ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Implementace zrychlené LR analýzy
Student:	Jakub Doupal
Vedoucí:	doc. Ing. Jan Janoušek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Seznamte se s implementací Automatové knihovny, která je vyvíjena na katedře teoretické informatiky FIT ČVUT a slouží k experimentům s programátorem definovovanými algoritmy a modely výpočtů. Seznamte se s metodou zrychlené LR syntaktické analýzy [1], kdy jsou operace redukce předpočítány a dále jsou výrazně omezeny informace, které jsou při analýze ukládány na zásobník. Navrhněte vhodné datové struktury pro začlenění uvedené metody do Automatové knihovny. Váš návrh datových struktur a metodu zrychlené LR analýzy implementujte v rámci Automatové knihovny a otestujte.

Seznam odborné literatury

[1] John Aycock, R. Nigel Horspool, Jan Janousek, Borivoj Melichar: Even faster generalized LR parsing. Acta Informatica 37(9): 633-651, Springer, (2001)

L.S.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 17. února 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

Implementace zrychlené LR analýzy

Jakub Doupal

Vedoucí práce: doc. Ing. Jan Janoušek, Ph.D.

17. května 2016

Poděkování

Chtěl bych poděkovat svému vedoucímu práce, doc. Ing. Janu Janouškovi, Ph.D., za jeho ochotu při konzultaci, jeho rady a připomínky a celkové vedení práce. Také bych chtěl poděkovat Ing. Janu Trávníčkovi za jeho čas a ochotu při konzultaci prostředí automatové knihovny.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 17. května 2016 .....

Odkaz na tuto práci

Doupal, Jakub. *Implementace zrychlené LR analýzy.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato bakalářská práce se zabývá implementací zrychlené LR syntaktické analýzy do automatové knihovny vyvíjené na Katedře teoretické informatiky na Fakultě informačních technologií ČVUT v Praze. Tato knihovna umožňuje práci s automaty, stromy, jazyky a gramatikami, a experimenty s početnými algoritmy z tohoto odvětví. Na knihovně se neustále pracuje a je rozvíjena, po dokončení jejího vývoje ji bude možné používat jako výukový nástroj v předmětech BI-AAG, BI-PJP, MI-SYP aj. Zatímco LR analýza se používá pro práci s bezkontextovými LR gramatikami, zobecněnou LR analýzu lze použít i u nejednoznačných gramatik. Vzhledem k velkému množství zásobníkových operací existují požadavky analýzu urychlit, k čemuž slouží metoda zrychlené LR analýzy, která snižuje počet operací výměnou za větší paměťovou náročnost. To může být využito zobecněnou LR analýzou. Cílem práce je seznámit se s prostředím automatové knihovny a relevantními algoritmy a implementovat zrychlenou metodu LR analýzy.

Klíčová slova zrychlená LR analýza, LR analýza, automatová knihovna, MI-SYP, LR jazyk, bezkontextová gramatika

Abstract

This bachelor thesis deals with implementing faster LR parser for automata library, which is being developed at Department of Theoretical Computer Science at Faculty of Information Technology at CTU in Prague. This library offers experimentation and work with automata, tree structures, languages, and grammars and numerous algorithms concerning the aforementioned. The library is constantly being developed and after its completion, it will be ready to use as a study tool in courses such as BI-AAG, BI-PJP, or MI-SYP. While LR parser is used for work with context-free LR grammars, generalized LR parser can be used to work with ambiguous grammars as well. Because of the number of stack operations in standard LR parsing, there is demand for speeding the analysis up. This can be done by applying the faster LR analysis method, which reduces the number of stack operations in exchange for space. This can be taken advantage of by generalized LR analysis. The goal of this thesis is to study the automata library and relevant algorithms and to implement the faster LR analysis.

Keywords faster LR parsing, LR parsing, automata library, MI-SYP, LR language, context-free grammar

Obsah

Seznam obrázků

Úvod

LR analýza je efektivní metodou zpracování deterministických bezkontextových jazyků. GLR metoda, která na LR analýze staví, dokáže zanalyzovat nejednoznačné gramatiky. Kvůli vysoké závislosti na zásobníku však může být tento algoritmus velmi neefektivní. Metoda zrychlené LR analýzy se proto snaží počet zásobníkových operací minimalizovat. Pokud je daná gramatika bez nelevé rekurze, lze pro ni sestavit takový zásobníkový automat, který při každém přechodu přečte jeden symbol. Této vlastnosti je posléze využito v GLR analýze.

Cílem této práce je nastudovat, implementovat a otestovat metodu zrychlené LR analýzy do automatové knihovny. Ta je vyvíjena pod vedením Ing. Jana Trávníčka na Katedře teoretické informatiky FIT ČVUT. Knihovna již obsahuje mnoho datových struktur a algoritmů a slouží k experimentům s automaty, gramatikami, regulárními výrazy a grafy. Knihovna je napsána v jazyce C++.

Tato bakalářská práce je rozdělena do čtyř kapitol. V první kapitole jsou vymezeny pojmy, které jsou využívány v druhé kapitole, která se zabývá metodou zrychlené LR analýzy a srovnáním se standardní LR analýzou.

Třetí kapitola se zabývá návrhem algoritmů pro řešení daného problému. Ve čtvrté kapitole je popsána samotná implementace do automatové knihovny a testování.

Kapitola 1

Vymezení pojmů a terminologie

V této práci je využíváno pojmů z teorie grafů, jazyků, automatů a syntaktické analýzy, a dále z logiky a teorie množin. U čtenářů této práce je obecné povědomí v těchto odvětvích předpokládáno, některé pojmy však v této kapitole budou zadefinovány.

Graf

Orientovaný graf je dvojice $(\mathcal{V}, \mathcal{H})$, kde $\mathcal{V}$ je konečná množina uzlů grafu a $\mathcal{H} \subseteq (\mathcal{V} \times \mathcal{V})$ je konečná množina hran. Hrana je značena jako (x, y), kde x značí počáteční uzel hrany a y značí konečný uzel hrany.
Konečnou posloupnost hran $(x_0, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n)$, ve které se žádný z uzlů neopakuje, nazveme *cestou* z uzlu x_0 do uzlu x_n o délce n.

Strom

Stromem nazveme takový graf $(\mathcal{V}, \mathcal{H})$, ve kterém existuje právě jeden uzel - *kořen stromu*, do kterého nevede žádná hrana, a zároveň z kořene existuje právě jedna cesta do všech ostatních uzlů. Uzel, ze kterého nevede žádná hrana, nazveme *list*.

Jazyk

Abeceda T je konečná a neprázdná množina symbolů.
Řetězec nad abecedou T je posloupnost symbolů z této abecedy. Prázdnou posloupnost nazveme prázdný řetězec a značíme ε.
Délka řetězce je počet symbolů, ze kterého se řetězec skládá. Délku řetězce x značíme $|x|$.
Množinu všech neprázdných řetězců nad abecedou značíme T^+. Množinu všech řetězců nad abecedou značíme T^*, $T^* = T^+ \cup \{\varepsilon\}$.
Jazyk L je libovolná podmnožina T^*.
Produkt jazyků L_1 a L_2 je jazyk $L = L_1.L_2 = \{xy : x \in L_1 \wedge y \in L_2\}$.

Gramatika

Gramatika je čtveřice $G = (N, T, P, S)$, kde N je konečná množina *neterminálních symbolů*, T je konečná množina *terminálních symbolů*, $P \subset (N \cup T)^*.N.(N \cup T)^* \times (N \cup T)^*$ je množina pravidel a $S \in N$ je startovní symbol. Pravidlo (α, β) zapisujeme ve tvaru $\alpha \to \beta$.
Bezkontextová gramatika je taková gramatika, kde všechna pravidla jsou ve tvaru $A \to \beta, A \in N, \beta \in (N \cup T)^*$.
Jako *derivaci* označíme relaci $\alpha \Rightarrow \beta$, jestliže $\alpha = \gamma A \delta$, $\beta = \gamma \omega \delta$, $\alpha, \beta, \gamma, \delta, \omega \in (N \cup T)^*$, $A \in N$, $A \to \omega \in P$.
Derivaci označíme jako pravou (*rightmost*), jestliže ji lze zapsat ve tvaru $\alpha A w \Rightarrow \alpha \beta w$, $\alpha, \beta \in (N \cup T)^*$, $A \in N$, $w \in T^*$, $A \to \beta \in P$.
Derivaci označíme jako levou (*leftmost*), jestliže ji lze zapsat ve tvaru $wA\alpha \Rightarrow w\beta\alpha$, $\alpha, \beta \in (N \cup T)^*$, $A \in N$, $w \in T^*$, $A \to \beta \in P$.
Reflexivně tranzitivní uzávěr derivace $\Rightarrow$ značíme $\Rightarrow^*$, tranzitivní uzávěr značíme $\Rightarrow^+$ a k-tou mocninu značíme $\Rightarrow^k$.
Pravou derivaci značíme $\Rightarrow_{rm}$, levou derivaci značíme $\Rightarrow_{lm}$.
Jazyk L generovaný gramatikou G je množina $L(G) = \{x : x \in T^* \wedge S \Rightarrow^* x\}$.
Větná forma $\gamma \in (N \cup T)^*$ je libovolný řetězec skládající se z terminálních a neterminálních symbolů gramatiky.

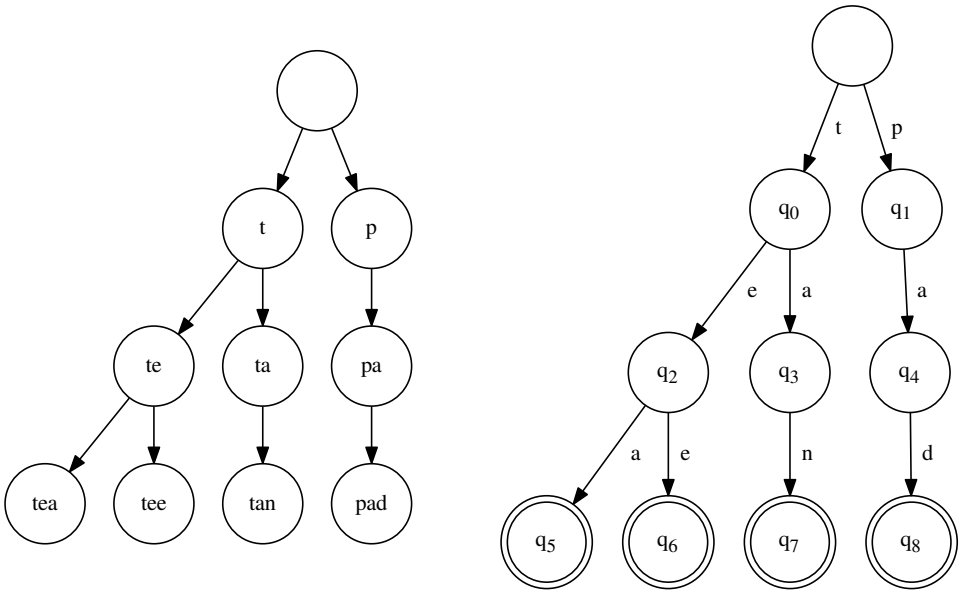
Automat

Konečný automat je pětice $A = (Q, T, \delta, q_0, F)$, kde Q je konečná množina stavů, T je vstupní abeceda, δ je přechodová funkce z $Q \times T$ do Q, $q_0 \in Q$ je počáteční stav a F je konečná množina koncových stavů.
Deterministický automat je takový automat, pro který platí: $\forall q \in Q, a \in T : |\delta(q, a)| \leq 1$. Tedy, pro každý stav dokážu na základě vstupního symbolu jednoznačně určit další akci.
Jako *konfiguraci* konečného automatu označíme dvojici (q, w), $q \in Q$, $w \in T^*$.
Jako *přechod* označíme relaci $(q, aw) \vdash (p, w)$, $q, p \in Q$, $w \in T^*$, $a \in T$, jestliže $p \in \delta(q, a)$.
Reflexivně tranzitivní uzávěr přechodu $\vdash$ značíme $\vdash^*$, tranzitivní uzávěr značíme $\vdash^+$ a k-tou mocninu značíme $\vdash^k$.
Jazyk L generovaný automatem A je množina $L(A) = \{w : w \in T^* \wedge \delta(q_0, w) \vdash^* (p, \varepsilon) \wedge p \in F\}$.

Trie

Trie je stromová datová struktura, obvykle využívaná pro uchovávání asociativního pole, kde klíči jsou obvykle řetězce. Pro naše účely lze na trie pohlížet jako na konečný automat, se startovním stavem v kořeni a koncovými stavy v listech.
Formálně lze jako automat trie zapsat pomocí následujícího vztahu: $A_{Trie} = (\mathcal{V}, T, \delta, q_{root}, F_{leaves})$, kde $\mathcal{V}$ je konečná množina uzlů v trie / stavů v au-

tomatu, T je vstupní abeceda, nad kterou je trie vystavěn, δ je přechodová funkce z $Q \times T$ do Q, q_{root} je kořen trie / počáteční stav automatu a F_{leaves} je konečná množina listů trie / koncových stavů automatu.
Pokud kořen trie označíme jako q_{root}, potom lze přechodovou funkci δ získat následovně: $\delta = \{(q, a) \times p$: existuje posloupnost $(q_{root}, q_0) \dots (q, p) \wedge$ jestliže řetězec v q je w, potom řetězec v p je $wa, a \in T, w \in T^*\}$.
Vztah trie a automatu lze snadno pochopit z obrázku níže.



