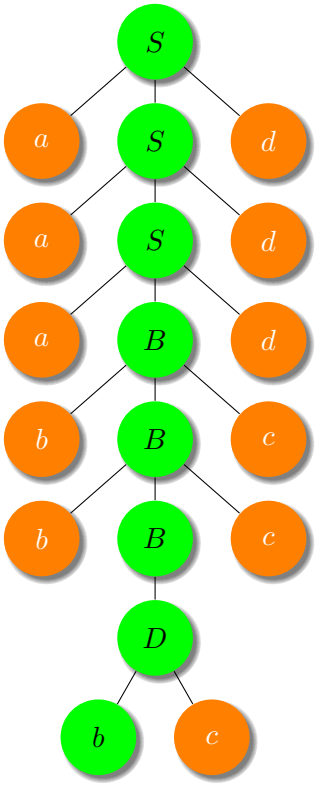
Příklad 3.2.2. Mějme gramatiku s řízenou cestou $H = (G, R)$, kde

- $G = (N, T, P, S)$, kde $N = \{S, B, D\}$, $T = \{a, b, c, d\}$, a množina pravidel P:

$$\begin{array}{lrcl} \textbf{1:} & S & \to & aSd \\ \textbf{2:} & S & \to & aBd \\ \textbf{3:} & B & \to & bBc \\ \textbf{4:} & B & \to & D \\ \textbf{5:} & D & \to & bc \end{array}$$

- $R = \{S^i B^i Db \mid i \geq 1\}$

Generovaný jazyk je $L(H) = \{a^j b^j c^j d^j \mid j \geq 1\}$.



Obrázek 3.1: Derivační strom pro $S \Rightarrow^* a^3b^3c^3d^3$ se zeleně vyznačenou cestou

Na obrázku 3.1 můžeme vidět derivační strom reprezentující řetězec $x = aaabbbcccddd$. Zeleně vyznačená cesta s po spojení symbolů označených uzlů vytvoří řetězec $word(s) = SSSBBBDb$ neboli také $word(s) = S^3B^3Db$. Z druhého vyjádření získaného řetězce je zřejmé, že $word(s) \in R$ a tedy že $x \in L(H)$.

Pro tento konkrétní příklad nám stačí najít pouze jednu cestu v derivačním stromě, která spadá do R. Existují však případy, kdy kontrola jediné cesty nestačí. Proto se později budeme zabývat gramatikou řízenou n cestami. Nyní si však ještě řekněme něco o generativní síle těchto gramatik.

Generativní síla gramatik řízených cestami

Pro každou bezkontextovou gramatiku G, existuje regulární jazyk, který popisuje všechny cesty v derivačním stromě řetězce $w \in G$ (více prop 1. v [5]). Neexistuje kontrolní regulární jazyk, který by zvyšoval generativní sílu gramatiky G (více prop 1. a prop 2. v [5]). Budeme tedy zkoumat neregulární kontrolní jazyky. Jak uvidíme v následujícím textu této kapitoly, dostatečně silným jazykem pro zvýšení generativní síly je jazyk lineární.

Gramatika řízená n cestami

Následující definice a příklady jsou převzaty z článku [4]. Zaveďme si jazyk ${}_{n-path}L(G,R)$, který tato gramatika generuje.

Definice 3.2.3. Pro všechny $x \in T^*, x \in {}_{n-path}L(G,R)$ existuje derivační strom $t \in {}_G\Delta(x)$ takový, že existuje množina Q_t, ve které je n cest ze stromu t takových, že pro všechny $p \in Q_t$ platí $word(p) \in R$.

Tedy pokud se nám podaří najít n cest v derivačním stromě daného řetězce x, které náleží do řídící množiny R, pak x patří do jazyka generovaného touto gramatikou ($x \in {}_{n-path}L(G,R)$).

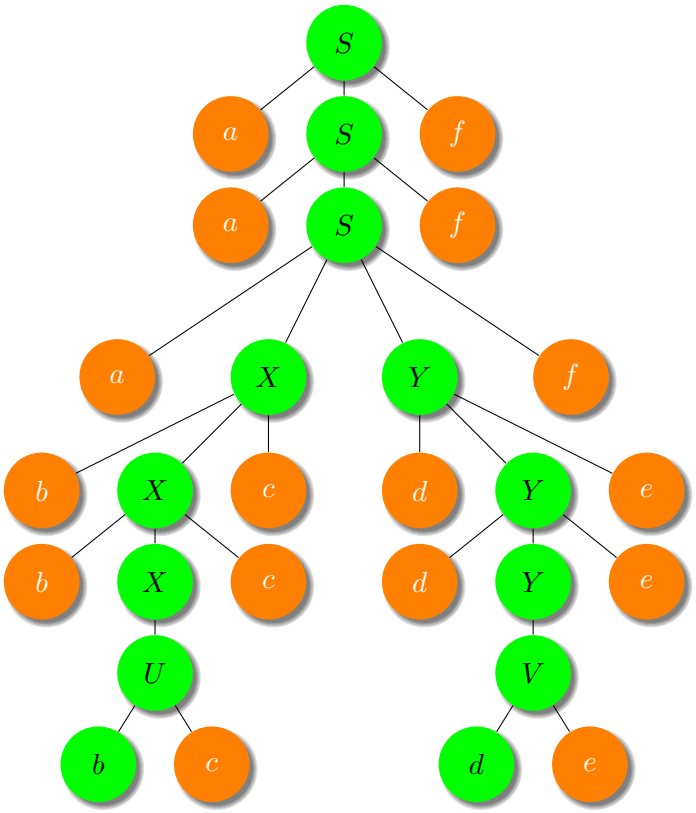
Příklad 3.2.4. Mějme stromem řízenou gramatiku generující ${}_{n-path}L(G,R)$ pro $n = 2$, kde

- $G = (N,T,P,S)$, kde $N = \{S,X,Y,U,V\}$, $T = \{a,b,c,d,e,f\}$, a množina pravidel P:

$$
\begin{array}{lrcl}
\textbf{1:} & S & \rightarrow & aSf \\
\textbf{2:} & S & \rightarrow & aXY \\
\textbf{3:} & X & \rightarrow & bXc \\
\textbf{4:} & Y & \rightarrow & dYe \\
\textbf{5:} & X & \rightarrow & U \\
\textbf{6:} & U & \rightarrow & bc \\
\textbf{7:} & Y & \rightarrow & V \\
\textbf{8:} & V & \rightarrow & de
\end{array}
$$

- $R = \{S^iX^iUb \cup S^iY^iVd \mid i \geq 1\}$

Generovaný jazyk je ${}_{2-path}L(G,R) = \{a^jb^jc^jd^je^jf^j \mid j \geq 1\}$. Tento jazyk není bezkontextový. Obrázek 3.2 ukazuje derivační strom pro $S \Rightarrow^* a^3b^3c^3d^3e^3f^3$. Na stejném obrázku lze také vidět, že pro rozhodnutí, zda daný řetězec patří do jazyka ${}_{2-path}L(G,R) = \{a^jb^jc^jd^je^jf^j \mid j \geq 1\}$ potřebujeme v derivačním stromě najít dvě cesty patřící do řídícího jazyka R. Kdyby byla v derivačním stromě pouze jedna cesta z R, pak bychom ověřili v podstatě jen „půlku“ kontrolovaného řetězce. Byli bychom schopni říct, zda se jedná o řetězec $a^jb^jc^j\alpha$, resp. $\alpha d^je^jf^j$, pro nějaké $j \geq 1$ a $\alpha \in T^*$.



Obrázek 3.2: Derivační strom pro $S \Rightarrow^* a^3b^3c^3d^3e^3f^3$ se zeleně označenými cestami

Příklad 3.2.5. Mějme stromem řízenou gramatiku generující ${}_{n-path}L(G,R)$ pro $n \geq 2$, kde

- $G = (N, T, P, S)$, kde $N = \{\{S\} \cup \{A_i, B_i \mid 1 \leq i \leq n\}\}$, $T = \{\{a_i \mid 1 \leq i \leq 2n+2\}\}$ a množina pravidel P pro $0 \leq i \leq n-1$:

$$\begin{array}{lrcl} \textbf{1:} & S & \to & a_1 S\, a_{2n+2} \mid a_1 A_1 A_2 \ldots A_n a_{2n+2} \\ \textbf{2:} & A_{i+1} & \to & a_{2i+2} A_{i+1} a_{2i+3} \mid B_{i+1} \\ \textbf{3:} & B_{i+1} & \to & a_{2i+2} a_{2i+3} \end{array}$$

- $R = \bigcup_{i=1}^{n} \{S^x A_i^k B_i a_{2i} \mid k \geq 1\}$

Zřejmě $R \in LIN$. Mějme derivaci $S \Rightarrow^k a_1^k S a_{2n+2}^k \Rightarrow a_1^k a_1 A_1 A_2 \ldots A_n a_{2n+2} a_{2n+2}^k \Rightarrow^{n \cdot k} a_1^{k+1} a_2^k B_1 a_3^k \ldots a_{2n}^k B_n a_{2n+1}^k a_{2n+2}^{k+1} \Rightarrow^n a_1^{k+1} a_2^{k+1} a_3^{k+1} \ldots a_{2n}^{k+1} a_{2n+1}^{k+1} a_{2n+2}^{k+1}$.

Pro $n \geq 1$ jsou cesty v derivačním stromě popsány řídícím jazykem R a v tomto případě (G, R) generuje ${}_{n-path}L(G,R) = \{a_1^k \ldots a_{2n+2}^k \mid k \geq 1\}$, přičemž tento jazyk není bezkontextový.

Příklad 3.2.6. Mějme stromem řízenou gramatiku generující ${}_{n-path}L(G,R)$ pro $n=2$, kde

- $G=(N,T,P,S)$, kde $N=\{A,B,C,D,E,F,G,H,I\}$, $T=\{a,b,c,d\}$, a množina pravidel P:

$$
\begin{array}{lrcl}
\textbf{1:} & A & \rightarrow & aA \mid aB \\
\textbf{2:} & B & \rightarrow & Bb \mid C \\
\textbf{3:} & C & \rightarrow & cC \mid D \\
\textbf{4:} & D & \rightarrow & Dd \mid HHH \\
\textbf{5:} & E & \rightarrow & Ea \mid I \\
\textbf{6:} & F & \rightarrow & bF \mid E \\
\textbf{7:} & G & \rightarrow & Gc \mid F \\
\textbf{8:} & H & \rightarrow & dH \mid G \\
\textbf{9:} & I & \rightarrow & a
\end{array}
$$

- $R=\{A^rB^sC^tD^uH^uG^tF^sE^rIa \mid r,s,t,u \geq 0\}$

Jazyk generovaný touto gramatikou je ${}_{3-path}L(G,R)=\{(a^rc^td^ub^s)^4 \mid r>0, s,t,u \geq 0\}$. Tento jazyk také není bezkontextový.

Kapitola 4

Návrh metody

Cílem této práce je navrhnout metodu pro paralelní syntaktickou analýzu. Jak už bylo uvedeno dříve, nebudeme se zabývat pouze klasickou analýzou, ale zaměříme se na stromem řízené gramatiky, kde je třeba navíc hledat cesty v derivačním stromě. Získáme tak rychlejší a silnější syntaktickou analýzu.

V této kapitole si podrobně popíšeme sekvenční metodu z článku [4], která vychází ze syntaktické analýzy shora dolů a navrhneme její paralelní zpracování. Následně se pokusíme navrhnout jak sekvenční, tak paralelní metodu využívající přístup zdola nahoru.

Ještě než začneme se samotným návrhem, tak je důležité si uvědomit závislost mezi samotnou syntaktickou analýzou a analýzou derivačního stromu (hledání cest). Abychom mohli derivační strom kontrolovat, tak jej potřebujeme nejdříve vygenerovat. V důsledku tohoto je logické, že analýza derivačního stromu nemůže nikdy skončit dříve, než samotná syntaktická analýza. A nejen že nemůže skončit dříve, ale ani během výpočtu nemůže být před syntaktickou analýzou. Z toho plyne, že nemusíme hledat řešení jak urychlit analýzu derivačního stromu, pokud nejdříve neurychlíme syntaktickou analýzu.

4.1 Analyzované gramatiky

Na této gramatice si ukážeme, jak jednotlivé metody pracují, tedy jak provádí syntaktickou analýzu a jak hledají cesty v daném derivačním stromě. Stejná gramatika pro všechny metody nám zároveň poskytne jejich přímé srovnání. Stejnou gramatiku, alespoň pro vytváření derivačního stromu, pak budeme používat v kapitole 5.3 pro testování.

Příklad 4.1.1. Mějme gramatiku $G = (N, T, P, S)$ a stromem řízenou gramatiku generující jazyk ${}_{n-path}L(G, R)$, kde

- $N = \{S, A, B\}$, $T = \{a, b, c, d, e, k\}$ a množina pravidel P (čísla představují jedinečné označení pravidel):

$$
\begin{array}{llcl}
\textbf{1:} & S & \rightarrow & AA \\
\textbf{2:} & A & \rightarrow & aAd \\
\textbf{3:} & A & \rightarrow & bBc \\
\textbf{4:} & A & \rightarrow & e \\
\textbf{5:} & B & \rightarrow & bBc \\
\textbf{6:} & B & \rightarrow & k
\end{array}
$$

- $R = \{SA^m B^{m-1} k \mid m \geq 1\}$

Generovaný jazyk gramatikou G je

$$L(G) = \{a^i(b^j k\, c^j + e)d^i a^s(b^t k\, c^t + e)d^s \mid i, j, t, s \geq 0\}$$

Nyní mějme řetězec $w = abkcdaed$. Vidíme, že $w \in L(G)$. Zabývejme se však tím, zda platí $w \in {}_{n-path}L(G, R)$ pro nějaké $n = 1$ nebo $n = 2$.