Obrázek 1.1: Vztah mezi trie a automatem

Zásobníkový automat

Zásobníkový automat je sedmice $M = (Q, T, G, \delta, q_0, Z_0, F)$, kde Q je konečná množina stavů, T je vstupní abeceda, G je abeceda zásobníku, δ je přechodová funkce z $Q \times (T \cup \{\varepsilon\}) \times G^*$ do $Q \times G^*$, $q_0 \in Q$ je počáteční stav, $Z_0 \in G$ je počáteční obsah zásobníku a F je konečná množina koncových stavů.
Jako *konfiguraci* zásobníkového automatu označíme trojici (q, w, α), $q \in Q$, $w \in T^*$, $\alpha \in G^*$.
Jako *přechod* označíme relaci $(q, aw, \alpha\beta) \vdash (p, w, \gamma\beta)$, $q, p \in Q$, $w \in T^*$, $a \in T$, $\alpha, \beta, \gamma \in G^*$.
Reflexivně tranzitivní uzávěr přechodu $\vdash$ značíme $\vdash^*$, tranzitivní uzávěr značíme $\vdash^+$ a k-tou mocninu značíme $\vdash^k$.

Jazyk L generovaný zásobníkovým automatem M je množina $L(M) = \{w : w \in T^* \wedge \delta(q_0, w, Z_0) \vdash^* (p, \varepsilon, \varepsilon) \wedge p \in Q\}$.

Kapitola 2

LR analýza

V této kapitole se seznámíme s algoritmy, které jsou používány v analýze zdola nahoru (*bottom-up*), tedy LR analýze, o vlastnostech této metody a na konci také o jejích úskalích.
Dále se seznámíme s LR jazyky a gramatikami, které tyto jazyky generují.

2.1 Základní vlastnosti LR analýzy

Algoritmy využívající LR analýzy používají metodu zdola nahoru (*bottom-up*) – to znamená, že oproti LL analýze je jejich výstupem pravá derivace. Ideu nám o tomto přístupu poskytuje už samotný název LR analýzy – L zde značí čtení vstupního slova zleva (*left*), zatímco R značí formu výstupu, kterou je pravá (*right*) derivace. [2]
Gramatiky, nad kterými lze vystavět LR analýzu, se nazývají LR gramatiky.
Základní princip LR analýzy lze popsat následovně:
Nechť $G = (N, T, P, S)$ je bezkontextová gramatika a nechť $w \in T^*$, $w = a_1a_2a_3 \dots a_n$ je slovo z jazyka $L(G)$. Poté zcela jistě existuje pravá derivace $S = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_{m-1} \Rightarrow \alpha_m = w$. Vzhledem k tomu, že tato derivace je pravá, každou větnou formu $\alpha_1 \dots \alpha_{m-1}$ lze zapsat ve tvaru $\alpha_i = \gamma A a_j a_{j+1} \dots a_{m-1}$, $\gamma \in (N \cup T)^*$, $A \in N$, a řetězec $a_j \dots a_{m-1}$ je příponou řetězce w.
Za předpokladu, že $\alpha_{i-1} = \gamma Bz$ a pravidlo $B \to \beta$ je použito v derivačním kroku $\alpha_{i-1} \Rightarrow \alpha_i$ (tzn. $\gamma Bz \Rightarrow \gamma\beta z$), hlavním problémem deterministické analýzy zdola nahoru je nalezení správné větné formy βz ve větné formě α_i. Jestliže je nalezena, větná forma α_i může být redukována na větnou formu α_{i-1}. [1]

2.2 Zásobníkový automat jako LR analyzátor

Standardním modelem LR analýzy je zásobníkový automat. Pro gramatiku G existuje mnoho zásobníkových automatů M, které přijímají jazyk $L(G) = L(M)$. Při standardní konstrukci zásobníkového automatu nám vyjde automat $M = (\{q, r\}, T, N \cup T \cup \{\#\}, \delta, q, \#, r)$. Pravidla přechodu δ definujeme následovně:

Přesun (Shift):	$\delta(q, a, \varepsilon)$	$=$	$\{(q, a)\}\ \forall a \in T$	[1]
Redukce (Reduce):	$\delta(q, \varepsilon, \alpha)$	$=$	$\{(q, A) : A \to \alpha \in P\}$	
Přijetí (Accept):	$\delta(q, \varepsilon, \#S)$	$=$	$\{(\mathrm{r}, \varepsilon)\}$	

Je nutné si povšimnout, že takto sestrojený zásobníkový automat bude vždy nedeterministický. Hlavním úskalím při této konstrukci je fakt, že operace přesunu se provádí nezávisle na obsahu zásobníku. Proto je zjevné, že metodu konstrukce LR analyzátoru je třeba změnit.

2.3 Algoritmy pro tvorbu LR analyzátoru

Pro tvorbu LR analyzátoru existuje několik algoritmů. Jako rozhodný faktor při výběru vhodného algoritmu slouží klasifikace dané gramatiky. Obecně můžeme říci, že do čím užšího kruhu gramatika spadá (tzn. splňuje „náročnější“ podmínky), tím méně náročný pro ni existuje algoritmus pro sestrojení LR analyzátoru.
Vzhledem k tomu, že samotné kategorie gramatik a algoritmy pro ně relevantní nejsou hlavním předmětem této práce, tyto budou popsány do takové míry, aby o nich čtenář získal povědomí, nikoliv zcela kompletně a vždy formálně.

2.3.1 Podpůrné funkce a definice

V LL analýze jsme byli seznámeni s funkcemi FIRST_k a FOLLOW_k. Pro LR analýzu nově definujeme funkce EFF_k a BEFORE.

Nechť $G = (N, T, P, S)$ je bezkontextová gramatika, $X \in N$, $\alpha, \beta \in (N \cup T)^*$, $x \in T^*$. Funkce FIRST_k, FOLLOW_k, EFF_k a BEFORE definujeme následujícím způsobem: [2] [1]

Definice 2.1. Funkci FIRST_k definujeme následujícím způsobem:
$\mathrm{FIRST}_k(\alpha) = \{x : \alpha \Rightarrow^* x\beta \wedge |x| = k \text{ nebo } \alpha \Rightarrow^* x \wedge |x| \leq k\} \cup \{\varepsilon : \alpha \Rightarrow^* \varepsilon\}$.

Množina $\mathrm{FIRST}_k(\alpha)$ zahrnuje všechny řetězce terminálních symbolů, které lze získat derivací z větné formy α.

Definice 2.2. Funkci FOLLOW_k definujeme následujícím způsobem:
$\mathrm{FOLLOW}_k(\beta) = \{x : \beta \Rightarrow^* \alpha\beta\gamma, x \in \mathrm{FIRST}_k(\gamma)\}$.

Množina $\text{FOLLOW}_k(\beta)$ zahrnuje všechny řetězce terminálních symbolů, které se mohou nacházet bezprostředně po větné formě β.

Definice 2.3. Funkci EFF_k definujeme následujícím způsobem:
$\text{EFF}_k(\alpha) = \{w \in \text{FIRST}_k(\alpha) : \text{existuje derivace } \alpha \Rightarrow_{rm}^* \beta \Rightarrow_{rm} wx, \text{ kde } \beta \neq Awx \text{ pro žádný } A \in N\}$.

Zkratka EFF označuje *epsilon-free first*. Množina $\text{EFF}_k(\alpha)$ zahrnuje všechny řetězce terminálních symbolů z množiny $\text{FIRST}_k(\alpha)$, které nebyly získány derivací $\alpha \Rightarrow^* \beta \Rightarrow^* wx$ takovým způsobem, že první neterminální symbol z β byl přepsán prázdným řetězcem ε.

Definice 2.4. Funkci BEFORE definujeme následujícím způsobem:
$\text{BEFORE}(X) = \{Y : S \Rightarrow^* \alpha Y X \beta, Y \in (N \cup T)^*\} \cup \{\# : S \Rightarrow^* X\beta\}$.

Množina $\text{BEFORE}(X)$ zahrnuje všechny terminální nebo neterminální symboly, které se mohou nacházet bezprostředně před symbolem X v nějaké větné formě, a symbol $\#$, pokud existuje nějaká větná forma, kde X je prvním symbolem.

Definice 2.5. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika a $S \Rightarrow_{rm}^* \alpha Aw \Rightarrow_{rm} \alpha\beta w$, $A \in N$, $\alpha, \beta \in (N \cup T)^*$, $w \in T^*$ je pravá derivace v G.
Větnou formu γ nazveme *perspektivní předponou* (viable prefix), jestliže γ je předponou $\alpha\beta$.
Větnou formu γ nazveme *úplnou perspektivní předponou* (complete viable prefix), jestliže platí $\gamma = \alpha\beta$. [1]

Perspektivní předpony jsou důležitou součástí analýzy zdola nahoru. Jestliže se v nějakém momentě na vrcholu zásobníku objeví kompletní perspektivní předpona β, víme, že můžeme provést redukci podle pravidla $A \to \beta$.

Definice 2.6. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika a $S \Rightarrow_{rm}^* \alpha Aw \Rightarrow_{rm} \alpha\beta w \Rightarrow_{rm}^* xw$, $A \in N$, $\alpha, \beta \in (N \cup T)^*$, $w, x \in T^*$ je pravá derivace v G.
Větnou formu β označíme jako *redukční část* (handle) větné formy $\alpha\beta w$, která může být podle pravidla $A \to \beta \in P$ redukována na větnou formu αAw. [1]

Definice 2.7. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. Jako její *rozšířenou gramatiku* (augmented grammar) definujeme gramatiku $G' = (N', T, P', S')$, kde $N' = N \cup \{S'\}$ a $P' = P \cup \{S' \to S\}$.

Jediným rozdílem rozšířené gramatiky G' je oproti gramatice G přidání pravidla $S' \to S$ a nastavení S' jako nového startovního neterminálu. Nově přidané pravidlo označíme jako nulté a redukce užívající toto pravidlo bude signálem analyzátoru k přijetí řetězce.

2.3.2 LR(k) gramatiky

Velkou třídou gramatik jsou gramatiky, které můžeme nazvat *LR(k) gramatikami*. Jejich důležitou vlastností je, že pro ně vždy lze sestrojit deterministický LR analyzátor. Neformálně lze říci, že gramatika je LR(k), jestli pro její danou pravou derivaci $S \Rightarrow_{rm} \alpha_1 \Rightarrow_{rm} \alpha_2 \Rightarrow_{rm} \ldots \Rightarrow_{rm} \alpha_{n-1} \Rightarrow_{rm} \alpha_n = x$ dokážeme určit, že má být provedena redukce podle pravidla $A \to \beta$, a to pouze na základě řetězce α_i a k symbolů z nepřečtené části řetězce (*lookahead*). [1] [2]

Definice 2.8. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika a $G' = (N', T, P', S')$ je její rozšířená gramatika. O gramatice řekneme, že je LR(k), $k \geq 0$, pokud následující podmínky

1. $S' \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$
2. $S' \Rightarrow^*_{rm} \gamma A x \Rightarrow_{rm} \alpha\beta y$
3. $\mathrm{FIRST}_k(w) = \mathrm{FIRST}_k(y)$

implikují, že $\alpha A y = \gamma B x$ (tzn. $\alpha = \gamma$, $A = B$, $y = x$). [2]

Definice 2.9. O gramatice řekneme, že je LR, pokud je LR(k) pro nějaké $k \geq 0$. [2]

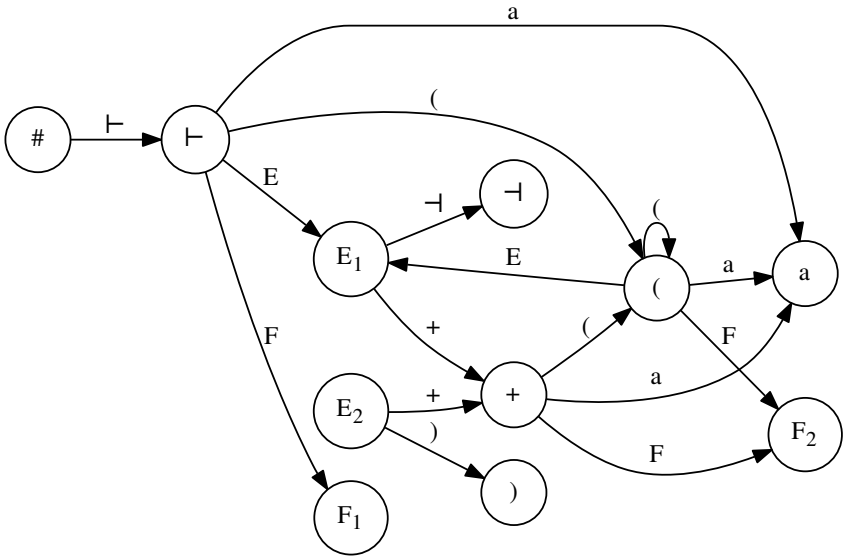
Příklad 2.1. Příklad LR gramatiky.

(0)	S'	$\to$	$\vdash E \dashv$	(3)	F	$\to$	a
(1)	E	$\to$	$E + F$	(4)	F	$\to$	(E)
(2)	E	$\to$	F				

Pro obecnou LR gramatiku použijeme pro konstrukci analyzátoru algoritmus, který je nejobecnější z celé skupiny algoritmů pro tvorbu LR analyzátoru. Pro tento algoritmus je důležité zavést nový pojem - *LR položky*, které jsou sestrojeny na základně historie analýzy pro danou gramatiku. LR položky jsou zapsány jako dvojice (P^*, w), kde P^* je upravené pravidlo zapsané ve tvaru $A \to \alpha.\beta$, $A \in P$, . je speciální znak – indikátor pozice, a $w \in \mathrm{FOLLOW}_k(A)$. Pro konstrukci analyzátoru je nezbytné sestrojení kolekci množin LR položek. Z této kolekce lze následně pomocí daných pravidel vytvořit *LR automat*, který je důležitou a nepostradatelnou částí LR analyzátoru. Algoritmus pro sestrojení kolekce množin LR položek a LR automatu následně z ní je uveden v [1].