- Pokud pro řetězec w platí $i = j$ nebo $s = t$ (nikoliv však obě rovnosti najednou), pak $w \in {}_{1-path}L(G, R)$.
- Pokud pro řetězec w platí $i = j$ a zároveň $s = t$, pak $w \in {}_{2-path}L(G, R)$.

Dodejme, že pokud v derivačním stromě řetězce nalezneme alespoň 1 cestu, pak daný řetězec patří do jazyka ${}_{1-path}L(G, R)$, pro který platí ${}_{1-path}L(G, R) \subset L(G)$. Podobně, pokud v derivačním stromě řetězce nalezneme alespoň 2 cesty, pak daný řetězec patří do jazyka ${}_{2-path}L(G, R)$, pro který platí ${}_{2-path}L(G, R) \subset L(G)$.

Podle známých algoritmů (viz. 2.4.2) sestrojíme LL tabulku pro analýzu shora dolů a tabulku pro analýzu zdola nahoru (viz. 2.4.2).

	a	b	c	d	e	k	**$**
S	1	1			1		
A	2	3			4		
B		5				6	

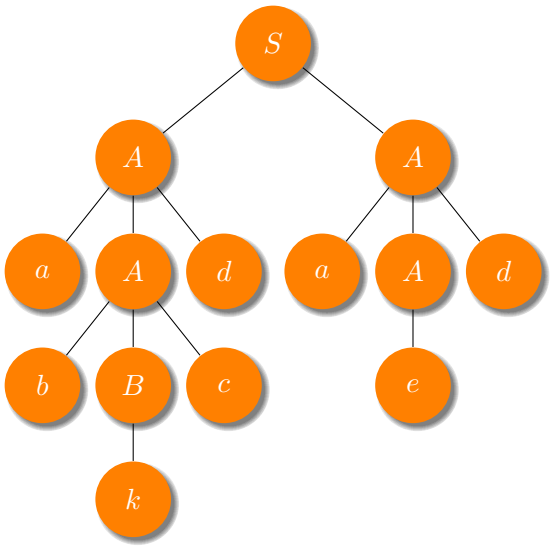
Tabulka 4.1: LL tabulka

	a	b	c	d	e	k	**$**	S	A	B
0	s3	s4			s5			1	2	B
1							√			
2	s3	s4			s5			6		
3	s3	s4			s5			7		
4		s9				s10			8	
5	r4	r4			r4					
6							r1			
7				s11						
8			s12							
9		s9				s10			13	
10			r6							
11	r2	r2		r2	r2		r2			
12	r3	r3		r3	r3		r3			
13			s14							
14			r5							

Tabulka 4.2: LR tabulka

Pro lepší představu si uveďme i grafickou podobu derivačního stromu řetězce $w = abkcdaed$. Bez jakékoliv analýzy si zřejmě všimneme, že v derivačním stromě je právě jedna cesta p, která

začíná v kořenovém uzlu S a končí v listovém k. Tedy $word(p) = SAABk$. Předpokládejme, že řetězec $w \in {}_{1-path}L(G, R)$. Dále v tomto textu si daný předpoklad potvrdíme pomocí výpočtu navržených metod.



Obrázek 4.1: Derivační strom pro řetězec $w = abkcdaed$.

Nakonec si ještě uveďme dvě gramatiky, které generují jazyk reprezentující cesty v daném stromě. Jedna shora dolů (4.1.2) a druhá zdola nahoru (4.1.3). V následujícím textu je pak využijeme pro lepší pochopení navržených metod. Dále předpokládejme nějaké zásobníkové automaty, které přijímají jazyky $L(G_1)$ a $L(G_2)$. Jejich konfigurace pro nás není příliš důležitá, potřebujeme pouze vědět, že existují a že tyto jazyky dokáží přijímat.

Definice 4.1.2. Mějme gramatiku $G_1 = (N, T, P, O)$, kde

- $N = \{O, P, R, T, V\}$
- $T = \{S, A, B, k\}$
- P (čísla představují jedinečné označení pravidel):

$$\begin{array}{lrcl} \textbf{1:} & O & \rightarrow & SP \\ \textbf{2:} & P & \rightarrow & AR \\ \textbf{3:} & R & \rightarrow & k \\ \textbf{4:} & R & \rightarrow & Tk \\ \textbf{5:} & T & \rightarrow & AV \\ \textbf{6:} & V & \rightarrow & TB \\ \textbf{7:} & V & \rightarrow & B \end{array}$$

Definice 4.1.3. Mějme gramatiku $G_2 = (N, T, P, O)$, kde

- $N = \{O, P, T, V\}$
- $T = \{S, A, B, k\}$
- P (čísla představují jedinečné označení pravidel):

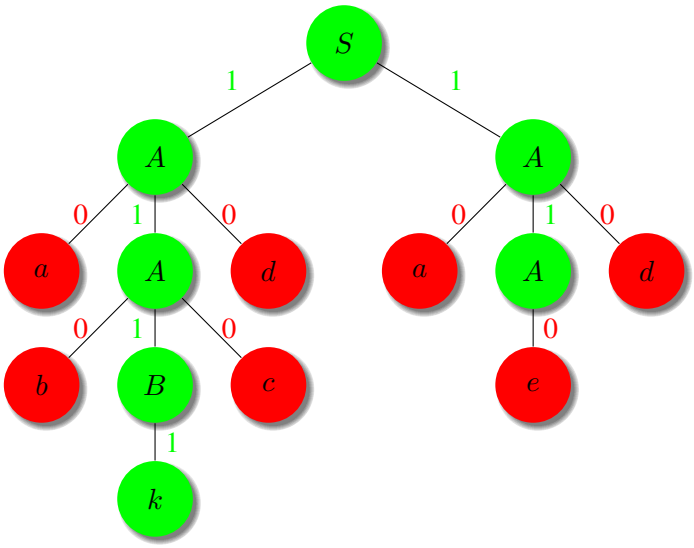
1: $O \rightarrow kP$
2: $P \rightarrow AS$
3: $P \rightarrow TAS$
5: $T \rightarrow BV$
6: $V \rightarrow A$
7: $V \rightarrow TA$

4.2 Přístup shora dolů

Podívejme se blíže na přístup shora dolů. Sekvenční část této metody a tedy i následující text vychází z článku [4]. Dále si také představíme paralelní zpracování tohoto přístupu, nejdříve si však neformálně ukažme způsob, jakým bude syntaktická analýza probíhat.

4.2.1 Princip metody

Princip metody si ukážeme na konkrétním příkladě. Použijeme definice gramatik z 4.1. Na obrázku 4.2 je derivační strom pro řetězec $w = abkcdaed$ s již označenými cestami. Vysvětleme si nyní, jak takový výsledek získáme. Každou hranu budeme označovat buď číslem 1 (patří do potenciální cesty) nebo 0 (nepatří do potenciální cesty). Důležité je uvědomit si, že pokud některý uzel nepatří do cesty, pak ani žádná další derivace vycházející z tohoto uzlu nebude patřit do žádné cesty. Kdybychom zjistili, že hned kořenový uzel nepatří do řídícího jazyka, pak s jistotou můžeme tvrdit, že v daném derivačním stromě neexistuje žádná potenciální cesta. Jak uvidíme později, tak právě tohle je obrovská výhoda oproti metodě přistupující zdola nahoru (viz. 4.3). Nemusíme totiž kontrolovat každou cestu zvlášť, protože jednotlivé cesty se „rozštěpují“ postupně v nižších úrovních stromu. Jejich prefix je proto z principu stejný.



Obrázek 4.2: Označený derivační strom pro řetězec $w = abkcdaed$.

4.2.2 Postup analýzy

Pro nějaké $n \geq 1$ mějme gramatiku $H = (G, R)$ a jazyk gramatikou H generovaný ${}_{n-path}L(G, R)$, kde $G = (N, T, P, S)$ je jednoznačná bezkontextová gramatika. Dále předpokládejme, že řídící jazyk R je generován gramatikou $G_R = (N_R, V, P_R, S_R)$, kde $V = (N \cup T)$.

Budeme konstruovat označený derivační strom s množinou označení $\Psi = \{0, 1\}$, jenž má následující význam. Mějme cestu p v derivačním stromě t a hranu e mezi každými dvěma uzly v dané cestě. Když nějakou hranu e, která patří do cesty p označíme jako $0 \in \Psi$, pak to znamená, že daná cesta p není popsána v jazyce R. Naopak pokud jsou všechny hrany v cestě p označeny jako $1 \in \Psi$, pak daná cesta p může být potenciálně popsána řídícím jazykem R.

Konstrukce derivačního stromu

Nyní budeme konstruovat označený derivační strom t pro řetězec $w \in L(G)$ podle známé metody uvedené v 2.4.2 nebo v [6]. Tedy začneme od S a dále vytváříme derivační strom t podle pravidel v G tak, že spojením všech terminálů, které představují listy stromu, dostaneme požadovaný řetězec w. Tedy derivační strom t nyní představuje derivaci $S \Rightarrow^* w$, kde S reprezentuje startující symbol v gramatice G.

Značení hran

Předpokládejme pravidlo $r : A \rightarrow A_1 A_2 \dots A_j \in P, j \geq 1$, které je použito v derivačním kroku $X \Rightarrow Y$, kde $X, Y \in (N \cup T)^*$. Nyní potřebujeme ohodnotit hrany mezi uzlem A a každým dalším uzlem A_j pro $j = 1, 2, \dots, n$. Řekněme, že t' bude derivační strom reprezentující $S \Rightarrow^* \omega_1 A_1 A_2, \dots A_j \omega_2$, pro nějaké $\omega_1, \omega_2 \in (N \cup T)^*$. Vidíme, že derivační strom t' je podstromem derivačního stromu t. A také platí, že každá cesta v t' je počátkem alespoň jedné cesty v t. Dále budeme rozlišovat dva případy:

- Jestliže jsou všechny hrany derivačního stromu t' označeny, můžeme pokročit k další derivaci.
- Jestliže nějaká hrana v t' není označena, potřebujeme zjistit její hodnotu

Pro každou neoznačenou hranu e mezi uzly A a A_j cesty p' v t' zkontrolujeme, zda G_R může generovat řetězec formy $word(p')word(A_j)$. Jelikož je velikost $word(p')$ konečná a velikost $|word(A_j)| = 1$, tak můžeme tuto kontrolu provést v polynomiálním čase. Jestliže daný řetězec je v G_R, označíme hranu e jako $1 \in \Psi$, jinak značíme jako $0 \in \Psi$. Dále budeme rozlišovat následující případy.

- Jestliže t' neobsahuje list se vstupní hranou označenou 1, pak $w \notin_{n-path} L(G, R)$.
- Jestliže t' obsahuje alespoň jeden list označený symbolem z množiny N, pak pokračujeme dalším derivačním krokem.
- Jestliže všechny listy z t' jsou označeny terminálním symbolem a pro alespoň n listů z t existuje hrana označená 1, pak $w \in {}_{n-path}L(G, R)$.

Syntaktický analyzátor

Syntaktický analyzátor používá dva zásobníkové automaty a množinu trojic (q, α, p), kde q je stav druhého automatu, α aktuální obsah zásobníku a p ukazatel na symbol prvního zásobníku. Lépe řečeno, každá trojice představuje jednu cestu, potom p můžeme chápat jako označení symbolu, který

v dané cestě má následovat. Druhý automat totiž zná pravidlo, které první automat použil. Z toho může odvodit následující symboly (pravá strana pravidla) pro aktuálně kontrolovaný. Jakmile se první automat potom dostane ke kontrole symbolu následujícího symboly, který p reprezentuje, tak se právě tato trojice stane aktuální a druhý automat pokračuje v její kontrole. Nyní první automat simuluje konstrukci derivačního stromu pomocí LL tabulky podle známé metody.

- Jestliže je na vrcholu zásobníku neterminál A, první vstupní symbol je a a existuje pravidlo $A \rightarrow x$ na pozici $[A, a]$ v LL tabulce, pak automat přepíše A na zásobníku obráceným řetězcem x - tzn. $reversal(x)$.
- Jestliže a je na vrcholu zásobníku a terminál a je první na vstupu, pak automat přečte a ze vstupu a odstraní jej ze zásobníku
- V ostatních případech jde o syntaktickou chybu.

Mějme na vstupu řetězec $abkcdaed$. Na vrcholu zásobníku začínáme se symbolem S. Jelikož a je první vstupní symbol, tak hledáme v LL tabulce pravidlo na pozici $[S, a]$. Použijeme pravidlo $1 : S \rightarrow AA$. Automat přepíše S, které je aktuálně na vrcholu zásobníku na $reversal(AA)$.

Během výpočtu druhý automat hledá potenciálně správné cesty v derivačním stromě. V množině trojic je ze začátku pouze jedna položka (q_0, ϵ, A), kde $A \in (N \cup T)$ je ukazatel na nějaký symbol na prvním zásobníku a ϵ je startující symbol na druhém zásobníku. Jestliže první automat umožňuje výpočet kroku se symbolem a a v množině existuje trojice (q, α, p), kde p je ukazatel právě na symbol a, pak druhý automat položí α na jeho zásobník, přesune se do stavu q a provádí kroky pro a, dokud jej nepřečte ze vstupu, resp. nepotřebuje další vstupní symbol. Například po expanzi S na AA, druhý automat nalezne v množině trojici (q_0, ϵ, S), tak se posune do stavu q_0 a vloží ϵ na zásobník.

To simuluje krok $\epsilon q_0 S \vdash \beta q$, kde $\epsilon q_0 S \rightarrow \beta q$ je přepisovací pravidlo druhého automatu. Nyní rozlišujeme, zda vstupem druhému automatu byl neterminální nebo terminální symbol.

- Na vstupu je neterminální symbol $a = A$. Dále předpokládejme, že první automat provedl expanzi podle pravidla $A \rightarrow A_1 A_2 \ldots A_n$, pro nějaké $n \geq 1$. Syntaktický analyzátor odstranil použitou trojici z množiny a jestliže druhý automat neodmítl vstup, tak zpět do množiny vloží n trojic (q, β, A_m), kde $1 \leq m \leq n$, q je aktuální stav automatu a β je aktuální obsah zásobníku. Nakonec si řekněme, že každý symbol A_m je v podstatě jen ukazatelem na A_n, kde $m = n$.
- Na vstupu je terminální symbol $a = a$. Předpokládejme, že porovnání na prvním zásobníku proběhlo v pořádku. Pak jestliže druhý automat přijal, tak ukazatel p v aktuálně použité trojice je přepsán na 0, kde 0 značí přijatou cestu. Pokud automat nepřijal, tak je aktuální trojice odstraněna.

Příklad 4.2.1. Ukažme si nejdříve kompletní postup této metody při analýze řetězce $w = abkcdaed$ a poté si příklad rozeberme podrobněji. V tabulce 4.3 je uvedena právě kompletní analýza tohoto řetězce. Každý krok je odlišen jinou barvou, kde oranžová znamená práci prvního zásobníku a modrá značí práci druhého zásobníku. Spodní indexy u některých symbolů uvádíme pouze pro lepší orientaci. Samozřejmě nemají vliv na samotnou analýzu. Symbol A_1 je pro syntaktickou analýzou stále symbol A, stejně jako A_2. Nakonec si uveďme, že v tabulce 4.3 je uveden vždy konečný stav po aktuálním kroku. V jednom kroku totiž může automat provést více přechodů (viz. kapitola o testování 5.3).

Krok	1. ZA	2. ZA	Aktuální trojice	Množina trojic
0	$Sqabkcdaed$			(q_0, ϵ, S)
1	$A_1A_2qabkcdaed$	q_0		
2		q_0	(q_0, ϵ, S)	$(q_1, \epsilon, A_1), (q_1, \epsilon, A_2)$
3	$aA_3dA_2qabkcdaed$			
4		q_1	(q_1, ϵ, A_1)	$(q_1, A, a), (q_1, A, A_3),$ $(q_1, A, d), (q_1, \varepsilon, A_2)$
5	$A_3dA_2qbkcdaed$			
6		Aq_1	(q_1, A, a)	$(q_1, A, A_3), (q_1, A, d),$ (q_1, ε, A_2)
7	$bBcdA_2qbkcdaed$			
8		Aq_1	(q_1, A, A_3)	$(q_1, AA, b), (q_1, AA, B),$ $(q_1, AA, c), (q_1, A, d),$ (q_1, ε, A_2)
9	$BcdA_2qkcdaed$			
10		q_1AA	(q_1, AA, b)	$(q_1, AA, B), (q_1, AA, c),$ $(q_1, A, d), (q_1, \varepsilon, A_2)$
11	$kcdA_2qkcdaed$			
12		Aq_1	(q_1, AA, B)	$(q_1, AA, c), (q_1, A, d),$ $(q_1, \varepsilon, A_2), (q_1, A, k)$
13	$cdA_2qcdaed$			
14		q_1	(q_1, A, k)	$(q_1, AA, c), (q_1, A, d),$ $(q_1, \varepsilon, A_2), (q_1, A, 0)$
15	dA_2qdaed			
16		AAq_1	(q_1, AA, k)	$(q_1, A, d), (q_1, \varepsilon, A_2),$ $(q_1, A, 0)$
17	A_2qaed			
18		AAq_1	(q_1, A, d)	$(q_1, \varepsilon, A_2), (q_1, A, 0)$
19	aA_4dqaed			
20		Aq_1	(q_1, ε, A_2)	$(q_1, A, a), (q_1, A, A_4),$ $(q_1, A, d), (q_1, A, 0)$
21	A_4dqed			
22		Aq_1	(q_1, A, a)	$(q_1, A, A_4), (q_1, A, d),$ $(q_1, A, 0)$
23	$edqed$			
24		AAq_1	(q_1, A, A_4)	$(q_1, AA, e), (q_1, A, d),$ $(q_1, A, 0)$
25	dqd			
26		AAq_1	(q_1, AA, e)	$(q_1, A, d), (q_1, A, 0)$
27	q			
28		Aq_1	(q_1, A, d)	$(q_1, A, 0)$

Tabulka 4.3: Ukázka algoritmu syntaktické analýzy

Nyní si rozeberme daný příklad podrobněji. Znovu mějme řetězec $w = abkcdaed$, který může být reprezentovaný derivačním stromem uvedeným na obrázku 4.1. Dále předpokládejme existenci nějakého zásobníkového automatu, který přijímá jazyk popsaný množinou R.

- **1. krok** – Začíná pracovat pouze první automat. Na vrcholu zásobníku má symbol S a na vstupu a. V LL tabulce zjistí, jaké má použít pravidlo. Po získání pravidla odstraní ze zásobníku S a vloží na něj obrácenou pravou stranu pravidla.

1. ZA	Postup
$Sqabkcdaed$	Použije pravidlo $S \rightarrow A_1A_2$.
$A_1A_2qabkcdaed$	

- **2. krok** – Předpokládejme, že první automat již provedl expanzi symbolu S na AA. Jako vstup dostane od prvního automatu dvojici (S, AA), kde S je právě zpracovaný symbol (uzel stromu) a AA je pravá strana pravidla. V derivačním stromě představují synovské uzly pro uzel S. Druhý automat vybere trojici, která koresponduje se vstupním symbolem S a podle ní nastaví druhý automat. Na jeho vstupu je nyní symbol S, se kterým první automat provedl expanzi. Tento symbol neodmítá a tak syntaktický analyzátor může odstranit použitou trojici (q_0, ϵ, S) a zpět do množiny vložit dvě (podle $|AA|$) nové trojice (q_1, ϵ, A_1), (q_1, ϵ, A_2), kde A_1, A_2 jsou ukazatelé na symboly na prvním zásobníku. Podle těchto ukazatelů se v množině vyhledává. Až bude první automat zpracovávat symbol A_2, tak se druhý automat může vrátit k aktuálnímu nastavení zásobníku, které je uložené v dané trojici.

Vstup	2. ZA	Trojice	Množina	Postup
(S, A_1A_2)			(q_0, ε, S)	Vybere trojici.
	q_0S	(q_0, ε, S)		Nastaví automat.
	q_1	(q_0, ε, S)		Přechod $q_0S \rightarrow q_1$.
	q_1	(q_0, ε, S)	(q_1, ε, A_1), (q_1, ε, A_2)	Uloží nové trojice.

- **3. krok** – Na vrcholu prvního zásobníku je symbol A_1, který je následně podle pravidla $A \rightarrow aAd$ přepsán na $reversal(aAd)$. Analyzátor generuje zprávu pro druhý zásobník. Obsahovat bude dvojici (A_1, aA_3d).

1. ZA	Postup
$A_1A_2qabkcdaed$	Použije pravidlo $A \rightarrow aAd$
$aA_3dA_2qabkcdaed$	

- **4. krok** – Druhý automat vezme trojici z množiny (q_1, ϵ, A_1), posune se do stavu q_1 a na svůj zásobník vloží ϵ. Jelikož automat symbol A neodmítl, tak syntaktický analyzátor odstraní použitou trojici (q_1, ϵ, A_1) a zpět do množiny vloží 3 (podle $|aAd|$) nové trojice (q_1, A, a), (q_1, A, A_3) a (q_1, A, d).

Vstup	2. ZA	Trojice	Množina	Postup
(A_1, aA_3d)			(q_1, ε, A_1), (q_1, ε, A_2)	Vybere trojici.
	q_1A_1	(q_1, ε, A_1)	(q_1, ε, A_2)	Nastaví automat.
	q_1A_1	(q_1, ε, A_1)		Přechod $q_0A_1 \rightarrow Aq_1$.
	Aq_1	(q_1, ε, A_1)	(q_1, ε, a), (q_1, A, A_3),	Uloží nové trojice.
			(q_1, A, d), (q_1, A, A_2)	

- **5. krok** – Opět pracuje první automat. Zde dochází ke zpracování prvního symbolu ze vstupního řetězce. Na vrcholu prvního zásobníku je symbol a, který po úspěšném zpracování posílá na vstup druhého zásobníku.

1. ZA	Postup
$aA_3dA_2qabkcdaed$	
$A_3dA_2qbkcdaed$	Použije přechod $aqa \to q$.

- **6. krok** – Vstupním symbolem je a. Všimněme si, že zde neposíláme pravou stranu pravidla. Důvod je jednoduchý. Při zpracování listového uzlu, čímž a rozhodně je, může druhý automat pouze odmítnou, nebo přijmout cestu. V dané cestě už tedy určitě pokračovat nebude a proto nepotřebuje ukládat svůj stav. V tomto konkrétním případě automat vstup odmítá, což znamená, že cestu nenalezl. Přesto, že se nyní zabýváme sekvenčním přístupem, řekněme si něco o paralelním vylepšení této metody. Všimněme si trojice (q_1, A, d). Tato trojice zřejmě také nepotřebuje žádné další informace od prvního automatu. Pokud tedy máme k dispozici volný procesor, můžeme provést její kontrolu paralelně s kontrolou trojice (q_1, ε, a) už v tomto kroku. Více si o možnostech paralelního zpracování řekneme v 4.2.3.

Vstup	2. ZA	Trojice	Množina	Postup
a			(q_1, ε, a), (q_1, A, A_3), (q_1, A, d), (q_1, A, A_1)	Vybere trojici.
	q_1a	(q_1, ε, a)	(q_1, A, A_3), (q_1, A, d), (q_1, A, A_1)	Nastaví automat.
	q_1a	(q_1, ε, a)		Odmítá!
	q_1	(q_1, ε, a)	(q_1, A, A_3), (q_1, A, d), (q_1, A, A_1),	Neukládá nic.

Dovolme si přeskočit několik kroků, které vypadají velmi podobně jako již zde uvedené. Podrobněji se podívejme až na krok, kdy první automat zpracovává symbol B a poté k, jenž vede k nalezení cesty.

- **12. krok** – Vysvětleme si proč ukládáme pouze A jako aktuální stav zásobníku, když před tímto krokem byl stav AA. Vše je dobře vidět v tabulce pro tento krok. Je to z toho důvodu, že se na vstupu zásobníku objevil symbol B, který předem vložená A začíná mazat. Samozřejmě záleží na konfiguraci druhého automatu. Zřejmě existují i další způsoby jak daný jazyk přijímat.

Vstup	2. ZA	Trojice	Množina	Postup
(B, k)		(q_1, AA, B)	(q_1, AA, c), (q_1, A, d), (q_1, A, A_2)	Vybere trojici.
	AAq_1B	(q_1, AA, B)	(q_1, AA, c), (q_1, A, d), (q_1, A, A_2)	Nastaví automat.
	AAq_1B	(q_1, AA, B)	(q_1, AA, c), (q_1, A, d), (q_1, A, A_2)	Přechod $Aq_1B \to q_1$.
	Aq_1	(q_1, AA, B)	(q_1, A, k), (q_1, AA, c), (q_1, A, d), (q_1, A, A_2)	Ukládá jednu trojici.

- **13. krok** – Pouze přijímá symbol k a generuje zprávu pro druhý automat.

1. ZA	Postup
$kcdA_2qkcdaed$ $cdA_2qcdaed$	Použije přechod $kqk \to q$.

- **14. krok** – Po úspěšném porovnání terminálu k na prvním automatu i druhý automat přijímá. Na vstupu má k a na zásobníku je poslední A. Daný kontrolní jazyk $R = \{SA^mB^{m-1}k \mid m \geq 1\}$ naznačuje, že $|A^m| = |B^{m-1}k|$, kde $m \geq 1$. Automat tedy označí aktuální trojici hodnotou 0, což znamená potencionálně správnou cestu.