Příklad 2.2. Kolekce množin LR(0) položek pro gramatiku 2.1

$$
\begin{array}{lcll}
\# & = & \{ & S' \to . \vdash E \dashv \} \\
\vdash & = & \{ & S' \to \vdash .E \dashv, \\
 & & & E \to .E + F, \\
 & & & E \to .F, \\
 & & & F \to .a, \\
 & & & F \to .(E) \ \} \\
E_1 & = & \{ & S' \to \vdash E. \dashv, \\
 & & & E \to E. + F \ \} \\
F_1 & = & \{ & E \to F. \ \} \\
a & = & \{ & F \to a. \ \} \\
\dashv & = & \{ & S' \to \vdash E \dashv . \ \}
\end{array}
\qquad
\begin{array}{lcll}
(& = & \{ & F \to (.E), \\
 & & & E \to .E + F, \\
 & & & E \to .F, \\
 & & & F \to .a, \\
 & & & F \to .(E) \ \} \\
+ & = & \{ & E \to E + .F, \\
 & & & F \to .(E), \\
 & & & F \to .a \ \} \\
F_2 & = & \{ & E \to E + F. \ \} \\
E_2 & = & \{ & F \to (E.), \\
 & & & E \to E. + F \ \} \\
) & = & \{ & F \to (E). \ \}
\end{array}
$$



Obrázek 2.1: LR automat pro gramatiku 2.1, sestrojený z kolekce množin LR položek 2.2 pomocí algoritmu uvedeného v [1].

Z kolekce množin LR položek lze kromě LR automatu zkonstruovat i tabulku analyzátoru, která je druhou nepostradatelnoou součástí samotného analyzátoru. Po jejím sestrojení jsme připraveni přejít k algoritmu samotné analýzy.

Algoritmus 2.1. Konstrukce tabulky analyzátoru pro LR gramatiky
Vstup: Kolekce množin LR(k) položek $\mathcal{C}$ pro gramatiku $G = (N, T, P, S)$, kde M_i jsou jednotlivé množiny.
Výstup: Tabulka analyzátoru p pro gramatiku G.

1. $p(M_i, u) =$ *přesun*, jestliže $[A \to \beta_1.\beta_2, v] \in M_i$, $\beta_2 \in T(N \cup T)^*$, $u \in \text{FIRST}_k(\beta_2 v)$.
2. $p(M_i, u) =$ *redukce(j)*, jestliže $j \geq 1$, $[A \to \beta., u] \in M_i$, $A \to \beta$ je j-té pravidlo v P.
3. $p(M_i, \varepsilon) =$ *přijetí*, jestliže $[S' \to S., \varepsilon] \in M_i$.
4. $p(M_i, u) =$ chyba ve všech ostatních případech. [1]

Příklad 2.3. Tabulka analyzátoru pro gramatiku 2.1
red j značí redukci podle j-tého pravidla

	$\vdash$	a	(	)	+	$\dashv$	ε
#	přesun						
$\vdash$		přesun	přesun				
F_1				red 2	red 2	red 2	
a				red 3	red 3	red 3	
F_2				red 1	red 1	red 1	
)				red 4	red 4	red 4	
E_1					přesun	přesun	
E_2				přesun	přesun		
+		přesun	přesun				
(		přesun	přesun				
$\dashv$							přijetí

Po sestrojení LR automatu a tabulky analyzátoru můžeme přejít k algoritmu pro analýzu LR gramatik. Vzhledem k tomu, že všechny speciální třídy gramatik, které si uvedeme v následující kapitole, jsou podtřídami LR(k) gramatik, tento algoritmus je aplikovatelný pro širokou škálu gramatik.

Algoritmus 2.2. Algoritmus analýzy pro LR gramatiky.
Vstup: LR automat a tabulka analyzátoru pro gramatiku $G = (N, T, P, S)$, vstupní řetězec $w \in T^*$ a počáteční symbol na zásobníku (v souladu s [1] značený #).
Výstup: Pravá derivace řetězce w pokud $w \in L(G)$, v ostatních případech chyba.
Metoda: Algoritmus čte symboly ze vstupního řetězce w a za pomoci zásobníku na výstupu vytváří sekvenci pravidel užitých při redukci. Algoritmus se opakuje do té doby, dokud není řetězec přijat, nebo nedojde k chybě.

V níže uvedených krocích X značí symbol na vrcholu zásobníku a u značí k nepřečtených symbolů řetězce w. Funkce *goto* je přechodovou funkcí LR automatu.

1. Přečteme vstup v tabulce analyzátoru $p(X, u)$:
   a) jestli $p(X, u) =$ *přesun*, potom přečteme jeden symbol a pokračujeme krokem 2.
   b) jestli $p(X, u) =$ *redukce(i)*, potom najdeme i-té pravidlo v P, které nechť je ve tvaru $A \to \alpha$. Z vrcholu zásobníku odstraníme $|\alpha|$ symbolů a k výstupu přidáme i-té pravidlo. Pokračujeme krokem 2.
   c) jestli $p(X, u) =$ *přijetí*, analýza byla dokončena, vstupní řetězec w je přijat a výstup je pravou derivací řetězce w.
   d) v ostatních případech $p(X, u) =$ *chyba* a analýza končí signalizací chyby.
2. Nechť Y je symbol, který má být vložen na zásobník (tzn. buď vstupní symbol přečtený v kroku 1a, nebo neterminální symbol z levé strany pravidla z kroku 1b). Potom:
   a) jestli $goto(X, Y) = Z$, vložíme Z na zásobník a pokračujeme krokem 1.
   b) jinak $goto(X, Y)$ není definováno a analýza končí signalizací chyby. [1]

Příklad 2.4. Příklad analýzy gramatiky 2.1 pomocí algoritmu 2.2

Vstupní řetězec	**Zásobník**	**Výstup**
$\vdash (a + (a)) \dashv$	$\#$	ε
$(a + (a)) \dashv$	$\# \vdash$	ε
$a + (a)) \dashv$	$\# \vdash ($	ε
$+(a)) \dashv$	$\# \vdash (a$	ε
$+(a)) \dashv$	$\# \vdash (F_1$	3
$+(a)) \dashv$	$\# \vdash (E_2$	32
$(a)) \dashv$	$\# \vdash (E_2+$	32
$a)) \dashv$	$\# \vdash (E_2 + ($	32
$)) \dashv$	$\# \vdash (E_2 + (a$	32
$)) \dashv$	$\# \vdash (E_2 + (F_1$	323
$)) \dashv$	$\# \vdash (E_2 + (E_2$	3232
$) \dashv$	$\# \vdash (E_2 + (E_2)$	3232
$) \dashv$	$\# \vdash (E_2 + F_2$	32324
$) \dashv$	$\# \vdash (E_2$	323241
$\dashv$	$\# \vdash (E_2)$	323241
$\dashv$	$\# \vdash F_1$	3232414
$\dashv$	$\# \vdash E_1$	32324142
ε	$\# \vdash E_1 \dashv$	32324142

2.3.2.1 Jednoznačnost LR gramatik

Z definice 2.8 vyplývá, že každá LR gramatika je jednoznačná (*unambiguous*). Uvažme pravou derivaci $S' \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w$ – to znamená, že v gramatice G se nachází nejvýše jedno pravidlo ve tvaru $A \to \beta$, díky čemuž může být větná forma $\alpha\beta w$ redukována na αAw. To znamená, že v LR(k) gramatikách existuje pouze jedna pravá derivace pro každý řetězec $w \in T^*$. [1]

2.3.3 LALR(k), jednoduché LR(k), LR(0) a silné gramatiky

Pokud LR(k) gramatiky navíc splňují některé z dalších podmínek, mohou být zahrnuty do užší skupiny gramatik a algoritmus pro sestrojení analyzátoru pro ně může být snazší. Zrychlená metoda LR analýzy však toto nebere v potaz a proto pro tuto práci nejsou podstatné.
Pro LALR(k) gramatiky spočívá zjednodušení konstrukce analyzátoru v relaxaci podmínky při konstrukci kolekce množin LR položek – při kontrole jejich duplicity se kontroluje pouze upravené pravidlo, nikoliv celá položka jako u LR(k) gramatik. [3] Po jejím sestrojení jsou algoritmy stejné, jako u LR(k) gramatik.
U jednoduchých LR(k) gramatik lze konstrukci kolekce množin LR položek zjednodušit ještě více, a to *de facto* vypuštěním řetězce w z LR položky (jelikož jednoduché LR gramatiky využívají LR(0) položek). Ta je potom zapsána ve tvaru $A \to \alpha.\beta$. Algoritmy po tomto kroku jsou opět stejné.
Speciálním případem LR(k) gramatik jsou LR(0) gramatiky. 0 zde značí počet nepřečtených symbolů (*lookahead*), který potřebujeme znát. Vzhledem k tomu, že nepotřebujeme znát ani jeden, tabulka analyzátoru je výrazně zjednodušena. Algoritmus analýzy je proto zjednodušen, jelikož vyhledání v tabulce analyzátoru je zjednodušeno z $p(X, u)$ na $p(X)$.
Nejužší skupinou LR gramatik, kterou uvádíme, jsou silné gramatiky. Silná LR(k) gramatika je taková gramatika, pro kterou existuje deterministický LR analyzátor, který pro rozhodování o dalším kroku využívá pouze informace o nejvýše k nepřečtených symbolů ve vstupním řetězci.
Bezkontextová gramatika $G = (N, T, P, S)$ je silnou LR(k) gramatikou, jestliže pro ni platí: jestli pro neterminální symbol X existuje dvojice neterminálních symbolů A, B tak, že A i B ovlivňují FOLLOW$_k(X)$, potom FOLLOW$_k(A) \cap$ FOLLOW$_k(B) = \emptyset$.
Toto pravidlo nám zaručí, že jsme vždy schopni se rozhodnout mezi redukcí a přesunem a v případě redukce jsme schopni se rozhodnout pro tu správnou v závislosti na nejvýše k dosud nepřečtených vstupních symbolů.[1]
Pro analýzu silných LR gramatik je dostačující tabulka přechodů (*parsing table*) $((N \cup T \cup \{\#\}) \times T)) \times (Q)$, tedy rozhodnutí o dalším kroku lze učinit pouze na základě znalosti symbolu na vrcholu zásobníku a k nepřečtených vstupních symbolů. U silných LR(0) gramatik tedy dokonce nemusíme znát vstupní symbol vůbec. [1]

2.4 Generalizovaná LR analýza

V kapitole 2.3.2 byly popsány gramatiky, které jsou LR(k), a tedy jsou jednoznačné. Navstává však otázka, co s gramatikami, které jsou nejednoznačné (*ambiguous*). Zde přichází na řadu generalizovaná LR analýza (dále GLR analýza, *generalized LR parsing*).
GLR analýza funguje na stejném principu jako LR analýza. V čem se však rozchází, je její pohled na jednoznačnost. Zatímco LR analýza nepovoluje v tabulce přechodů více stavů pro danou konfiguraci analyzátoru, GLR analýza ano. V tabulce přechodů GLR analyzátoru se tak mohou nacházet konflikty přesun–redukce (*shift–reduce conflict*) a konflikty redukce–redukce (*reduce–reduce conflict*).
V případě, že GLR analyzátor na některý z těchto konfliktů narazí, paralelně pokračuje od stavu, ve kterém konflikt nastal, pro všechny platné možnosti. GLR analyzátor takto nalezne všechny pravé derivace pro daný řetězec a danou gramatiku. [4] [5]

2.5 Úskalí zásobníkového automatu

V sekci 2.3.2 byla popsána konstrukce obecného analyzátoru pro LR(k) gramatiky. Pro nejednoznačné gramatiky byl v sekci 2.4 popsán princip GLR analýzy. Mohlo by se zdát, že algoritmus pro analýzu nejednoznačných gramatik existuje a není tak třeba nic díle řešit – opak je však pravdou.
Pro analýzu tohoto problému uvažme vysoce nejednoznačnou gramatiku - GLR analyzátor na výstup posílá oproti LR analýze všechny možné pravé derivace. Aby tohoto bylo možné dosáhnout, každý z „podprocesů“, který vznikne rozdělením svého rodičovského „procesu“, má k dispozici vlastní kopii zásobníku. Tento model lze optimalizovat - více procesů může sdílet „předponu“ zásobníku. Tato technika je nazvána grafově strukturovaný zásobník (*graph structured stack*).[6] Nicméně, pokud je zásobník měněn při téměř každém kroku, cena vytváření kopií je stále ohromná. [5]

2.6 Zrychlená LR analýza

V předchozí sekci byly ukázány překážky, které GLR analyzátoru nastavuje standardní konstrukce LR analyzátoru. Do zrychlení LR analýzy bylo investováno mnoho času a tento výzkum přirozeně přinesl výsledky – všechny z nich však jsou pro GLR analýzu použitelné pouze v omezené míře. Vzhledem ke své fundamentální vlastnosti, kterou je produkce všech možných pravých derivací pro daný řetězec, je zřejmé, že je třeba si pamatovat zásobník pro mnoho podprocesů.
GLR analyzátor využívá algoritmu pro LR analýzu, jediným rozdílem je chování při nalezení více možných přechodů. Proto je zřejmé, že zrychlením LR

analyzátoru dojde i ke zrychlení GLR analýzy. Kandidátem pro zrychlení je využití zásobníku – ve standardní metodě na něj je spoléháno ve velké míře. Pokud bychom dokázali snížit závislost na zásobníku, výrazně bychom snížili režii.
Pokud bychom se dokázali zásobníku zbavit úplně, znamenalo by to, že bychom bezkontextové jazyky dokázali analyzovat pomocí konečného automatu – což je teoreticky nemožné. Namísto toho se k této situaci snažíme co nejvíce přiblížit využíváním efektivního konečného automatu dokud to jde, než využijeme v omezené míře zásobník. [5]
Pokud bychom pro danou bezkontextovou gramatiku měli k dispozici množinu všech úplných perspektivních předpon Φ, byli bychom schopni z ní sestavit trie - tedy datovou strukturu, která by mohla být použita stejným způsobem jako LR automat při standardní LR analýze. Oproti standardnímu LR automatu má však trie podstatnou výhodu - cesta pro každou redukční část je jednoznačná a existuje pouze jedna. Proto známe nejen redukční část, kterou jsme našli, ale celou úplnou perspektivní předponu – tradiční metoda LR analýzy se v tomto ohledu spoléhá na zásobník. Náš automat sestrojený z trie však má zásadní problém - může být nekonečně velký. [1]
Při použití standardního LR analyzátoru je preferována levá rekurze – jelikož redukční část je nashromážděna na vrcholu zásobníku a je redukována hned, jak to je možné. Pro naši zrychlenou metodu je však nelevá rekurze zásadní překážkou. [5] V metodě zrychlené analýzy budeme gramatiku $G = (N, T, P, S)$ rozšiřovat pravidlem $S' \to \vdash S \dashv$ a nový počáteční stav nastavíme na S'.

Příklad 2.5. Množina úplných perspektivních předpon pro 2.1 - kvůli pravé rekurzi je nekonečná

$$\{\vdash E \dashv, \vdash E{+}F, \vdash F, \vdash E{+}a, \vdash a, \vdash (E), \vdash (F), \vdash ((E)), \vdash ((F)), \vdash (((E))), \ldots\}$$

2.6.1 Limitní body

Překážku v podobě nelevé rekurze je možné překonat úpravou vstupní gramatiky. Pokud dokážeme identifikovat neterminály, které způsobují nelevou rekurzi, můžeme je nahradit *limitními body*. Limitní bod se nachází na pravé straně pravidla z P a nahrazuje daný neterminální symbol. Limitní bod pro neterminální symbol A zapíšeme jako speciální terminální symbol $\bot_A$. [5]

Příklad 2.6. Gramatika 2.1 upravená pomocí limitního bodu

(0)	S'	$\to$	$\vdash E \dashv$	(3)	F	$\to$	a
(1)	E	$\to$	$E + F$	(4)	F	$\to$	$(\bot_E)$
(2)	E	$\to$	F				

2.6.1.1 Nalezení limitních bodů

Hlavním a nepodmíněným kritériem limitních bodů je odstranění všech nelevých rekurzí. Není však kritériem jediným – vzhledem k tomu, že při setkání s limitním bodem se uchýlíme ke spolupráci se zásobníkem, je důležité, aby množina limitních bodů, která zapříčiní eliminaci nelevých rekurzí, byla co nejméně mohutná.

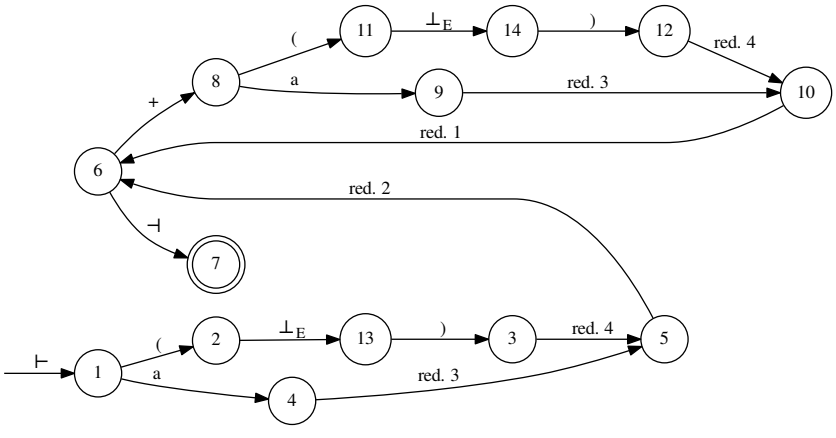
2.6.2 Sestrojení konečného automatu

Po nalezení limitních bodů a sestrojení trie úplných perspektivních předpon je z trie sestrojen konečný automat.

Algoritmus 2.3. Algoritmus pro sestrojení konečného automatu.
Vstup: Trie sestrojený z množiny úplných perspektivních předpon
Výstup: Konečný automat sestrojený z trie

1. Přidáme „redukční přechody“, které indikují redukci daným pravidlem gramatiky. Vezměme všechny perspektivní předpony $\beta\alpha$, kde α je redukční částí pravidla $A \to \alpha$. Nechť s je počáteční stav a q_0 je stav na konci cesty $\beta\alpha$, začínající v s. q_1 je stav na konci cesty βA, také začínající v s. Pokud je pravidlo $A \to \alpha$ označeno k, přidáme přechod ze stavu q_0 do stavu q_1, označený *redukce*(k).
   Jako speciální případ, stav na konci cesty $\vdash S \dashv$ nastavíme jako konečný.

2. Odstraníme přechody označené neterminálními symboly – tyto nemohou být přečteny ze vstupního řetězce. [5]



Obrázek 2.2: Konečný automat pro gramatiku 2.6

2.6.3 Sestrojení zásobníkového automatu

Konečný automat sestrojený v předchozí sekci přijímá pouze podmnožinu jazyka $L(G)$. Abychom přijímali celý jazyk $L(G)$, je třeba ke konečnému automatu přidat zásobník a udělat tak z něj zásobníkový automat. Místa, kde je třeba se spolehnout na zásobníkový automat, jsou limitní body.
V zásadě platí, že v momentu, kdy automat dosáhne přechodu $\perp_A$, nemáme dostatek informací, abychom si pamatovali dostatečný počet informací bez využití zásobníku. Jeho využitím tento nedostatek eliminujeme.

Algoritmus 2.4. Algoritmus pro sestrojení zásobníkového automatu.
Vstup: Konečný automat FA_G, sestrojený podle algoritmu 2.3
Výstup: Zásobníkový automat
Metoda: Dokud v FA_G existují $\perp$ přechody, provádíme následující kroky:

1. Vybereme přechod $\perp_A$.
2. Z G sestrojíme novou gramatiku, $G_\perp$. Na počátku do $G_\perp$ vložíme všechna pravidla z G. Poté nastavíme počáteční symbol $G_\perp$ na A a odstraníme z $G_\perp$ všechny neterminální symboly a pravidla, které jsou nedosažitelné z A. Gramatiku $G_\perp$ rozšíříme pravidlem $A' \to A$ *pop*.
3. Pro $G_\perp$ sestrojíme konečný automat metodou uvedenou v předchozí sekci – nazvěme jej $\mathrm{FA}_\perp$. Když FA_G dosáhne přechodu $\perp_A$, na zásobník vloží „návratový stav“ a přejde do počátečního stavu $\mathrm{FA}_\perp$. Jakmile $\mathrm{FA}_\perp$ dosáhne přechodu *pop*, přejde do stavu, který je odebrán z vrcholu zásobníku.
4. Řekněme, že přechod $\perp_A$ byl v FA_G učiněn ze stavu q_0 do stavu q_1. Potom z FA_G tento přechod odstraníme a nahradíme jej přechodem z q_0 do počátečního stavu $\mathrm{FA}_\perp$ a tento přechod označíme *push* q_1.
5. Spojíme $\mathrm{FA}_\perp$ s FA_G. [5]

Vzniklý automat je vidět v obrázku 2.3.

2.7 Optimalizace zrychleného analyzátoru

Pro gramatiky bez nelevé rekurze lze dokázat užitečnou vlastnost pro GLR analýzu. Po přesunu vstupního symbolu je počet redukcí omezen konstantou. Důkaz je uveden v [7]. Automat sestrojený v sekci 2.6.3 provádí přechody přesun, redukce, *push* a *pop* mezi jednotlivými stavy. Lze ukázat, že pokud gramatika neobsahuje nelevou rekurzi, každý cyklus v tomto automatu obsahuje alespoň jeden přechod přesunu. Díky této vlastnosti dokážeme automat optimalizovat tak, že při každém přechodu provede jeden přesun, může změnit vrchol zásobníku a upravit výstup. Ve srovnání s neoptimalizovanou verzí

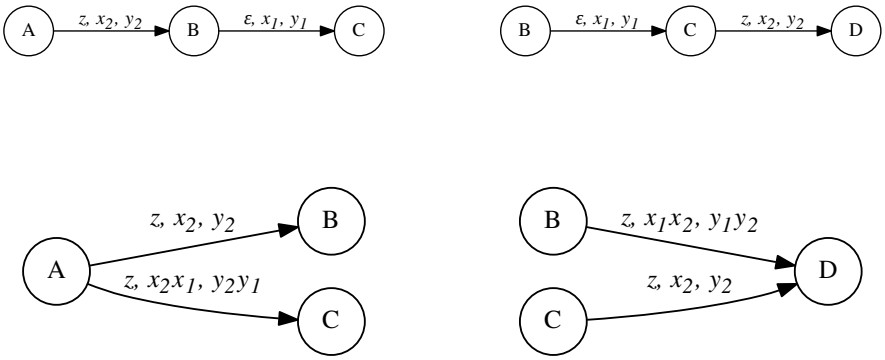
tento automat má méně stavů a provede méně přechodů – při samotném přesunu však vykoná větší porci práce. [7]
Důvod toho, že každý cyklus musí obsahovat alespoň jeden přechod přesunu, je následující: přechody *push* a *pop* skáčí z a do „podprocesu“ v místech, kde limitní body narušily rekurzi. Tato rekurze není levá, a proto platí, že se mezi dvěma skoky do stejného „podprocesu“ musí nacházet alespoň jeden přesun. Z toho vyplývá, že každý cyklus obsahující push musí obsahovat i přesun.
Také lze dokázat, že nemůže existovat cyklus sestávající pouze z *pop* přechodů (jelikož *pop* vždy následuje po redukci), ani z přechodů redukcí – jelikož počet redukcí je omezen konstantou.
Díky těmto vlastnostem můžeme optimalizovat automat P sestrojený v sekci 2.6.3. Optimalizovaný automat nazveme P'. [7]

Algoritmus 2.5. Algoritmus pro optimalizaci automatu vytvořeného pomocí algoritmu 2.4

1. Všechny přechody *pop* z P jsou nahrazeny všemi možnými přechody *pop* X.

2. P' vytvoříme z koncového stavu P a takových stavů P, ze kterých jsou provedeny akce přesun nebo *pop*.
   Poté pro každou sekvenci přechodů $l_0, l_1, l_2, \ldots l_n$ v P (na cestě z A do B), kde l_0 je přesun nebo *pop*, $l_1, \ldots, l_n$ jsou redukce nebo *push*, vytvoříme v P' přechod z A do B, kde z je vstup k přečtení (určený l_0), x je změna na zásobníku (určená $l_0 \ldots l_n$) a y je výstup (určený $l_1 \ldots l_n$).

3. V P' spojíme přechody tak, aby každý přechod četl jeden symbol. Dokud P' obsahuje stav B, z kterého vedou hrany, které nečtou žádný symbol, uplatníme jednu z níže uvedených transformací.



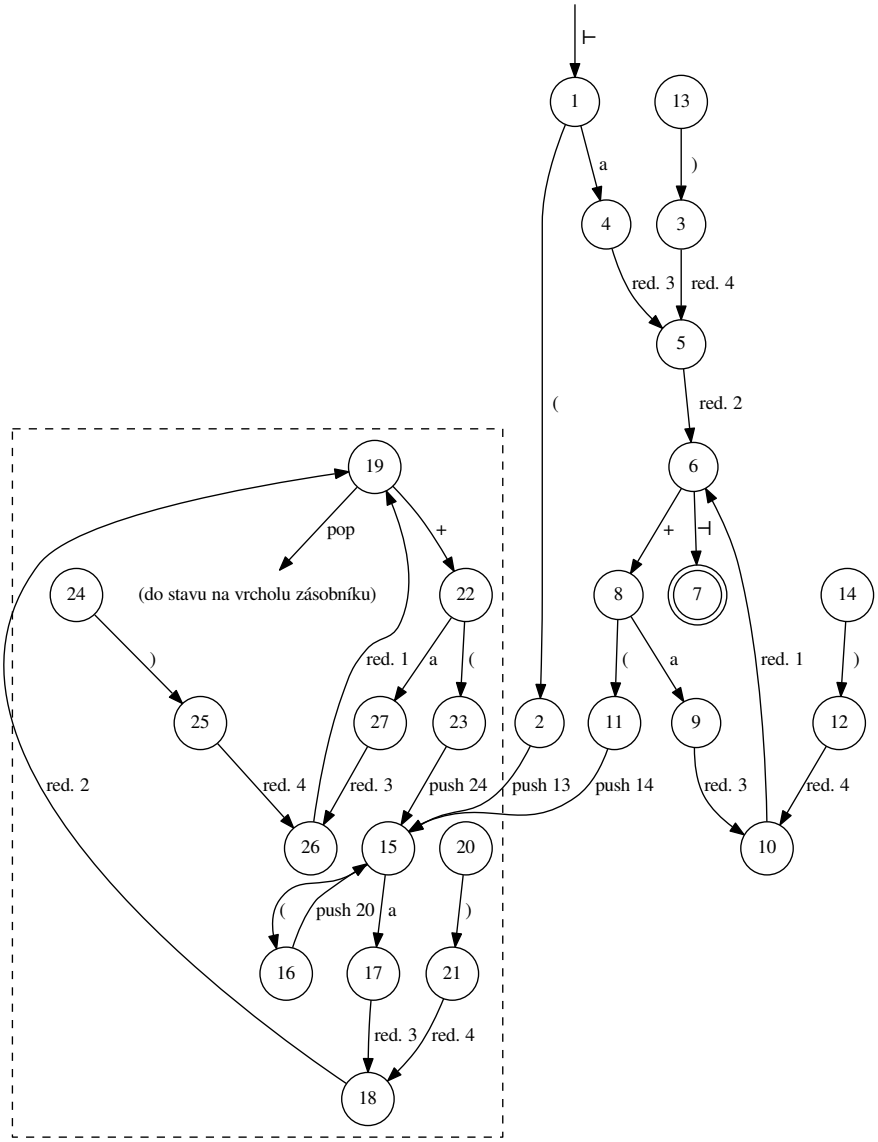
4. Odstraníme z P' neužitečné stavy a jejich hrany. [7]

Výsledný optimalizovaný automat je ukázán v obrázku 2.4. V každém kroku je nyní přečten jeden vstupní symbol. Změny na zásobníku x jsou zapsány ve tvatu $-x_1 + x_2$, kde x_1 a x_2 jsou (eventuálně prázdné) sekvence symbolů odebraných a vložených na zásobník. [7]