Vstup	2. ZA	Trojice	Množina	Postup
k		(q_1, A, k)	(q_1, AA, c), (q_1, A, d), (q_1, A, A_2)	Vybere trojici.
	Aq_1k	(q_1, A, k)	(q_1, AA, c), (q_1, A, d), (q_1, A, A_2)	Nastaví automat.
	Aq_1k	(q_1, AA, B)	(q_1, AA, c), (q_1, A, d), (q_1, A, A_2)	Přechod $Aq_1k \to q_1$.
	q_1	(q_1, AA, B)	(q_1, AA, c), (q_1, A, d), (q_1, A, A_2), $(q_1, \varepsilon, 0)$	Přijímá! Aktualizuje trojici.

Další kroky již probíhají analogicky jako zde zmíněné a žádný z nich nevede k nalezení další cesty. Což je patrné z tabulky popisující kompletní analýzu tohoto příkladu, kde v posledním kroku vidíme pouze jedinou trojici s označením 0. Po nalezení jedné cesty můžeme tvrdit, že daný řetězec $w = abkcdaed$ patří do $L(G)$ neboli ${}_{1-path}L(G, R)$, kde $i = j$ nebo $s = t$. Avšak do jazyka ${}_{2-path}L(G, R) = L(G)$ by tento řetězec nepatřil.

Poznámka 4.2.2. *V tabulkách jednotlivých kroků si můžeme všimnout, že automat většinou provádí více než jeden přechod. Je to tak proto, abychom byli později schopni porovnávat jednotlivé metody. Existují totiž jazyky, které je možné generovat vícero gramatikami, kde každá gramatika může použít jiný počet pravidel pro přijmutí stejného symbolu. A kdybychom počítali každý takový přechod jako krok, potom bychom byli závislí na vlastnostech gramatik a jen těžko bychom mohli porovnávat metody shora dolů s metodami přistupujícími zdola nahoru.*

4.2.3 Paralelní přístup

Dnešní možnosti počítačů jsou teoreticky neomezené. Zřejmě každý počítač je již vybaven alespoň dvěma procesory. Neobvyklé však nejsou ani počítače se čtyřmi nebo i více procesory. Nejlepší by asi bylo, kdybychom dokázali navrhnout metodu, která využije veškerých možných prostředků, které má k dispozici. Bohužel u přístupu shora dolů tohle není úplně jednoduchý úkol. Asi nejintuitivnější formou paralelizace je rozdělení vstupních dat na více procesorů, které nad nimi provedou výpočet a výsledky se poté sjednotí do celku. Kdybychom pak dokázali co nejvíce omezit komunikaci mezi procesory, tak bychom zřejmě dostali výsledek N krát rychleji, kde N je počet procesorů. Řekněme si, proč to takhle udělat nemůžeme.

Rozdělení vstupu

Mějme náš předchozí příklad a řetězec $w = abkcdaed$. Pokud se podíváme na derivační strom tohoto řetězce (4.2), tak zjistíme, že nejlepší by bylo rozdělit vstup na $w = xy$, kde $x = abkcd$ a $y = aed$. Za předpokladu, že bychom měli teoreticky neomezené množství procesorů, tak nejlepší by byla situace, kdybychom tuto operaci rozdělení použili v každém uzlu stromu. To znamená, že řetězec x opět rozdělíme v uzlu A na $x_1 = a, x_2 = bkc$ a $x_3 = d$, přičemž x_2 by se znovu dále dělil. V tomto konkrétním případě bychom takový derivační strom dokázali zkontrolovat za pouhých 5 kroků (každou úroveň v jednom kroku) a to je obrovské zlepšení oproti sekvenčním 28 krokům. Bohužel my nevíme, jak řetězec na vstupu rozdělit. Existuje pouze jediná možnost – vyzkoušet všechna možná rozdělení. U takto krátkých vstupů to zřejmě můžeme udělat, avšak představme si, že bychom zkoušeli rozdělit například stořádkový zdrojový kód v nějakém programovacím jazyce. Spočítejme si možnosti u řetězce w.

Počet synů	Počet možností
1	1
2	7
3	21
4	35
5	35
6	21
7	7
8	1

To máme 128 možností, jak rozdělit vstup v případě, že o dané gramatice nic nevíme. Znalostí gramatiky si však můžeme pomoci a počet možností zredukovat. Například víme, že existuje pouze jedno pravidlo $S \to AA$, tedy možností už je jen 7. Přesto, kdybychom chtěli pro každou možnost použít jeden procesor a zkontrolovat všechny možnosti v jednom kroku, potřebovali bychom 7 procesorů. Nyní si představme, že budeme mít pravidlo $S \to AAAA$, to bychom museli kontrolovat hned 35 možností, tzn. 35 procesorů na jeden krok. Samozřejmě můžeme těchto 35 možností zkontrolovat například v 5 krocích (potřebovali bychom pouze 7 procesorů), nicméně zcela jistě bychom pro většinu příkladů dostávali horší výsledky, než produkuje samotná sekvenční metoda.

Zredukovat počet možností můžeme i tak, že se podíváme do $First$ množiny pro daný neterminál a podle ní pak řetězec rozdělujeme. Tedy pro uzel S a jeho množinu $First = \{a, b, e\}$ nemá cenu, abychom řetězec dělili například takto $x = ab$ a $y = kcdaed$. Je totiž zřejmé, že tato možnost nepovede k úspěchu. Pokud využijeme této znalosti, tak jsou pouze tři možnosti, jak daný řetězec rozdělit.

Zřejmě existují i další metody, jak bychom mohli redukovat počet možností. Myslím si však, že režie spojená s touto „predikcí“ by byla příliš vysoká a metoda by se nám nevyplatila.

Paralelní automaty

Podívejme se proto na metodu, která bude zcela jistě a vždy produkovat lepší výsledky nežli metoda sekvenční. Každý automat bude obsluhovat jeden procesor. Potřebujeme pouze vyřešit komunikaci mezi oběma automaty. Komunikace musí být „neblokující“. Pokud bude první automat předávat informace druhému automatu, který zrovna zpracovává některý předešlý požadavek, pak nesmí čekat na převzetí dat. Nejlepší zřejmě bude použít sdílenou frontu, do které bude první automat

zapisovat a druhý automat z ní bude číst. Když si uvědomíme, že tuto frontu nemusíme zpracovat postupně, pak můžeme do kontroly cest zapojit teoreticky až m procesorů. Kde m je počet všech potenciálních cest v derivačním stromě.

Otázkou pak je, zda něčeho takového dokážeme v praxi využít. Předpokládejme náš modelový příklad. Fronta dat od prvního procesoru se plní podle toho, jakým způsobem generuje derivační strom. Postupně jsou to dvojice $(S, AA), (A, aAd), (a, \varnothing), (A, bBc)$, atd. Druhý zásobník může začít analyzovat, až jakmile bude ve frontě alespoň jedna dvojice. Řekněme, že se právě dostala do fronty dvojice (S, AA), druhý automat začne ihned s analýzou a mezitím první pokračuje dále v generování derivačního stromu. Jejich práce, ať už kontrola cesty nebo generování stromu, bude zřejmě časově velmi podobná. Kdybychom zde spustili třetí procesor, tak by jen čekal na další dvojici od prvního automatu a až by dvojici dostal, tak by naopak zase čekal druhý procesor, který zrovna dokončil analýzu dvojice (S, AA). Zkrátka, první procesor nedokáže generovat požadavky takovou rychlostí, aby je mohlo zpracovávat více procesorů.

Přesto existuje možnost, jak zapojit více jako dva procesory. Podívejme se na 4. krok v tabulce 4.3. Druhý automat právě přijal uzel A a vzhledem k pravidlu $A \rightarrow aAd$ přidal do množiny trojic tři nové – $(q_1, A, a), (q_1, A, A_3), (q_1, A, d)$. Zaměřme se na trojici (q_1, A, d). K její kontrole se druhý automat dostane až v 18. kroku. Pravda však je, že druhý automat už od prvního nic víc nepotřebuje k tomu, aby tento uzel zkontroloval ihned. Právě ke kontrole trojic, které obsahují terminální uzly můžeme využít třetí procesor. Samozřejmě se může stát, že takovou kontrolu bude třetí procesor provádět zbytečně, protože první automat k tomuto uzlu nemusí vůbec dojít (např. kvůli syntaktické chybě). Troufám si však tvrdit, že ve většině případů takovéto rozšíření povede k lepším výsledkům.

4.3 Přístup zdola nahoru

Derivační strom je generován zdola nahoru, od listů stromu po jeho kořen. Z pohledu předchozí metody, od konce cesty k jejímu počátku. Z předešlých vět je patrné, že pro kontrolu cesty nemůžeme použít stejnou gramatiku jako u metody shora dolů. Není to však takový problém, protože stačí pouze navrhnout gramatiku, která generuje jazyk $reversal(R)$. Určitá nevýhoda však vychází ze samotné podstaty metody. Uvedeme si nyní myšlenku tohoto přístupu.

4.3.1 Princip metody

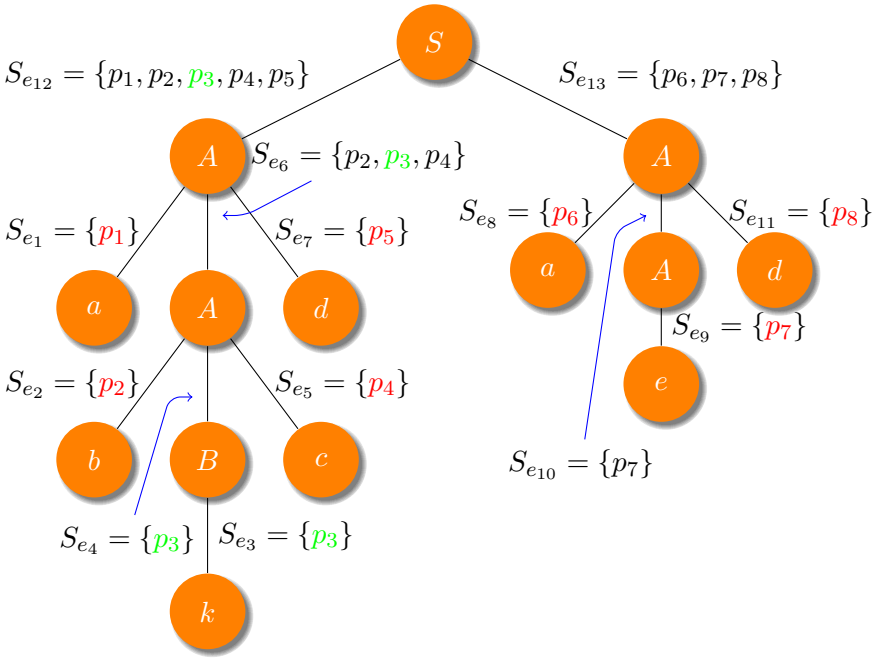
U přístupu shora dolů jsme začali kontrolovat cesty v kořeni stromu, respektive na počátku všech potenciálních cest. Je tedy zřejmé, že všechny cesty mají společný počátek. Některé cesty mohou být dokonce totožné až na listový uzel. Pokud jsme označili hranu nějakého uzlu jako nevyhovující, pak už jsme nemuseli kontrolovat žádné další derivace vycházející z tohoto uzlu. U přístupu zdola nahoru si toto dovolit nemůžeme a každou cestu musíme kontrolovat samostatně. Pro lepší demonstraci si ukažme derivační strom (obr. 4.3) řetězce $w = abkcdaed$, který generuje gramatika uvedená v 4.1.

Zjednodušeně si nyní řekněme, jak bude tato metoda pracovat. Na obrázku 4.3 vidíme derivační strom včetně vytvořených S_{e_x} množin, kde x označuje číslo hrany. Množiny S_{e_x} budeme využívat k jednoznačnému rozpoznání cest, které vedou přes danou hranu. Pokud například spojíme množiny S_{e_x}, které patří hranám vycházejícím z kořenového uzlu, tak dostaneme všechny potencionální cesty v daném derivačním stromě. Právě kořenový uzel je totiž počátkem všech potencionálních cest.

Syntaktický analyzátor bude vytvářet derivační strom s takto označenými cestami. Dostane-li se ke kontrole nějakého uzlu a tento uzel neodmítne, pak spojí množiny S_{e_x}, které patří jeho vycházejícím hranám a toto sjednocení posílá společně se svým označením ke kontrole potencionálních cest. Automat, který bude kontrolovat cesty, pak pro každé označení cesty v předané množině postupuje následovně.