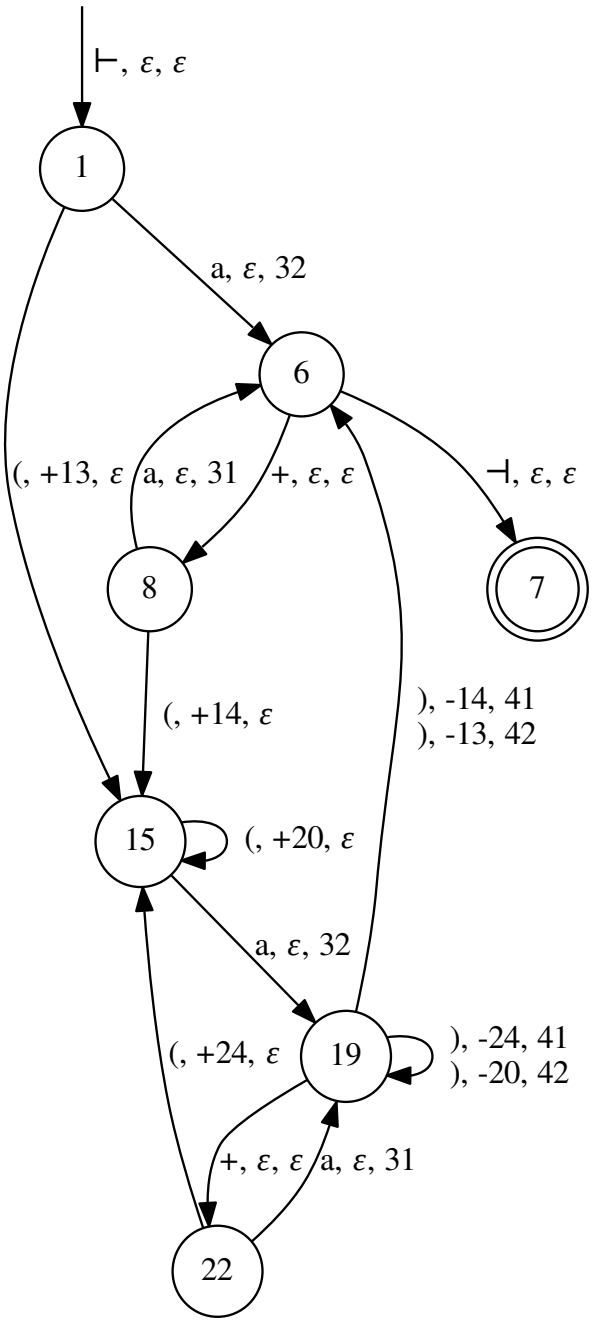
Příklad 2.7. Příklad analýzy gramatiky 2.6 pomocí optimalizovaného automatu 2.4

Vstupní řetězec	**Stav**	**Zásobník**	**Výstup**
$\vdash (a + (a)) \dashv$	0	#	ε
$(a + (a)) \dashv$	1	#	ε
$a + (a)) \dashv$	15	#13	ε
$+(a)) \dashv$	19	#13	32
$(a)) \dashv$	22	#13	32
$a)) \dashv$	15	#13, 24	32
$)) \dashv$	19	#13, 24	3232
$) \dashv$	19	#13	323241
$\dashv$	6	#	32324142
ε	7	#	32324142

Jak můžeme vidět v příkladu 2.7, výstup je stejný jako při standardní analýze 2.4 – na zásobníku je však vykonáváno výrazně méně operací.
Lze snadno ověřit, že výstup je opravdu pravou derivací vstupního řetězce.



Obrázek 2.3: Zásobníkový automat pro gramatiku 2.6, vzniklý transformací z automatu 2.2



Obrázek 2.4: Optimalizovaný zásobníkový automat 2.3

Kapitola 3

Návrh

V této kapitole bude popsáno rozdělení jednotlivých podproblémů a návrh algoritmů pro implementaci zrychleného LR analyzátoru.
Konstrukci analyzátoru můžeme rozdělit do pěti podproblémů.

1. Nalezení množiny limitních bodů, která gramatiku zbaví všech nelevých rekurzí
2. Sestrojení trie z množiny úplných perspektivních předpon
3. Transformace trie na konečný automat pomocí metody uvedené v [5]
4. Převedení konečného automatu vytvořeného v předchozím kroku na zásobníkový automat pomocí metody uvedené v [5]
5. Optimalizace vytvořeného zásobníkového automatu pomocí metody uvedené v [7]

3.1 Nalezení limitních bodů

Velmi důležitou součástí sestavení zásobníkového automatu je nalezení limitních bodů. Kvůli tomu, že pro každý neterminální symbol nahrazený limitním bodem budeme vytvářet vlastní automat („podproces“), přirozeně chceme mít limitních bodů co nejméně.
Vzhledem k tomu, že limitní body vyhledáváme kvůli eliminaci případů, kdy množina úplných perspektivních předpon je nekonečně mohutná, vybíráme limitní body tak, aby byly přerušeny všechny nelevé rekurze. Levá rekurze nám nevadí.

Příklad 3.1. Gramatika s levou rekurzí

(0)	S'	$\rightarrow$	$\vdash A \dashv$	(2)	A	$\rightarrow$	a
(1)	A	$\rightarrow$	Ba	(3)	B	$\rightarrow$	Aa

a pro ni sestrojená množina úplných perspektivních předpon

$\{\vdash A \dashv, \vdash Ba, \vdash a, \vdash Aa\}$

Příklad 3.2. Gramatika s nelevou rekurzí

(0)	S'	$\rightarrow$	$\vdash A \dashv$	(2)	A	$\rightarrow$	a
(1)	A	$\rightarrow$	aB	(3)	B	$\rightarrow$	aA

a pro ni sestrojená množina úplných perspektivních předpon

$\{\vdash A \dashv, \vdash a, \vdash aB, \vdash aaA, \vdash aaaB, \vdash aaaaA, \vdash aaaaaA, \ldots\}$

Postup, který jsem vybral, je ve zkratce následující:

Algoritmus 3.1. Algoritmus pro výběr limitních bodů.

1. Vytvoříme orientovaný graf z neterminálních symbolů gramatiky G. Uzly grafu budou neterminální symboly z gramatiky. [8]

2. Hrany značí derivaci v gramatice. Pokud v G existuje pravidlo $A \rightarrow \alpha B \beta$, na základě hodnoty α určíme typ hrany:

   a) $\alpha \neq \varepsilon$: typ hrany určíme jako *NL* - kandidát na **ne**l**e**vou rekurzi

   b) $\alpha = \varepsilon$: typ hrany určíme jako *L* - kandidát na **l**evou rekurzi

3. Jestliže pro dvojici (A, B) existuje více pravidel ve tvaru $A \rightarrow \alpha_i B \beta_i$, potom typ hrany určíme následovně:

   a) Typ hrany určíme jako *NL*, pokud alespoň v jednom z pravidel platí $\alpha_i \neq \varepsilon$.

   b) Jinak typ hrany určíme jako *L*.

4. Pomocí prohledání do hloubky (DFS) najdeme všechny cykly, které obsahují alespoň jednu hranu označenou jako *NL*.

5. Abychom vytvářeli co nejméně zásobníkových automatů, chceme co nejméně limitních bodů. Uplatníme hladový algoritmus. Po označení neterminálního symbolu A jako limitního bodu upravíme gramatiku G následujícím způsobem:

   - Všechny cykly, které obsahují neterminál A, jdou zapsat ve tvaru: $(\ldots, B_{m-1}, B_m, A, B_1, B_2, \ldots)$. To znamená, že pro neterminál B_m existuje pravidlo ve tvaru $B_m \rightarrow \alpha A \beta$. Toto pravidlo nahradíme pravidlem $B_m \rightarrow \alpha \perp_A \beta$ a cyklus označíme jako vyřešený.

6. Vyřešíme triviální případy, kde existuje hrana (A, A) a tato hrana je *NL*. V těchto případech máme pouze jednu volbu, kde umístit limitní bod – v neterminálním symbolu A.

7. Dokud v grafu stále existují nevyřešené cykly:

   a) Pokud existuje cyklus, který obsahuje neterminální symbol A, který už byl označen jako limitní bod, tento cyklus vyřešíme pomocí označení A jako limitního bodu i v něm

   b) Jinak upřednostňujeme neterminální symbol B, které se nachází v nejvíce nevyřešených cyklech. Tyto cykly potom náležitě označíme jako vyřešené pomocí limitního bodu B.

8. Po dokončení předchozí smyčky máme gramatiku G upravenou tak, že pro ni lze sestrojit konečně velký trie úplných perspektivních předpon.

Alternativní metodou je převod na *feedback arc set problem*. Feedback arc set je taková množina hran, po jejímž odstranění nebude graf obsahovat žádné cykly. Problém nalezení minimálního FAS je NP-těžký, existují však aproximační algoritmy. Je zřejmé, že pokud by všechny vrcholy byly libovolně uspořádány do sekvence $s = v_1, v_2, \ldots v_n$, potom by do feedback arc setu náležely všechny hrany $(v_j, v_i), j \geq i$. Algoritmus uvedený v [9] se snaží nalézt takové uspořádání, které nebude optimálním, bude však dobrým řešením.

Algoritmus 3.2. Algoritmus pro výběr limitních bodů pomocí feedback arc set problému.

1. Gramatika je převedena do grafové reprezentace, obdobně jako v algoritmu 3.1. [8]

2. Nechť $d^+(v)$ je výstupní stupeň uzlu a $d^-(v)$ je vstupní stupeň uzlu. Potom $\delta(v) = d^+(v) - d^-(v)$.

3. Dokud v grafu existují cykly:

   a) Dokud graf obsahuje stoky, tzn. $d^+(v) = 0$, vybereme stoku v a připojíme ji k sekvenci s_2 – $s_2 = vs_2$. v odstraníme z grafu.

   b) Dokud graf obsahuje zdroje, tzn. $d^-(v) = 0$, vybereme zdroj v a připojíme jej k sekvenci s_1 – $s_1 = s_1v$. v odstraníme z grafu.

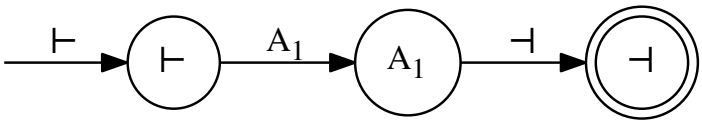
   c) Vybereme vrchol, pro který je $\delta(v)$ maximální, a připojíme jej k sekvenci s_1 – $s_1 = s_1v$. v odstraníme z grafu. [9]

3.2 Sestavení trie a konečného automatu

V předchozí kapitole jsme nalezli množinu limitních bodů a upravili gramatiku tak, že neobsahuje nelevou rekurzi a lze tedy pro ni sestrojit trie úplných perspektivních předpon. Tento trie posléze převedeme na konečný automat, pomocí pravidel uvedených v [5]. Pro sestrojení konečného automatu jsem vybral následující postup, který slučuje konstrukci trie a konečného automatu (tzn. přeskočíme fázi trie).

Algoritmus 3.3. Algoritmus pro konstrukci konečného automatu z gramatiky rozšířené o limitní body.

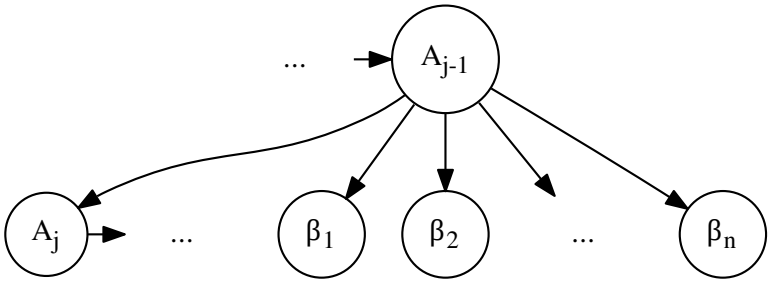
1. Jestliže nulté pravidlo je ve tvaru $S' \to \vdash A_1 \dashv$, potom přirozeně první úplnou perspektivní předponou je $\vdash A_1 \dashv$. Počáteční vzhled automatu je tak popsán grafem níže.



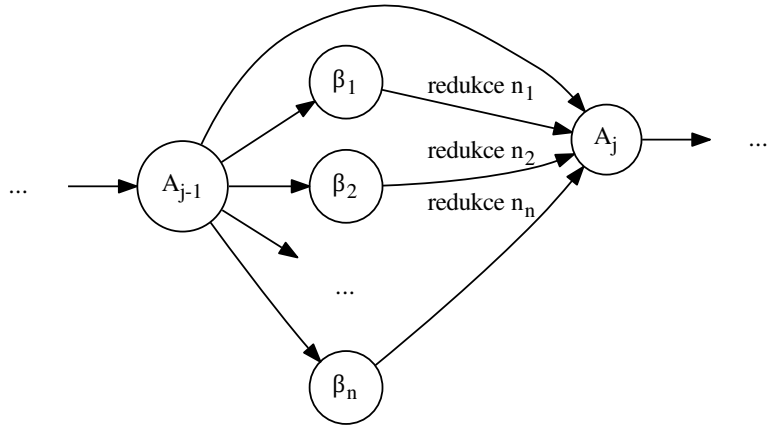
2. Dokud se v automatu nachází nové „listy“ oproti předchozímu kroku:

   a) Jestliže byl tento „list“ vygenerován pravidlem $A_i \to \beta$, jdu zpětně po hranách maximálně o $|\beta|$ kroků, dokud nenarazím na první neterminální symbol.

   b) Jakmile narazím na neterminální symbol A_j, potom pro všechna pravidla ve tvaru $A_j \to \beta_k$ jdu zpětně po hraně do stavu, ze kterého vede hrana do A_j. Z tohoto stavu poté přidám cesty pro všechny β_k.



c) Pro „list“ každé větné formy β_k přidám hranu *redukce n* vedoucí do A_j, pokud $A_j \to \beta_k$ je n-té pravidlo.



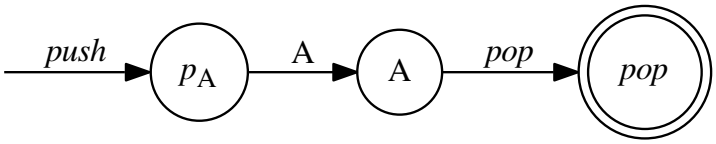
3. Z automatu odstraním všechny přechody označené neterminálními symboly, protože tyto se nemůžou nacházet ve vstupním řetězci.

3.3 Konstrukce zásobníkového automatu

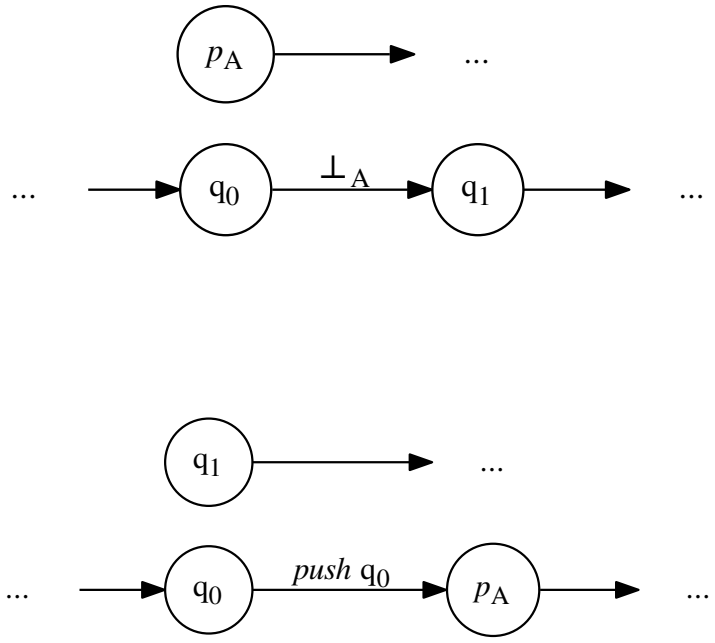
Po sestrojení konečného automatu pro gramatiku G zdaleka nejsme hotovi - náš zásadní problém spočívá v přechodech do limitních bodů a samotnou jejich přítomností. Proto dalším krokem je transformace konečného automatu na zásobníkový. Samozřejmě, jako první bude třeba sestrojit konečné automaty - „podprocesy“ pro jednotlivé limitní body. Abychom tohoto dosáhli, uplatníme následující algoritmus:

Algoritmus 3.4. Algoritmus pro transformaci konečného automatu vytvořeného algoritmem 3.3 na zásobníkový.

1. Pro každý z limitních bodů sestrojíme konečný automat pomocí algoritmu 3.3 s mírnou úpravou: pro limitní bod $\perp_A$ přidáme pravidlo $A' \to p_A\ A\ pop$ a neterminální symbol A' nastavíme jako nový počáteční symbol. Počáteční vzhled automatu poté bude mírně upraven. Limitní body znovu nehledáme pro danou gramatiku upravenou pro limitní bod.



2. Konečné automaty postupně napojíme na počáteční automat následujícím způsobem: jestliže v automatu mám přechody označené $\perp_A$ vedoucí ze stavu q_0 do stavu q_1, potom tento přechod odstraním a přidám přechod vedoucí ze stavu q_0 do počátečního stavu automatu A, p_A, označený jako *push* q_1. Jelikož limitní body byly vybrány při rekurzi, je zaručeno, že se postupně dostaneme ke všem.



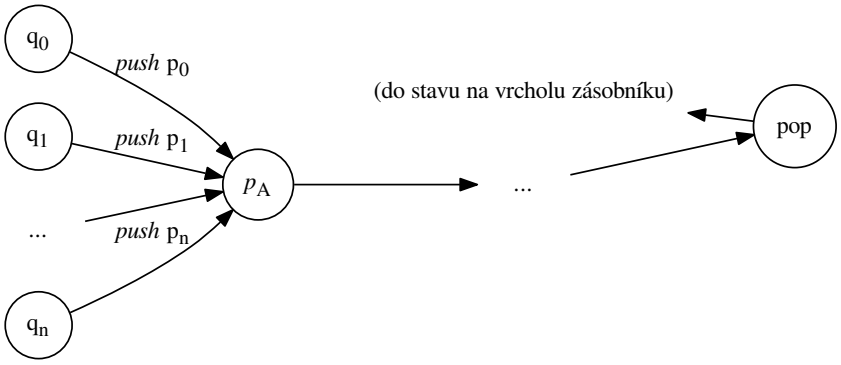
3.4 Optimalizace zásobníkového automatu

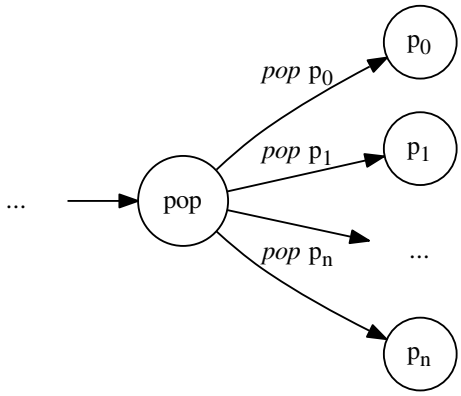
Postup optimalizace zásobníkového automatu vzniklého v předchozí sekci je jasně dán [7]. Při každém přesunu se přečte jeden symbol ze vstupního řetězce

a může se změnit obsah zásobníku a/nebo výstup. P značí původní automat, P' značí optimalizovaný automat.

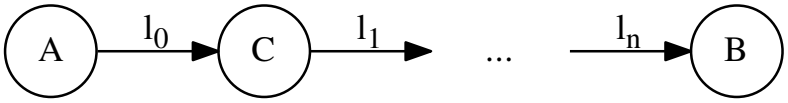
Algoritmus 3.5. Algoritmus pro optimalizaci zásobníkového automatu vytvořeného metodou 3.4.

1. Všechny přechody v P označené *pop* nahradíme přechody do všech možných stavů, které se mohou při tomto přechodu nacházet na vrcholu zásobníku – jinými slovy, pro „podproces“ limitního bodu $\perp_A$ povedou tyto přechody do stavů, které jsou vkládány na zásobník při přechodu do počátečního stavu tohoto „podprocesu“.

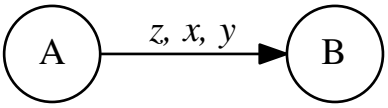




2. Vytvoříme P' z takových stavů P, ze kterých vedou přechody přesun nebo *pop*, a z konečného stavu P. Poté pro každou sekvenci přechodů

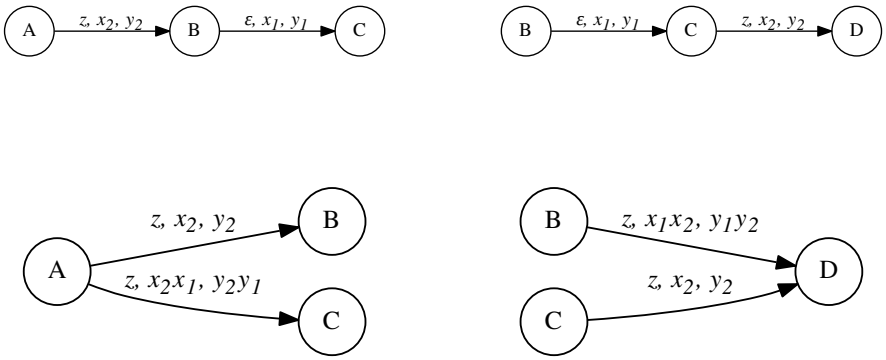


v P, kde l_0 je přesun nebo *pop* a $l_1 \ldots l_n$ jsou redukce nebo *push*, vytvoříme v P' přechod



kde z je vstupní symbol, x je změna na zásobníku ve tvaru $-x_1 + x_2$, kde x_1 a x_2 jsou sekvence odebíraných a respektive přidaných symbolů na zásobník, a y je výstup.

3. V P' spojíme přechody tak, aby každý přechod četl jeden symbol. Dokud P' obsahuje stav B, z kterého vedou hrany, které nečtou žádný symbol, uplatníme jednu z níže uvedených transformací.



Vzhledem k tomu, že přechod *pop* (jediný, který bude obsahovat $z = \varepsilon$) vždy následuje po akci redukce [7] a na konci budeme odstraňovat nedosažitelné stavy, upřednostňujeme transformaci uvedenou vpravo na předchozím obrázku.

4. Odstraníme nedosažitelné stavy a z nich vedoucí přechody v P'.

Po dokončení tohoto algoritmu máme sestrojený optimalizovaný zrychlený LR analyzátor. [9]

Kapitola 4

Implementace a testování

4.1 Automatová knihovna

Automatová knihovna je psána v programovacím jazyce C++, konkrétněji jeho standardu C++11. Je sestavována pomocí nástroje `make`. [10]
C++ je multiparadigmatický programovací jazyk, který je rozšířením jazyka C. Byl vyvinut Bjarnem Stroustrupem v 80. letech.
C++ je hojně rozšířený a používaný programovací jazyk, a díky jeho využití v mnoha předmětech získají jeho znalost i studenti FIT ČVUT. Právě tento fakt je vedle verzatility tohoto jazyka důkazem vhodné volby.
Automatová knihovna je koncipována „UNIX-like“ přístupem – poskytuje několik spustitelných programů, které přijímají vstup a odesílají výstup nezávisle na sobě. K ilustraci může sloužit sekvence příkazů uvedená v příkladu 4.1.

Příklad 4.1. V této sekvenci příkazů program `arand2` vygeneruje bezkontextovou gramatiku, výstup pošle na vstup programu `anormalize2` a ten gramatiku převede do Chomského normální formy.

```
./arand2 -t CFG | ./anormalize2 -f CNF
```

Díky této vlastnosti je vhodné, aby vstup a výstup jednotlivých programů byl nějakým způsobem standardizovaný. Toho je dosaženo použitím jazyka XML. XML (*Extensible Markup Language*) je obecný značkovací jazyk. Jeho výhodou je jednoduchost, univerzálnost a možnost definování vlastních značek. V příkladu 4.2 je uveden úsek XML souboru, ve kterém je popsáno pravidlo $A \rightarrow a$ gramatiky v Chomského normální formě, která byla vygenerována sekvencí příkazů 4.1.

Příklad 4.2. Pravidlo gramatiky v XML souboru

```
...
<rules>
  <rule>
    <lhs>
```

```
        <labeledSymbol>
          <primitiveLabel>
            <String>A</String>
          </primitiveLabel>
        </labeledSymbol>
      </lhs>
      <rhs>
        <labeledSymbol>
          <primitiveLabel>
            <String>a</String>
          </primitiveLabel>
        </labeledSymbol>
      </rhs>
    </rule>
    <rule>
  ...
```

V příkladu 4.2 je vidět, že byly definovány značky `rules` pro množinu pravidel, `rule` pro samotné pravidlo, `lhs` a `rhs` pro levou a pravou stranu, atd. Převod mezi vnitřními strukturami v knihovně a samotným XML je realizován pomocí rozhraní SAX - *Simple API for XML*. SAX poskytuje sériový, nikoliv náhodný přístup k XML datům – při komunikaci s programy automatové knihovny chceme jako vstup a výstup načíst nebo vypsat celou datovou strukturu, proto tato jednodušší varianta je naprosto dostačující.
Samotné programy využívají několika knihoven, které obsahují společná data, jako algoritmy, datové struktury a obecné vlastnosti, jako např. výše zmíněný XML parser. Níže uvedený je seznam spustitelných souborů a knihoven relevantních pro tuto práci.

alib2data Knihovna obsahující datové struktury jako gramatiky, automaty, pro ně jak specifické, např. stavy, neterminální symboly, tak i společné, jako řetězec, symbol, aj.

alib2algo Knihovna obsahující algoritmy nad strukturami definovanými v `alib2data` – např. funkce *FIRST* pro gramatiky, funkce *run* pro automaty, aj.

alib2common Knihovna obsahující společné struktury pro celou automatovou knihovnu, jako převaděč z XML a do XML, obecné datové struktury jako objekt, výjimka, globální data aj.

arun2 Spustitelný soubor, který na vstupu přijímá automat a řetězec. Pro tuto dvojici je poté možné vybrat algoritmus, který se má provádět (např. *Accept*, který říká, jestli automat řetězec přijímá)

V knihovně je hojně využíváno dynamického výběru (*dispatch*). Při dynamickém výběru je z polymorfních metod vybrána na základě datového typu ta odpovídající až za běhu, oproti statickému výběru, kdy je vybrána při kompilaci. Jako příklad může sloužit kód v příkladu 4.3.
Samotné algoritmy jsou implementovány jako statické metody dané třídy. V příkladu 4.3 je tato struktura ukázána pro algoritmus *accept*, který určuje, zda daný automat přijímá zadaný řetězec.

Poznámka. V prostředí automatové knihovny náleží třídy do různých jmenných prostorů (*namespaces*). Je zřejmé, že třída `Grammar` bude náležet do jmenného prostoru `grammar`, atp. Proto, pokud to nebude situace vyžadovat, bude v ukázkách kódu jmenný prostor vynecháván.

Příklad 4.3. Implementace algoritmu *accept* v automatové knihovně

```
// obecný automat
bool Accept::accept ( const Automaton & automaton, const
    LinearString & string ) {
   return dispatch ( automaton.getData(), string );
}
// DFA - deterministický konečný automat
bool Accept::accept ( const DFA & automaton, const
    LinearString & string );
// NFA - nedeterministický konečný automat
bool Accept::accept ( const NFA & automaton, const
    LinearString & string );
// ... - metody pro další typy automatů
```

4.2 Použité existující datové struktury

V automatové knihovně bylo obsaženo již hodně struktur a prototypů, které byly v práci využity. Kromě samotných datových typů jako bezkontextová gramatika nebo symbol poskytuje knihovna kvalitní zázemí i co se týče převodu mezi objekty a XML souborem, podpůrných funkcí jako základ pro *dispatch* (dynamický výběr), nebo rozšíření STL (*standard template library*) [11]. Návrh automatové knihovny je vysoce polymorfní a není neobvyklé, aby třída dědila i od čtyř jiných tříd. Takto lze seskládat třídu z již obsažených, případně přidání několika dalších.