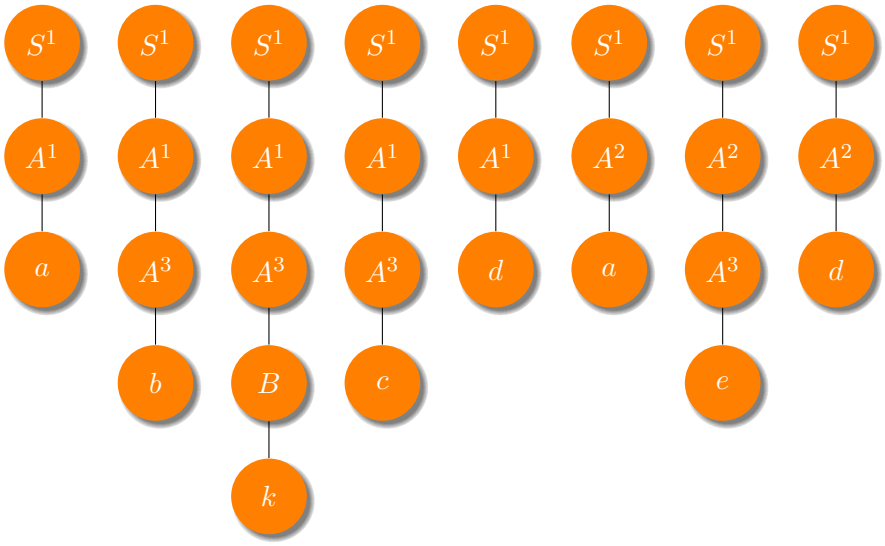
- Pokud se jedná o terminální uzel, pak jej zkontroluje. Jestliže neodmítne, tak vytváří nový stav.
- Jedná-li se o neterminální uzel, tak hledá uložený stav, který je označen danou cestou.
  - Pokud jej nenajde, pak již byla cesta odmítnuta v některém z předešlých kroků a nemá smysl kontrolovat další uzly.
  - Pokud nějaký stav najde, pak kontroluje, zda přijatý uzel patří do dané cesty (ukládá nový stav) či nikoliv (pouze zahazuje aktuálně kontrolovaný stav a dál už danou cestu nekontroluje).

Jednotlivé množiny S_{e_x} na obrázku 4.3 obsahují označení cest, které jsou graficky znázorněny černou, červenou nebo zelenou barvou. Pokud je označení červené, pak to znamená, že druhý automat cestu odmítl a právě na dané hraně končí s kontrolou této cesty. Pokud je označení zelené, pak daná hrana patří do této cesty. Pokud je označení černé, pak automat cestu nekontroluje, protože ji odmítl v některé z předešlých kontrol.



Obrázek 4.3: Derivační strom pro řetězec $w = abkcdaed$.

S tímto postupem je spojena jedna velká nevýhoda. Na obrázku 4.4 vidíme, že existuje 8 potenciálních cest v derivačním stromě z obrázku 4.3. Pokud by všechny cesty byly správné, museli bychom zkontrolovat celkem 29 uzlů, přičemž například jen kořenový uzel S bychom kontrolovali hned osmkrát. Pro připomenutí, u metody shora dolů jsme ve stejném derivačním stromě kontrolovali pouze 13 uzlů. Díky tomuto se může zdát, že daný přístup nevede k uspokojivým výsledkům. Avšak uvědomíme-li si, že většina cest nepatří do daného řídícího jazyka a s jejich analýzou končíme už v listovém uzlu (tedy na počátku), pak nám pro tento konkrétní příklad zbývá zkontrolovat 12 uzlů a to už je o 1 uzel lepší než u metody shora dolů. Uvidíme, že u sekvenční metody tento problém zřejmě nedokážeme řešit a bude záležet pouze na konkrétním derivačním stromě, zda bude metoda použitelná, či nikoliv. Avšak u paralelního přístupu jej můžeme částečně eliminovat a tato metoda bude porovnatelná s předchozí, alespoň co se rychlosti týče.



Obrázek 4.4: Potencionální cesty z modelového příkladu 4.1. Stejné písmeno a horní index značí totožný uzel.

4.3.2 Postup analýzy

Pro nějaké $n \geq 1$ mějme gramatiku $H = (G, R)$ a jazyk gramatikou H generovaný ${}_{n-path}L(G, R)$, kde $G = (N, T, P, S)$ je jednoznačná bezkontextová gramatika. Dále předpokládejme, že řídící jazyk R je generován gramatikou $G_R = (N_R, V, P_R, S_R)$, kde $V = (N \cup T)$.

Mějme nějaké $m \geq 1$, které značí počet potenciálních cest v derivačním stromě (konkrétně v našem příkladě z 4.1 je $m = 8$). Budeme konstruovat derivační strom a označovat jeho jednotlivé hrany značením z množiny $\Psi = \{p_i \mid 1 \leq i \leq m\}$.

Definice 4.3.1. Mějme hranu e, která je mezi dvěma uzly konstruovaného derivačního stromu. Množinu $S_e \subseteq \Psi$ nazýváme *množinou potenciálních cest* pro hranu e.

Definice 4.3.2. Velikost množiny S_e značíme jako $|S_e|$ a představuje počet prvků v množině S_e. Pokud je $S_e = \emptyset$, pak je $|S_e| = 0$.

Konstrukce derivačního stromu

Nyní budeme konstruovat derivační strom pro $w \in L(G)$ podle známé metody uvedené v [2]. Začínáme od terminálních symbolů, představující jednotlivé symboly z řetězce w. Tyto terminály podle pravidel z gramatiky G spojujeme do neterminálních symbolů tak, abychom zpracovali celý vstupní řetězec a zároveň získali jediný neterminální symbol, který reprezentuje startující symbol dané gramatiky.

Značení hran v derivačním stromě

Předpokládejme pravidlo $r : A \rightarrow B_1B_2 \dots B_j \in P$, pro nějaké $j \geq 1$, $B \in (N \cup T)$ a $A \in N$. Dále pokud $B_j \in N$, pak předpokládejme existenci nějakého pravidla $B_j \rightarrow C_1C_2 \dots C_k \in P$, pro nějaké $k \geq 1$, $C \in (N \cup T)$. Řekněme, že t_1 bude derivační strom reprezentující $A \Rightarrow^* \omega_1B_1B_2 \dots B_j\omega_2$, pro nějaké $\omega_1, \omega_2 \in (N \cup T)^*$. Podobně t_2 bude derivační strom reprezentující

$B_j \Rightarrow^* \omega_1 C_1 C_2 \dots C_k \omega_2$. Nyní pro každou hranu e mezi $A \to B_j$ vytvoříme množinu S_e, kterou vypočteme takto.

- Pokud B_j je **terminální symbol**, tedy listový uzel.
  Do množiny S_e přidáme p_i, kde $p_i \in \Psi$ a i volíme tak, aby bylo jednoznačným označením dané cesty. Pokud jde o první listový uzel bude $i = 1$, pokud jde o druhý listový uzel, bude $i = 2$, pokud jde o poslední listový uzel, tak bude $i = m$.

- B_j je **neterminální symbol**.
  Jelikož jde o neterminální symbol, pak předpokládáme existenci nějakého pravidla $B_j \to C_1 C_2 \dots C_k \in P$, jak už jsme si napsali výše. Také lze předpokládat, že všechny hrany mezi uzlem B_j a uzly C_k, pro nějaké $k \geq 1$, jsou již označeny. Pro přehlednost si hranu mezi uzlem B_j a každým jedním uzlem C_k označme jako c_k. Množina S_e bude sjednocením všech množin S_{c_k}. Což je možná lépe vidět na obrázku 4.3. Tedy $S_e = S_{c_1} \cup S_{c_2} \cup \dots \cup S_{c_k}$, pro nějaké $k \geq 1$.

Analýza cesty

Dříve než začneme analyzovat označený derivační strom, zaveďme si množinu, kam si budeme ukládat potencionálně správné cesty, respektive řetězce, které danou cestu reprezentují.

Definice 4.3.3. Množinu $Paths \subseteq L(R)$, kde R je řídící jazyk, nazýváme *množinou potenciálních cest pro derivační strom*.

Opět uvažujme pravidlo $r : A \to B_1 B_2 \dots B_j \in P$, pro nějaké $j \geq 1$, $B \in (N \cup T)$ a $A \in N$. Předpokládejme, že všechny hrany e_j mezi uzlem reprezentujícím neterminál A a každým dalším uzlem B_j jsou již označeny množinou S_{e_j}. Pro každou množinu S_{e_j} mezi uzlem A a uzlem B_j a pro každou její položku p_i postupujeme takto.

- Pokud je B_j **terminální symbol**, pak zkontrolujeme, zda G_R může generovat řetězec formy $word(B_j)$. (V tomto případě bude velikost $|word(B_j)| = 1$, protože se kontroluje pouze listový uzel.)

  - Jestliže daný řetězec je v G_R, pak označme $word(B_j)$ jako řetězec p_i ($p_i = word(B_j)$) a přidejme jej do množiny $Paths$. Tedy bude platit $p_i \in Paths$.
  - Jestliže daný řetězec nepatří do G_R, pak nic nepřidáváme.

- Pokud je B_j **neterminální symbol**, pak se pokusíme nalézt v množině $Paths$ řetězec označený jako p_i. Pokud $p_i \notin Paths$, pak nic nekontrolujeme a pokračujeme další položkou. Pokud $p_i \in Paths$, pak jej z množiny vybereme a vytvoříme nový řetězec, který vznikne konkatenací s uzlem B_j. Tedy $p'_i = p_i word(B_j)$. Zkontrolujeme, zda G_R může generovat řetězec p'_i.

  - Jestliže daný řetězec je v G_R, pak řekněme, že $p_i = p'_i$ a opět přidáme p_i do množiny $Paths$.
  - Jestliže daný řetězec nepatří do G_R, pak nic nepřidáváme. Uvědomme si, že jsme řetězec před konkatenací a samotnou analýzou nejdříve z množinu $Paths$ vyjmuli. Danou cestu už tedy nebudeme dále analyzovat.

Postup

- Pokud je t_1 derivační strom reprezentující $\alpha \Rightarrow^* w$, kde $\alpha \in N^+$ a pokud $Paths = \emptyset$, pak $w \notin {}_{n-path}L(G, R)$. Tedy pokud zpracoval celý vstupní řetězec (už nemůže vzniknout nová cesta) a nezůstává žádná potenciálně správná cesta, pak končí neúspěchem.
- Pokud je t_1 derivační strom reprezentující $S \Rightarrow^* w$, kde S reprezentuje startující symbol gramatiky G a pokud $|Paths| \geq n$, pak $w \in {}_{n-path}L(G, R)$.
- Pokud žádný z předešlých bodů neplatí, pak pokračujeme další derivací.

Syntaktický analyzátor

Syntaktický analyzátor používá dva zásobníkové automaty a množinu trojic (q, α, p), kde q je stav druhého automatu, α obsah druhého zásobníku a p označuje cestu, které daný stav patří.

První automat simuluje konstrukci derivačního stromu pomocí LR tabulek podle známé metody uvedené v [2]. Mějme na vstupu řetězec $w = abkcdaed$, který generuje gramatika uvedená v 4.1. Na vrcholu zásobníku je dvojice $\langle \$, q_0 \rangle$. Na vstupu je symbol a. V LR tabulce tedy hledáme dvojici $[q_0, a]$, která dává výsledek $s3$. Na vrchol zásobníku vkládáme dvojici $\langle a, 3 \rangle$. V tomto místě metoda zároveň generuje informace o postupu prvního automatu pro druhý automat, který hledá cesty v derivačním stromě. Konkrétněji jde o dvojici (a, S_e), kde a je aktuálně zpracovávaný uzel a S_e množina označení potenciálních cest.

Druhý automat dostane informaci o úspěšném zpracování symbolu a a započne svou práci. (Pokud by první automat ukončil svou práci neúspěchem, tak nemá smysl, aby druhý automat pokračoval v hledání cest.) Na vstupu má tedy symbol a a informaci o tom, které cesty přes zpracovávaný symbol procházejí. Nahlédne do množiny trojic a pro každé obdržené značení cesty $p_i \in S_e$ zjistí, zda již existuje trojice spojená právě s touto cestou (tedy $p = p_i$). Zde si můžeme všimnout hlavní nevýhody této metody. Pro jeden symbol ze vstupu, tedy pro jeden uzel stromu, může naleznout více trojic, protože S_e může obsahovat více značení. Jelikož jde v tomto konkrétním případu o terminální symbol a, pak mohou nastat následující dva případy.

- Žádnou trojici nenalezne. V takovém případě je automat uveden do startujícího stavu a začíná analýzu nové cesty.
- Nalezne trojici pro danou cestu. Z trojice získá stav a obsah zásobníku. Zásobníkový automat posune do tohoto stavu a nastaví obsah zásobníku.

Poznámka 4.3.4. *Kdyby šlo o neterminální symbol, pak pokud by nenalezl trojici, tak analýzu dané cesty končí a pokračuje další, jinak by byla reakce stejná jako u terminálního symbolu.*

Jakmile nastaví zásobníkový automat, tak může provádět potřebné kroky za účelem zpracování vstupního symbolu a. Pokud daný symbol neodmítne, tak si do množiny trojic ukládá aktualizovanou položku, případně nahraje novou, pokud začíná analyzovat novou cestu. Pokud by se stalo, že druhý automat odmítne symbol, tak je zřejmé, že tudy cesta nevede a nemusí jí dále kontrolovat. To znamená, že do množiny nic přidávat nebude. Po těchto krocích předává řízení zpět prvnímu automatu a čeká na další vstupní symbol.

Příklad 4.3.5. Ukažme si, jak by vypadalo zpracování řetězce $w = abkcdaed$ pro modelový příklad 4.1. Stejně jako u metody shora dolů i zde oranžový krok znamená, že pracuje první automat a modrý, že pracuje druhý automat.