4.2.1 Gramatiky

V algoritmech popsaných v nadcházející kapitole bylo využíváno bezkontextových gramatik - obecných a v Chomského normální formě. Návrh těchto tříd je popsán v ukázce kódu 4.1.

Kód 4.1. Ukázka struktury tříd `CFG` a `CNF` a třídy, z které dědí

```
class CFG: public GrammarBase, public
    TerminalNonterminalAlphabetInitialSymbol {
map < Symbol, set < vector < Symbol > > > rules;
bool addRule ( Symbol leftHandSide, vector < Symbol >
    rightHandSide );
}

class CNF: public GrammarBase, public
    TerminalNonterminalAlphabetInitialSymbol {
map < Symbol, set < variant < Symbol, pair < Symbol, Symbol >
    > > > rules;
bool addRule ( Symbol leftHandSide, variant < Symbol, pair <
    Symbol, Symbol > > rightHandSide );
bool addRule ( Symbol leftHandSide, Symbol rightHandSide );
bool addRule ( Symbol leftHandSide, pair < Symbol, Symbol >
    rightHandSide );
}

class TerminalNonterminalAlphabetInitialSymbol {
set<Symbol> terminalAlphabet;
set<Symbol> nonterminalAlphabet;
Symbol initialSymbol;
}
```

Jak lze vidět, třída *TerminalNonterminalAlphabetInitialSymbol* obsahuje základ pro všechny gramatiky, konkrétnější formy gramatik se rozlišují mimo jiné např. ve tvaru pravidel. U třídy *CNF* můžeme sledovat obvyklý jev v automatové knihovně, kde jsou díky přetížení poskytovány metody, které je snazší zavolat, a interní reprezentace. Díky této vlastnosti mohu pohodlně zavolat funkci `addRule` bez toho, abych se zabýval vytvářením instance `variant`.

4.2.2 Symbol, stav

Často využívanými třídami v této práci jsou `Symbol` a `State`. Symbol je využíván v gramatikách jako neterminální a terminální, u automatů je použit ve vstupní, zásobníkové a výstupní abecedě. Porovnání symbolu se děje na základě vnitřních dat – pokud tedy chci např. najít hodnotu pro symbol A v asociativním poli (`map`), mohu vytvořit nový symbol s hodnotou *"A"* a ten předat jako parametr. Obdobně funguje porovnání u stavů, kde symboly mohou být vnitřními daty.

4.3 Implementace navržených algoritmů

V této sekci bude popsána realizace návrhu algoritmů ze sekce 3 v automatové knihovně, rozhodnutí, která jsem musel učinit a odchylky od návrhu.

4.3.1 Nalezení limitních bodů

Pro nalezení limitních bodů jsem se rozhodnul implementovat dvě metody: hladový algoritmus a feedback arc set problem. Zatímco feedback arc set problem se snaží minimalizovat počet hran, které je třeba odstranit, a tím pádem i počet přechodů *push*, hladový algoritmus nachází řešení, ve kterém je vytvořeno co nejméně „podprocesů“ pro jednotlivé limitní body.
Hladový algoritmus navíc poskytuje optimálnější výsledky, co se týče samotné detekce cyklů, díky lepší schopnosti rozlišovat hrany označujících potenciální levou a nelevou rekurzi. Díky této vlastnosti jsem se nakonec rozhodl, že primární volbou při nalezení limitních bodů bude právě hladový algoritmus.
Potenciálním kámenem úrazu je větší časová náročnost u složitějších gramatik, proto byla zachována možnost volby.
Třída `FindLimitPoints` je navržena způsobem uvedeným v kódu 4.2.

Kód 4.2. Nalezení limitních bodů

```
// nalezení limitních bodů pomocí hladového algoritmu
map <Symbol, set<Symbol> > FindLimitPoints::findLimitPoints (
    const CFG & g );
// nalezení limitních bodů pomocí feedback arc set
map <Symbol, set<Symbol> > FindLimitPoints::findLimitPointsFAS (
    const CFG & g );
```

Obě metody vrací výsledek ve stejné formě, jak by se dalo očekávat. Návratovou hodnotou je asociativní pole, kde klíčem je symbol na levé straně pravidla a hodnotou je množina všech symbolů na pravé straně pravidla, kde bude neterminální symbol nahrazen limitním bodem.

Příklad 4.4. Pro množinu nalezených limitních bodů $\{(A, \{B, C\}), (B, \{C\})\}$ bude gramatika ve tvaru

(0)	S'	$\rightarrow$	$\vdash A \dashv$	(3)	B	$\rightarrow$	BAC	(6)	A	$\rightarrow$	a
(1)	A	$\rightarrow$	aB	(4)	C	$\rightarrow$	aA	(7)	B	$\rightarrow$	b
(2)	A	$\rightarrow$	bC	(5)	C	$\rightarrow$	bB	(8)	C	$\rightarrow$	c

upravena následujícím způsobem:

(0)	S'	$\rightarrow$	$\vdash A \dashv$	(3)	B	$\rightarrow$	$BA \perp_C$	(6)	A	$\rightarrow$	a
(1)	A	$\rightarrow$	$a \perp_B$	(4)	C	$\rightarrow$	aA	(7)	B	$\rightarrow$	b
(2)	A	$\rightarrow$	$b \perp_C$	(5)	C	$\rightarrow$	bB	(8)	C	$\rightarrow$	c

Tuto reprezentaci jsem vybral pro usnadnění vyhledávání, zda neterminál na pravé straně bude nahrazen limitním bodem při konstrukci konečného automatu. Vzhledem k tomu, že metoda je volána pouze při konstrukci automatu a je specifická pro metodu zrychlené LR analýzy, jako návratovou hodnotu jsem zvolil datový typ složený ze standardních kontejnerů a symbolů z dané

gramatiky.
Z důvodů specificity problému nalezení limitních bodů je i grafová struktura vytvořená z neterminálních symbolů realizována pomocí kontejnerů a datového typu `Symbol`.

4.3.2 Konstrukce zásobníkového automatu

Po nalezení limitních bodů lze plynule přejít ke konstrukci zásobníkového automatu. K této menší odchylce od návrhu jsem se rozhodl proto, že původně sestrojený konečný automat je převeden na zásobníkový automat a postupně jsou na něj napojovány automaty pro jednotlivé limitní body. Díky tomuto lze samotnou konstrukci konečného automatu pojmout jako opakující se součást sestavení výsledného automatu.
Jako první však bylo nutné vyřešit problém s oddělovači a nultým pravidlem $S' \rightarrow \vdash S \dashv$, které gramatiku rozšiřují, a pouze díky jejich přítomnosti lze zaručit, že bude při každém přechodu zásobníkového automatu přečten jeden symbol. Po diskuzi s oponentem práce, který spravuje automatovou knihovnu, jsem se rozhodl, že nejlepším řešením bude vyžadovat gramatiku v korektním tvaru, tzn. obohacenou o toto pravidlo. V případě, že gramatika není v této formě, algoritmus je zastaven už na začátku a metoda vyhodí výjimku. Pro usnadnění rozšíření gramatiky byla implementována metoda *augment*, která bude využita i při testování.

4.3.2.1 Konstrukce konečného automatu

Konečný automat s počátečním symbolem S je sestrojen v souladu s návrhem. Největší dilema navstalo při rozhodování o návratovém typu samotné metody. Po zvážení možností jsem metodu udělal bez návratového typu a výstup je předáván přes referenci. Tímto výstupním parametrem je struktura `PDAConstructionInformation`, která obsahuje čísla stavů (společná pro zásobníkový automat) a množinu přechodů před optimalizací.

Kód 4.3. Vytvoření konečného automatu

```
void createFAFromGrammar ( const CFG & grammar,
    PDAConstructionInformation & info, Symbol initialSymbol,
    limitPoints );
```

V úseku kódu 4.3 lze vidět, že pro konstrukci konečného automatu je potřeba znát gramatiku pro získání pravidel a symbolů, limitní body pro určení typu hrany a počáteční symbol pro daný konečný automat. Výstupním parametrem je pak dříve zmíněná struktura. Datový typ `limitPoints` je stejný, jako návratový typ metody 4.2.
Při rozhodování se mezi typem hrany *push* a přečtením symbolu je třeba dávat si pozor na speciální případ. Pokud je neterminální symbol B v množině limitních bodů pro A, pak pro expanzi pravidlem $A \rightarrow B\alpha$ typ hrany bude

přečtení symbolu (později bude hrana odstraněna) – nesmí zde být typ hrany *push*, jelikož by mohlo dojít k zacyklení. Neterminální symbol B bude nahrazen limitním bodem $\perp_B$ pouze v pravidlu $A \to \beta B \alpha$.
Také je třeba si dávat pozor na pravidla ve tvaru (i) $A \to A$. Jelikož by toto pravidlo znamenalo hranu z A do A jako *redukce* i, při spojování cest v dalším kroku by opět došlo k zacyklení.

4.3.2.2 Spojení konečných automatů

Po získání konečného automatu pro počáteční symbol a pro jednotlivé limitní body je potřeba odstranit přechody, které čtou neterminální symbol, a podle návrhu upravit *push* přechody. Nyní mám k dispozici množiny všech typů přechodů a podle návrhu zkonstruuji finální zásobníkový automat. Problém možného nedeterminismu jsem vyřešil pomocí průchodu do hloubky – na zásobník si ukládám konfiguraci (stav, *push*, výstup). Symbol k přečtení a *pop* je určen přechodem ze stavu, ve kterém tento průchod začínám. Po optimalizaci přechodů vytvořím z vnitřní reprezentace v algoritmu nedeterministický zásobníkový překladový automat, který metoda vrátí.

4.3.2.3 Polymorfismus

Třídu pro tvorbu automatu jsem udělal polymorfní následujícím způsobem: ze spustitelného souboru je zavolána metoda s obecnou gramatikou jako parametrem. Podle metody dynamického výběru je následně vybrán buď běh pro `CFG` nebo `CNF`.

4.4 Návrh nových datových struktur

V samotných algoritmech je možné využívat již existujících datových struktur jako `Symbol`, bezkontextová gramatika, aj. Po dokončení algoritmu je však nutné automatovou knihovnu o některé struktury rozšířit.

4.4.1 Překladový zásobníkový automat

Zvoleným modelem reprezentace výsledku je nedeterministický překladový zásobníkový automat (`NPDTA`). Ten sice v automatové knihovně obsažen nebyl, při jeho návrhu se však šlo inspirovat již existující strukturou `NPDA`, tedy standardním nedeterministickým zásobníkovým automatem.
Hlavním problémem bylo rozšíření o výstup. Ten jsem se rozhodl realizovat jako sekvenci čísel korespondující pravidlům, která odpovídá pravé derivaci vstupního řetězce – tato pravidla jsou seřazena abecedně. Proto byla implementována nová třída, `outputAlphabet`, která bude sloužit jako společný základ pro překladové automaty. Její návrh je popsán v kódu 4.4.

Kód 4.4. Ukázka třídy `outputAlphabet`

```
class OutputAlphabet {
   set<Symbol> outputAlphabet;
   bool addOutputSymbol (Symbol symbol);
   void addOutputSymbols (set<Symbol> symbols);
   void setOutputAlphabet (set<Symbol> symbols);
   virtual bool removeOutputSymbol (const Symbol & symbol) = 0;
}
```

Jakmile byla tato třída implementována, nic nebránilo implementaci třídy `NPDTA`. V kódu 4.5 jsou popsány takové metody, které slouží k demonstraci návrhu této třídy.

Kód 4.5. Ukázka implementace překladového zásobníkového automatu

```
class NPDTA: public AutomatonBase, public
    SingleInitialSymbolPushdownStoreAlphabet, public
    SingleInitialState, public InputAlphabet, public
    OutputAlphabet {
   map<tuple<State, variant<Epsilon,Symbol>, vector<Symbol> >,
       set<tuple<State, vector<Symbol>, vector<Symbol> > > >
       transitions;
   // parametry popisují strukturu výše uvedených transitions
   bool addTransition ( State from, variant<Epsilon, Symbol>
       input, vector<Symbol> pop, State to, vector<Symbol> pop,
       vector<Symbol> output );
   // metoda removeTransition je obdobná addTransition
   // implementace virtuální metody zděděné z třídy
       OutputAlphabet
   virtual bool removeOutputSymbol ( const Symbol & symbol );
   // metody getTrasitions a getTransitionsFromState vrací
       transitions
   map<...> getTransitions;
   map<...> getTransitionsFromState ( const State & from );
   // statická metoda, která čte tokeny získané z XML souboru a
       vrací NPDTA
   static NPDTA parse (deque<Token> & input);
   // pomocná metoda, volaná metodou parse
   static void parseTransition ( deque<Token> & input, NPDTA &
       automaton );
   // metoda, která z instance automatu vytvoří posloupnost
       tokenů
   void compose ( deque<Token> & out );
   // pomocná metoda, volaná metodou compose
   void composeTransitions ( deque<Token> & out );
}
```

Aby binární soubory mohly vytvořený zásobníkový automat sestrojit z XML souboru nebo jej jako XML soubor posílat na výstup, bylo třeba rozšířit třídy `AutomatonFromXMLParser` a `AutomatonToXMLComposer`. Přidány byly metody `(compose|parse)(OutputAlphabet|TransitionOutputSymbolMultiple)`. Metody této třídy jsou poté volány z příslušných metod ve třídě `NPDTA`.