Krok	1. ZA	2. ZA	Cesta	Množina trojic
0	$(\$,0)qabkcdaed$			
1	$(a,3)(\$,0)qbkcdaed$			
2		Oqa	p_1	
3	$(b,4)(a,3)(\$,0)qkcdaed\$$			
4		Oqb	p_2	
5	$(k,10)(b,4)(a,3)(\$,0)qcdaed\$$			
6		Oqk	p_3	(q,P,p_3)
7	$(B,8)(b,4)(a,3)(\$,0)qcdaed\$$			
8		PqB	p_3	(q,VAS,p_3)
9	$(c,12)(B,8)(b,4)(a,3)(\$,0)qdaed\$$			
10		Oqc	p_4	(q,VAS,p_3)
11	$(A,7)(a,3)(\$,0)qdaed\$$			
12		$VASqA$	p_3	(q,AS,p_3)
13	$(d,11)(A,7)(a,3)(\$,0)qaed\$$			
14		Oqd	p_5	(q,AS,p_3)
15	$(A,2)(\$,0)qaed\$$			
16		$ASqA$	p_3	(q,S,p_3)
17	$(a,3)(A,2)(\$,0)qed\$$			
18		Oqa	p_6	(q,S,p_3)
19	$(e,5)(a,3)(A,2)(\$,0)qd\$$			
20		Oqe	p_7	(q,S,p_3)
21	$(A,7)(a,3)(A,2)(\$,0)qd\$$			(q,S,p_3)
22	$(d,11)(A,7)(a,3)(A,2)(\$,0)q\$$			(q,S,p_3)
23		Oqd	p_8	(q,S,p_3)
24	$(A,6)(A,2)(\$,0)q\$$			(q,S,p_3)
25	$(S,1)(\$,0)q\$$			
26		SqS	p_3	(q,ε,p_3)
27	$(S,1)(\$,0)q\$$			

Tabulka 4.4: Ukázka algoritmu syntaktické analýzy

Nyní si projdeme daný příklad podrobněji. Znovu mějme řetězec $w = abkcdaed$, který může být reprezentovaný derivačním stromem uvedeným na obrázku 4.1. Jeho rozklad na cesty je pak uveden na obrázku 4.4. Postupujme podle výše uvedeného algoritmu krok po kroku pro gramatiku uvedenou v 4.1.1. Dále předpokládejme existenci nějakého zásobníkového automatu, který přijímá jazyk popsaný množinou R. První řádek v tabulce, nebudeme blíže popisovat, protože se jedná pouze o inicializaci prvního automatu. Druhý automat je inicializovaný až první vstupní informací.

- **1. krok** – Logicky patří prvnímu automatu, který generuje derivační strom. Na zásobníku je dvojice $(\$,0)$. Z LR tabulky získáme akci $s3$. Na zásobník vkládáme dvojici $(a,3)$. Zároveň se zde generuje zpráva pro druhý automat. Bude v ní vstupní symbol pro druhý automat (a) a označení cest, které přes symbol (respektive uzel v derivačním stromě) procházejí. V tomto konkrétním případě jde o terminální symbol nebo-li listový uzel, začínáme tedy novou cestu. A jelikož jde o první listový uzel, tak vybíráme jednoznačné označení p_1. V podstatě jsme tak pro zbytek syntaktické analýzy označili cestu vedoucí z kořene stromu do uzlu a značením p_1.

1. ZA	Postup
$(\$, 0)qabkcdaed$	Akce $s3$
$(a, 3)(\$, 0)qbkcdaed$	

- **2. krok** – Předpokládejme, že první automat přijal symbol, vygeneroval část derivačního stromu a odeslal informace. V tomto kroku druhý automat získá vstupní symbol a a označení cesty p_1. Pokouší se nalézt značení p_1 v množině trojic, avšak ten je prázdný, tedy začínáme novou cestu. Druhý automat může přijmout cesty začínající pouze symbolem k. Vstupní symbol a nepřijímá a to znamená, že pouze vrací řízení zpět prvnímu automatu.

Vstupní data	2. ZA	Cesty	Množina trojic	Postup
$(a, \{p_1\})$				Hledá trojice s cestou p_1
	Oqa	p_1		Nenalezl. Začíná novou cestu.

- **3. krok** – Řízení opět získává první automat, který bude dále generovat derivační strom. Na vstupu je symbol b. Z LR tabulky získáme $s4$. Vložíme dvojici $(b, 4)$ na vrchol prvního zásobníku a generujeme informace pro druhý zásobník. Zpráva bude obsahovat vstup b a označení cesty p_2.

1. ZA	Postup
$(a, 3)(\$, 0)qbkcdaed\$$	Akce $s4$
$(b, 4)(a, 3)(\$, 0)qkcdaed\$$	

- **4. krok** – Situace v tomto kroku je pro druhý automat velmi podobná situaci z předešlého kroku, jen s tím rozdílem, že na vstupu je symbol b místo a. Avšak ani b druhý automat nepřijímá, vrací řízení zpět prvnímu zásobníku a čeká na další vstup.

Vstupní data	2. ZA	Cesty	Množina trojic	Postup
$(b, \{p_2\})$				Hledá trojice s cestou p_2
	Oqb	p_2		Nenalezl. Začíná novou cestu.

- **5. krok** – Pro první automat se nic nemění. Opět nalezne akci v LR tabulce, která bude podobná jako v předchozích dvou krocích a uloží na zásobník vstupní symbol. Poté generuje zprávu pro druhý zásobník.

1. ZA	Postup
$(b, 4)(a, 3)(\$, 0)qkcdaed\$$	Akce $s10$
$(k, 10)(b, 4)(a, 3)(\$, 0)qcdaed\$$	

- **6. krok** – Na vstupu máme symbol k. Druhý automat tento symbol neodmítne (v tabulce jsou rozepsány kroky, vedoucí ke zpracování symbolu) a proto může uložit svůj stav pro další zpracování této cesty. Tato trojice bude (q, P, p_3). Znamená to, že až příště dostanu nějaký vstup označený cestou p_3, pak je uzel k zcela jistě jedním se synovských uzlů daného vstupu. Právě uložení aktuálního stavu nám dovoluje příště navázat tam, kde jsme nyní skončili.

Vstupní data	2. ZA	Cesty	Množina trojic	Postup
$(k, \{p_3\})$				Hledá trojice s cestou p_3.
	Oqk	p_3		Nenalezl. Začíná novou cestu.
	$kPqk$	p_3		Provádí pravidlo $O \to kP$.
	Pq	p_3	(q, P, p_3)	Přechod automatu $kqk \to q$.

- **7. krok** – První automat opět nahlédne do LR tabulky a získá akci. Tentokrát se však jedná o redukci, což znamená neterminální uzel a pokračování v některé jíž započaté cestě. Získá vyhovující pravidlo (zde $B \to k$), ze zásobníku vybere pravou stanu pravidla (k) a vloží zpět levou část pravidla a stav, který získá z `goto` tabulky. Vygeneruje zprávu pro druhý automat. Vstup je symbol B a označení cesty je p_3, protože jde o otcovský uzel předchozího k, což je zřejmé i z použitého pravidla.

1. ZA	Postup
$(k, 10)(b, 4)(a, 3)(\$, 0)qcdaed\$$	Akce $r6$.
$(B, 8)(b, 4)(a, 3)(\$, 0)qcdaed\$$	Z `goto` pro $(4, B)$ získá 8.

- **8. krok** – Na vstupu má B a cestu p_3. V množině trojic nalezne cestu, která vyhovuje označení p_3. Načteme tedy stav a obsah zásobníku a trojici z množiny odstraníme. Zkoumáme, zda z daného „startujícího“ stavu lze přijmou vstup B. Jak vidíme níže v tabulce, tak tento symbol neodmítáme. Vkládáme aktualizovanou trojici do množiny a vracíme řízení zpět prvnímu automatu.

Vstupní data	2. ZA	Cesty	Množina trojic	Postup
$(B, \{p_3\})$			(q, P, p_3)	Hledá trojice s cestou p_3.
	PqB	p_3		Odebírá jí a nastavuje 2. automat.
	$TASqB$	p_3		Provádí pravidlo $P \to TAS$.
	$BVASqB$	p_3		Provádí pravidlo $T \to BV$.
	$VASq$	p_3		Přechod automatu $BqB \to q$.
	$VASq$	p_3	(q, VAS, p_3)	Ukládá zpět do množiny aktuální stav.

Dovolme si nyní přeskočit několik kroků, které jsou velmi totožné z výše uvedenými. Zastavme se až u posledního, tedy u kroku, kdy už proběhla analýza kořene stromu prvním automatem a je předáno řízení druhému automatu, který nachází cestu.

- **26. krok** – Nyní se v derivačním stromě nacházíme nejvýše, tedy v kořenovém uzlu. V něm se z podstaty definování cest schází všechny potenciální cesty, které derivační strom obsahuje. I přesto však ve vstupních datech vidíme pouze cestu p_3. Je to proto, že právě tento příklad takto vychází. Celkově je v derivačním stromě 8 cest a pouze jediná dospěla až ke kořeni stromu (ostatní druhý automat odmítl v některém z dřívějších kroků).

Vstupní data	2. ZA	Cesty	Množina trojic	Postup
$(S, \{p_3\})$			(q, S, p_3)	Hledá trojice s cestou p_3.
	SqS	p_3		Odebírá jí a nastavuje 2. automat.
	q	p_3		Přechod automatu $BqB \rightarrow q$.
	q	p_3	(q, ε, p_3)	Ukládá zpět do množiny aktuální stav.

Vidíme, že v množině trojic nám zůstala pouze jedna jediná trojice, což samozřejmě značí, že v derivačním stromě existuje pouze jediná cesta. Po nalezení jedné cesty můžeme tvrdit, že daný řetězec $w = abkcdaed$ patří do $L(G)$ neboli ${}_1 - pathL(G, R)$, kde $i = j$ nebo $s = t$. Avšak do jazyka ${}_2 - pathL(G, R) = L(G)$ právě když $i = j$ a zároveň $s = t$, by tento řetězec nepatřil.

4.3.3 Paralelní přístup

Paralelní zásobníky

Podívejme se nyní na syntaktickou analýzu podobnou té, kterou jsme si popsali v 4.2.3. Připomeňme si problém s opakovaným kontrolováním jednoho uzlu, pokud jím prochází více jako jedna cesta a zaměřme se na jeho řešení.

Jak jsme již psali výše v tomto textu, tak tento problém můžeme řešit pouze částečně. Představme si situaci, kdy v množině trojic máme $(q, \alpha_1, p), (q, \alpha_2, p)$, kde q je stav automatu, α_i pro $i \in \{1, 2\}$ je nějaký obsah zásobníku a p je označení stejné cesty. V datech od prvního zásobníku dostaneme dvojici $(A, \{p\})$, kde A je nějaký symbol. Vidíme, že přijaté p můžeme asociovat se dvěma trojicemi a to znamená 2 sekvenční kroky (pro každou trojici jeden). Uvědomíme-li si, že v době přijetí dat už máme vše potřebné pro kontrolu obou trojic, pak lze spustit právě dvě vlákna a provést kontrolu paralelně v jednom kroku. Obecně nyní předpokládejme n vyhovujících trojic, kde $n \geq 1$. Abychom všechny mohli zkontrolovat v jednom kroku, tak bychom potřebovali $N = n$ procesorů. Budeme-li mít například 4 vyhovující trojice a pouze dva procesory, pak je dokážeme zkontrolovat za dva kroky. To znamená, že daný problém řešíme pouze částečně a jsme odkázáni na výpočetní sílu počítače.

Téměř vždy budeme mít k dispozici alespoň dva procesory. Lze tedy paralelizovat generování derivačního stromu (první procesor) a hledání cest v derivačním stromu (druhý procesor). Opět zde zavedeme sdílenou frontu, do které bude první automat pouze zapisovat a druhý z ní bude pouze číst. Oba automaty tak mohou pracovat nezávisle na sobě (samozřejmě jen v případě, že mají co zpracovávat). Pokud první automat během generování derivačního stromu narazí na syntaktickou chybu, pak končí a stejně tak končí i druhý automat. Nemá totiž smysl hledat cesty v derivačním stromě, který není kompletní. Navíc i kdybychom v takovém stromě cestu našli, tak řetězec w by stejně nemohl patřit do jazyka ${}_{n-path}L(G, R)$, protože by logicky nepatřil ani do $L(G)$ a z předešlého textu víme, že ${}_{n-path}L(G, R) \subseteq L(G)$.

Kapitola 5

Implementace a testování

Součástí této práce je navržené metody implementovat a otestovat. Popíšeme aplikaci, která je k práci přiložena, navrhneme metodu, jak jednotlivé přístupy kontrolovat a také pomocí aplikace několik syntaktických analýz spustíme tak, abychom mohli zhodnotit dosažené výsledky. Nečekejme zde však podrobný manuál pro práci s aplikací (ten lze najít na přiloženém CD, stejně jako programovou dokumentaci), řekneme si spíše jaké možnosti máme a tedy co samotná aplikace umí.

5.1 Implementace a ovládání

Program byl vyvíjen v prostředí Microsoft Visual Studia 2010 v jazyce $C\#$, jako Windows Presentation Foundation (WPF) aplikace. Samotná aplikace se nezabývá jen porovnáním metod, ale snaží se uživateli co nejdetailněji zprostředkovat postup, jakým je syntaktická analýza prováděna a to včetně vykreslování derivačního stromu. Právě k vykreslování derivačního stromu je použita knihovna `GraphViz`, která je volně dostupná online na `www.graphviz.org`. Zajímavé je také to, že aplikace není implementována paralelně, ale kvaziparalelně. Principy kvaziparalelního zpracování jsou popsány v [7]. Tato skutečnost nám dovoluje testovat paralelní zpracování i na jednom reálném procesoru. Navíc uživatel si může sám zvolit, kolika procesory chce výpočet provést.

5.1.1 Správa gramatik

Aplikace bude již při prvním spuštění obsahovat ukázkové gramatiky, které budou uvedeny i dále v textu (viz. 5.3). Uživatel má samozřejmě možnost, jak tyto gramatiky měnit, tak přidávat nové. Může jednoduše specifikovat nové terminální a neterminální symboly, startující symbol a pravidla. Takto nově vytvořená gramatika se uloží do souboru, který je při příštím otevření aplikace opět načten. Uživatel není nucen zadávat LL tabulku nebo LR tabulku, případně definici zásobníkového automatu. Vše je provedeno automaticky podle výše uvedených algoritmů (viz. 2.3) a to ihned při vytváření gramatiky.

5.1.2 Ovládání

Po vytvoření gramatik je potřeba některou z gramatik vybrat pro syntaktickou analýzu a některou z gramatik vybrat pro analýzu derivačního stromu, resp. hledání cest v derivačním stromě. Po výběru gramatik zadáme vstupní řetězec a můžeme sledovat jaké výsledky jednotlivé metody produkují. Pokud budeme chtít detailnější informace o průběhu syntaktické analýzy, pak zvolíme některou z metod. Zobrazí se nám podrobný výpis o průběhu vybrané metody, dále pak stav zásobníků a grafická podoba derivačního stromu. Navíc můžeme metody restartovat a začít krokovat od začátku

analýzy. Během tohoto krokování se nám samozřejmě bude měnit stav na zásobnících, derivační strom a výpis prováděných akcí.

5.2 Návrh testování

Při vývoji paralelních aplikací nám jde zřejmě hlavně o to, abychom na výpočet nějakého problému spotřebovali co nejméně času. Nabízí se tedy možnost porovnávat čas spotřebovaný jednotlivými metodami. Představme si však jaké časové výsledky bychom získali analýzou příkladu 4.1. Vzhledem k jeho velikosti by byly jednotlivé časy velmi malé. Mějme naopak velmi rozsáhlou gramatiku a velmi dlouhý vstup. Sice bychom získali rozumné časové vytížení, ztratili bychom však „přehlednost“ metody a zároveň i funkci, ke které je aplikace primárně určena. Těžko bychom chtěli manuálně analyzovat takto rozsáhlou gramatiku.

Mnohem lépe se jeví varianta, kdy výsledkem bude počet kroků použitých pro syntaktickou analýzu. Nyní musíme určit, co bude představovat jeden krok výpočtu metody. Předně můžeme vyloučit možnost uvažovat jeden přechod zásobníkového automatu jako jeden krok syntaktické analýzy. Stejný jazyk totiž může generovat několik ekvivalentních gramatik a zásobníkové automaty jsou zde vytvářeny právě z těchto gramatik. Může se tedy stát, že pro kontrolu stejného vstupního symbolu jeden automat provede 1 přechod a druhý automat provede 5 přechodů.

Jako jeden krok výpočtu pro první i druhý automat uvažujme kontrolu jednoho uzlu derivačního stromu. Kolik přechodů se použije k jeho kontrole nás nezajímá a nejsme tak závislí na specifikaci gramatiky. Jako jeden krok syntaktického analyzátoru uvažujme kontrolu minimálně jednoho uzlu. Našim cílem je tedy zkontrolovat co nejvíce uzlů v jednom kroku syntaktického analyzátoru. Přitom samozřejmě každý uzel odlišným procesorem.

5.3 Testování

V testech pro syntaktickou analýzu využijeme opět gramatiku, která je definována v 4.1. Budeme však měnit řídící jazyk tak, aby u jednotlivých metod vynikly jejich výhody a nevýhody.

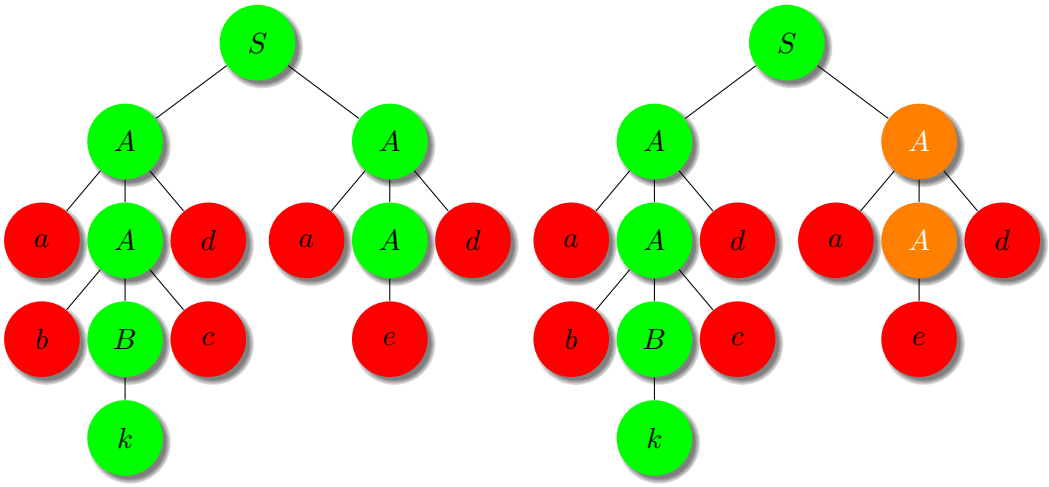
Příklad 5.3.1. Mějme řídící jazyk $R = \{SA^mB^{m-1}k \mid m \geq 1\}$ a řetězec $w = abkcdaed$. Na obrázku 5.1 vidíme výsledné derivační stromy pro metody shora dolů (vlevo) a zdola nahoru (vpravo).

	Celkem		**SA**		**Cesty**	
	Cest	Kroků	Počet uzlů	Počet kroků	Počet uzlů	Počet kroků
Shora dolů	1	28	14	14	14	14
Zdola nahoru	1	26	14	15	12	12

Tabulka 5.1: Pro 1 procesor.

Vidíme, že paralelní metody (tabulky 5.2 a 5.3) jsou v obou případech lepší než metoda sekvenční (tabulka 5.1). Při použití dvou a tří procesů se zlepšila pouze metoda shora dolů.

- Shora dolů – Větším počtem procesorů umožňujeme kontrolovat trojice, které už nepotřebují vstup, protože jde o listové uzly (viz. 4.2.3). Díky tomuto vylepšení ušetříme několik kroků ke kontrole derivačního stromu, což je ostatně vidět v tabulkách. Celkově jsme však ušetřili pouze jeden krok. Druhý automat, který analyzuje derivační strom, je v podstatě téměř vždy



Obrázek 5.1: Vlevo metoda shora dolů a vpravo metoda zdola nahoru.

pozadu oproti prvnímu automatu, který mu dodává vstupní informace. Jedinou možností, jak se dostat před první automat je, že bude kontrolovat uzly, které první automat ještě nevytvořil (neanalyzoval). Tohle si může dovolit pouze u listových uzlů, protože už nepotřebuje žádné další informace od prvního automatu. V tomto případě jsme ušetřili jeden krok právě proto, že druhý automat „předběhl“ první, zkontroloval poslední uzel d dřív, než jej první automat vytvořil a jelikož to byl poslední uzel v derivačním stromě, tak druhý automat končí a první pouze dokončí již zkontrolovaný uzel.

- Zdola nahoru – V daném derivačním stromě je pouze jedna jediná cesta správná. Abychom mohli spustit více procesorů ke kontrole derivačního stromu, tak by musely existovat alespoň dvě cesty a navíc by musely procházet stejným uzlem. To znamená, že v tomto konkrétním případě nám nebylo umožněno spustit více procesorů. Proto analýza dvěma a více procesory musí být stejná.

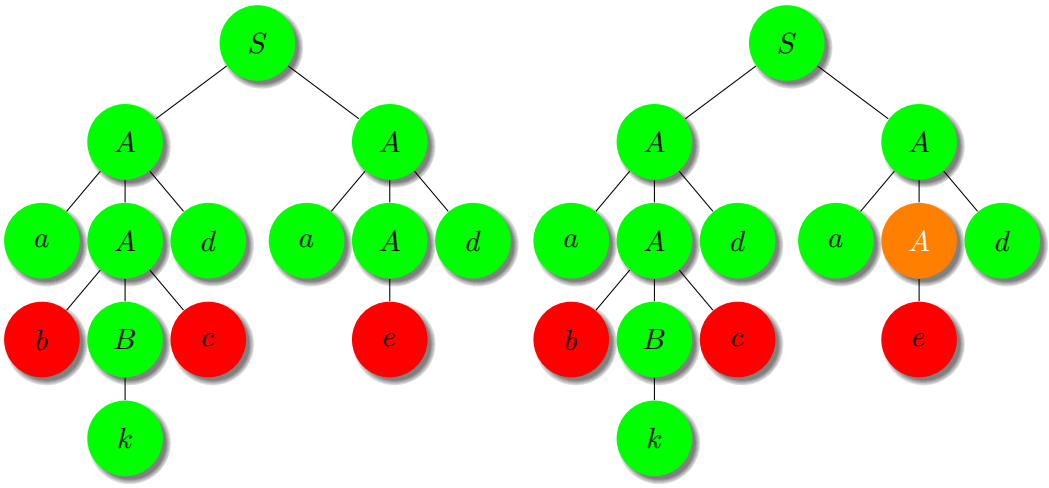
	Celkem		**SA**		**Cesty**	
	Cest	Kroků	Počet uzlů	Počet kroků	Počet uzlů	Počet kroků
Shora dolů	1	15	14	14	14	14
Zdola nahoru	1	15	14	14	12	12

Tabulka 5.2: Pro 2 procesory.

	Celkem		**SA**		**Cesty**	
	Cest	Kroků	Počet uzlů	Počet kroků	Počet uzlů	Počet kroků
Shora dolů	1	14	14	14	14	11
Zdola nahoru	1	15	14	14	12	12

Tabulka 5.3: Pro 3 procesory.

Příklad 5.3.2. Mějme řídící jazyk $R = \{SA^mB^{m-1}k \cup S\ A^ma \cup S\ A^md \mid m \geq 1\}$ a opět řetězec $w = abkcdaed$. Rozšířili jsme řídící jazyk tak, abychom přijali více jako jednu cestu.



Obrázek 5.2: Vlevo metoda shora dolů a vpravo metoda zdola nahoru.

	Celkem		SA		Cesty	
	Cest	Kroků	Počet uzlů	Počet kroků	Počet uzlů	Počet kroků
Shora dolů	5	28	14	14	14	14
Zdola nahoru	5	34	14	14	20	20

Tabulka 5.4: Pro 1 procesor.

Již u sekvenčního přístupu můžeme pozorovat změny oproti předchozímu příkladu 5.3.1. Při analýze stejného derivačního stromu obě metody nalezly pět potenciálních cest. Přístup shora dolů k výpočtu potřeboval 28 kroků, což je stejné jako u předchozího příkladu. Rozdíl je však u přístupu zdola nahoru. Ten spotřeboval 34 kroků, tedy o 7 více než u příkladu 5.3.1. Důvod jsme si popsali v 4.3.1. Zjednodušeně jde o to, že jedním uzlem prochází více cest a my jej musíme kontrolovat pro každou cestu zvlášť. V tomto případě jsme například museli pětkrát kontrolovat kořenový uzel S. Na druhou stranu, nemuseli jsme kontrolovat uzel A (na obrázku 5.2 označený oranžovou barvou), protože přes něj evidentně nevede žádná cesta. Avšak ani to nám nepomohlo a metoda zdola nahoru je u sekvenčního přístupu výrazně horší, než metoda shora dolů.

	Celkem		SA		Cesty	
	Cest	Kroků	Počet uzlů	Počet kroků	Počet uzlů	Počet kroků
Shora dolů	5	15	14	14	14	14
Zdola nahoru	5	19	14	14	20	20

Tabulka 5.5: Pro 2 procesory.

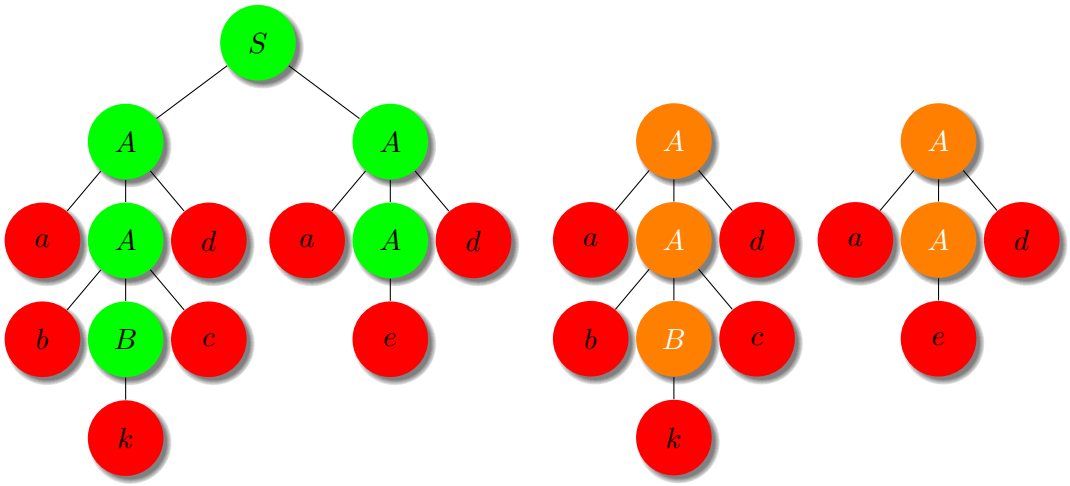
Oproti sekvenčnímu přístupu je paralelní téměř o polovinu kroků lepší. Ovšem v porovnání s paralelní verzí předchozího příkladu 5.3.1, je metoda shora dolů stejná a metoda zdola nahoru je horší. Abychom se dostali na stejná čísla, tak bychom museli použít minimálně 4 procesory (viz.

tabulka 5.6). U obou vidíme celkově 15 kroků na výpočet. U přístupu zdola nahoru nám samozřejmě zůstává nutnost zkontrolovat 20 uzlů, ale díky větší výpočetní kapacitě můžeme cesty procházející stejným uzlem kontrolovat paralelně.