4.4.2 Běh automatu

Po sestrojení automatu je vyžadováno, aby bylo možné pro zadaný řetězec rozhodnout, jestli je danou gramatikou generován (a tím pádem přijímán sestrojeným automatem), či nikoliv. Pro tuto funkčnost je využito již implementovaných tříd `Run` a `Accept`. Metoda `Run::calculateState(s)` přijímá jako parametry automat a řetězec, simuluje běh automatu se zadaným řetězcem a vrátí uspořádanou n-tici hodnot (*tuple*), která mimo jiné obsahuje logickou hodnotu, zda automat řetězec přijímá, nebo stav, ve kterém průchod skončil. Metoda `calculateState(s)` je polymorfní v závislosti na typu automatu. Není volána přímo ze spustitelného souboru, ale metodou `Accept::accept`. Jelikož metoda `accept` může dostat na vstupu více typů automatu, opět v ní probíhá dynamický výběr. Strukturu metod tříd `Run` a `Accept` popisuje kód 4.6.

Kód 4.6. Třídy `Run` a `Accept`, kde `AutomatonType` je nahrazen konkrétním typem automatu.

```
tuple<bool,...> Run::calculateState(s) ( const AutomatonType &
    automaton, const LinearString & string );
bool Accept::accept ( const Automaton & automaton, const Object &
     object )  {
   return dispatch ( automaton.getData(), object.getData() );
}
bool Accept::accept ( const AutomatonType & automaton, const
    LinearString & string ) {
   tuple<bool,...> res = Run::calculateState(s) ( automaton,
       string );
   return get<0>(res); // bool
}
```

Ve třídě `Run` byla implementována metoda `calculateState` pro `DPDA`. Inspirovat se touto metodou šlo minimálně, jelikož bylo zapotřebí implementovat běh nedeterministického automatu. Varianta pro `NFA` zase neposkytovala vhodnou strukturu kvůli absenci ε-přechodů.
Běh automatu `NPDTA` je realizován pomocí průchodu do šířky (BFT). Proto je využita fronta, která obsahuje jako konfiguraci čtveřici *(stav, nepřečtená část řetězce, stav zásobníku, výstup)*. U průchodu do šířky je možné, že se po chvíli rozvětví natolik, že kopírovat nepřečtenou část řetězce, zásobník a výstup se stane neúnosným. Tento naivní způsob byl implementován pouze na začátku pro kontrolu správnosti.
Pro finální implementaci bylo nejsnazším vyřešit problém s nepřečtenou částí řetězce – namísto kopírovaní řetězce se do konfigurace předává iterátor. Pro zásobník a výstup byla vytvořena struktura `graphStructuredStack`. Její název sice napovídá, že se jedná o grafově strukturovaný zásobník, ve skutečnosti je to však acyklický orientovaný graf a název je odvozen z [5]. Po dosažení finálního stavu a kontrole prázdného zásobníku je výstup rekonstruován prů-

chodem přes rodiče. Operace *pop* a *push* se stávají triviální záležitostí změnou ukazatele.

Kód 4.7. Struktura `graphStructuredStack`

```
struct graphStructuredStack {
    graphStructuredStack * parent;
    Symbol data;
    graphStructuredStack ( graphStructuredStack * parent, Symbol
        data );
};
```

Kód 4.8. Metoda `Run::calculateStates`

```
tuple<bool, set<State>, set<vector<Symbol> > > Run::
    calculateStates ( const NPDTA & automaton, const LinearString
     & string ) {
    deque<tuple<State, vector<Symbol>::const_iterator,
        graphStructuredStack*, graphStructuredStack*> > bftQueue;
    auto configuration = make_tuple ( initialState, symbolIterator
        , stackNode, outputNode );
    bftQueue . push_back ( configuration );
    while ( ! bftQueue . empty () ) {
        configuration = bftQueue . pop_front ();
        if ( konec řetězce && konečný stav && prázdný zásobník ) {
            projdi outputNode a přidej do allOutputs;
            res = true;
            states . insert ( state );
        }
        // BFT
    }
    return make_tuple ( res, states, allOutputs );
}
```

Metoda `Accept::accept` byla rozšířena o variantu pro `NPDTA` a vrací logickou hodnotu *true*/*false*.
Nově byla přidána třída `Translate`. Ta byla založena na třídě `Accept`, rozdíl je v navrácené hodnotě – metoda `Translate::translate` vrací množinu řetězců. Návrh třídy je popsán v kódu 4.9.

Kód 4.9. Třída `Translate`

```
class Translate : public DoubleDispatch<Translate, set<
    LinearString>, AutomatonBase, ObjectBase> {

    set<LinearString> translate (const Automaton & automaton,
        const Object & object);
    set<LinearString> translate (const Automaton & automaton,
        const LinearString & string);
    set<LinearString> translate (const NPDTA & automaton, const
        LinearString & string);
}
```

Je vidět, že třída dědí z generické třídy `DoubleDispatch` s parametry `Translate` (implementovaná třída), `set<LinearString>` (návratový typ) a `AutomatonBase` a `ObjectBase` - tj. datové typy, pro které se bude provádět dynamický výběr. Díky této formě polymorfismu je možné v budoucnu rozšířit tuto třídu i o metody pro jiné překladové automaty, než `NPDTA`.

4.5 Spustitelné soubory

Kapitola 4.1 zmiňuje „UNIX-like“ přístup ke spustitelným souborům. Každý soubor má vymezenou množinu operací, kterou provádí. Běh binárního souboru lze ve zkratce popsat třemi kroky: jako první z argumentů určí algoritmus, který se bude provádět a z XML souboru nebo poskytnutého vstupu načte data. Tato data poté předá jako argumenty příslušné metodě a na standardní výstup vypíše XML soubor získaný z navrácené hodnoty.
Pro běh automatu je využit binární soubor `arun2`. Pro typ běhu *accept* nebylo třeba v tomto souboru dělat žádné změny – datový typ je rozpoznán za běhu díky dynamickému výběru. Jediná vyžadovaná změna tak byla zahrnutí souboru `Translate.h` a přidání typu běhu *translate*.
Pro konstrukci překladového zásobníkového automatu byl vytvořen nový spustitelná soubor, `aparse2`. Struktura všech souborů je stejná, proto byly pouze zaměněny typy běhu. Kromě samotné konstrukce obsahuje `aparse2` i možnost *augment*, která rozšíří gramatiku do takového tvaru, že bude přijat metodou `createPDAFromGrammar`.

Příklad 4.5. Konstrukce a běh automatu pro gramatiku

(0)	S'	$\rightarrow$	$\vdash E \dashv$	(3)	F	$\rightarrow$	(E)
(1)	E	$\rightarrow$	$E + F$	(4)	F	$\rightarrow$	a
(2)	E	$\rightarrow$	F				

v souboru `grammar.xml` a řetězec $\vdash (a + a) \dashv$ v souboru `string.xml`

```
./aparse2 -g grammar.xml -t pda > npdta.xml
./arun2 -a ./npdta.xml -i string.xml -t translate
```

Výstupem je množina obsahující posloupnost 424132, což je pravá derivace tohoto řetězce.

4.6 Testování

Manuálně byly otestovány různé případy, nicméně pro korektnost byly implementovány testy náhodné. Ty jsou v automatové knihovně realizovány pomocí *shell* skriptů.

4.6.1 Generování náhodného řetězce

Běh automatu přijímá jako argument řetězec. Jediný generátor náhodných řetězců, který byl v automatové knihovně implementován, byl `GenerateUpToLength`, který však generuje množinu řetězců do určité délky. Proto byla implementována třída `GenerateRandomString`. Ta jako parametr přijímá gramatiku a požadovanou délku řetězce. O tuto možnost byl rozšířen spustitelný soubor `agenerate2`.

4.6.2 Shell skript

Po prostudování ostatních testovacích skriptů a inspirací se nimi byl přidán skript `tests.aparse.sh`. Tento skript vygeneruje pomocí příkazu `./arand2 -t CFG` bezkontextovou gramatiku a pro ni vygeneruje náhodný řetězec pomocí `agenerate2`. Gramatika je poté rozšířena, převedena do Chomského normální formy a je porovnán výsledek příkazu `./arun2 -t accept` a `./agenerate -t CYK`, který rozhodne, zda řetězec je gramatikou generován. Očekávání je takové, že se výstupy těchto příkazů budou rovnat.

Závěr

Cílem této bakalářské práce bylo nastudovat algoritmy pro tvorbu zrychleného LR analyzátoru a implementovat je do automatové knihovny. Těmto algoritmům se věnují první dvě kapitoly.
Návrhem a implementací se zabývají třetí a čtvrtá kapitola. Algoritmus pro tvorbu LR analyzátoru se mi do automatové knihovny podařilo naimplementovat a otestovat jeho funkčnost. Navíc byla naimplementována funkčnost běhu takto vzniklého nedeterministického automatu, a to za využití grafově orientovaného zásobníku.
Výsledek této práce může být vylepšen implementací paralelního běhu nedeterministického automatu. Implementovaná verze je sice vylepšením oproti naivnímu algoritmu, využití vláken by však algoritmus urychlilo ještě výrazněji. Na naimplementované datové struktury lze navázat při implementaci jiných metod, např. využitím třídy `outputAlphabet` pro překladové automaty. Za zvážení by také stálo k výstupu v podobě posloupnosti čísel pravidel asociovat samotná pravidla.

Literatura

[1] Melichar, B.; Janoušek, J.; Vagner, L.: *Parsing and translation*. České vysoké učení technické v Praze, první vydání, 2013, ISBN 978-80-01-05192-4.

[2] Aho, A. V.; Ullman, J. D.: *The Theory of Parsing, Translation, and Compiling*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1972, ISBN 0-13-914556-7, 368–388 s.

[3] Chapman, N. P.: *LR Parsing: Theory and Practice*. New York, NY, USA: Cambridge University Press, 1987, ISBN 0-521-30413-X, 87 s.

[4] GNU: Writing GLR parsers. Online; navštíveno 30. dubna 2016. Dostupné z: `http://www.gnu.org/software/bison/manual/html_node/GLR-Parsers.html`

[5] Aycock, J.; Horspool, N.: *Compiler Construction: 8th International Conference, CC'99, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999. Proceedings*, kapitola Faster Generalized LR Parsing. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, ISBN 978-3-540-49051-7, s. 32–46, doi:10.1007/978-3-540-49051-7_3. Dostupné z: `http://webhome.cs.uvic.ca/~nigelh/Publications/cc99-paper.pdf`

[6] Tomita, M.: *An Efficient Context-free Parsing Algorithm for Natural Languages and Its Applications*. Dizertační práce, Carnegie Mellon University, Pittsburgh, PA, USA, 1985, aAI8517539.

[7] Aycock, J.; Horspool, N.; Janoušek, J.; aj.: Even faster generalized LR parsing. *Acta Informatica*, ročník 37, č. 9, 2001: s. 633–651, ISSN 1432-0525, doi:10.1007/PL00013319. Dostupné z: `http://webhome.cs.uvic.ca/~nigelh/Publications/czech.pdf`

[8] Šimáček, J.: *Přerušování cyklů v gramatice*. Diplomová práce, České vysoké učení technické v Praze, Fakulta elektrotechnická, Praha, 2006.

[9] Eades, P.; Lin, X.; Smyth, W. F.: A Fast and Effective Heuristic for the Feedback Arc Set Problem. *Inf. Process. Lett.*, ročník 47, č. 6, Říjen 1993: s. 319–323, ISSN 0020-0190, doi:10.1016/0020-0190(93)90079-O. Dostupné z: `http://dx.doi.org/10.1016/0020-0190(93)90079-O`

[10] FIT-GitLab: Automata library. 2016, online; navštíveno 16. května 2016. Dostupné z: `https://gitlab.fit.cvut.cz/travnja3/automata-library`

[11] cppreference: Containers library. 2016, online; navštíveno 15. května 2016. Dostupné z: `http://en.cppreference.com/w/cpp/container`

Příloha A

Seznam použitých zkratek

FIT Fakulta informačních technologií

ČVUT České vysoké učení technické v Praze

GLR Generalizovaná LR

neterminál neterminální symbol

terminál terminální symbol

XML Extensible Markup Language

CYK Cocke-Younger-Kasami

Příloha B

Manuál k automatové knihovně

B.1 Požadavky

Požadavky pro kompilaci jsou uvedeny v souboru *README.md*. Dále je potřebné mít nainstalovanou knihovnu `libxml2`.
Automatová knihovna by měla jít zkompilovat na jakékoliv Linuxové distribuci. V případě, že váš primární operační systém je jiný, je dostačující použít virtuální operační systém.

B.2 Instalace

Po změně složky spusťte příkazem `make release` v adresáři automatové knihovny. Spustitelné soubory se poté nachází ve složce *bin-release*. V případě změny kódu pouze některé z knihoven nebo spustitelných souborů lze příkaz `make` pustit i v dané složce. Poté je však třeba do složky *../bin-release* nakopírovat *bin-release/** nebo *lib-release/**.

B.3 Spuštění aplikace

Po změně adresáře na *bin-release* se daný program spustí obvyklým způsobem, například `./arun2`. Přepínač `-h` nebo `--help` vypíše základní informace o daném binárním souboru.

arun2 Očekává na standardním vstupu nebo přes parametr `-a, --automaton` automat. Parametr `-t, --type` určuje typ běhu. Vstupní řetězec je zadán parametrem `-i, --input`.

aparse2 Očekává na standardním vstupu nebo přes parametr `-g, --grammar` bezkontextovou gramatiku ve správném tvaru, nebo gramatiku v Chomského normální formě. Přepínač `-t, --type` určuje typ běhu. Výchozím

typem běhu je `pda`, tedy konstrukce zásobníkového automatu. Bezkontextovou gramatiku lze do správného tvaru upravit pomocí typu běhu `augment`.

B.4 Příklady

Ukázky obsahuje složka *examples2*. Tyto příklady jsou logicky rozděleny do příslušných podsložek.

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `automata-library.zip` ..... zdrojové kódy implementace
- `text` ..... text práce
  - `src` ..... zdrojové soubory práce ve formátu LaTeX
  - `BP_Doupal_Jakub_2016.pdf` ..... text práce ve formátu PDF