	Celkem		**SA**		**Cesty**	
	Cest	Kroků	Počet uzlů	Počet kroků	Počet uzlů	Počet kroků
Shora dolů	5	14	14	14	14	11
Zdola nahoru	5	15	14	14	20	20

Tabulka 5.6: Pro 4 procesory.

Příklad 5.3.3. Upravme si nyní řídící jazyk tak, abychom v derivačním stromě nenalezli žádnou cestu, přičemž chyba bude v listových uzlech. Tedy například $R = \{SA^mB^{m-1}n \mid m \geq 1\}$. Metoda zdola nahoru by zde měla mít velkou výhodu, protože v podstatě končí na začátku každé cesty. Opět vytvořme stejný derivační strom pro řetězec $w = abkcdaed$.



Obrázek 5.3: Vlevo metoda shora dolů a vpravo metoda zdola nahoru.

Na obrázku 5.3 vidíme vlevo výsledek pro metodu shora dolů a vpravo pak nekompletní derivační strom pro přístup zdola nahoru. Podle algoritmu metody zdola nahoru (viz. 4.3), končíme analýzu, pokud selže první automat a nebo pokud je už jasné, že v derivačním stromě nebudou žádné cesty. V tomto případě nastala druhá jmenovaná možnost. Analýzou posledního listu d jsme došli k závěru, že v derivačním stromě nemůže být žádná cesta. Nemá tak cenu, aby první automat pokračoval ve vytváření derivačního stromu – metoda končí neúspěchem.

	Celkem		**SA**		**Cesty**	
	Cest	Kroků	Počet uzlů	Počet kroků	Počet uzlů	Počet kroků
Shora dolů	0	28	14	14	14	14
Zdola nahoru	0	20	12	12	8	8

Tabulka 5.7: Pro 1 procesor.

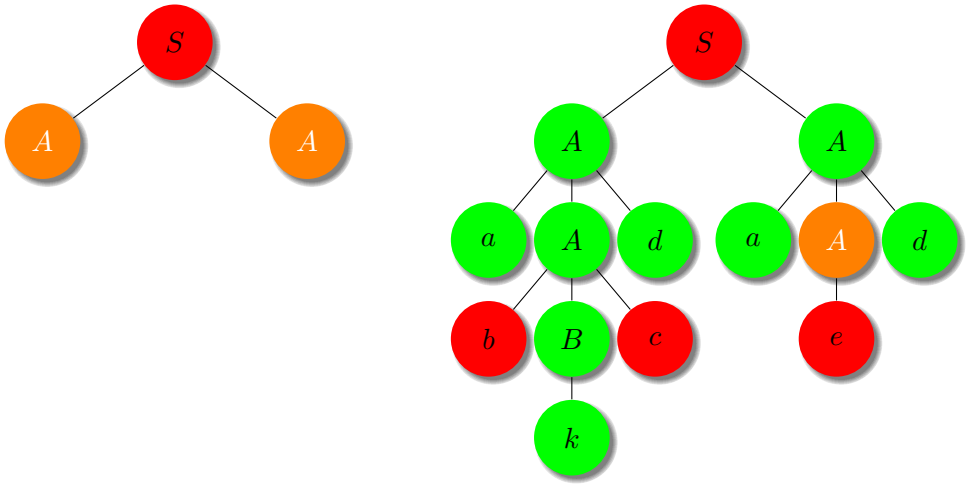
Přístup shora dolů vykazuje opět stejné výsledky jako u předchozích příkladů. Přístup zdola nahoru zde končí vždy u listových uzlů a v tabulce 5.7 vidíme, že spotřeboval relativně málo kroků. Je to sice zatím nejlepší výsledek, avšak vzhledem k výhodě, jakou v tomto případě měl, to není velmi uspokojivá hodnota.

Dále se podívejme na paralelní přístup. Ukážeme si výpočet pouze pro dva procesory (viz. tabulka 5.8). Výsledky pro více procesorů jsou velmi podobné, protože u přístupu zdola nahoru je využití více procesorů podmíněno existencí alespoň dvou cest a u přístupu shora dolů se vylepší analýza pouze o jeden krok (podobně jako je tomu v příkladu 5.3.2). Pro dva procesory se u metody shora dolů opět nic nemění, strom se stále skládá ze 14 uzlů, které musí metoda zkontrolovat. Polepšila si však analýza zdola nahoru. Ke kontrole jí stačilo pouze 13 kroků a to hlavně díky tomu, že nemusela vytvářet kompletní derivační strom, jinak by se vyrovnala přístupu shora dolů.

	Celkem		**SA**		**Cesty**	
	Cest	Kroků	Počet uzlů	Počet kroků	Počet uzlů	Počet kroků
Shora dolů	0	15	14	14	14	14
Zdola nahoru	0	13	13	13	8	8

Tabulka 5.8: Pro 2 procesory.

Příklad 5.3.4. Nakonec si upravme řídící jazyk tak, abychom v derivačním stromě nenalezli žádnou cestu, přičemž chyba bude v kořenovém uzlu. Tímto bychom měli zvýhodnit metodu shora dolů. Tedy například $R = \{CA^mB^{m-1}k \cup TA^ma \cup TA^md \mid m \geq 1\}$. Opět budeme analyzovat řetězec $w = abkcdaed$.



Obrázek 5.4: Vlevo metoda shora dolů a vpravo metoda zdola nahoru.

Na obrázku 5.4 vidíme nekompletní derivační strom metody shora dolů a vpravo pak derivační strom metody zdola nahoru. U přístupu shora dolů můžeme využít vlastnosti, že všechny cesty mají společný prefix (resp. minimálně kořenový uzel). Pokud zjistíme, že kořenový uzel nepatří do řídícího jazyka, pak můžeme s jistotou tvrdit, že v daném derivačním stromě neexistuje žádná potenciální cesta. Tento příklad jsme postavili právě na tom, že kořenový uzel nebude druhým automatem přijat. První automat tak nemusí pokračovat ve vytváření derivačního stromu, protože je jasné, že žádná další cesta neexistuje. U přístupu zdola nahoru je situace o poznání horší. Že

v derivačním stromě neexistuje žádná cesta zjistí až v posledním kroku, při analýze kořenového uzlu. Tento uzel navíc musí zkontrolovat pětkrát, protože v daném stromě existuje právě pět potenciálních cest.

	Celkem		**SA**		**Cesty**	
	Cest	Kroků	Počet uzlů	Počet kroků	Počet uzlů	Počet kroků
Shora dolů	0	2	1	1	1	1
Zdola nahoru	0	34	14	14	20	20

Tabulka 5.9: Pro 1 procesor.

Podívejme se na výslednou tabulku pro sekvenční přístup (tabulka 5.9). Zde jasně vítězí metoda shora dolů. Velmi podobně je tomu tak i v paralelní verzi (tabulka 5.10). Metoda zdola nahoru si sice polepšila téměř o polovinu, avšak metoda shora dolů stále zůstává na dvou krocích. To i přesto, že v paralelní verzi vygenerovala o jednu derivaci větší strom.

	Celkem		**SA**		**Cesty**	
	Cest	Kroků	Počet uzlů	Počet kroků	Počet uzlů	Počet kroků
Shora dolů	0	2	2	2	1	1
Zdola nahoru	0	19	14	14	20	20

Tabulka 5.10: Pro 3 procesory.

Kapitola 6

Závěr

V této práci jsme v kapitole 4 navrhli přístupy, jak provést paralelní syntaktickou analýzu, založenou na stromem řízené gramatice. Popsali jsme přístup shora dolů a přístup zdola nahoru. Zároveň jsme zde upozornili na možné nevýhody jednotlivých metod.

V kapitole 5 jsme se zabývali implementací a testováním popsaných metod. Řekli jsme si, jaké nástroje byly pro implementaci použity a jaké funkce obsahuje výsledná aplikace. Ta samozřejmě primárně slouží k porovnání obou navržených přístupů, avšak součástí je i prezentace podrobného postupu obou metod – včetně vykreslování derivačního stromu. Obsahuje také jednoduchou správu gramatik. Uživatel si tak může předem připravit několik gramatik a jejich postupným výběrem porovnávat dosažené výsledky.

Dále jsme v této kapitole provedli několik testů, které byli založeny na syntaktické analýze jednoho stejného řetězce, který generuje pro všechny testy stejná gramatika. Měnili jsme pouze řídící jazyk tak, abychom postupně zvýhodňovali jednotlivé metody. Jak se z testů ukázalo, tak ve většině případů byla rychlejší metoda shora dolů. Navíc se zdá být stabilnější. Tedy pro jeden řetězec a stejnou gramatiku produkuje velmi podobné výsledky, ať už je v derivačním stromě jakýkoliv počet potenciálních cest. Výjimkou je nulový počet cest. Zde záleží na tom, kde přesně metoda zjistí, že daný strom neobsahuje žádné cesty. Pokud by to bylo například v kořenovém uzlu, pak by metodě stačily jen dva kroky na odmítnutí řetězce. Přístup shora dolů má totiž oproti přístupu zdola nahoru velkou výhodu v tom, že se cesty postupně rozcházejí v nižších úrovních derivačního stromu. Některé cesty tak mohou být totožné třeba až po listový uzel. Každý uzel tak stačí kontrolovat maximálně jednou. Ovšem když procházíme strom zdola nahoru, tak jeden uzel kontrolujeme právě tolikrát, kolikrát přes něj prochází nějaká cesta. Tento problém jsme řešili právě paralelním přístupem, kdy jeden stejný uzel kontrolovalo hned několik procesorů – pro každou cestu jeden. Tímto se však dostáváme na výsledky, které metoda shora dolů produkuje pro paralelní přístup s dvěma procesory. Přístup zdola nahoru byl lepší pouze v případě, kdy derivační strom neobsahoval žádnou cestu a chyby byly vždy až v listových uzlech.

Pokud bychom dokázali lépe řešit problém metody zdola nahoru, pak by se zřejmě vyrovnala metodě shora dolů i bez použití více procesorů. Potom by záleželo pouze na specifikaci řídícího jazyka a konkrétním derivačním stromě. Zajímavá by také mohla být kombinace těchto dvou metod. Abychom však mohli analyzovat derivační strom zároveň shora dolů i zdola nahoru, museli bychom takto tento derivační strom zároveň i generovat.

Podařilo se nám tedy navrhnou a implementovat dvě metody paralelní syntaktické analýzy založené na stromem řízené gramatice. Tyto metody otestovat a zjistit, že z hlediska rychlosti je výhodnější použít metodu shora dolů.

Literatura

[1] Abraham, S.: Some questions of language theory. In *COLING*, Proceedings of the Conference, 1965, s. 1–11.

[2] Aho, A. V.; Sethi, R.; Ullman, J. D.: *Compilers principles, techniques, and tools*. Reading, MA: Addison-Wesley, 1986.

[3] Dassow, J.; Paun, G.: *Regulated rewriting in formal language theory*. Springer-Verlag, 1989, ISBN 978-0-387-51414-7.

[4] Čermák, M.; Koutný, J.; Meduna, A.: Parsing Based on n-Path Tree-Controlled Grammars. *Theoretical and Applied Informatics*, ročník 2011, č. 23, 2012: s. 213–228, ISSN 1896-5334. URL http://www.fit.vutbr.cz/research/view_pub.php?id=9679

[5] Marcus, S.; Martín-Vide, C.; Mitrana, V.: A new-old class of linguistically motivated regulated grammars. In *CLIN*, *Language and Computers - Studies in Practical Linguistics*, ročník 37, Computational Linguistics in the Netherlands, 2000, s. 111–125.

[6] Meduna, A.: *Automata and Languages*. London: Springer-Verlag, 2000, ISBN 1-85233-074-0.

[7] Rábová, Z.; Zendulka, J.; Češka, M.; aj.: *Modelování a simulace*. Nakladatelství VUT Brno, 1992.

[8] Rosenkrantz, D. J.: Programmed grammars and classes of formal languages. In *Journal of the ACM*, ročník 16, Association for Computing Machinery, 1969, s. 107–131.

[9] van der Walt, A. P. J.: Random context grammars. In *Proceedings of Symposium on Formal Languages*, 1970.